Introduction

The XPages Extension Library provides a set of new XPages artifacts that are ready to use. It is provided as an open source project on openNTF: http://extlib.openntf.org. The entire source code is available from a Subversion repository: https://svn-166.openntf.org/svn/xpages.

It had been initially contributed by IBM to support the enhancement of Notes/Domino templates. It contains a set of new XPages controls that supplement the existing by providing new capability, like:

- An application layout object for rendering the main frame
- Improved UI components, like dialog, in context form, ...
- A set of data pickers (value and name pickers)
- .... and a lot more!

It is built on top of Notes/Domino 8.5.2, using the XPages Extension API provided with this release. As such, it is also a nice example showing how to use this API.

How to get and install the library

If your goal is to use the library as is, then follow the instructions bellow. If your goal is to contribute to the library, then you'll have to install a development environment and connect to the Subversion repository. This is explained in another document.

The library should be installed on both Domino Designer to provide the components at design time, and on the Domino server to have the components at runtime. Here are the steps:

1- Get the pre-built version of the library

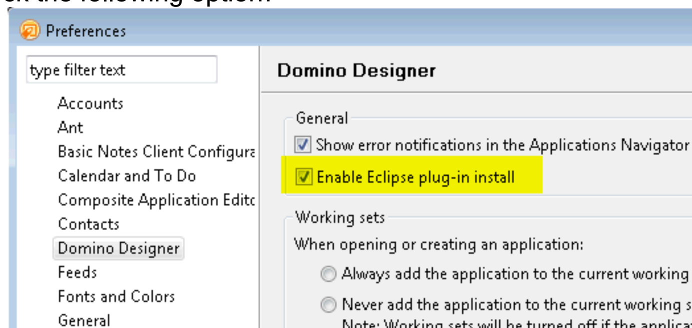
The binary files are available as an OpenNTF project available here:
    http://extlib.openntf.org/

From this site, you should get the following files:
- updateSite.zip
    This contains the Eclipse updateSite to be installed in Designer and the server.
- XPagesExt.nsf
    The sample database showing the library capability

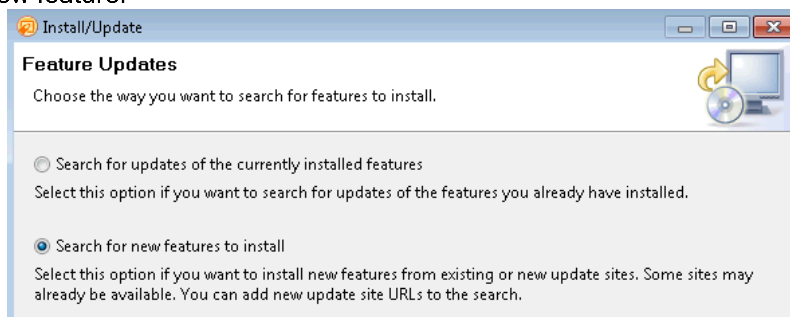
To IBMers: an internal version is available on an internal community source project located here:
    https://cs.opensource.ibm.com/projects/xpagesext/

2- Installing the library in Domino Designer

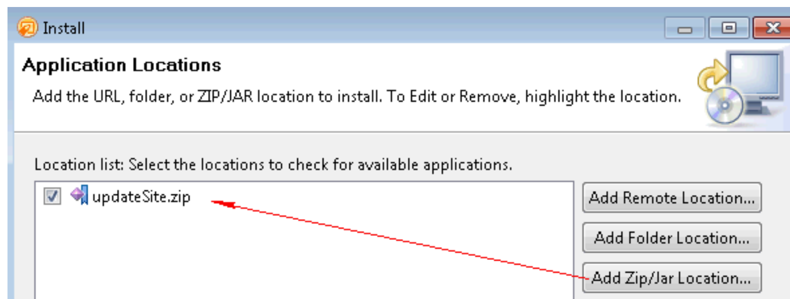
The library is provided as an Eclipse update site which should be installed using the Eclipse update manager. To enable the update manager, you should display the Designer preferences dialog and check the following option:



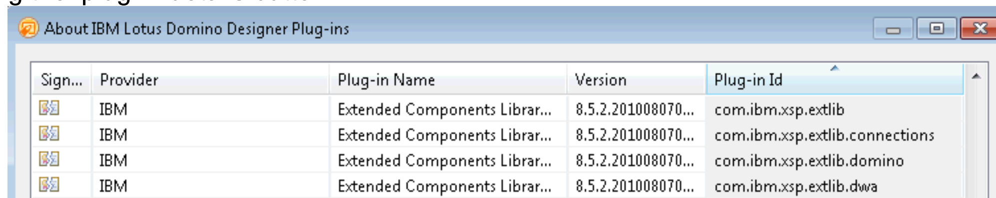
Now, the new `File->Application->Install...` menu is enabled. Select it and choose to install a new feature:



Add the Extension Library update site zip file:

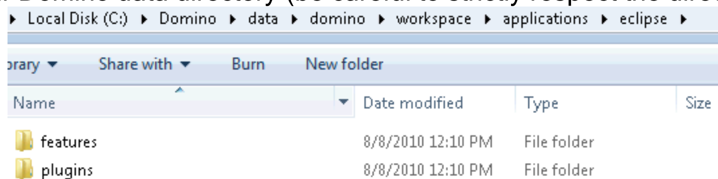


Reply to the questions and restart Designer
Then verify that the plug-ins are properly installed from the menu `Help->About Domino Designer` using the plug-in details button:



3- Installing the library in the Domino Server

The library is also installed as a set of plug-in in the Domino server, but there is no update manager UI currently available in the Domino server.
Instead, you should unpack the content of the updateSite.zip file into the following directory located in your Domino data directory (be careful to strictly respect the directory hierarchy):

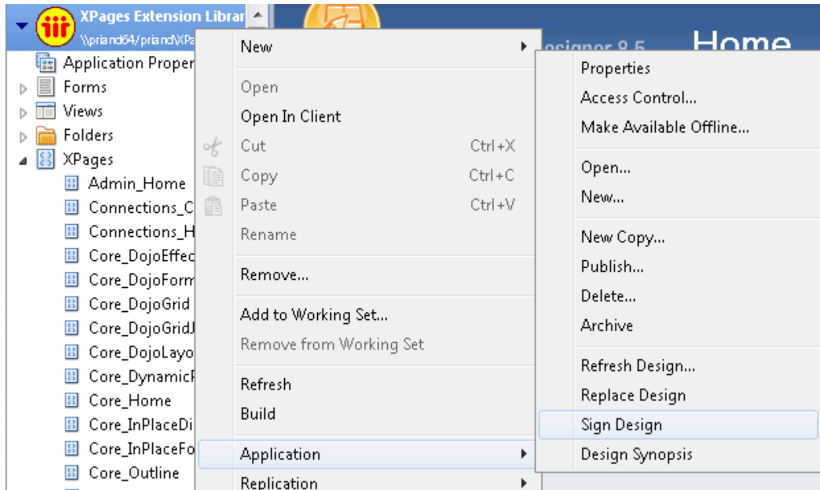


Then restart your server, or the http task. The library should be available.
To verify that the library is properly installed and running, emit the command bellow. You should get a result showing the plug-ins at least with in the "resolved" state.

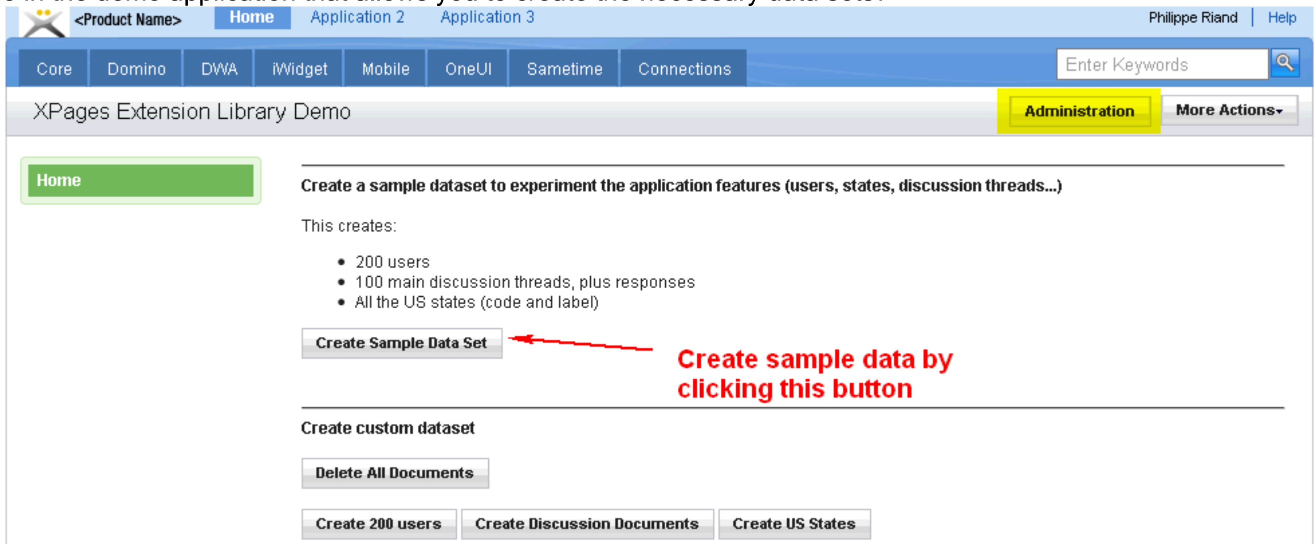
```
> tell http osgi ss com.ibm.xsp.extlib
08/08/2010 12:10:21 PM  Framework is launched.
08/08/2010 12:10:21 PM  id      State       Bundle
08/08/2010 12:10:21 PM  104     RESOLVED    com.ibm.xsp.extlib.connections_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Master=111
08/08/2010 12:10:21 PM  105     RESOLVED    com.ibm.xsp.extlib.domino_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Master=111
08/08/2010 12:10:21 PM  106     RESOLVED    com.ibm.xsp.extlib.dwa_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Master=111
08/08/2010 12:10:21 PM  107     RESOLVED    com.ibm.xsp.extlib.iwidget_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Master=111
08/08/2010 12:10:21 PM  108     RESOLVED    com.ibm.xsp.extlib.mobile_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Master=111
08/08/2010 12:10:21 PM  109     RESOLVED    com.ibm.xsp.extlib.oneui_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Master=111
08/08/2010 12:10:21 PM  110     RESOLVED    com.ibm.xsp.extlib.sametime_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Master=111
08/08/2010 12:10:21 PM  111     <<LAZY>>    com.ibm.xsp.extlib_8.5.2.201008070116NGT
08/08/2010 12:10:21 PM                      Fragments=104, 105, 106, 107, 108, 109, 110
>
```

4- Installing/running the demonstration application

The XPagesExt.nsf should be installed in the data directory of either the client of server. The design elements have to be signed from Designer or the Domino administrator to avoid security errors at runtime.



At runtime, the application requires some sample data to work properly. There is a Administration page in the demo application that allows you to create the necessary data sets:



Deploying the library to a set of servers and clients

TODO...

Installing a development environment for contributing to the library

The XPages Extension Library is entirely built in Java using Eclipse. Although Designer is based on top of Eclipse and comprises the JDT (Java Development Toolkit), it is highly advised to install a separate, pure Eclipse, development environment. This is because the extension library plug-ins should be exported to the Designer runtime, so it is more convenient from a different Eclipse instance. Moreover, it also the debugging of Designer if necessary, by starting the client in debug mode or by using the eXpeditor toolkit.

0- Prerequisites

Both the Notes client and the Domino server 8.5.2, at a minimum, should be installed. To make the library deployment experience easy, both should be installed locally, on the same machine where the Eclipse development environment is also installed. Working with remote environments is more challenging as some tools won't work (see bellow).

1- Non IBMers

1/ Installing Eclipse

The set of extra tools provided by IBM currently requires Eclipse 3.4 or 3.5 (3.6 is not yet supported). Here is the URL to download Eclipse from:

http://www.eclipse.org/downloads/packages/release/galileo/r.

>>> IBMers: for legal reasons, you must get your own copy from:
http://fullmoon.ottawa.ibm.com/downloads/

Get the Eclipse SDK release.

2/ Configuring Eclipse

It advised to change the eclipse default configuration and increase the memory allocated to Eclipse. Change the following parameters in `eclipse.ini` (located in your eclipse directory):

```
-vmargs
-Xms40m
-Xmx512m
```

3/ Installing the SVN plugins

Once installed, you should install the SVN connectors plugins. There are 2 in the market and we advise you to use the one called 'subversive'

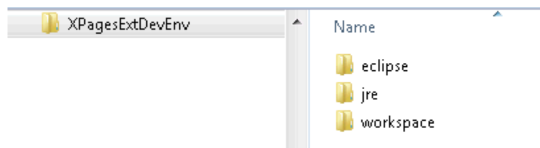
To install the plug-ins, follow these steps:

In the menu, select "Help" ->"Install new Software..."
Click "Add..."
For the location, the URL is:
http://community.polarion.com/projects/subversive/download/eclipse/2.0/update-site/
Select all of the available packages for installation and continue through the prompts until the SVN Plug-in has been installed.
When Eclipse restarts, you will be prompted to select a number of connectors to install. Select and install all of them.
Once this installation is complete, you will now be able to open the SVN Repository Exploring perspective.

Be also sure to get a copy of the IBM JRE 1.5 to compile the Java code. Even though a 1.6 JRE is supported at runtime, some platforms currently only propose a 1.5 JRE.

2- IBMers

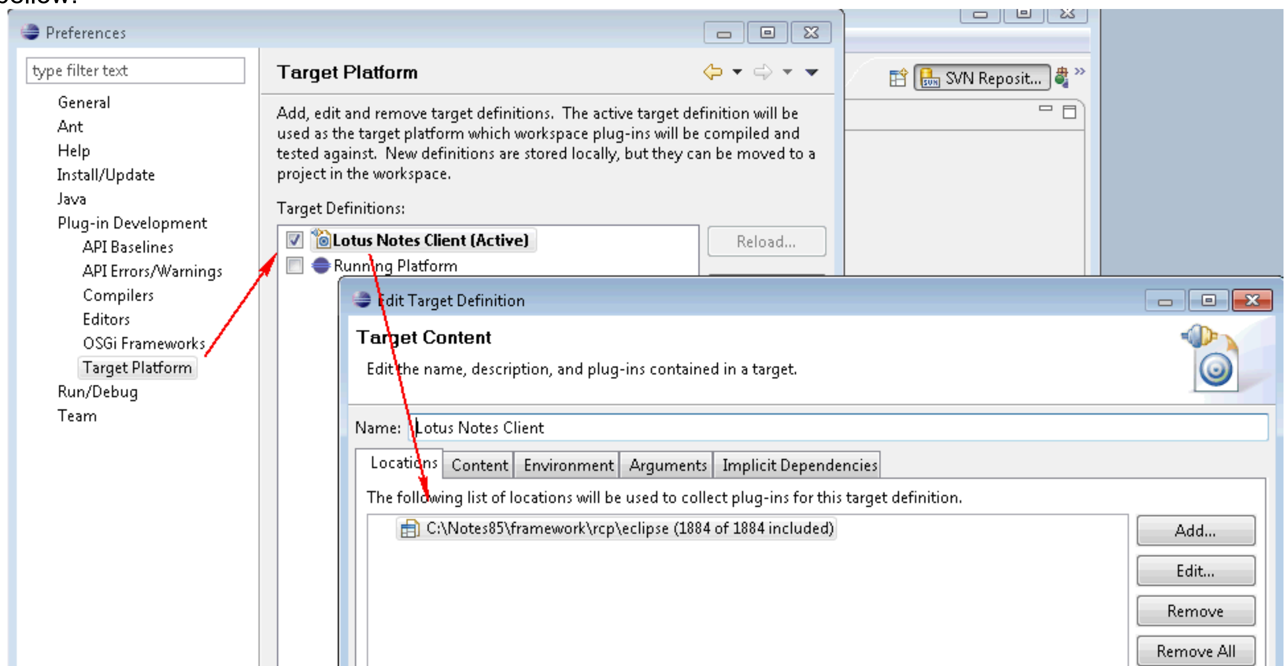
Even though the steps bellow work for everybody, there is a pre-configured version of Eclipse, windows 32 bits, available at https://fshgsa.ibm.com/projects/x/xpages-ext/ (you need a GSA account). Get the `XPagesExtDevEnv.zip` file and unpack it in its own directory:

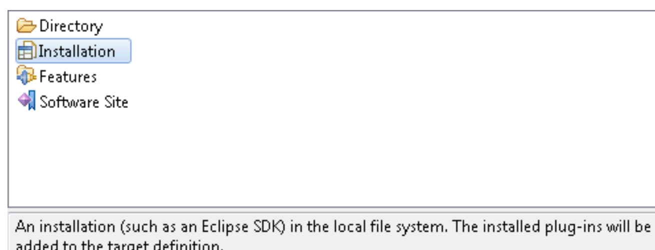


The dev environment is then ready to go, with its own copy of the IBM 1.5 JRE.
Launch the eclipse exe from the eclipse directory. For other platforms than Win32, then you'll have to follow the install in 1- (note that Win32 works also well on Win64)

3- Configuring Eclipse

To compile the library, you need to set the Target Platform of the Eclipse Plug-in Development Environment (aka PDE) set to Domino Designer. This makes all the runtime plug-ins available to your Eclipse environment.
In Eclipse 3.5, you have to create a new target platform, name it 'Lotus Notes Client' and configure it as bellow:

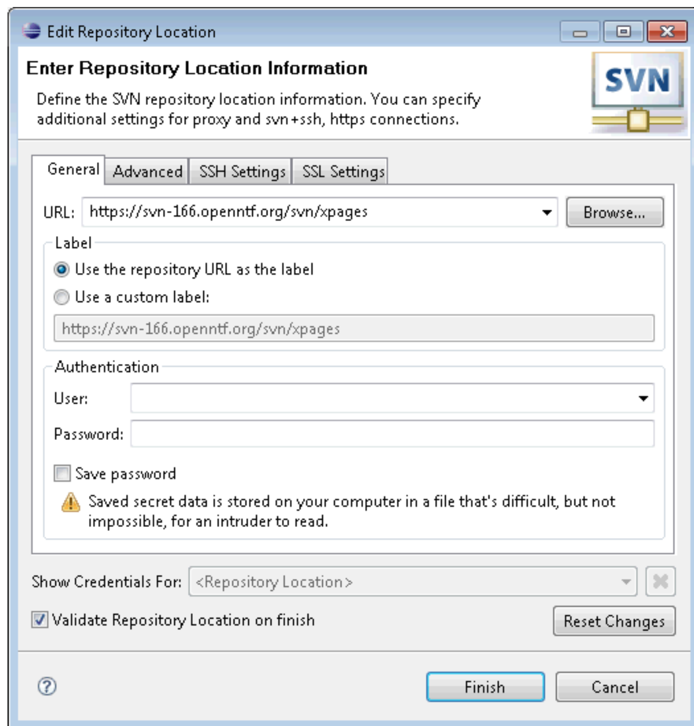




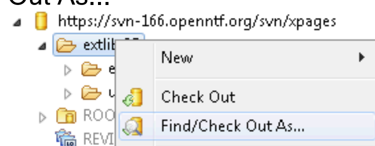
The new target platform should show something like 1800+ plug-ins. You should make it the default target platform.

4- Getting the source code from the SVN repository

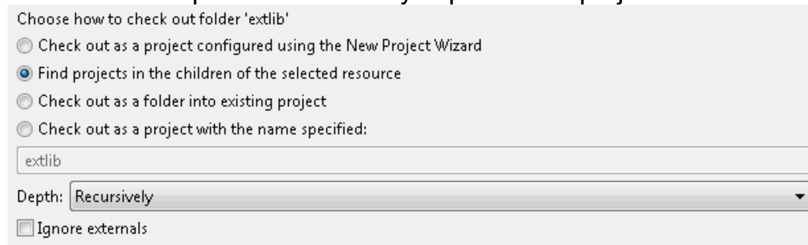
Switch to the SVN Repository Exploring perspective and add a connection to the IBM W3 community source server. Use your IBM id and your W3 password



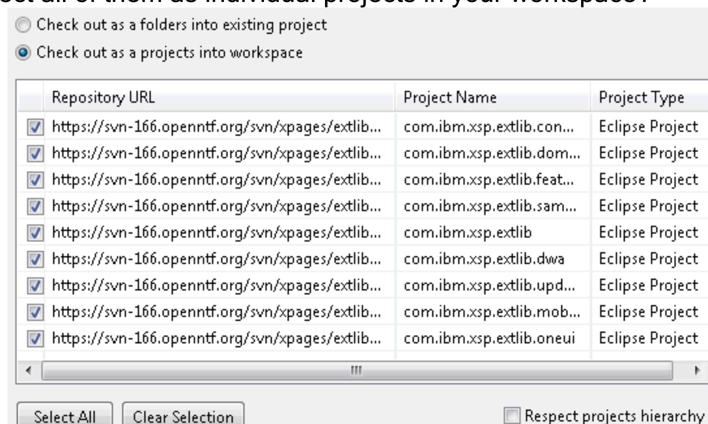
From this repository, import the projects located in the `extlib` subdirectory by selecting "Find/Check Out As..."



Select the second option to recursively import all the project in this hierarchy :



Select all of them as individual projects in your workspace :



WARN: we observed, from time to time, some timeout errors from the SVN server, particularly when connecting from the IBM network. If this happens, please retry or import the project one by one.

5- Deploying the library to Domino Designer

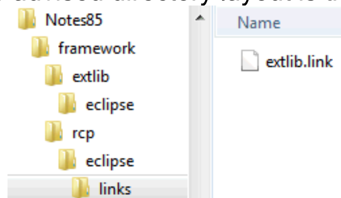
Although exporting and installing the update site definitely works, it can be cumbersome when developing the library as it requires frequent deployments. But hopefully there are some ways to make it simplier. This assumes that Domino, Notes and Eclipse are installed.

One solution is to download and install the eXpeditor toolkit (http://www14.software.ibm.com/webapp/download/nochargesearch.jsp?cat=&q0=&pf=&k=ALL&pn=&pid=&rs=&S_TACT=104CBW71&status=Active&S_CMP=&b=&sr=1&q=Lotus+Expeditor+Toolkit+&ibm-search=Search) and install it in your Eclipse development environment. This ensures that the plug-ins added to the workspace are added to your Notes/Designer runtime environment through its custom launcher. But, unfortunately, these plug-ins are *not* recognized by the JDT as part of the target platform. In short, you'll get compile errors when compiling XPages as the classes from your library won't be available to the Java compiler. To over come this issue, you'll anyway have to export the plug-ins to the platform, as explained bellow.

The other solution is to export the plug-ins from Eclipse directly to the Notes runtime, and restart Notes. This has to be done each time the XPages registry is updated (the `xsp-config.xml` files). It doesn't have to be done, for example, when the code of a renderer is changed, the registry is not updated and Designer doesn't have to be updated (unless you'd like to test the code in the Notes client rather than the Domino server).

When exporting the plug-ins directly to the Notes installation directory for development purposes, it is advised to isolate these plug-ins in a separate directory. This is done by creating a directory matching the structure expected by eclipse (eclipse/features & eclipse/plugins), and add a `.link` file into the `framework/rcp/eclipse/links` directory.

The advised directory layout is the following:



As a convenience, the following zip is containing the necessary directories and the link file, to be unzipped into your Notes directory.

ExtLib-Notes85.zip

If your Notes client is not installed in `c:\Notes85`, then you'll have to update the `extlib.link` file from this zip file.

Now, for security reasons, the Notes client does not accept plug-ins installed this way without modifying the following file:

`data/workspace/.config/org.eclipse.update/platform.xml`

!!! WARN: do not do these changes on a production environment . Only do it in your development environment. !!!

2 sets of changes have to be done:

1/ The `transient` attribute in the first config tag should be set to `false`.

2/ All the occurrences of `MANAGED-ONLY` value should be replaced by `USER-EXCLUDE` (generally in **3 places**)

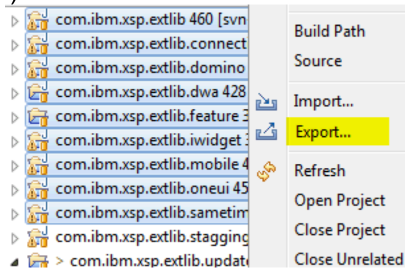
```
<?xml version="1.0" encoding="UTF-8"?>
<config date="1281358289804" transient="false" version="3.0">
<site enabled="true" policy="USER-EXCLUDE" updateable="true" url="platform:/base/">
```

This only has to be done once.

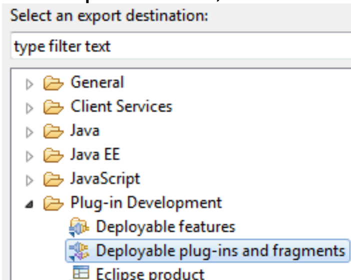
During development time, the library plug-ins should be exported from Eclipse using the following steps

1/ Select the plug-ins from your Eclipse workspace and the 'Export...' popup menu action (right

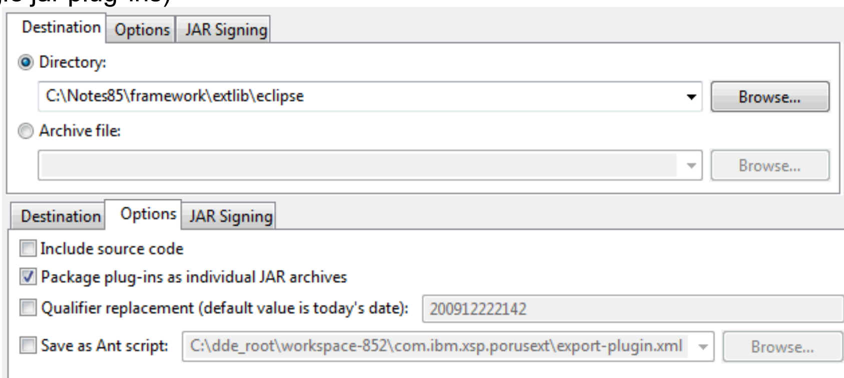
click)



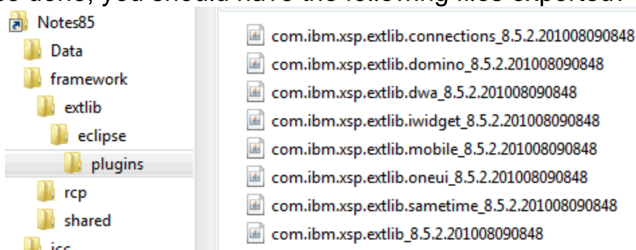
2/ In the export wizard, select the deployment of plug-ins/fragments



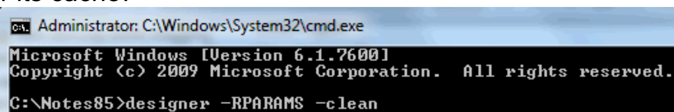
3/ Set the export options (the export directory pointing to your Notes platform, and the option for single jar plug-ins)



Once done, you should have the following files exported:



Finally, after exporting the plug-ins, you should restart Designer from the command line, asking it to clear its cache:



6- Running the Domino server in debug mode

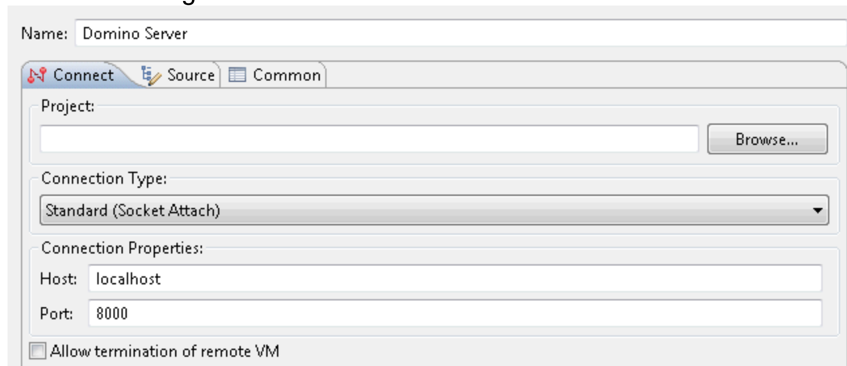
To debug your Java classes, you have to start the Domino JVM in debug mode. This is done by putting the following settings in your server notes.ini:

```
; Enabling Java debug
JavaEnableDebug=1
JavaDebugOptions=transport=dt_socket,server=y,suspend=n,address=8000
```

When restarted, the server should display the following message in its console:

```
08/13/2010 05:59:32 PM  Administration Process started
08/13/2010 05:59:32 PM  JVM: WARNING: Remote Java Debugging is enabled, resulting in decreased performance and potentially co
Listening for transport dt_socket at address: 8000
08/13/2010 05:59:33 PM  LDAP Server: Started
```

Then, in Eclipse, you need to create a launch configuration for remote debugging, pointing to port #8000. This is straight forward:

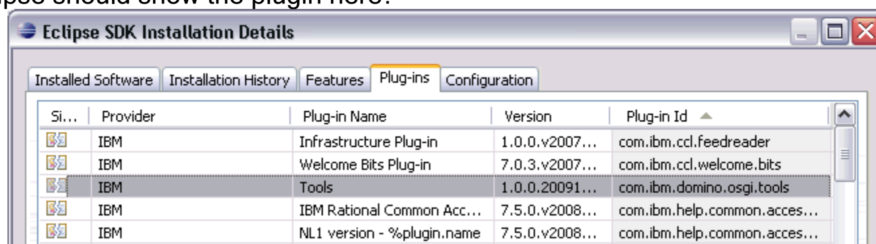


7- Deploying the library to the Domino server

*** *IBM ONLY FOR NOW* ***

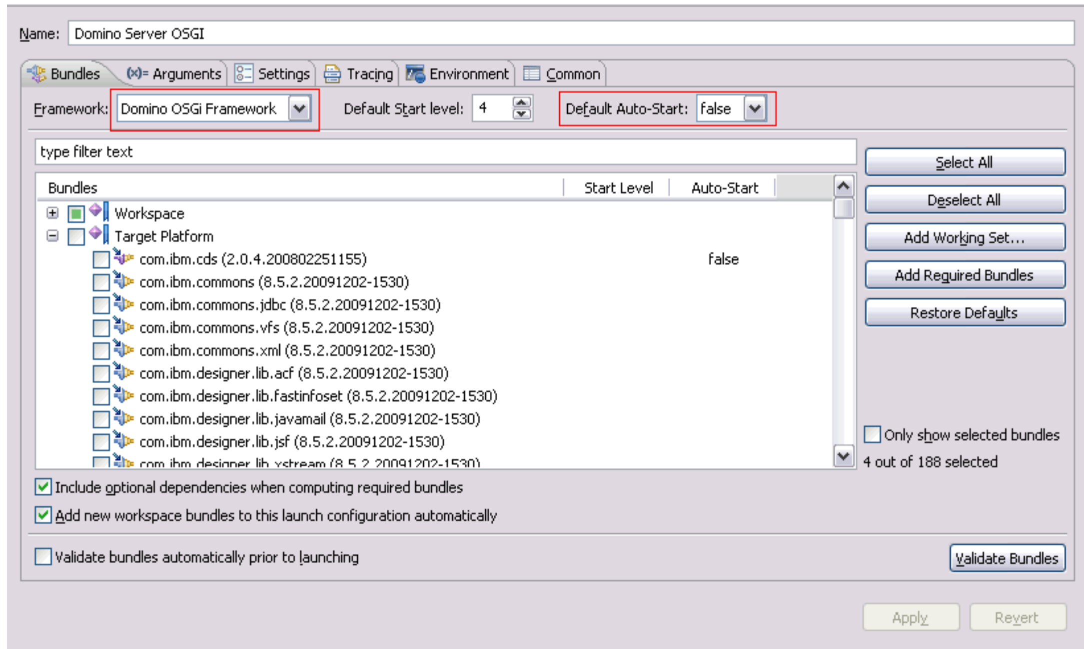
The Domino server OSGi integration features a development mechanism that let the runtime pick-up the plug-ins from an Eclipse workspace instead of the one installed in the Domino server configuration. This gives a very nice XPages development experience as the changes happening inside Eclipse are automatically reflected to the Domino runtime, without a deployment process.

This requires an extra plug-in to be installed in your Domino environment. **IBMers:** If you used the pre-configured Eclipse instance available on GSA, then this plugin is preinstalled
Eclipse should show the plugin here:

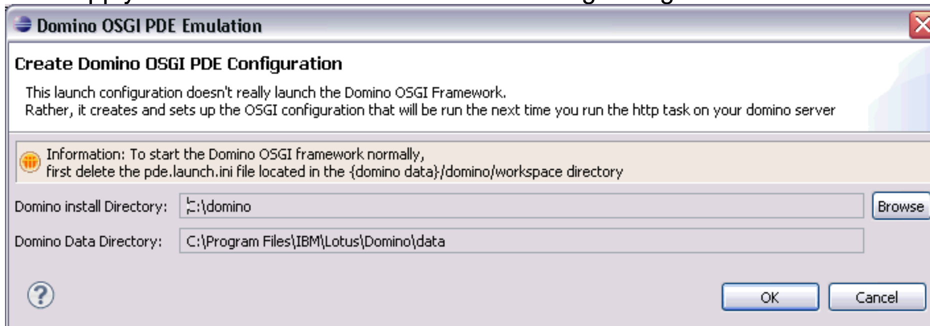


Once the plugin is install, you should create a special launch configuration. Here are the steps:

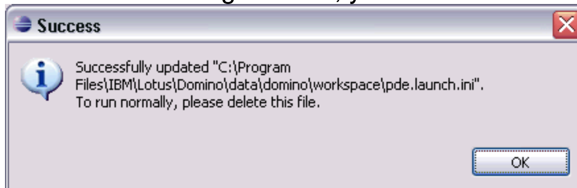
1. Use Run\Run Configurations menu
2. Select OSGI Framework on the left handside, right click and select new menu
3. **Very important:**
   - Select "Domino OSGI Framework" as the type
   - False as the default Auto-Start
   - Deselect all the plugin in the target platform (the tool will automatically add them from the domino install)
   - Select all the plugins you want to launch from the Workspace



4. Click Apply and Run. You should see the following dialog box:



5. Click OK and if all goes well, you should see this confirmation dialog



6. At this time, a config area has been created in your dev workspace metadata directory and a file called pde.launch.ini pointing to the config has been created in {notesdata}/domino/workspace
    1. Launch the http task and you will see a special warning showing that the runtime is now using the pde.launch.ini file.

```
08/13/2010 05:59:34 PM  HTTP Server: DSAPI Domino Off-Line Services HTTP extension Loaded successfully
08/13/2010 05:59:35 PM  HTTP JVM: WARNING: Using pde configuration Domino OSGi located in C:\Domino\data\domino\work
08/13/2010 05:59:38 PM  XSP Command Manager initialized
08/13/2010 05:59:38 PM  HTTP Server: Started
```

Library plug-ins organization

From a user standpoint, the library is currently using a single XML namespace:

```
xmlns:xe=" http://www.ibm.com/xsp/coreex "
```

The default prefix is 'xe', while the URI belongs to the IBM namespace.

It is unadvised that you add you own private XPages artifacts to this namespace, unless it is going to be added to the XPages Extension Library. Instead, use your own namespace, in your company/personal domain.
-> For security reasons, plug-ins that use this namespace can be disabled in the future if the library is added as a core component of Notes/Domino, and if the plug-ins are not signed by IBM

1- The plug-in architecture

To ease the maintenance and the packaging of the library, it composed of one main plug-in and several plug-in fragment

`com.ibm.xsp.extlib`
: The main plug-in, containing the core classes. It registers the library to the XPages runtime system and Domino designer.

`com.ibm.xsp.extlib.domino`
: Contains the artifacts that are specific to the Domino platform

`com.ibm.xsp.extlib.dwa`
: Contains the code shared with iNotes (formally DWA)

`com.ibm.xsp.extlib.mobile`
: Contains the code specific to mobile device

`com.ibm.xsp.extlib.oneui`
: Contains the code that manages the OneUI rendering and behaviors

`com.ibm.xsp.extlib.sametime`
: Contains the code that connects with Lotus Sametime

`com.ibm.xsp.extlib.connections`
: Contains the code that connects with Lotus Connections

All these plugins are referenced by a single feature

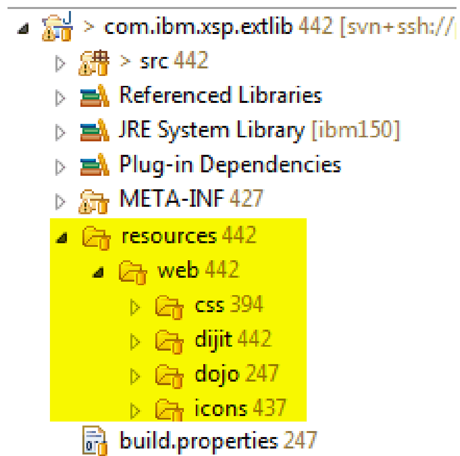
`com.ibm.xsp.extlib.feature`
: The feature referencing the main plug-in and its associated fragment

Finally, to ease the build of a ready to use update site, there is this last project

`com.ibm.xsp.extlib.updatesite`
: Update site builder, pointing to the feature

2- Accessing the resources

Some components require Dojo modules or CSS files to be added to the HTML page. These resources are served using a custom resource provider that reads their content from a directory located in the plug-in root:



The dojo resources are either located in the `dojo` or `dijit` directories, the style sheets in the `ccs` one and the images in the `icons` one.

From a client perspective, they are served using the following URL prefix: `/xsp/.ibmxspres/.extlib/`. Here is, for example, how the TagCloud control insert its required css: `/xsp/.ibmxspres/.extlib/css/tagcloud.css`. Note that these URLs might be transformed in the future if, instead of serving them through a servlet, we want to put a copy in the Domino directory

For the Dojo module, the library is automatically registering a Dojo module path with the following statement:

```
dojo.registerModulePath('extlib', '/xsp/.ibmxspres/.extlib')
```

Thus, all the Dojo modules in the `extlib` namespace will automatically be loaded using the resource provider.

Connecting to the SVN repository from Designer

The XPages Extension Library Demonstration database is also checked in the SVN repository. It uses a new extension to Designer that is available as a separate plug-in, running on top of Designer 8.5.2.

1- Installing the Designer plug -in

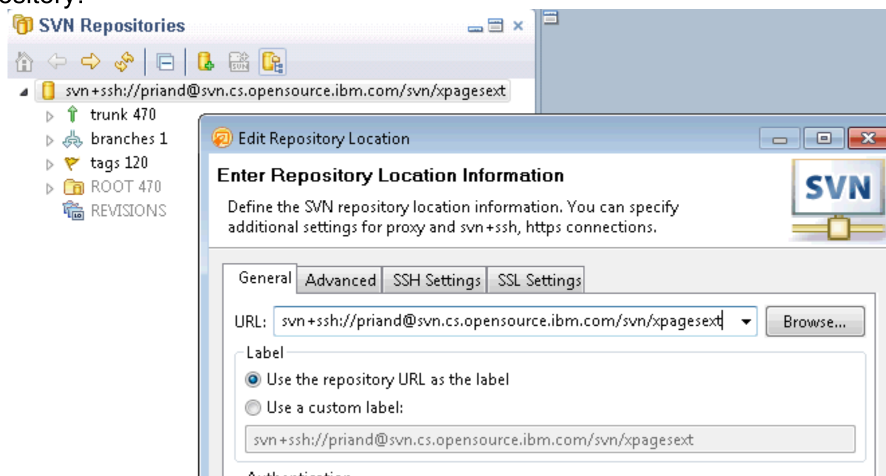
***** IBM only for now *** The instructions may change as this plug -in is being developed *****

See internal instruction here

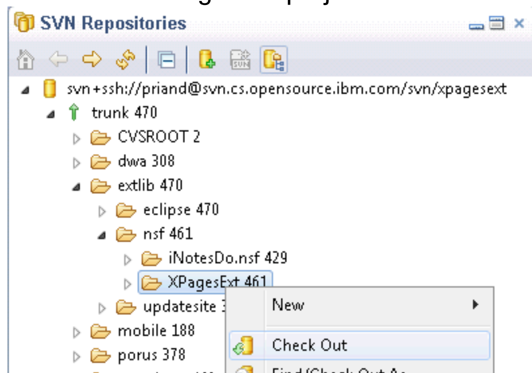
		Date	Topic
		03/04/2010	Updatesite for Code drop (RE: Source Control support in DDE)

2- Connecting to the SVN repository

You should select the SVN Repository Exploring and add a connection to the IBM community source repository:

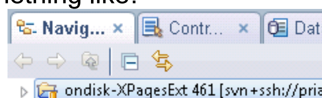


Then select the XPagesExt project and check it out (it is actually named `ondisk-XPagesExt`)

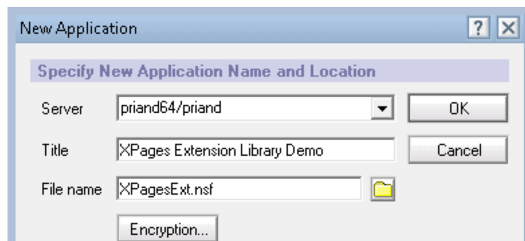


3- Connecting the SVN source code to an actual NSF

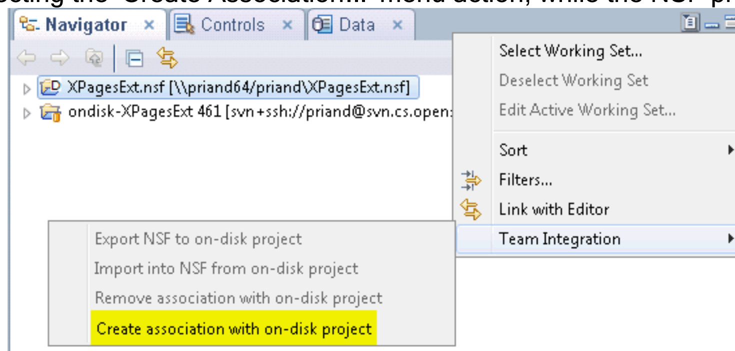
Switch back to the Designer perspective and display the Eclipse Navigator View. You should see something like:



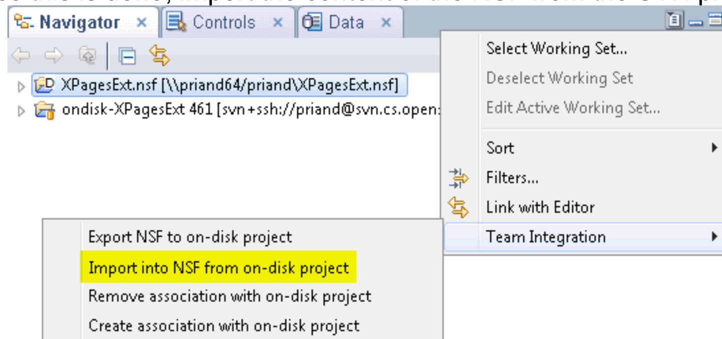
Now, create a brand new application on your server, and name its follow:



From the Eclipse Navigator view, make an association between the NSF and the SVN project by selecting the 'Create Association...' menu action, while the NSF project is being selected:



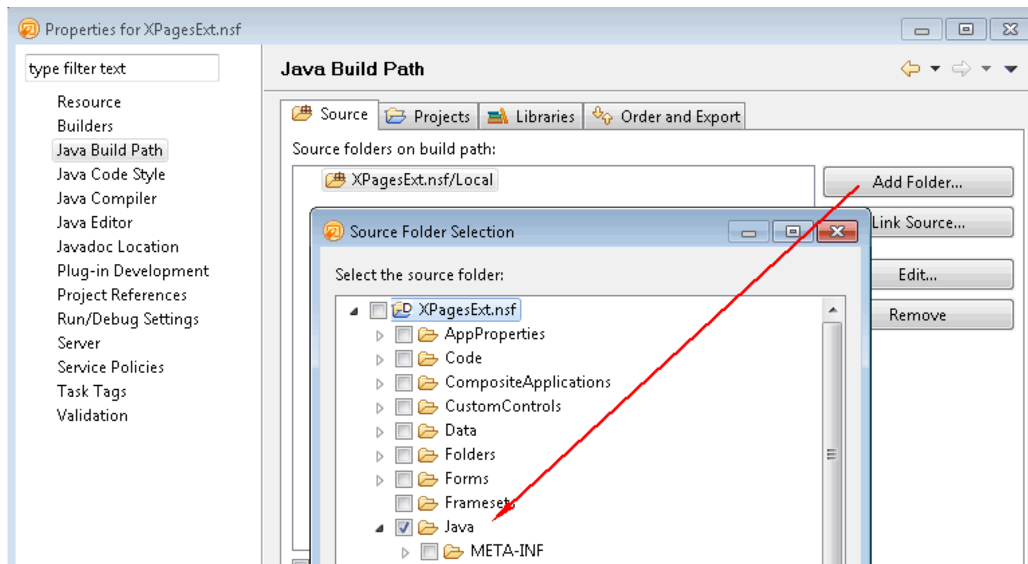
Once this is done, import the content of the NSF from the SVN project:



IMPORTANT - Current known issues with the SVN integration plug -in

4.1 - The buid path

The build path is not synchronized properly so you have to manually add the Java folder to the build path, from the project properties dialog:



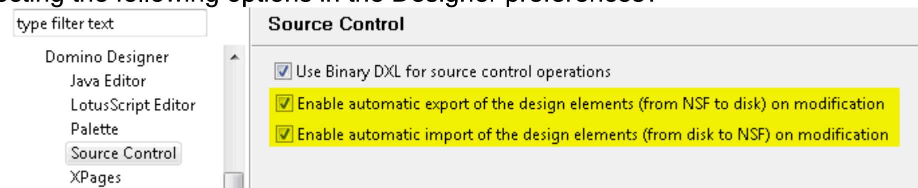
4.2 - Plug-in.xml dependencies

The custom entries in plugin.xml are not correctly check-out. You have to manually edit the plugin.xml file and add the line that imports the `com.ibm.domino.xsp.adapter` plug-in. This line MUST be before the comment `<!--AUTOGEN-START-BUILDER`

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin class="plugin.Activator" id="priand64_2fpriand_XPagesExt_nsf"
  name="Domino Designer" provider="TODO" version="1.0.0">
  <requires>
    <import optional="true" plugin="com.ibm.domino.xsp.adapter"/>
    <!--AUTOGEN-START-BUILDER: Automatically generated by null. Do not
modify.-->
    <import plugin="org.eclipse.ui"/>
```

4.3 - Synchronization

Before this comes the default behavior, ensure that the NSF and SVN projects are always in sync, by selecting the following options in the Designer preferences:



Contributing to the library

Everybody is welcomed to contribute to the library. This can be through patches when bugs are found, or brand new artifacts (controls, simple actions, data sources, dojo diijts...). The more external contributions we have, the better the XPages platform will be!

But, to maintain the library integrity and control the pedigree of the source code, the SVN repository on openNTF only enables a small set of committers. If you're not one of them and wish to submit some code, please contact Philippe Riand (priand@us.ibm.com).

Architecture & Patterns

The XPages Extension Library uses a set of patterns

Accessing request parameters

When a control or a complex property is driven by parameters in the URL, it generally uses the `requestParamMap` provided by JSF (or the `params` global object). Based on the JSF spec, this map is read only, which means that its content cannot be changed. To overcome this situation, the built-in XPages datasource try to read the request parameters from both the `requestParamMap` and the `requestMap`. As the `requestMap` is not read only (it is, in fact, the `requestScope`), custom code can push some parameters to this page. This is used, for example, by the dynamic control as it creates it children dynamically and needs to pass them parameters.

Thus, it is advised that a component/complex property reads a parameter from the map using the following pattern:

```
Object value = _requestMap.get(key);
if (value == null) {
   value = _requestParameterMap.get(key);
}
```

Separating the component from its rendering mechanism

A little background on JSF

A JSF page is made of an hierarchy of JSF components that are processed through JSF phases. One of them, in fact the last one, generates the response for the browser and is called the rendering phase. In this phase, the components are processed in order, deep first, and their following methods are called:

```
    public abstract void encodeBegin(FacesContext context) throws
IOException;
    public abstract void encodeChildren(FacesContext context) throws
IOException; /* if getRendersChildren() returns true*/
    public abstract void encodeEnd(FacesContext context) throws
IOException;
```

But this forces the component to know about its rendering. To avoid that, JSF provides the concept of renderers and render-kit. A renderer is a Java class that renders a component, while a render-kit is a set of renderers targeting a particular platform. XPages comes with a default HTML render-kit that supports HTML based browsers.
The default implementation of the component rendering methods are looking for the render to apply to the component (using its `renderType` and `family` properties) by calling one of the standard JSF function. Then, once the renderer is found, they simply delegate the actual rendering to it. Note that they also delegate the 'decoding' operation, when a page is submitted to the server.

Assigning renderers to components

Nowadays, it is accept that XPages should only render HTML. Early attempts to render SWT or WML are obsolete, as all the target platforms (web browser, Lotus Notes client, mobile devices) are accepting HTML. This ensures the maximum reuse between the platform. That said, there is no need for brand new render-kits, but simply HTML renderers for the new components, or for the existing components we'd like to render slightly differently.

The association between a component and the renderer to use is one through 2 component properties:

`family`
: Assigned by overriding the `getFamily()` method. required by JSF, if not done by a base component.

`rendererType`
: Assigned by a call to `setRendererType()`. Note that most of the existing components call this method in their constructor, to assign a default renderer.

Then, the renderer should be implemented as a Java class implementing `Renderer` class, and register it to the runtime inside a `faces-config.xml` file. As we're only going to add renderers to the existing HTML render-kit, it should be as simple as:

```
<render-kit>
...
   <renderer>
      <component-family>[My component family]</component-family>
      <renderer-type>[The name of the renderer to register]</
      renderer-type>
      <renderer-class>[Java class name implementing the actual renderer]
      </renderer-class>
   </renderer>
```

The XPages extension Library pattern

In the Extension Library, all the components are rendered through a renderer. But, for some of them, multiple renderers are available depending on the expected result. For example, the Outline component that is parameterized using a list of `ITreeNode` can be rendered as a Dojo menu, a basic

HTML list, a Dojo accordion...

TODO: review this

The user is then given 2 choices:

1- Use the base `xe:outline` component and manually assign the renderer to use

2- Use one of the derived components, like `xe:navigator`

These derived component simply assign the default renderer to a different value. They can also carry some optional parameters.

It is a very good to propose derived components, as it makes the user experience easier.

Utility classes

ExtLibResources

This class contains the resources (dojo modules, css files...) that can be added by a component during its render phase. This avoids the creation of many objects in memory. Moreover, there is method to efficiently add a render time resource to the UIViewRootEx,.

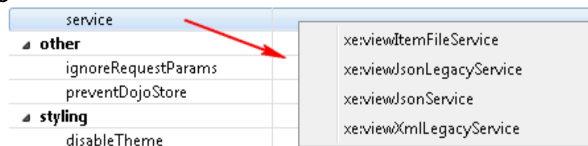
Here is an example on how to use the resources from this class:

```
ExtLibResources.addEncodeResource (rootEx, ExtLibResources.dijitToolbar );
```

Separating UI and data using Complex properties

The XPages Extension Library had been designed to clearly separate the UI (in fact, the components) from the data. For example, there is one REST service component but it can serve multiple data sources (Domino Views, Domino documents, custom sources...) that are completely unrelated. Moreover, it can serve them in different format (JSON, XML...).

This pattern is implemented using "complex properties". They do not exist in JSF, although converters/validators are specific cases of them, but are very powerful XPages. In XPages, a component can not only have basic properties (string, number, boolean...), but it can have properties that are objects.
If the class of the objects are properly registered is the XPages registry, then the "All Properties" panel in Domino Designer can display the list of classes (and their associated tag) matching the property type



Pattern implementation

The registry should define a base class, or an interface, representing the property type. For example, there is a `IRestService` interface. In the corresponding `xsp-config` file, it is defined as a simple entry with no associated tag (abstract type):

```
    <complex-type>
        <description>
            <p>Abstract type which must be extended by all REST service implementations</p>
        </description>
        <display-name>REST service interface</display-name>
        <complex-id>com.ibm.xsp.extlib.component.rest.IRestService</complex-id>
        <complex-class>com.ibm.xsp.extlib.component.rest.IRestService</complex-class>
    </complex-type>
```

Then, the REST service component (`UIRestService`) is featuring a property of this kind:

```
      <component>
            <description>A control that exposes REST services using the pathinfo information</description>
            <display-name>REST Services</display-name>
            <component-type>com.ibm.xsp.extlib.UIRestService</component-type>
            <component-class>com.ibm.xsp.extlib.component.rest.UIRestService</component-class>
            ....
            <property>
              <description>Specifies the service implementation</description>
              <display-name>Service</display-name>
              <property-name>service</property-name>
              <property-class>com.ibm.xsp.extlib.component.rest.IRestService</property-class>
              <property-extension>
                <designer-extension>
                    <category>basics</category>
                </designer-extension>
              </property-extension>
            </property>
            ....
```

Finally, concrete implementation of the REST service complex type can be provided. They don't have to be in the same `xsp-config` file, nor even to be in the same plug-in. Finally, a concrete implementation can define its own properties. Note that the hierarchy can be as complex as possible, including intermediate classes carrying some common properties (see, for example, the Domino View REST service implementation, through the `AbstractRestService` and the `DominoService` classes)

```
      <complex-type>
          <description>Base class for all the REST based services</
description>
          <display-name>Abstract Rest service</display-name>
          <complex-id>com.ibm.xsp.extlib.component.rest.AbstractRestService
</complex-id>
          <complex-class>
com.ibm.xsp.extlib.component.rest.AbstractRestService</complex-class>
              ...
          <complex-extension>
            <base-complex-id>com.ibm.xsp.extlib.component.rest.IRestService
</base-complex-id>
          </complex-extension>
      </complex-type>

      <complex-type>
          <description>Base class for all the domino based services</
description>
          <display-name>Abstract Domino service</display-name>
          <complex-id>com.ibm.xsp.extlib.component.rest.DominoService</
complex-id>
          <complex-class>com.ibm.xsp.extlib.component.rest.DominoService</
complex-class>
            <property>
              <description>Domino Database Name</description>
              <display-name>Database Name</display-name>
              <property-name>databaseName</property-name>
              <property-class>java.lang.String</property-class>
            </property>
          <complex-extension>
            <base-complex-id>com.ibm.xsp.extlib
.component.rest.AbstractRestService</base-complex-id>
          </complex-extension>
      </complex-type>

      ...
```

Using managed beans to customize controls and complex properties

Even though controls (and complex properties) can have as many properties as needed, it is sometimes not sufficient to fully customize it. It might require some custom code to execute. One solution is to provide a new component (or complex property) that inherits from the initial one, and adds the desired business logic. But this might be using a sledgehammer to crack a nut :-)

A easy solution comes by assigning a JavaBean (managed by JSF or not) to the control. JSF provide a way with the `binding` property of every control, or it can be implemented by a property in the control referencing the bean. The later is used by different controls in the XPages Extension Library.

1- Defining a bean for a control /complex property

A bean is assigned to a control through a property of type `String`. It can either be the name of a managed bean (as defined in the `faces-config.xml` file), or simply the name of a class. With a managed bean, the life cycle of the bean is well defined. With a class name, the object is created when needed.
Here is the implementation for such a bean. The `UIDynamicViewPanel` is used as an example:

1- Create a property in the control

```
private String customizerBean;
public String getCustomizerBean() {
   if (customizerBean == null) {
         ValueBinding vb = getValueBinding("customizerBean");
//$NON-NLS-1$
         if (vb != null) {
               return (String)vb.getValue(FacesContext.
getCurrentInstance());
         }
   }
   return customizerBean;
}
public void setCustomizerBean(String customizerBean) {
   this.customizerBean = customizerBean;
}
```

2- Create a method that returns the actual bean object by loading of if necessary
This method should be called by the actual bean implementation when

```
protected Customizer loadCustomizationBean(FacesContext context) {
   String bean = getCustomizerBean();
   if(StringUtil.isNotEmpty(bean)) {
         return (Customizer)ManagedBeanUtil.getBean(context,
bean);
   }
   return null;
}
```

Note the use of the method `ManagedBeanUtil.getBean`. This utility method find a bean using its name or, if the name contains at least a dot '.', it loads the class and creates a new instance of the class using Java reflection.
For performance optimization, the bean instance can also be cached in a transient variable for the component. It should never be serialized, or saved as part of a `StateHolder` property.

2- Delegating a whole complex property behavior to a bean

This technique exists to avoid the registration of new complex property classes. This is convenient for a quick customization within an application, without having to feed the registry using xsp-config files. In this case, there is one single complex property implementation that references a bean, and delegate all its methods to the bean. The bean class must implement the same interface than the complex property.

The `BeanValuePickerData` class is a good example of a value picker complex property that delegates its behavior to a bean.

Navigation and Layout

The concept of 'Trees' in the UI

When looking at a typical Web UI, a lot of parts

Application Layout

Navigator and outline components

New XPages Controls

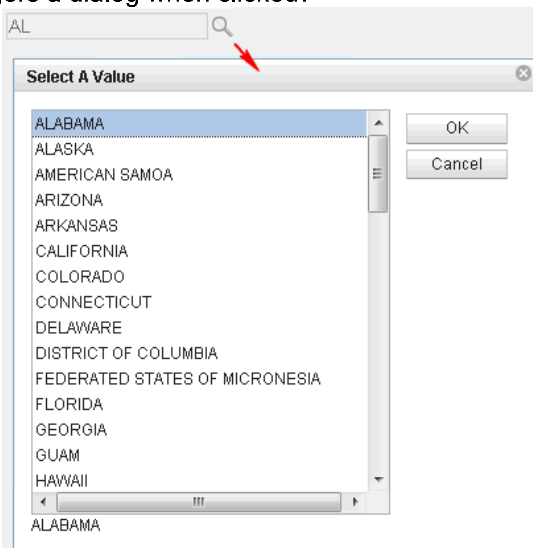
The value and name pickers - See Core_ValuePicker.xsp/Domino_Pickers.xsp

The XPages Extension Library is equipped with two new powerful controls that enhance the user experience:

- A value picker
  The value picker let the end user choose one or multiple values within a list.
- A name picker
  The name picker let the end user choose one or multiple names within a list

Using the pickers

Both pickers work similarly. When the XPages renders its content, each picker renders an icon which triggers a dialog when clicked:



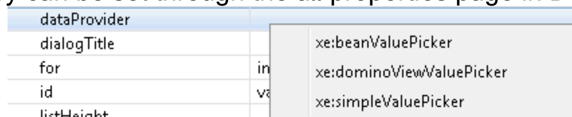
A picker is in fact a true JSF component, that is connected to an InputText through its `for` property. When the picker dialog is displayed, then it grabs the value(s) of the InputText to set the initial selection. When the dialog is closed, then it puts the new selected items back to the InputText .

As the pickers have access to the InputText during the render phase, they know if the it accepts single or multiple values, thus they adapt their behavior accordingly. They also grab the character used to separate the multiple values if appropriate.

Although there is only one ValuePicker and one NamePicker component in the palette, they can display data from various sources and have different renderings, depending on the needs. This is controlled via a set of properties

Selecting the data to display

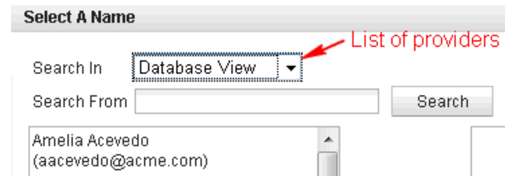
The pickers use a complex property member, `dataProvider`, to access the data. This clearly separates the UI from the data, thus allowing different data sources to be plugged in.
As any other complex property, each data provider can have its own set of parameters (properties). They can be set through the all properties page in Designer:



This list shows the different providers coming out of the box. But this is fully extensible, on a global or per application basis.

The name picker also allows the end user to optionally from choose multiple directories. This is done by using a data provider aggregator, which is a custom data provider that delegates to some other concrete implementations. Here is, for example, a name picker that provides data from both a Domino address book and a view in the current database:

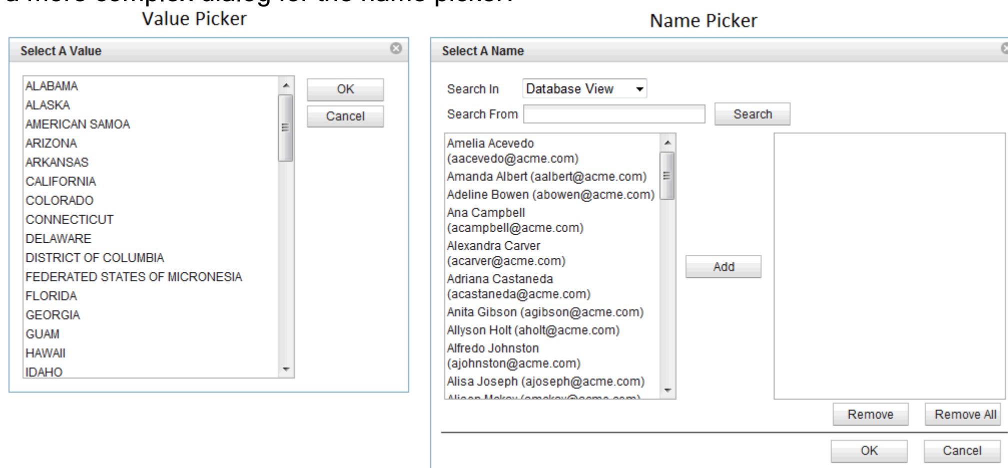
dataProvider	xe:namePickerAggregator
dataProviders	Aggregator
dominoNABNamePicker [0]	
addressBookDb	Domino NAB
addressBookSel	
groups	
loaded	Domino View
people	
dominoViewNamePicker [1]	
databaseName	
label	Database View
labelColumn	Name
loaded	
viewName	AllEMails



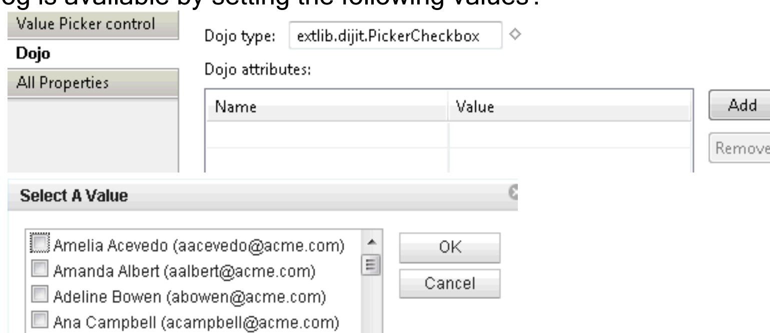
Note that the values returned by the data providers can have an optional label. In this case, the label is being displayed by the dialog instead of the value. But the value is still going to be stored by in the input field.

Setting the Picker UI

Both the value and the name picker come with a default rendering. It is a basic list for the value picker, and a more complex dialog for the name picker:



But other user interfaces can be provided. As the UI is provided through Dojo dijits, a custom UI can be assigned using the `dojoType` and `dojoAttributes` properties. For example, a checkbox based dialog is available by setting the following values:



Enabling typeahead

XPages also comes with a built-in TypeAhead component which can complement the pickers. Similarly to the value picker, it requires to get access to data and, for this purpose, can be connected to the picker. It then takes advantage of its data provider. The picker JSF components feature an adapter that adapts the data coming from the data provider to what the typeahead control expects, whenever it is a basic list of data or some custom markup.

Here are the steps to connect the typeahead control to the picker:

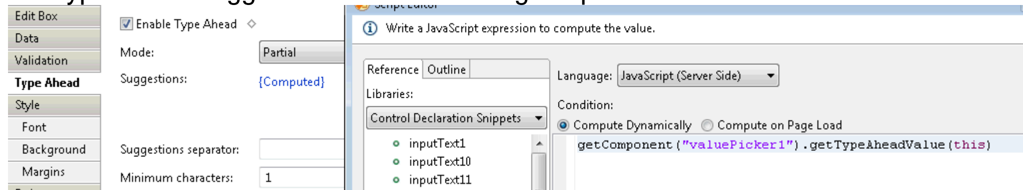
1- Add a typeahead control to the target input text and set the following properties:

```
<xp:typeAhead mode="partial"  minChars="1"  preventFiltering=
```

```
"true" >
```

`preventFiltering` ensures that the typeahead doesn't filter the entries by itself, but let the data provider doing it. This is not absolutely required, but makes the control performing better. Samething for `mode="partial"`, which ensures that only the typeahead component is triggered when a typeahead request comes in.

2- Make the typeahead "suggestions formula" calling the picker code:



The getTypeAheadValue() method returns what is expected by the typeahead control.

A better integration between the 2 components might be done in the future, but it will require some changes in the typeahead control itself. This is beyond what a library can provide.

Pickers architecture

On the client side, a picker is a custom Dojo dijit that pops up a dialog. The entire dialog is defined only using client side HTML and JavaScript. It doesn't require any XPages to run on the server.
There is also a generic dojo module, `extlib.dijit.Picker`, that adds a function to the global `XSP` object. This very lightweight module is automatically included in each page that contains a picker. When the function is called, it dynamically loads the desired dialog dijit and activates it.

The communication between the dijit and the data provider on the server is done through REST calls. The picker server side component acts as a REST service provider, which returns some compacted JSON data, complying to the format expected by the Dojo `ItemFileReadStore` object. A URL to the service can be constructed at runtime by calling the picker `getUrl()` method (see: `AbstractPickerRenderer`). The content of the REST service response is automatically composed by calling the data provider `readEntries()` method.

The REST service is expecting some parameters in the URL to drive the result:

- `start`: index of the first item in the list.
- `count`: maximum number of items to return.
- `source`: index of the source to lookup (e.g.: the index of the directory to lookup)
- `key`: the key used to filter the data. Only the items starting with this key will be returned
- `startKey`: a slight variation of the previous parameter. It returns the values starting from the first item with a key that is greater or equal to this value. The key doesn't have to match, just have to be greater or equal.

Providing custom data providers

Custom data providers can be provided beyond the out of the box ones. Here are the different possibilities.

1- Using JavaScript for the value picker - `simpleDataProvider`

The data provider implementation allows a developer to execute a piece of Server Side JavaScript. To keep the JavaScript simple, this provider can only return string values, without associated labels.

Property	Value
◢ dataProvider	xe:simpleValuePicker
labelSeparator	
loaded	
valueList	# var a = new Array("a","b","c","d","e","f","...
valueListSeparator	

Here the kind of SSJS formula that can be coded:

```
var a = new Array("a","b","c","d","e","f","g","h")
return a
```

To make it even easier, it can return a single string along if `valueListSeparator` is not empty

```
return "a,b,c,d,e,f,g,h"
```

2- Using a managed bean for a value/name picker - `beanXXXPicker`

This method gives the full capability of a data provider, without having to register a brand new complex type to the XPages runtime. Basically, this method can be chosen when the provider is specific to an application and doesn't have to be shared across applications.

The steps are easy:

1- Create a Java class that implements either `IValuePickerData` or `INamePickerData`, depending of the kind of data. Look, for example, at the `SimplePicker` class in the demo application

2- Add a `beanXXXPicker` data provider and make it points to your bean

If a class name is specified, then the an instance is created when needed. To get a better control of the life cycle of the bean, use a managed bean and use its published name in the `dataBean` property.

Property	Value
◢ dataProvider	xe:beanValuePicker
dataBean	extlib.pickers.SimplePicker
loaded	

Providing a custom UI

Similarly to data providers, a developer can provide a custom UI for a picker. It has to be done via a custom Dojo module that defines a new dijit. This dijit should extends `extlib.dijit.TemplateDialog`. Although it is not formely required, extending this base class provides help for displaying the popup dialog, managing the browser differences and handling some common events (ok, cancel...).

The XPages runtime generates a set of parameters that are passed to the dijit at runtime, like the url of the REST service to call. Here are the typical parameters that are passed to the dijit

- `dlgTitle`: the title of dialog
- `control`: the client id of the associated input text
- `url`: the REST service URL
- `sources`: an optional array of source (directory list)

Note that this can be extended, if needed, by using custom picker renderers.

For more information, please look at the `PickerCheckbox.js` file, which defines the simplest dijit. It shows how the REST service should be called, how it communicates with the InputText and how it creates checkboxes based on the data list.

The Tag Cloud

Tooltips

Extended UI: beyond the static pages

XPages are generally static pages defined in Domino Designer. When a page is requested by a browser, the the XPages runtime creates an hierarchy of JSF components in memory and start to process it through the JSF lifecycle. Once created, the hierarchy of components remains identical in memory, until the page is discarded and another one loaded. This happens, for example, after a redirect.

The XPages Extension Library goes beyond this standard behavior by allowing the JSF tree created from the page definition to change at runtime. This permits some new designs where the UI is adapting to a context (ex: in context editing) or when a temporary UI is needed (ex: modal dialog). Although the principle is easy to understand, some particular care has to be taken when dynamically changing the tree of components. In fact, XPages extends JSF in multiple ways (FacesComponent, Themes...) and the dynamic creation of controls must comply with those extensions

The basis of a dynamic UI

An XPages in represented in memory by an hierarchy of components, inheriting from the base `UIComponent` class. Each JSF component can contain a list of children, as well as a list of named facets. Even if both collections can be accessed using the standard `getChildren()` or `getFacets()` methods of the `UIComponent` class, some custom code should *not* programmatically add new components to these collections. It might seem to work, but some subtle side effects can be seen afterwards. Let's explain why, and what should be done.

This chapter explains the basis of the dynamic component creation.

1- Constructing an XPages

When an XPage is constructed, then an in-memory hierarchy of components is created, starting from the root component. The creation is done recursively, by automatically creating the children of a component right after the component is created. This is know as a "deep first" tree processing.

For example, when using the page bellow

```
<?xml version="1.0"  encoding="UTF-8" ?>
<xp:view xmlns:xp=" http://www.ibm.com/xsp/core " >
   <xp:panel id="panel1" >
         <xp:panel id="panel2" >
               <xp:panel id="panel3" ></xp:panel>
         </xp:panel>
   </xp:panel>
   <xp:panel id="panel4" ></xp:panel>
</xp:view>
```

The components are created in the following order:

- 1: `panel1`
  - 1.1: `panel2`
    - 1.1.1: `panel3`
- 2: `panel4`

2- How components can customize the creation process

Although the default creation process if fine for most of the components, some other would like to customize this process. For example, an `include` component will read its children from another page.

To let a component customize the tree creation, XPages added the optional `FacesComponent` interface. When a component implements it, then it gets notified when it is constructed, and gets a chance to change the tree creation. Here are the methods exposed by the `FacesComponent` interface:

```
public void initBeforeContents(FacesContext context) throws FacesException;
```

Called after the component is created and all of its properties assigned. It gives it a chance to change the property values before the children are constructed.

```
public void buildContents(FacesContext context, FacesComponentBuilder builder) throws FacesException;
```

Called after `initBeforeContents` to construct the actual children of the component, including the facets. It generally simply delegate to the `FacesComponentBuilder` object, which knows how to create the actual component children from the XML definition.

```
public void initAfterContents(FacesContext context) throws FacesException;
```

Called after the component and all its children are created. It gives to the component an opportunity to re-arrange the hierarchy of children, of change some of the children properties.

If the `FacesComponent` interface is optional, the components that implement it expect the 3 methods

above to be called at the appropriate time. Failing to call them would result in undetermined behaviors.

Although components should be notified during their creation, they are not getting any notification from when they are discarded. As a result, some custom code can safely remove children or facets from a control by simply removing them from the collections.

3- Injecting controls into the tree at runtime : ControlBuilder

The easiest way for creating and inserting new control into the JSF tree is by using the `ControlBuilder` helper class. It construct the JSF tree as it if was coming from the XML definition. The definition of the control content is defined using an hierarchy of `ControlBuilder.IControl` objects. Each of this object should be able to create the runtime `UIComponent` with all its property values set, as well as the children/facets `IControls`.

This technique can successfully be used when a piece of the UI should be created dynamically, based on an external definition of the page. For example, a dynamic view panel can create its columns based on the view design element definition. Or a survey tool can create an entry form based on the list of questions.

A `ControlImpl` helper makes it easier when using this builder. Here is, as an example, how `UIDynamicViewPanel` control dynamically constructs its columns:

```
// The view control already exists, it is simply wrapped into a
ControlImpl
// We then create the columns  and ask the control builder to actually
// add the columns to the view panel and call the FacesComponent methods
ControlImpl viewControl = new ControlImpl(this);
int index = 0;
for(Iterator<ColumnDef> it=viewDef.iterateColumns(); it.hasNext();
index++) {
    ColumnDef colDef = it.next();
    IControl viewCol = createColumn(context,bean,index,colDef);
    viewControl.addChild(viewCol);
}
ControlBuilder.buildControl (context, viewControl,true);
```

4- Controlling the tree creation

Another way to deal with dynamic trees is to control the tree creation, and delay the construction of some components. Let's take an example of a component that creates it children lazily, on demand. In this case, it should should prevent the children from being created when the page is loaded first, and create them when asked for by a method call.

Here how the method that creates the controls should be implemented

```
public void addChildren(FacesContextEx context) throws FacesException {
    DynamicUIUtils.removeChildren (this, true);
    DynamicUIUtils.createChildren (context, this, getId());
}
```

Note that most of the low level details are implemented within the `DynamicUIUtils` utility class. Its *`createChildren`* method loads the Java class generated by Domino Designer and calls it to create the desired component, identified by its id, and its children hierarchy. In our case, as the component already exists, its has to move the children from the newly created component to the one we already have, and just discard the new one. All of that is encapsulated in the utility class, but has a drawback: as a new, temporary, component is created, it should not block the creation of its children while being constructed. Thus, its implementation of `buildContents` should be done as follow:

```
public void buildContents(FacesContext context, FacesComponentBuilder
builder) throws FacesException {
    // If we are building a dynamic component, then we should create the
children
    if(DynamicUIUtils.isDynamicallyConstructing (context)) {
```

```
            // Temporarily reset this flag so another InPlace container,
    child of this one, won't be constructed.
            DynamicUIUtils.setDynamicallyConstructing (context, false);
            try {
                buildDynamicContents(context, builder);
                return;
            } finally {
                DynamicUIUtils.setDynamicallyConstructing (context, true);
            }
        }
        // Normal stuff here...
        super.buildContents(context, builder);
    }
```

The `DynamicUIUtils` class is providing a flag that a component can check

.

Page processing : execution id and partial refresh id

1- The component ajax ids

When an Ajax request is sent by the browser to the XPages runtime, it can contain 2 optional ids (in the form of client ids):

1- The execution id

When a POST request is sent by the browser, the XPages runtime execute the full page lifecycle (all the phases, including the apply value phase, the validation one). By default, this is applied to the full page unless an execution id is specified. In this case, only a portion of the tree, starting from the component with this id, is processed. This means that the components that are not is this subtree are not updated with values coming from the client, the values are not validated and even the data models (ex: the domino document) are not updated.

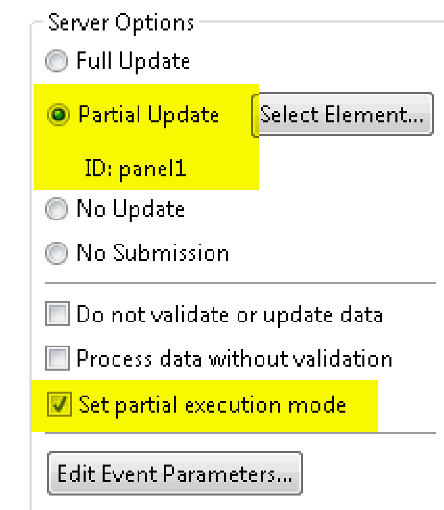
2- The partial refresh id

It defines the part of the UI that should be refreshed during the rendering phase. Only the HTML generated by this component and its children is returned by the server. The client then replaces the old HTML snippet corresponding to this component with the new content

A developer controls the value of those 2 ids from the event handler component. Each time a user attaches an event using the Domino Designer events panels, an event handler component is generated in the source code. Here is for example the code generated when handling a button `onclick` event:

```
<xp:button value="Label"  id="button1" >
   <xp:eventHandler event="onclick"  submit="true" refreshMode=
"complete" >
         <xp:this.action><![CDATA[#{javascript:print("You clicked me!")}
]]></xp:this.action>
   </xp:eventHandler>
</xp:button>
```

Both the exec id and the refreshid are controlled using some event handler properties, available from the event panel or the All Properties panel:



or

execId	
execMode	partial
handlers	
id	
loaded	
navigate	
refreshId	panel1
refreshMode	partial

In short, they have to be specified on a per component basis.

2- Forcing the component Ajax ids

In some cases, we like to 'isolate' a portion of the JSF tree and ensure that only this portion of the tree is processed when something happens inside it. For example, when a modal dialog is displayed, a server action triggered by a button should only refresh the content of the dialog, and not the entire page. It should also only process the data pertaining to the controls within the dialog, and not the one outside it. This is the behavior we want unless the developers specifies something else.

It is actually achieved by forcing the event handlers contained in a sub tree to trigger ajax requests. The `FacesContextEx` has a property called `subTreeComponent` that instructs the event handler when the processing should be restricted to a portion of the panel. This property should be set by a component, in its `encodeBegin` method, and restored in its `encodeEnd` method.

Here is, as an example, how the a typical control would implement this capability:

```
public void encodeBegin(FacesContext context) throws IOException {
    FacesContextEx ctx = (FacesContextEx)context;
    UIComponent oldSubTree = ctx.getSubTreeComponent();
    ctx.setSubTreeComponent(this);
    // Use a boolean for a null value as the AttributeMap does not
support nulls
    getAttributes().put("xsp.subtree.component",oldSubTree!=null
?oldSubTree:Boolean.TRUE );
    super.encodeBegin(context);
}

public void encodeEnd(FacesContext context) throws IOException {
    super.encodeEnd(context);

    if(getAttributes().containsKey("xsp.subtree.component")) {
          FacesContextEx ctx = (FacesContextEx)context;
          Object oldSubTree = getAttributes().get("xsp.subtree.component"
);
          ctx.setSubTreeComponent(oldSubTree instanceof
UIComponent?(UIComponent)oldSubTree:null);
    }
}
```

Many components in the XPages Extension Library make use of this capability. A common, generic implementation is located in `UIDynamicControl`.
`

Dynamic Views - See Domino_DynamicView.xsp

As exposed in the previous chapter, the dynamic view panel control is probably the simplest example of a dynamic UI. It dynamically creates a set of columns depending on the view it points to. It uses the `ControlBuilder` utility class to actually create the controls and inject them in the JSF tree.

1- Creating the columns

The columns are created from the view referenced by the data source. But this view name van be computed, and the resulting name might change between 2 calls. The dynamic view panel should adapt to such changes and recreate a new set of columns when the view has changed.
The creation of the columns is done when the view panel is rendered. The columns are left as is if had been already created for the view, or recreated as necessary:

```
public void encodeBegin(FacesContext context) throws IOException {
    updateColumns(context);
    super.encodeBegin(context);
}
```

The design of the view is encapsulated into a set of classes located in the `ViewDesign` one.

2- Caching the view design element

To optimize the performance of this control, the definition of the view are loaded by the `ViewFactory` class, declared in the `ViewDesign`. A default implementation factory is caching the view design in the application scope. This ensures that the design is not read too many times from the NSF.

3- Column customization

The view panel control provides a lot of options for the columns. Some can be deduced automatically from the view design element (ex: the column title), some can be guessed (ex: which column should display a link, defaults to the first column), and the other should be provided through sample code.

To override the default rendering behavior, the dynamic view panel uses an optional Java bean implementing the `Customizer` class.

In Context Editing - See Core_InPlaceForm.xsp

1- What is "in context editing "

In context editing is one of the new Web UI design patterns, popularized with the use of Ajax technology. It consists in dynamically showing an entry form around some data, and generally after a user click action. The entry form let the end user edit the data close by.

Here is, for example a list of user:

User: Lorraine, Blevins Edit
User: Rosanna, Mccoy Edit
User: Phoebe, Livingston Edit
User: Jonathan, Jacobson Edit

Each user has an Edit link that inserts a form bellow the user entry and let people edit the information:

User: Lorraine, Blevins Edit
User: Rosanna, Mccoy Close

First name:	Rosanna
Last name:	Mccoy
E mail:	rmccoy@acme.com
City:	Chesapeake

Ok

User: Phoebe, Livingston Edit
User: Jonathan, Jacobson Edit

A click on the Ok button or on the Close link removes this entry form, with or without saving the data.

2- How this works

In the XPages source, there is an `inPlaceForm` control which contains the controls to be displayed in the form.

Previous 1|2|3|4|5 Next

User: {computedField1}, {computedField2} {link1}

xe:inPlaceForm

First name:	FirstName
Last name:	LastName
E mail:	EMail
City:	City

Ok

/xe:inPlaceForm

This `inPlaceForm` control is always created when the page is constructed, but its children are not. As a result, the runtime cost of the function is minimal when not used: none of the children controls are created in the tree, and none of the data sources defined in the children are loaded.

Now, the `inPlaceForm` control content can be displayed or hidden by calling the following server side methods:

```
var c = getComponent("inPlaceForm1")
c.show()
c.hide()
c.toggle()
```

The control will automatically create and inserts the child controls into the JSF tree, or delete them, as necessary. Moreover, the control can be located with a repeat and it maintains a list of contexts in which it is visible. The controls are removed from the JSF tree when there is no longer valid contexts.

Note that the `show()` or `toggle()` methods have an optional `Map` to emulate some URL parameters. The content of the map is temporarily pushed to the `requestScope` and removed after the request is processed.

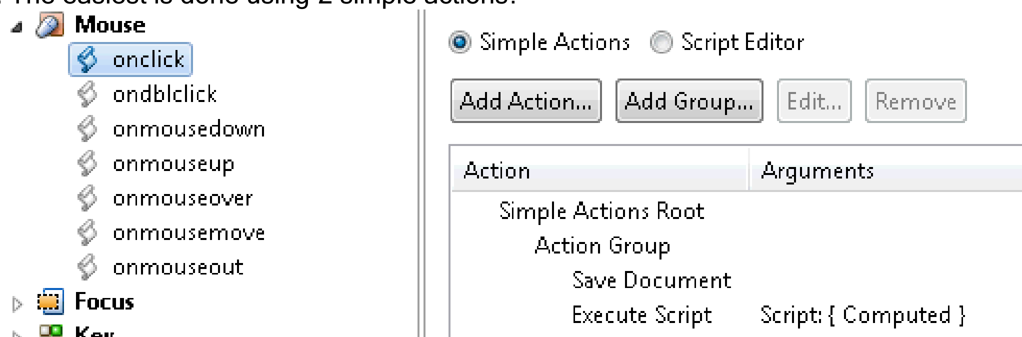
3- Implementing a Ok button in the form

Implementing such a button is not as trivial as calling show() or hide(), because some server side processing should be done (e.g.: saving the document). Moreover, the browser should not redirect to a different page if the update was successful. It should rather stay on the same page and refresh a portion of it (generally, the piece showing the data it just edited, to get it with the new values).

The steps are then the following:

1- Add a simple button as the ok button, and *not* a submit button

2- Add a server side event that both saves the document and then ask the form to hide its children. The easiest is done using 2 simple actions:

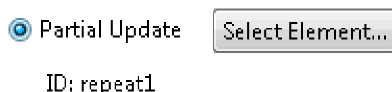


If the data source is correctly saved, then the 'Execute Script' is executed. It should simple hide the form like this:

```
var c = getComponent("inPlaceForm1")
c.hide()
```

3- Make the `onclick` event do a partial refresh of the edited value. The refresh id should at least contain the entire form plus the edited value to be refreshed.

In case the form is editing a row within the repeat control, refresh the entire repeat control:



4- Accessing the current context

The `inPlaceForm` control, as well as its children, can access the current page context. This means, for example, that an `inPlaceForm` located in a repeat can access the current repeat context. Here is an example of a form opening a document, using a document data source, for the current repeated NoteID:

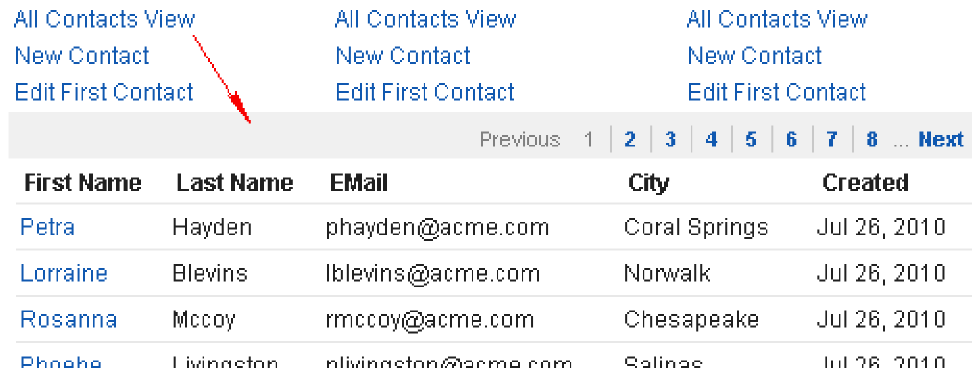
```
<xe:inPlaceForm id="inPlaceForm1" >
   <xp:panel>
         <xp:this.data>
               <xp:dominoDocument
                     var="document1"  formName="Contact"  action=
"editDocument"
                     documentId="#{ javascript:row.getNoteID() }"
                     ignoreRequestParams="true" >
               </xp:dominoDocument>
         </xp:this.data>
```

As we want to force the data source to use the computed properties, we prevent it from using the actual requests parameters using `ignoreRequestParams="true"` .

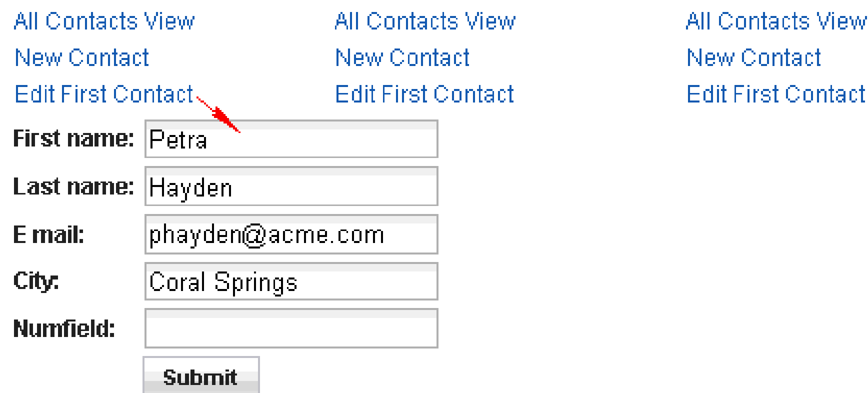
Dynamic UI parts - See Core_DynamicPage.xsp

The XPages Extension Library offers a new control that create a different UI depending on the context. Basically, this control can replace its entire set of children by a new hierarchy, presenting a completely different UI.
Here is an example, where the control can either display a view or a form when the user clicks a link. The browser in fact stays on the same page as the click triggers only a partial refresh of the content of the control. On the server side, the dynamic content control is replacing its children

After a click on 'All Contacts View':



After a click on 'Edit First Contact':



1- The Dynamic UI control (`UIDynamicContent`)

When added to a page, this control doesn't have any children but a set of facets.Each facet represents one of the UI that can be displayed. As the facets are named, their name is used by the control to select which one should be displayed. A dynamic can have as many facets as needed:

```
<xe:dynamicContent id="dynp" defaultFacet="view" useHash="true" >
   <xp:this.facets>
         <xp:panel xp:key="view" id="panel1" >
               <xp:viewPanel rows="10" id="viewPanel1" var="row" >
                     ...
               </xp:viewPanel>
         </xp:panel>
         <xp:panel xp:key="contact" id="panel2" >
               <xp:this.data>
                     <xp:dominoDocument var="document1" formName=
"Contact" ></xp:dominoDocument>
               </xp:this.data>
                     ...
         </xp:panel>
   </xp:this.facets>
</xe:dynamicContent>
```

This is done by calling a simple server side API. For example, the code bellow instructs the dynamic control component to discard all its children and replace them by the content of the facet 'view'.

```
var c = getComponent("dynp")
c.show("view")
```

The `show()` method also have a optional parameter used to pass URL parameters to the control children. Those parameters are added temporarily to the `requestParameterMap`, and can be consumed by the controls/complex properties. Built-in data sources are reading the parameters from this map, like the document data source:

```
var c = getComponent("dynp")
c.show("contact",{action:'editDocument'
,documentId:dataAccess.firstContactID})
```

A typical link implementation will then call the `show()` method and partial refresh the dynamic control.

Unless specified otherwise, the dynamic UI control forces a partial execution/refresh of its children. This can be reverted using this property:

partialExecute	false

2- Default content

When the control is displayed initially, it can either show nothing or use one of the facet as the default one:

Property	Value
basics	
autoCreate	
binding	
defaultFacet	view
id	dynp

3- Keeping track of the content

It is sometimes interesting to keep track of the current facet being displayed, as well as the parameters, in the browser URL. This allows a user to bookmark the page and to use the back button. Unfortunately, changes to the URL forces the browser to refresh and send a request to the server. As we'd like to stay on the same page only using Ajax refresh, the only URL update that doesn't trigger a page refresh is when the # part of the URL is modified. The technique of using this # part is well known by JavaScript developer. The dynamic UI control can automatically managed this for you, if the following property is set:

useHash	true

The URL is updated when a new facet is selected:

```
http://.../XPagesExt.nsf/Core_DynamicPage.xsp#content=contact&action=edi
tDocument&documentId=8FA
```

A Dojo listener is also installed for tracking the URL changes by the user (going to a bookmark, hitting back/next buttons...). When a change is detected, then an Ajax request is automatically sent to the server and the content gets updated.

As a result, and when this option is enabled, the dynamic UI control can be controlled by a piece of client side JavaScript:

```
XSP.showContent("#{id:dynp}","view")
```

or by a simple relative URL:

```
#content=view
```

On the server side, the parameters from the hash are automatically added to the `requestMap` or the `requestParameterMap`, thus making it transparent to the data sources and other controls

4- Implementing the Ok button

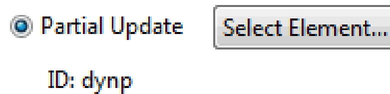
When there is an Ok button in the dynamic content, it cannot be a regular submit button as this would redirect to a different page on success. Instead, the button should be a simple button, better implemented with the following simple actions:

Action	Arguments
Simple Actions Root	
Action Group	
Save Document	
Execute Script	Script: { Computed }

The script in the second action is simply resetting the content to its default, like the 'view' here:

```
var c = getComponent("dynp")
c.show("view")
```

The button click should also only partial refresh the dynamic content panel:



5- Hooking a click on a view column

When a click happens on a view column in the dynamic UI, we'd like to dynamically switch to the entry form, without refreshing the entire page. A solution consist in adding an `onclick` handler to the view column, and use one of the technique to change the dynamic content (server side code, client side code or relative url). Here is an example using some SSJS:

```
var c = getComponent("dynp")
var id = row.getNoteID()
c.show("contact",{action:'editDocument',documentId:id})
```

6- Known issues

The first issue is related to Domino Designer, as the Facets are not rendered, which makes the edition of the page a source code experience:

The second have to deal with search engines. Even though the context is tracked using the # part of the URL, this part is not sent to the server by the browser. It requires some interpretation by some client side JavaScript code, which then triggers an Ajax requests. Unfortunately, the current robots from the major search engine providers do not interpret the JavaScript within the page. As a result, they are not able to retrieve and index the dynamic content.

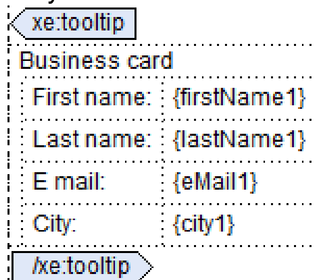
Dynamic Tooltips - See Core_Tooltip.xsp

Another interesting control that creates its dynamically is the tooltip control. From a rendering standpoint, it uses a Dojo tooltip dijits and it gets its content from an Ajax request.

As a tooltip is read only, the server side controls used to render the tooltip are created when the request comes in and are discarded right after they rendered. Said differently, the JSF controls are inserted temporarily to the tree when the tooltip is rendered, and removed right after. This ensure that the tree stays as clean and small as possible, thus bringing the best overall performance.

1- Adding a dynamic tooltip

Similarly to the `inPlaceForm` control, the `tooltip` one gets its children directly form the markup:



Its `dynamicContent` has to be set to `true` to make it the content being created dynamically:

dynamicContent	true

Of course, the children controls have access of the context where the tooltip is located, to for example fill the parameters of a data source:

```
<xe:tooltip id="tooltip2"  for="computedField2" dynamicContent="true" >
   <xp:panel>
         <xp:this.data>
               <xp:dominoDocument var="document1"
                     formName="Contact"  action="editDocument"
                     documentId="#{ javascript:row.getNoteID() }"
                     ignoreRequestParams="true" >
               </xp:dominoDocument>
```

Modal Dialog - See Core_InPlaceDialog.xsp

Another interesting capability we see nowadays in web 2.0 applications is the use of modal dialogs. Since pop-up windows are generally disabled in browsers, those dialogs are implemented using <div> tags with the page itself. Te implementation of such a feature is beyond the scope of this document but, fortunately for us, the Dojo library has the code implemented. This is what we're going to use.

1- Known issues with the dialogs

If you ever tried to use a Dojo dialog within an XPages, you probably faced some issues as it does not integrate well with the JSF lifecycle. There are multiple reasons for this:

1- The form tag

XPages actions require a form tag to be added to the page. This is transparently done by the XPages runtime so the developer doesn't have to worry about it.

```
<body ...>
    <form id="....>
```

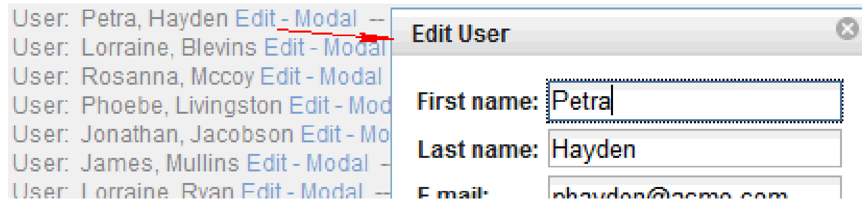
But the Dojo dialog implementation moves the dialog related tags and positions them as direct children of the `<body>` tag. As a result, the dialog is then moved outside of the form tag and any XPages action stops working. Some developers came with very creative solutions to work around it, but none is generic enough

2- Limiting the server side processing to the controls it contains

Unless the developer specifies an execution id and a partial refresh id to all the event handlers in the dialog content, the full page will be processed, while it have to be limited to the dialog content.

3- Let the dialog access the current context

For example, when displaying a view, a click on the 'Edit' link should display a dialog with the correct document:



It means that the construction of the dialog should be done just before it's getting displayed, and it should then get access to the page context.

4- Closing the dialog

Finally, a click on the Ok button should run some server side actions (saving the documents...) and close the dialog in the browser.

2- The XPages Extension Library solution

The XPages Extension Library provides a new control to the palette that deals with all the technical issues mentioned above. Similarly to the `inPlaceForm` control, the dialog content is defined as child controls. But those controls are actually not created until the dialog is displayed. This ensure the best possible performance as none of the control is initially created, or the data sources opened.

The content of the dialog is defined as any other XPages content, and can access the current context. The dialog above, for example, is located in a view row. It accesses the current note id or the row to

open the document:

```
<xe:dialog id="inPlaceDialog1"  title="Edit User" >
   <xp:panel>
         <xp:this.data>
               <xp:dominoDocument
                     var="document1"  formName="Contact"  action=
"editDocument"
                     documentId="#{ javascript:row.getNoteID() }"
                     ignoreRequestParams="true" >
               </xp:dominoDocument>
         </xp:this.data>
```

Moreover, the dialog implementation ensures that only the components in the dialog are processed when a server side action is triggered. It also forces all the event handler to only partial refresh the dialog body. Finally, it manages the uses of a different form tag, specific for the dialog.
That makes the use of dialogs almost transparent to the XPages developer.

3- Displaying the dialog

A modal is triggered by a client side action, using the following method:

```
XSP.openDialog('#{id:inPlaceDialog1}')
```

When this is called, an Ajax request is submitted to the server to create the dialog content in the JSF tree. A Dojo dialog window is created, displaying the result of the Ajax call.

4- Implementing the Ok button

Similarly to the other dynamic controls, the Ok button should be implemented with a standard button (not a submit one) featuring 2 simple actions:

Action	Arguments
Simple Actions Root	
Action Group	
Save Document	
Execute Script	Script: { Computed }

The first one saves the data sources, wile the second executes the following script:

```
var c = getComponent("inPlaceDialog1")
c.hide()
```

But, generally, a button action should partial refresh a another part of the page. This is particularly true when the Ok button is clicked, as it means that some data had been modified and the page should reflect the changes. Unfortunately, it is not as simple as setting the button onclick partial refresh target, as the dialog should not be closed, nor the refresh happen, if the ok action wasn't executed properly (after a validation failure, for example). In that case, the dialog should be refreshed.

The solution comes from an extended version of the `hide()` method, which takes some extra parameters, like the component to refresh after the dialog closed:

```
var c = getComponent("inPlaceDialog1")
c.hide("repeat1")
```

The first parameter is the id of the control to refresh. It can also feature a map of parameters that will be used when emitting the partial refresh request. This might be used in rare cases.

Note that the `hide()` method generates a piece of client side script that closes the dialog and optionally execute a partial refresh. The JSF controls are also properly removed from the JSF tree.

The dialog can also be closed by the user clicking the close button in the dialog title bar. In this case, no request is sent to the server thus leaving the tree with the dialog controls inserted. But they will be automatically removed the next time the page will be rendered,

Dojo Control Encapsulation

Dojo Form Controls

Dojo Layout Controls

REST services architecture

Extending the JavaScript Interpreter

Mobile Controls

The extension library contains a set of controls that can be used to develop easily mobile apps for webkit based browsers like iPhone, iPad, Android, Blackberry OS6, etc.

The following chapters describe how to use the demo application and how to use the controls to build your own mobile apps.

How to run the Demo App

In order to run the sample app you need to get in addition to the extension library the 'MobileDemo.nsf' plus you need an appropriate simulator or actual device to open the XPages page.

1 - Non IBMer: Get MobileDemo .nsf

Non IBMers can get the database from the OpenNTF project 'XPages Mobile Controls':
http://mobilecontrols.openntf.org/
TBD: add details

Extract the NSF into your Domino data directory.

2 - IBMer: Get MobileDemo .nsf

IBMers can get the database from the following SVN repository:
svn+ssh://[userName]@svn.cs.opensource.ibm.com/svn/xpagesext/trunk/extlib/nsf/diskMobileAppNsf

Create a new NSF called 'MobileDemo.nsf' in the Domino data directory and connect it to the local on disc project. See chapter 1.5 for details.

Additionally one thing need to be done manually for now since the SVN plugin cannot handle it yet:
Set the ACL appropriately (e.g. allow anonymous access for testing purposes)

Then open the XPages page Setup.xsp and hit the button 'Create Blog Documents'.

3 - Prerequisites

In order to try the sample app you can use different browsers or devices.

The best test is always a specific actual device. If you don't have this or if you need to debug your app you can use different browsers/simulators.

The mobile controls can even be run in Firefox since there is a compatible module which simulates the missing webkit functionality. This allows using Firebug to debug network transactions, to use the console, debug client side JavaScript, etc.

Safari and Chrome can be used too and they come closer to the actual experience since both are webkit based.

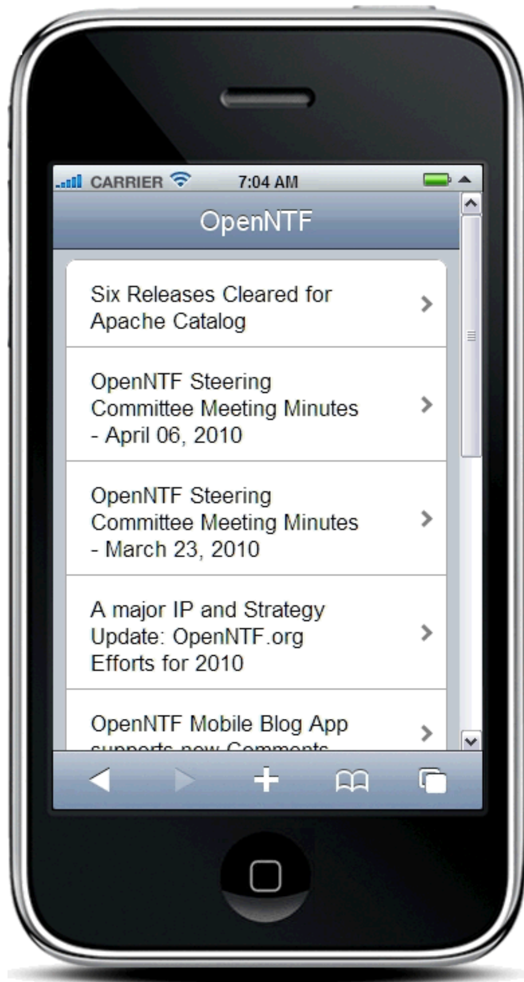
Simulators come often even closer to the actual experience and allow taking good looking screenshots and videos. The different SDKs all provide simulators. Another nice simulator is Genuitec MobiOne that you can get here: http://www.genuitec.com/mobile/download.html

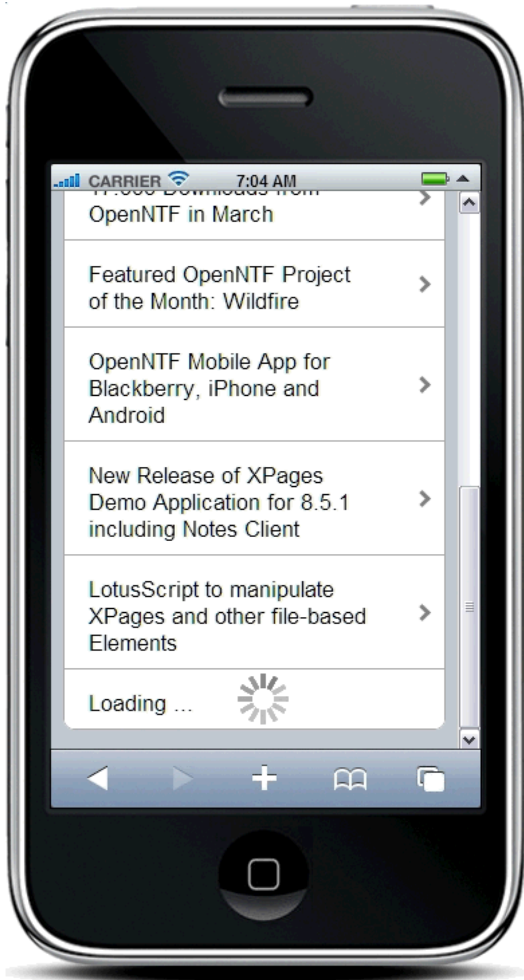
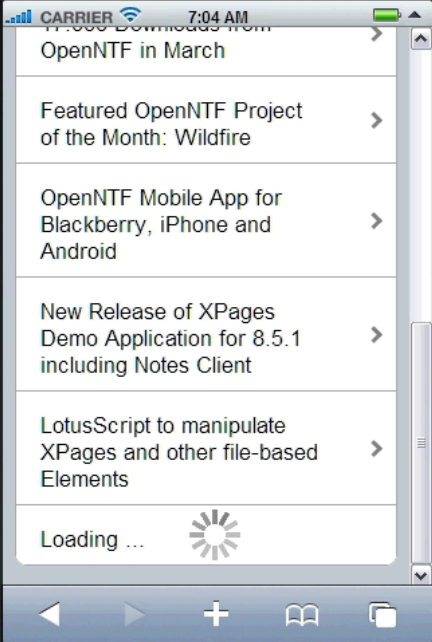
4 - Run the Sample

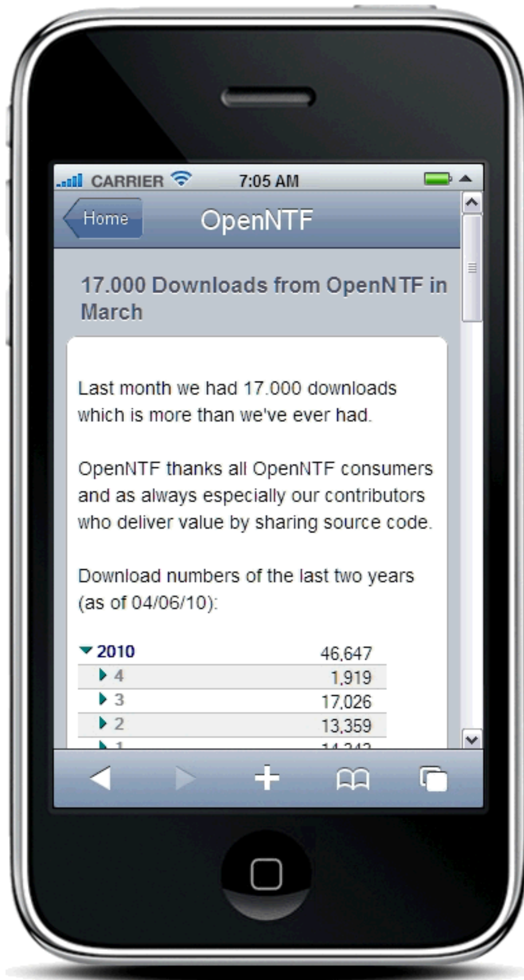
In order to run the sample simply open mJava.xsp, e.g. http://nheidloff-1/MobileDemo.nsf/mJava.xsp.

The following video describes the functionality. You can open views from different databases, can open documents, edit and save them etc.
TBD: add video

Here are some screenshots:


CARRIER
7:04 AM
OpenNTF
Six Releases Cleared for Apache Catalog
OpenNTF Steering Committee Meeting Minutes - April 06, 2010
OpenNTF Steering Committee Meeting Minutes - March 23, 2010
A major IP and Strategy Update: OpenNTF.org Efforts for 2010
OpenNTF Mobile Blog App



CARRIER
7:04 AM
OpenNTF in March
Featured OpenNTF Project of the Month: Wildfire
OpenNTF Mobile App for Blackberry, iPhone and Android
New Release of XPages Demo Application for 8.5.1 including Notes Client
LotusScript to manipulate XPages and other file-based Elements
Loading ...


CARRIER
7:05 AM
Home
OpenNTF
17.000 Downloads from OpenNTF in March
Last month we had 17.000 downloads which is more than we've ever had.
OpenNTF thanks all OpenNTF consumers and as always especially our contributors who deliver value by sharing source code.
Download numbers of the last two years (as of 04/06/10):
2010 46,647
4 1,919
3 17,026
2 13,359

CARRIER 7:05 AM

the catalog can have the same download number.

Niklas Heidloff
04/07/2010 08:32:03 AM CDT

Comments

I sure your videos are partly responsible for the increase.

Wayne Sobers
04/07/2010 08:32:03 AM CDT

Post a new comment

Your name:

CARRIER
7:05 AM
Post a new comment
Your name:
Your comment:
Save

Installation and Usage of Resources

The mobile controls require Dojo 1.5 or more precisely dojox/mobile from Dojo 1.5. Since IBM Lotus Domino 8.5.2 doesn't come with Dojo 1.5 some extra configuration needs to be done. There are two options available how to use dojox/mobile from Dojo 1.5. You can either install Dojo globally on your Domino 8.5.2 server or you can import it into your NSF.

Additionally there are themes and styles whose usage is described below.

1 - Install Dojo 1.5 on the Domino server

In general you can install Dojo 1.5 on Domino 8.5.2 and use it. However some issues might occur since testing hasn't been done yet.

1. Download Dojo 1.5: http://download.dojotoolkit.org/release-1.5.0/dojo-release-1.5.0.zip (IBMers: Use internal URL)

2. Go to %DOMINOROOT%\data\domino\js

3. Create a new directory named dojo-1.5.0

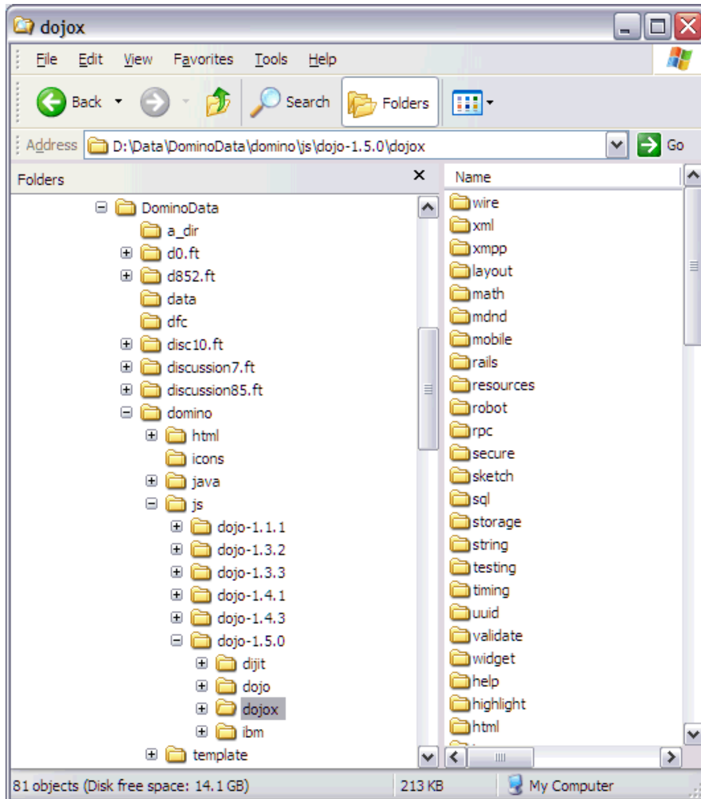
4. Put all of the Dojo 1.5 folders into this directory. It should now have 3 folders: dijit, dojo, dojox

5. Go back up to the js directory and go into the directory for the most recent Dojo build (likely 1.4.3)

6. Copy the ibm directory into your new dojo-1.5.0 directory

7. Go into the ibm directory in dojo-1.5.0. In this directory and all subdirectories, only where there are files with the extension .js.uncompressed.js, you must remove all files with an extension of .gz and .js (not .js.uncompressed.js). After removing the .js and .gz files, all of the leftover .js.uncompressed.js files need to be changed to just .js.

This is what you should see after the installation:



To tell the server to use the newest version of dojo, you need to edit xsp.properties in %DOMINOROOT%\data\properties. The file may be named xsp.properties.sample. Open this file and uncomment the xsp.client.script.dojo.version line and set it equal to 1.5.0. Save the file and rename it to xsp.properties if it does have sample on the end of the filename. Restart the server and you should be good to go.

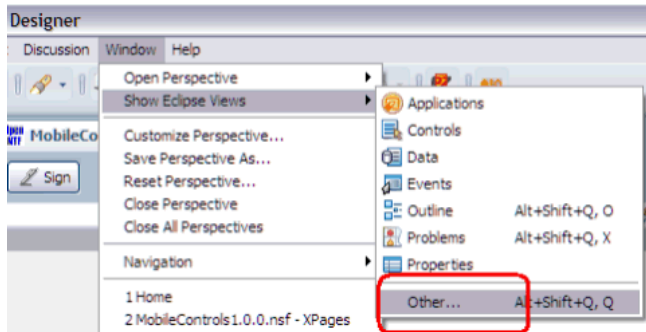
```
# ########################################
# CLIENT SIDE JAVASCRIPT
# ########################################

# The version of the Dojo Toolkit to use.
# By default the Dojo version is detected by examining the folder
# Data/domino/js/ for subfolders with names like dojo-<version>,
# and using the latest version available.
# Change this setting if you are installing different versions of Dojo
# in that folder and you need XPages to use a specific version.
# Note, using XPages with a Dojo version other than the default is unsupported;
# if you do so you will need to test for compatibility problems.
xsp.client.script.dojo.version=1.5.0

# Add parameters to the djConfig attribute of Dojo.
# Useful to switch Dojo to debug, using for example:
#   xsp.client.script.dojo.djConfig=isDebug:true
#xsp.client.script.dojo.djConfig=
```

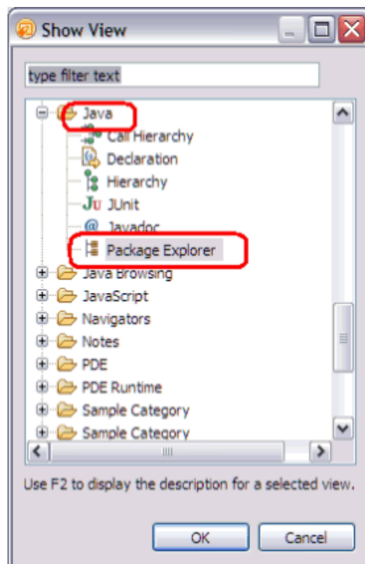
2 - Install Dojo 1.5 in your NSF

If for whatever reason you don't want to install Dojo 1.5 on your Domino 8.5.2 server you can also import the dojox/mobile part of Dojo 1.5 into your NSF. The mobile controls try automatically to use Dojo 1.5 on the Domino server but fall back to the current NSF if Dojo 1.5 cannot be found on Domino.

1. Download Dojo 1.5: http://download.dojotoolkit.org/release-1.5.0/dojo-release-1.5.0.zip (IBMers: Use internal URL)

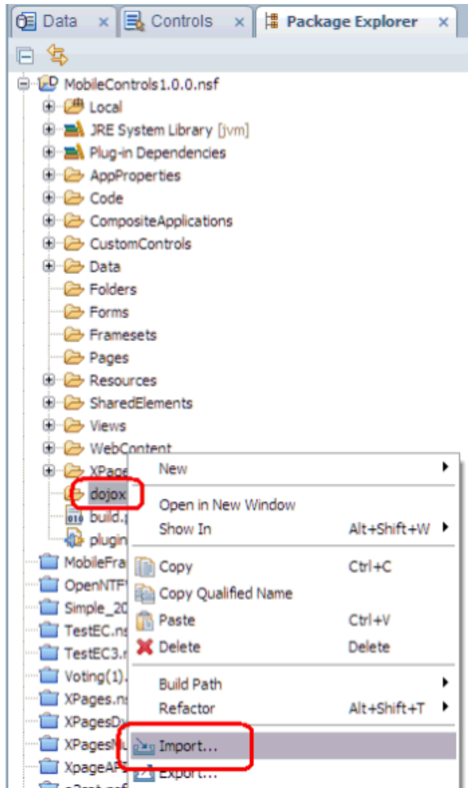
2. Open Designer and then the package explorer:

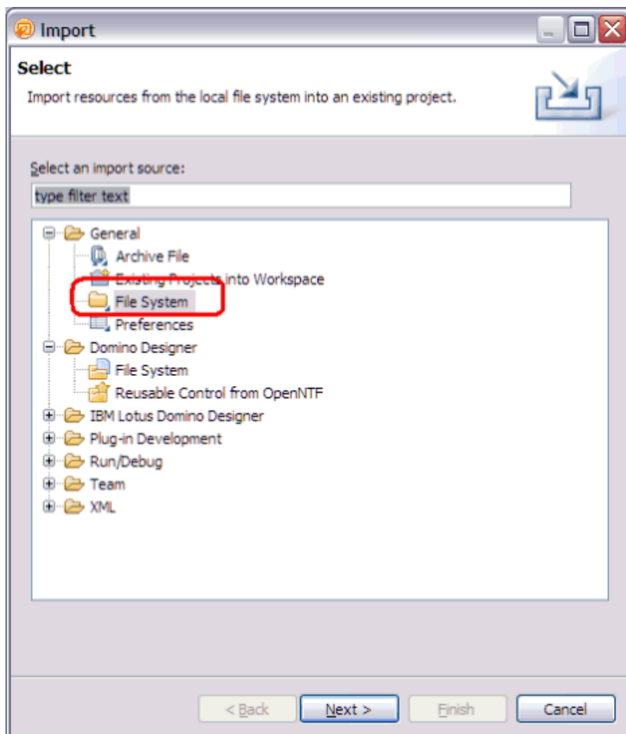




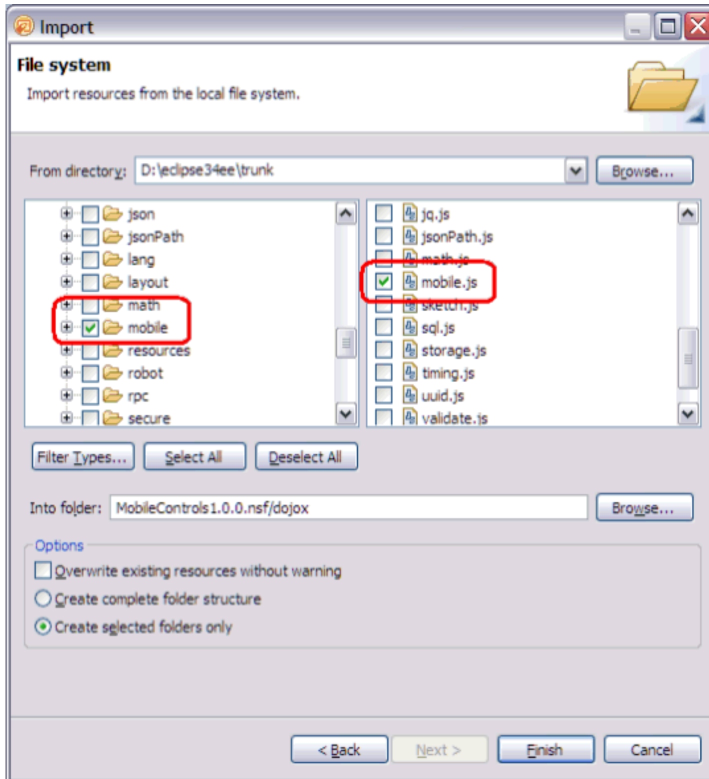
3. Open the project in the package explorer and create the folder 'dojox'

4. Right click dojox and choose import:

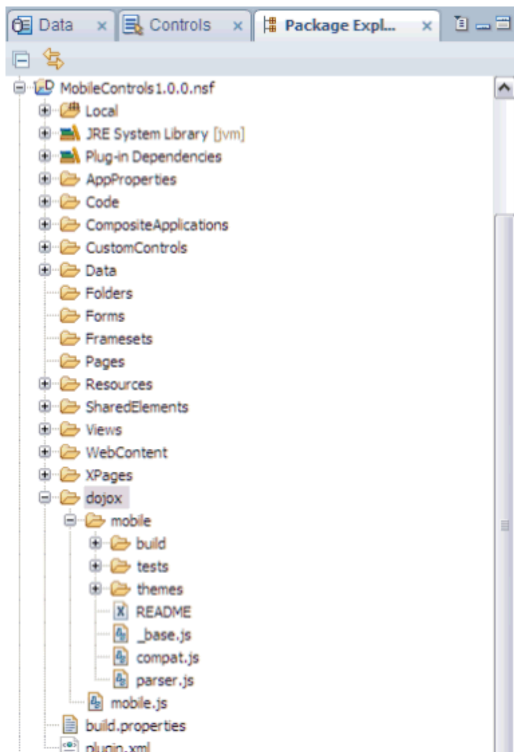




5. Choose the mobile directory and the file mobile.js:



You should see this now:



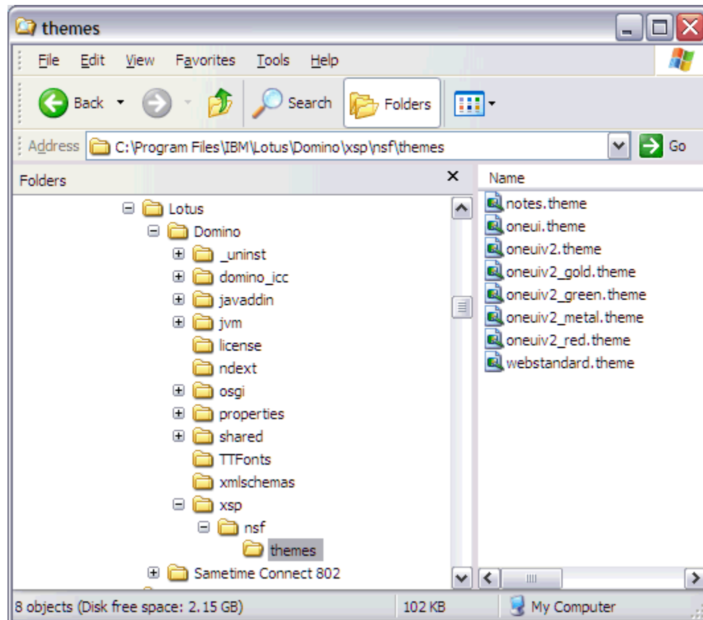
3 - Usage of Themes and Styles

Dojo 1.5 comes with a style for typical iPhone apps. This style is currently the default for mobile apps but it can be changed using themes.

There are three themes that you can find in the extension library at com.ibm.xsp.extlib.mobile\resources\mobile\themes (iphone.theme, android.theme and blackberry.theme). At this point there are all identical and all only refer to the iPhone css that is part of Dojo 1.5. You can however create your own themes and your own css.

As the Dojo files these themes can also be accessed from the Domino server or from a NSF.

If you don't want to put these themes in all your NSFs you can copy them on your Domino server in the directory Lotus/Domino/xsp/nsf/themes:

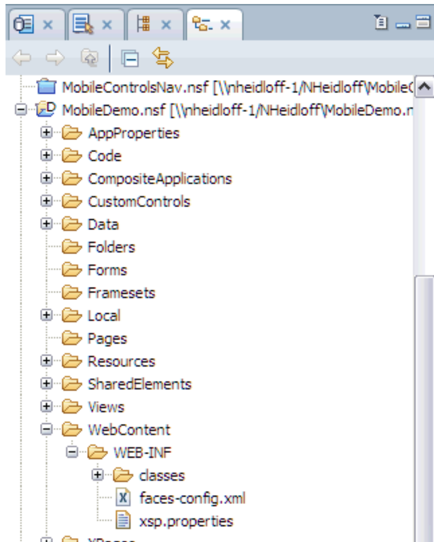


Alternatively you can copy the theme(s) you want to your NSF as this is done in the 'MobileDemo.nsf'. As with themes in general you can create your own themes, refer to your own stylesheets, etc.

By default the iphone theme is used when accessing the XPages page via an user agent 'iPhone', 'iPad', 'Android' and 'Blackberry'. You can overwrite this behavior and even use different themes for different user agents. To do this you can set the following properties in the xsp.properties file:

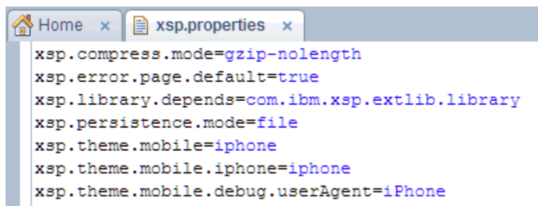
```
xsp.theme.mobile=iphone
xsp.theme.mobile.iphone=iphone
xsp.theme.mobile.android=iphone
xsp.theme.mobile.blackberry=iphone
```

The xsp.properties file can be opened from the package explorer:



For testing purposes it is often desired to overwrite the user agent. This allows for example opening the mobile app in Firefox so that Firebug can be used for debugging purposes. So another property can be set in the same xsp.properties file in the NSF to overwrite the user agent:

xsp.theme.mobile.debug.userAgent=iPhone




```
xsp.compress.mode=gzip-nolength
xsp.error.page.default=true
xsp.library.depends=com.ibm.xsp.extlib.library
xsp.persistence.mode=file
xsp.theme.mobile=iphone
xsp.theme.mobile.iphone=iphone
xsp.theme.mobile.debug.userAgent=iPhone
```

Usage of Mobile Controls

The extension library comes with the following controls which show up in the Domino Designer palette.

tbd: overview of controls