BuildXPages

Cameron Gregor

Table of Contents

Introduction

BuildXPages is a collection of tools which can aid in the Building and Deployment of XPages NSFs and XPages plugins.

If you would like to automate some or all of your XPages Deployment then there might be something in here for you!

What Can I use it for?

Some ideas of some tasks which this project can help you achieve

- Refreshing Database Designs
- Copying / Deleting Databases
- Configuring Database Template Inheritance
- Building NSFs from source code in an On-Disk Project
- Building Plugins and Features
- Deploying Plugins and Features to Servers
- Deploying Plugins and Features to Notes/Designer
- Starting / Stopping / Restarting Http Servers

What does the project provide?

The Project contains a library of tasks that can be organised and executed by Apache ANT. Apache ANT is a build tool which helps you write instructions on how to automate tasks.

If you haven't used Ant before, don't be scared! It is pretty simple once you get the hang of it. Yes there are other options, but Ant is well documented which makes it easier to understand. I have also provided some tutorials which should give you a feel for how it all works.

Ant has access to many commonly used steps such as manipluation of files in the filesystem (copy, move, delete, unzip etc.) and more complex tasks such as compiling java, running programs.

This project also includes a Plugin for Domino Designer which is used by the ANT task for building an NSF. The Plugin is basically an extension of the existing 'Headless Designer' system that was provided by IBM. The IBM Headless Designer system was a great start but could have used a little bit more love. There were a few annoyances, it was hard to tell when the build was finished, it was hard to tell that the build even started properly. There were some minor annoyances regarding project names and locations. The mechanism also involved writing a text 'command file', which was fed in when starting designer. This means each build would always have to starting designer, build and then shut it down. If designer did not shut down, then you couldn't start it up again and weird things like that.

So I decided to extend it and turn the functionality into a 'headless server'. This allow designer to remain open the whole time and listen for instructions from my ant tasks. If I need designer to restart and can tell it to shut down, and then start designer again and the 'headless server' starts again to wait for more instructions.

Getting Started

In order to use the BuildXPages there is a little bit of setup to be done. This section will hopefully guide you through the process. Please let me know via an Issue on the Github Project page if anything was left out.

Install IBM Notes/Designer

I won't tell you how to do that because you should know by now :)

Install a JDK (Java Development Kit)

I recommend installing a 32-bit JDK, because IBM Notes is 32-bit

NOTE You may actually be able to use the JDK in IBM Notes!

Install Ant

Installing Ant is a matter of downloading and extracting ant

If < FP8 use Ant 1.9

Set JAVA_HOME to Notes JRE

If you see 'Unable to locate tools.jar'ant

DORY has NOTES_PROGDIR on the path but listed as E:\Program Files etc did we try using Notes?

Install the BuildXPages Ant Library

In order for Ant to be able to use the BuildXPages Library, the **BuildXPages.jar** and **jna-4.1.0.jar** libraries must be made available to it.

You can make the library available to every ant project, or if you prefer you can make the library available just each individual project that needs it.

Installing System-wide

When ANT runs, there are 2 locations in which it will check for additional libraries. You should put BuildXPages.jar and JNA

${user.home}/.ant/lib

Libraries installed here will be available for the user who's home directory it is. For example my user directory on windows is **C:\Users\Cameron\.ant\lib** The Benefit of installing libraries here is that they will be available for both Ant in eclipse, and ant on the command line.

ANT_HOME/lib

Libraries installed here will be available for all users. For example on windows, if you have installed ant to **C:\ant**, then any libraries present in the **C:\ant\lib** directory will be available to be used.

TIP If you are trying to create the **.ant** directory in your user folder using Windows Explorer, it may not let you. So you need to use the command line, navigate to your user directory, and then issue the command `mkdir .ant`

Installing for a Single project

If you don’t want to install to the global library locations, you can just make it available for your single project.

Create a directory in your project called lib Put the BuildXPages.jar in that location

when running ant, you will need to specify the library location with -lib command

```
<taskdef
    uri="antlib:com.gregorbyte.buildxpages.ant"
    resource="com/gregorbyte/buildxpages/ant/antlib.xml"
    classpath="lib/BuildXPagesAntLib.jar"></taskdef>
```

Install Eclipse (Plugin Building only)

TIP You only need to install Eclipse if you are **Building plugins**. It is not required for Building/Deploying NSFs or Deploying Plugins

Go to eclipse.org and download 'Eclipse for RCP and RAP Developers' Follow the instructions!

Environment Variables

NOTES_PROGDIR NOTES_DATADIR DOMINO_PROGDIR DOMINO_DATADIR ECLIPSE_BASE

IMPORTANT On windows, if your Notes/Domino is within a directory with spaces e.g. 'Program Files (x86)' it may be a good idea to use the 8.3 format of that directory e.g. 'Progra~2'

TIP Once a console prompt has opened up, it has loaded it’s environment variables and will not receive any new or modified variables. So if you have added or changed environment variables, you will need to start a new console prompt to see the effect.

Install the Headless Designer Plugin to Notes/Designer

Removing a Password from Notes ID

In order for the Notes C API tasks to run smoothly, the ID file on the computer needs to have no password.

It goes without saying that this is a security risk but you can decide for yourself if you feel you can secure the computer so that nobody can access it.

Alternatively, you can ensure that IBM Notes is running, in which case the automated tasks will *hopefully* not be prompted for a password.

To Remove the Password

The ID file that you have been issued by an administrated must be created with settings that allow the password to be removed.

- Open IBM Notes, and make sure you are logged in with the user.id
- File → Security → User Security
- Click Change Password
- If you don’t see a 'No Password' button then you cannot remove the password because of policy or something

If you can’t remove the password, then make sure you will always keep IBM Notes running and logged in, and that you tick the **Don’t Prompt for a password from other Notes-based programs (reduces security)**

[NotesDontPromptPasswordOthers] | *images\NotesDontPromptPasswordOthers.JPG*

Build4XPages Designer plugin

This project provides a plugin that is installed to Domino Designer. The plugin runs a 'server' which waits for instructions that are issued by teh build tasks. For example an instruction to build an NSF, a request to shutdown the designer

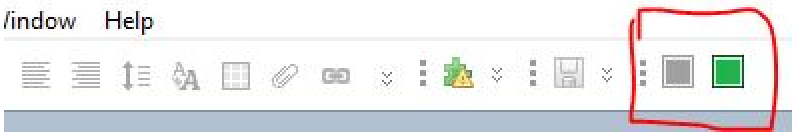
Installation

Obtain the latest version of the BuildXPages project from the projects releases page and install the **com.gregorbyte.designer.headless.updatesite** update site to Domino Designer using the instructions in How to Install Plugins to Domino Designer

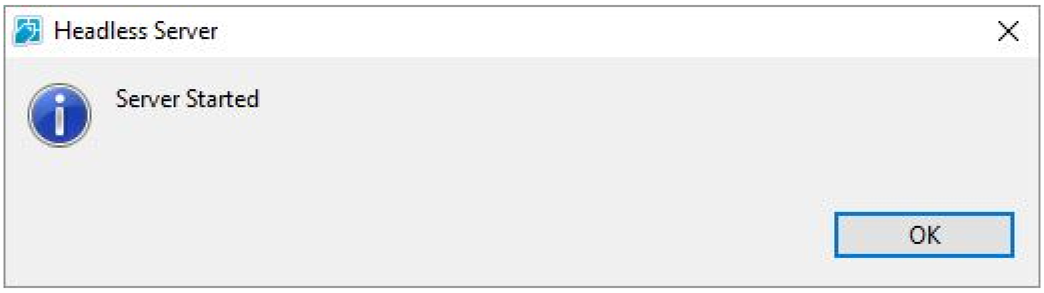
Starting and Stopping the Headless Server

By Default, the headless server does auto-started when you start Domino Designer (you can change this, see Preferences), so you must know how to start and stop it!

You can find the Headless toolbar, and you will see one button grayed out, and one coloured button. The green button is the 'start' button, so if you can see it, that means the server is not running.



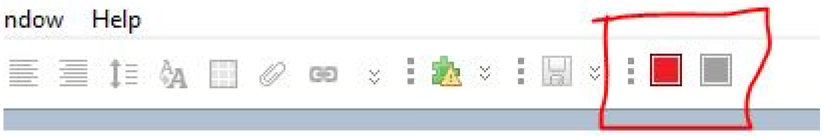
If you click the green button it will start the server, and you will see the message


Headless Server

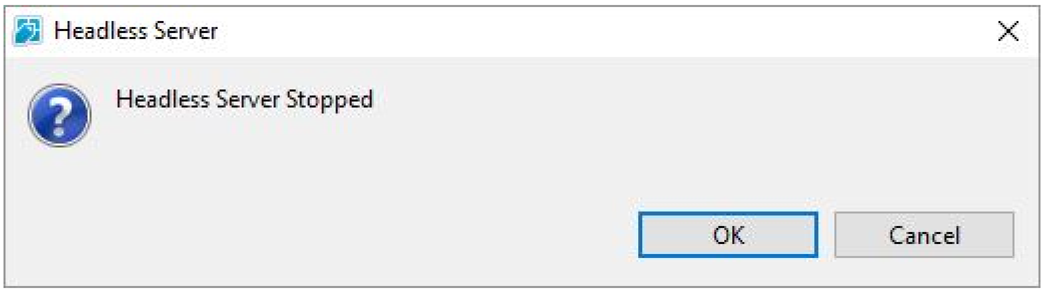
Server Started

OK

You should then be able to see the red button



and if you click the red button you will stop the headless server and see the message


Headless Server

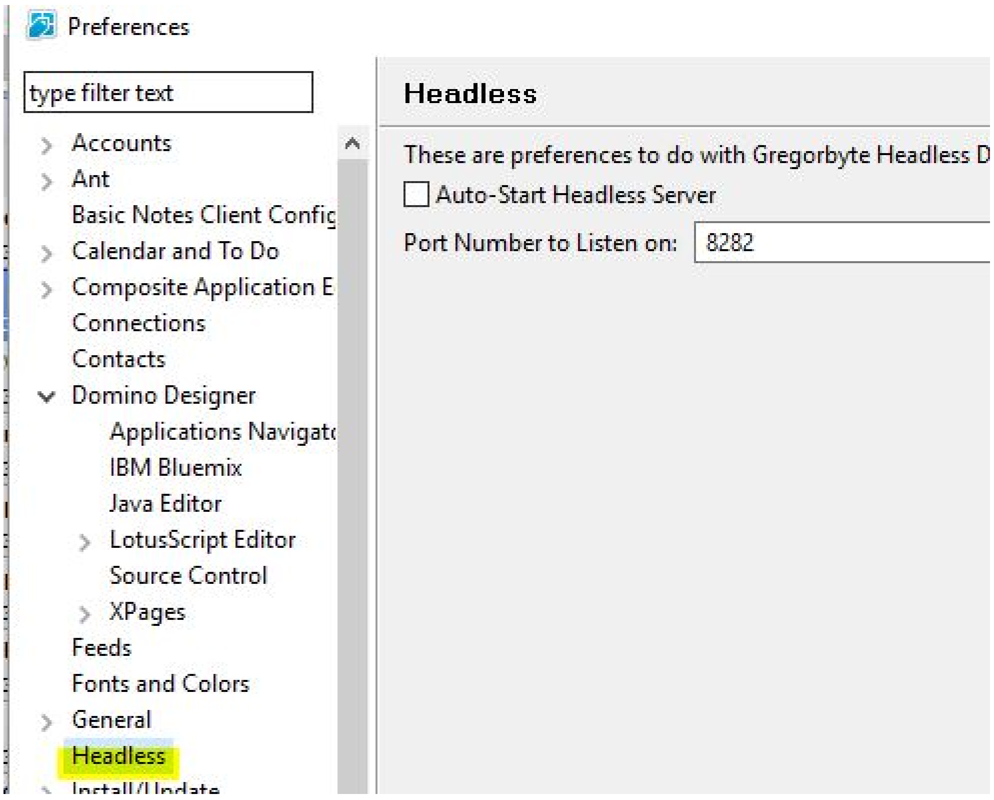
Headless Server Stopped

OK Cancel

Preferences

To access the preferences for Headless Server, goto File → Preferences → Headless

You can set the Headless Server to auto-start when Domino Designer starts, and also change the Port Number that it will be listening on.



Preferences

type filter text

Headless

These are preferences to do with Gregorbyte Headless D

☐ Auto-Start Headless Server

Port Number to Listen on: 8282

Tutorials

These Tutorials are designed to demonstrate the different tasks that are available. It is intended that these tutorials can be used to verify that everything is working correctly, and you can then take the build files for these tutorials and customise them for your own projects.

Introduction to Apache Ant

IMPORTANT Before doing this tutorial make sure you have performed necessary setup detailed in the **gettingstarted** page

This tutorial is a very basic introduction to the concepts of Ant. If you already have used ant before then you can skip this tutorial.

We are going to cover the following concepts

- The Build File (build.xml)
- Targets
- Tasks
- Properties
- Task Dependencies

Diving Straight In

Ant is a command line program, which you run by using the command `ant` and then supplying some command line arguments.

Let's run our first ant build, our sample build will just print a hello message to the console.

1. Open your command line
2. Change directory to the the **demo/antintro** folder
3. issue the command `ant`

You should see the following output

```
Buildfile: V:\Projects\BuildXPages\demo\antintro\build.xml
hello:
     [echo] Hello Alice!
BUILD SUCCESSFUL
Total time: 289 milliseconds
```

So we told Ant to run, but how did Ant know what we wanted to do?

The Build File (build.xml)

The build file is an xml file that contains all the instructions of what you want to automate.

By default Ant will look for a build file named **build.xml**. You are able to call it something else if you want, but you will need to specify it as a command line argument.

Lets have a look at our build file for this project

```
<project name="Introduction to Ant" default="hello">

    <description>This is a simple sample to demonstrate some concepts of
ant</description>

    <!-- tag::properties[] -->
    <property name="firstname" value="Alice"></property>
    <!-- end::properties[] -->

    <!-- tag::targets[] -->
    <target name="hello" description="Says Hello">
        <echo message="Hello ${firstname}!"></echo>
    </target>

    <target name="goodbye" depends="hello" description="Says Goodbye">
        <echo message="Goodbye ${firstname}!"></echo>
    </target>
    <!-- end::targets[] -->

</project>
```

We can see the root element of the build file is 'project'. We have a name for this project and a default target 'hello'.

There is a longer description of the build project, then some property definitions, and finally some targets each with one task within it.

Targets

Targets are used to create a sequence of tasks that you want to perform, and refer to that sequence by a name.

In this build file we have 2 targets **hello** and **goodbye**.

```
    <target name="hello" description="Says Hello">
        <echo message="Hello ${firstname}!"></echo>
    </target>

    <target name="goodbye" depends="hello" description="Says Goodbye">
        <echo message="Goodbye ${firstname}!"></echo>
    </target>
```

You can see in the root project element, **hello** is specified as the default target, and therefore this is the target that was executed when we told ant to run

Tasks

Tasks are the building blocks of what you can automate. In this sample we are keeping it very simple and only using the 'echo' task to print messages to the console. But there are a whole bunch of default tasks available.

The Apache Ant project site is very well documented and I recommend having a look through the Overview of Apache Ant Tasks

Custom Tasks

If there is no task that suits your need, you can make custom tasks yourselves! This is precisely what I have done for BuildXPages. See the other tutorials for demonstration of the custom tasks made available by the BuildXPages project

Properties

Properties are useful to allow you to use the same build file, but customise it for different projects and environments.

Properties can be provided using a properties file, and they can also be provided directly in the build file.

In this build file, our **echo** tasks use a property called **firstname**

This **firstname** property is specified at the top of the build file.

```
    <property name="firstname" value="Alice"></property>
```

Once a property is set, it cannot be changed, so the property value will remain whatever it was first set as.

Loading from a Properties file

It is often easier to keep custom property values in a properties file, and supply this as an argument when running Ant.

In our tutorial directory we have a demo properties file called **build.properties** which specifies the firstname to be bob

```
# Set firstname to bob
firstname=Bob
```

We can tell ant to use this property file using the `-propertyfile` option

```
ant -propertyfile build.properties
```

```
Buildfile: V:\Projects\BuildXPages\demo\antintro\build.xml

hello:
     [echo] Hello Bob!

BUILD SUCCESSFUL
Total time: 0 seconds
```

We now see a message to Bob because the firstname property is being set from the **build.properties** file *before* it is attempted to be set as **Alice** within the **build.xml** file.

Specifying a Property as Command line Argument

You can also provide a property directly on the command line when running ant. The argument takes the form `-D<propertyname>=<value>`

Try it now by putting your own name in. Because this property is set before the **build.properties** file is loaded

```
ant -Dfirstname=Cam
```

```
Buildfile: V:\Projects\BuildXPages\demo\antintro\build.xml

hello:
     [echo] Hello Cam!

BUILD SUCCESSFUL
Total time: 0 seconds
```

Task Dependencies

A task can be set to depend on another task, in our example, the **goodbye** target is set to depend on the **hello** target (using the *depends* attribute). This means before we say **goodbye** we must say **hello**!.

```
    <target name="hello" description="Says Hello">
        <echo message="Hello ${firstname}!"></echo>
    </target>

    <target name="goodbye" depends="hello" description="Says Goodbye">
        <echo message="Goodbye ${firstname}!"></echo>
    </target>
```

Let’s try it out. This time because we are not running the default target we must specify it. Issue the following command

```
ant goodbye
```

```
Buildfile: V:\Projects\BuildXPages\demo\antintro\build.xml

hello:
     [echo] Hello Alice!

goodbye:
     [echo] Goodbye Alice!

BUILD SUCCESSFUL
Total time: 0 seconds
```

Conclusion

So this was a very short, very simple introduction to the concepts of Ant. We learned about the concepts of The Build File, Targets, Tasks, Properties and Task Dependencies.

These concepts will hopefully give you a good start to comprehend the rest of the tutorials in BuildXPages, in which we will use much more complicated targets and tasks to achieve some tasks for building and deploying XPages applications.

Building an NSF

IMPORTANT Before doing this tutorial make sure you have performed necessary setup detailed in the **gettingstarted** page

In this tutorial we will build a Simple XPages Application from an on On-Disk Project, using Domino Designer Headless Plugin + BuildXPages Ant Tasks.

After the successful build, the nsf will be available in your local workspace.

The files for this Demonstration are in the **demo/buildnsf** folder of the Build4XPages distribution files. The sample On-Disk Project is located within the tutorial folder as **sampleodp**. The Application has a Custom Control and an XPage which uses that custom control.

build.xml

Our 'recipe' for the build is contained in the **build.xml** file located in the **demo/buildnsf** directory. **build.xml** is the default name for an Ant build file, you can call it another filename but if so you would need to specify extra command line argument `-b myfile.xml`

Within the build.xml file are things called *targets*. A target is a bit like a code block. Targets can be configured to depend on another target, so that means executing a *target* will cause the other targets that it depends on to be executed first.

Within this build.xml file are 2 *targets* that are used to setup the demo **clean** and **init**

clean is used to clean up any previous attempt at this tutorial, and **init** is used to initialise a new attempt.

You can see that **init** has been set to depend on **clean** so that before the **init** target is executed, the **clean** target will be executed.

```
    <target name="clean">
        <delete dir="odp"/>
        <delete dir="odperror"/>
    </target>

    <target name="init" depends="clean">

        <copy todir="odp">
            <fileset dir="sampleodp"></fileset>
        </copy>

        <copy todir="odperror">
            <fileset dir="sampleodp"></fileset>
        </copy>

        <!-- Delete the Custom Control from the odp that we are using for the error
demo -->
        <delete>
            <fileset dir="odperror/CustomControls"></fileset>
        </delete>

    </target>
```

Run a Successful Build

Within this **build.xml** file is a target called **build**. This will build the NSF that should not have any problems. This target is listed as the default target in the root element of the build.xml file (not shown here).

This target makes use of the **bxp:buildnsf** task which will connect to the Headless Designer Plugin, and instruct it to build the nsf as specified by the task attributes.

```
    <target name="build" depends="init">

        <bxp:checkDesignerRunning/>

        <!-- We assign the .project file path to a property using the 'location'
attribute,
        this way the property will be expanded to the absolute path of the .project
file-->
        <property name="odp.projectfile" location="odp/.project" />

        <bxp:buildnsf project="${odp.projectfile}" projectname="${odp.projectname}"
nsf="${nsf.filepath}" />

    </target>
```

You can see that this target depends on **init**. **init** depends on clean, so when we execute this **build** target what will really happen is the following order of execution:

```
clean -> init -> build
```

To run the Build

1. Open your command line
2. Change directory to the the **demo/buildnsf** folder
3. issue the command `ant`

Ant knows to look for **build.xml** and the default target (which is our **build** target). The build process will then begin, and your nsf will (hopefully) be built into your local workspace.

You should see some output on the console like so:

```
Buildfile: V:\Projects\BuildXPages\demo\buildnsf\build.xml

clean:
   [delete] Deleting directory V:\Projects\BuildXPages\demo\buildnsf\odp
   [delete] Deleting directory V:\Projects\BuildXPages\demo\buildnsf\odperror

init:
     [copy] Copying 20 files to V:\Projects\BuildXPages\demo\buildnsf\odp
     [copy] Copying 20 files to V:\Projects\BuildXPages\demo\buildnsf\odperror

build:
[bxp:buildnsf] Attempt to Create Socket
[bxp:buildnsf] Socket Created
[bxp:buildnsf] CONNECTED TO DESIGNER! What can we do for you?
[bxp:buildnsf] Issuing Refresh Import Build Command for
V:\Projects\BuildXPages\demo\buildnsf\odp\.project
[bxp:buildnsf] BUILD JOB ABOUT TO RUN
[bxp:buildnsf] BUILD JOB RUNNING
[bxp:buildnsf] BUILD JOB STATUS: SUCCESS
[bxp:buildnsf] PROBLEMS STATUS: SUCCESS
[bxp:buildnsf] No ProblemMarkers found after Building

BUILD SUCCESSFUL
Total time: 2 seconds
```

Also, in your domino Designer workspace you should see a new NSF with filepath temp\BuildXPagesDemo.nsf

Run a Build with Errors

Let’s see an example of running an NSF Build that results in some errors. To do this we will use the same sample On-Disk Project, but before we run the build we will sneakily delete the custom controls. This should cause an error when Designer tries to build the XPages that use those missing custom controls.

The *target* that I have written to do this is called **buildfail**.

Note it is pretty much the same as **build** but is pointing to a different location for the on-disk project, and specifies different NSF and projectname.

```
    <target name="buildfail" depends="init">

        <bxp:checkDesignerRunning/>

        <!-- We assign the .project file path to a property using the 'location'
attribute,
        this way the property will be expanded to the absolute path of the .project
file-->
        <property name="odp.project" location="odperror/.project" />

        <bxp:buildnsf project="${odp.project}" projectname="BuildXPagesDemoNSFError"
nsf="temp\BuildXPagesDemoError.nsf" />


    </target>
```

Since it is not the default target, we will tell ant that that is the target we want to build by including the target name in the command line arguments.

1. issue the command `ant buildfail'

You should then see an output like so:

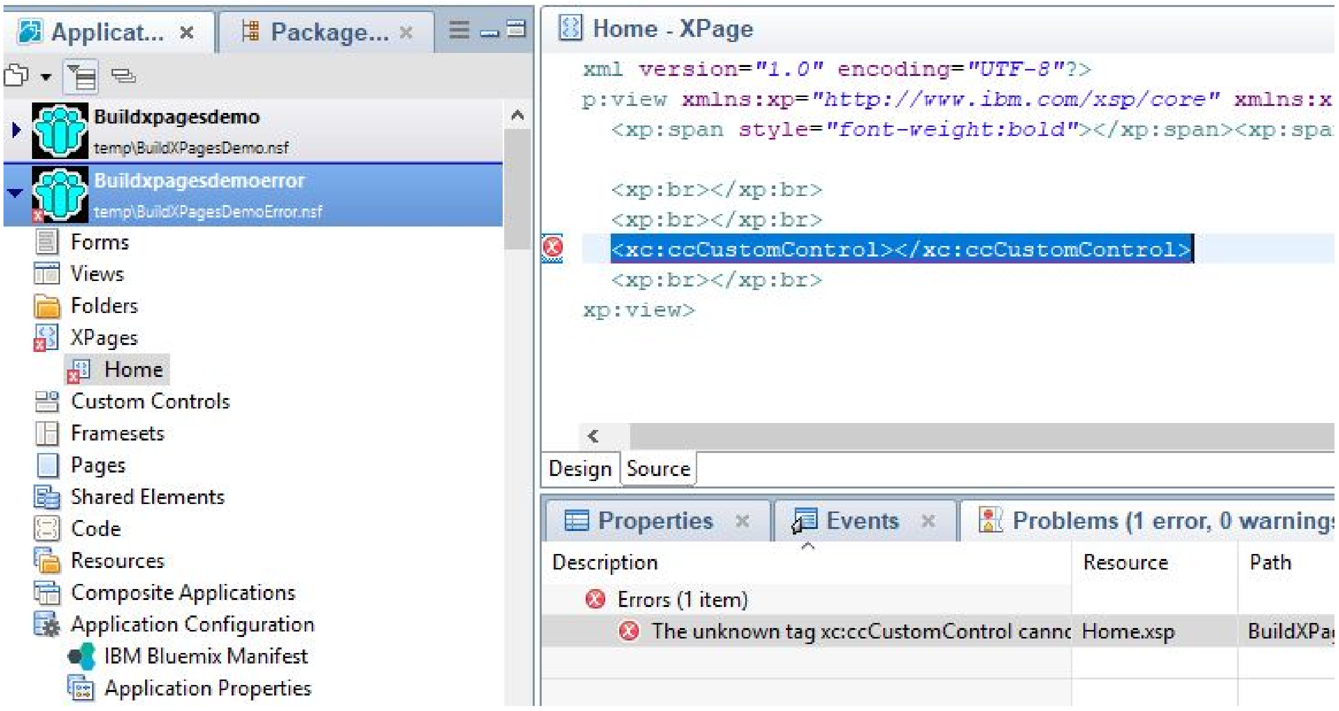
```
Buildfile: V:\Projects\BuildXPages\demo\buildnsf\build.xml

clean:
   [delete] Deleting directory V:\Projects\BuildXPages\demo\buildnsf\odp
   [delete] Deleting directory V:\Projects\BuildXPages\demo\buildnsf\odperror

init:
     [copy] Copying 20 files to V:\Projects\BuildXPages\demo\buildnsf\odp
     [copy] Copying 20 files to V:\Projects\BuildXPages\demo\buildnsf\odperror

buildfail:
[bxp:buildnsf] Attempt to Create Socket
[bxp:buildnsf] Socket Created
[bxp:buildnsf] CONNECTED TO DESIGNER! What can we do for you?
[bxp:buildnsf] Issuing Refresh Import Build Command for
V:\Projects\BuildXPages\demo\buildnsf\odperror\.project
[bxp:buildnsf] BUILD JOB ABOUT TO RUN
[bxp:buildnsf] BUILD JOB RUNNING
[bxp:buildnsf] BUILD JOB STATUS: SUCCESS
[bxp:buildnsf] PROBLEMS STATUS: FAIL
[bxp:buildnsf] The following ProblemMarkers were present after building
[bxp:buildnsf] Error: Home.xsp: The unknown tag xc:ccCustomControl cannot be used as a
control.
```

Notice that the 'PROBLEMS STATUS' is fail and the last line is telling us the exact problem. If you were to look in Designer you would see the problem there too.



```
xml version="1.0" encoding="UTF-8"?>
p:view xmlns:xp="http://www.ibm.com/xsp/core" xmlns:x
  <xp:span style="font-weight:bold"></xp:span><xp:spa

  <xp:br></xp:br>
  <xp:br></xp:br>
  <xc:ccCustomControl></xc:ccCustomControl>
  <xp:br></xp:br>
xp:view>
```

Description	Resource	Path
Errors (1 item)		
The unknown tag xc:ccCustomControl cannc	Home.xsp	BuildXPa

Deploying an NSF

Building an NSF is one thing, but it is no good if you don’t get it into the live system! Let’s have a look at how we can acheive that.

There is more than one way to deploy an NSF, not everyone is going to do it the same way. This tutorial will hopefully demonstrate some of the tasks you could use to acheive your desired deployment.

Scenario 1 - Super Simple

After building an NSF, you simply want to make it available to someone who will perform manual deployment.

Automated deployment is not for everyone and maybe manual deployement

Scenario 2 - Refresh an Existing Template

This is my preferred option.

- I have 'local' nsfs located on the headless build machine
- I have a Production Template
- I have one or more Production Applications

I set the 'master template' name of the built nsf, and the inherit of the template I refresh the production template I either allow the design task to run overnight, or if doing an immediate deployment, I issue the load design task console command

Building Features and Plugins

In this sample we build the plugins for a feature, and prepare an Update Site that can be used to deploy to Domino or Domino Designer

First Time Setup

Before we build plugins, we need to make sure we are set up properly

- Install Eclipse for RCP and RAP Development
- Install Ant
- Install Build4XPages Ant tasks

Configure Your Notes and Domino Program Directories

In order to run the build steps, we need to know where Notes and Domino are

- Set them up as Environment Variables
- Specify them in the build.properties file

Configure Eclipse Properties

The build script needs to know 3 things in order to build your plugins

What is the base directory of Eclipse?

This may be something like `C:\eclipse`

Where can I find the PDE Build.xml file?

This is a file which provides the actual recipe to build plugins. In older versions of eclipse it will be under <eclipse>/plugins/org.eclipse.pde.build_<version>/scripts/build.xml In Newer versions of eclipse it will be located under your user home directory's p2 pool of plugins <userhome>/.p2/pool/plugins/org.eclipse.pde.build_<version>/scripts/build.xml

Where can I find the equinox launcher jar?

This is an executable jar which launches the plugin build process. In older versions of eclipse it will be under <eclipse>/plugins/org.eclipse.equinox.launcher_<version>.jar In Newer versions of eclipse it will be located under your user home directory's p2 pool of plugins <userhome>/.p2/pool/plugins/org.eclipse.equinox.launcher_<version>.jar

These 3 things should be the same for every project you build on the computer, so you can set these up as Environment Varables as well

If you don't want to set them up as environment variables, you can specify them in the accompanying build.properties file

Run the Build!

1. Open your console and navigate to the demo/buildfeature folder

2. Type `ant`
3. Cross your fingers

After the build is finished you should see a zip file in the demo/buildfeature/BuildXPages directory.

Deploying Plugins

TODO

BuildXPages Ant Task Reference

The following is a list of the Ant tasks provided by the project and some documentation and examples of how to use them.

Deploying NSFs to a Server

Building and Deploying Plugins

Interacting with Domino Designer for Building NSFs

Controlling a Domino Server

NSF Deployment Related Tasks

The following tasks are usually useful after you have built NSF’s and need to move them around, refresh templates etc. to acheive deployment of an NSF that has been successfully built.

copynsf

Copies an NSF from one place to another using the Notes C API’s 'NSFDbCreateAndCopy' function.

NOTE	Currently this tasks only copies the Note class 'ALLNONDATA' which means design elements only. It could easily be modified to include all documents if you like. Let me know if you want this done.

Attributes

srcserver

The Server of the Database to be copied. Optional - defaults to Local

srcfilepath

The Filepath of the Database to be copied

dstserver

The Destination Server for the newly created database. Optional - defaults to Local

dstfilepath

The Filepath the Database to be created.

Example

```
<copynsf srcfilepath="demo\\GitTest.nsf" dstfilepath="demo\\DidItwork.nsf" />
```

deletensf

Deletes an NSF using the Notes C API’s NSFDbDelete method.

Attributes

server

The Server of the NSF that you want to Delete. Optional - defaults to Local

filename

The filepath of the NSF that you want to Delete

Example

```
<!-- Delete a Local NSF -->
<deletensf filename="demo\\DidItwork.nsf" />
```

```
<!-- Delete an NSF on another Server -->
<deletensf server="Domino02" filename="demo\\DidItwork.nsf" />
```

settemplatenames

Updates the Template Inheritance settings of an NSF. You can set an NSF To be a master template, or you can set an NSF to Inherit from another Template, or Both! You can also clear the 'inherit from' or 'master template' settings.

Attributes

server

The Server of the NSF that you are modifying template settings for

database

The filepath of the NSF that you are modifying template settinsg for

clearinheritfrom

Defaults to false. When set to 'true' will clear any inheritance settings if they exist

inheritfrom

The Name of the Master Template that you would like the NSF To inherit from

clearmastername

Defaults to false. When set to 'true' will clear the 'Is Master Template' settings of the nsf if they exist *mastername*

This is the Template Name that you would like this NSF to be known as.

Example

```
<!-- Set an NSF to inherit from 'DemoTemplate' -->
<bxp:settemplatenames database="MyFolder\MyNSF.nsf" server="Domino01"
inheritfrom="DemoTemplate">
</bxp:settemplatenames>
```

```
<!-- Set an NSF to be a Master Template called 'DemoTemplate' -->
<bxp:settemplatenames database="MyFolder\MyNSF.nsf" server="Domino01"
mastername="DemoTemplate">
</bxp:settemplatenames>
```

```
<!-- Clear the Inherit From settings so the NSF will no longer inherit from a template
-->
<bxp:settemplatenames database="MyFolder\MyNSF.nsf" server="Domino01"
clearinheritfrom="true">
</bxp:settemplatenames>
```

```
<!-- Clear the Master Template settings so the NSF will no longer be a Master template
-->
<bxp:settemplatenames database="MyFolder\MyNSF.nsf" server="Domino01"
clearmastername="true">
</bxp:settemplatenames>
```

scxd

Sets the Single Copy XPage Design settings of an NSF. You can set both the relevant properties of the SCXD path, and the Flag which determines whether to use the SCXD template or not.

Attributes

server

The server of the NSF that you would like to set SCXD settings for

database

The filepath of the nsf that you would like to set SCXD settings for

scxdpath

The path of the XPages database which you would like to use as the SCXD

scxdflag

true/false determines whether the 'Use Single Copy XPage Design' checkbox is will be ticked in the Database Properties

Example

```
<!-- Set an NSF to use 'scxd\Awesome.nsf' as it's single copy xpage design -->
<bxp:scxd database="MyFolder\MyNSF.nsf" server="Domino01" scxdpath="scxd\Awesome.nsf"
scxdflag="true" />
```

```
<!-- Set an NSF to not use any SCXD -->
<bxp:scxd database="MyFolder\MyNSF.nsf" server="Domino01" scxdpath="" scxdflag="false"
/>
```

refreshdbdesign

Refreshes an NSF’s design from a server using the Notes C API DesignRefresh method.

The console output shows all the design elements that are modified which is great to verify what has changed.

This task uses existing template settings of the NSF so if you need to change them you should use the settemplatenames task before using this task.

Attributes

server

The server of the NSF that you would like to Refresh the Design of

database

The filepath of the NSF that you would like to refresh the Design of

templateserver

The server which has the Template that will be refreshed from

Example

```
<!-- Refresh MyNSF.nsf from the Domino03 server -->
<refreshdbdesign server="Domino02" database="MyFolder\\MyNSF.nsf"
templateserver="Domino03" />
```

Plugin Related Tasks

The following tasks are all related to the building and deployment of OSGi plugins for XPages and Notes/Designer.

buildfeature

CAUTION This Task and Documentation needs a little bit of love before it is truly re-usable. If you are keen to use this then let me know and I will improve it!

This task will build all the Plugins listed in a feature, using the headless eclipse PDE build system.

Before using this task, you will need to have prepared the build directory with the source code of the plugins that are to be built, and the the feature that is to be built.

Properties that are used in this task

This task is actually a macro, and currently relies on some properties to have already been set in ant

featureId
: The feature to be built

eclipseBase
: Root directory of eclipse

pdeBuildVersion
: PDE Build version

equinoxLauncherVersion
: Equinox Launcher Version

buildId
: Usually the build number

buildLabel
: Project Name

buildConfigDir
: Directory of build.properties

buildDir
: The working directory in which the plugins will be built

Attributes

pluginPath
: This is a semi-colon delimited string of the target platform plugin paths

Example

CAUTION You really need a lot more information than this to use this task but I have just put this all here as a starting point. Please feel free to contact me to make me provide more information.

```
<!-- Plugin path is supplied by a build.properties file -->
<bxp:buildfeature pluginpath="${pluginPath}" />
```

importplugins

Imports plugins/features from an update site on the filesystem into an Open Eclipse NSF Update Site

Attributes

server

The server that the **Open Eclipse NSF Update Site** is located on

database

The filepath of the **Open Eclipse NSF Update Site**

sitexml

The location of the site.xml of the **Filesystem update site** that you want to import

deletefirst

true/false, defaults to false. If set to true, it will clear the updatesite nsf of all existing plugins/features

Example

```
<bxp:importplugins server="Domino02" database="UpdateSite\MyUpdateSite.nsf"
sitexml="C:\MyFolder\MyProject\com.my.updatesite\site.xml" />
```

copyplugintobuilddir

This task is used during preparation for a headless eclipse plugin build. It is used to copy the source code of of a plugin, into the *build directory* which is a special working directory used to build the plugins and features.

Properties used

pluginsDir

The Destination directory, usually the eclipse/plugins folder of the build Directory

Attributes

plugin

the folder name of the plugin to copy

Example

```
<bxp:copyplugintobuilddir plugin="com.acme.myplugin" />
```

unpackplugin

Usually you update Domino Designer with new plugins by going through the normal installation process. But there is another way in which you can build plugins into a particular folder, and Domino Designer always picks up new versions.

When deploying plugins this way, some plugins need to be 'unpacked' which means they don’t exist as a jar file, but the contents are unpacked into a folder.

This task will unpack a plugin to a folder. You just need to give the plugin name, and it will find the version information automatically

Attributes

dir
: The directory that the plugin is in

pluginid
: the plugin id that you want to unpack

Example

```
<bxp:unpackplugin dir="eclipse/plugins" pluginid="com.acme.superplugin" />
```

generatesitexml

After building your plugins headlessly with eclipse PDE, you will then have a directory of plugins and features, but before importing plugins into an Open Eclipse NSF Update Site you will also need a **site.xml** file.

Newer versions of eclipse use a *p2 repository* format which does not require a site.xml, but with Notes/Domino running on a much older version of eclipse, it still uses an older format which uses the **site.xml** file.

The build process does not generate one for us so we do it ourselvels with this task, which scans the features directory and builds the necessary xml.

Attributes

eclipsedir
: The root folder that contains the plugins and features directory. I call it eclipse directory because often you will find it in the structure ../something/**eclipse**/plugins and ../something/**eclipse**/features

Example

```
<bxp:generatesitexml eclipsedir="testgen" />
```

clearupdatesite

NOTE This task does not refer to clearing an NSF update site, this task is for clearing and update site on the filesystem

After building plugins, sometimes you want to put them in an update site on the filesystem. You may want to make sure the update site is empty before you put your plugins there, so this task can clear out an existing update site.

Attributes

eclipsedir

The root folder that contains the plugins and features directory. I call it eclipse directory because often you will find it in the structure ../something/**eclipse**/plugins and ../something/**eclipse**/features

Example

```
<bxp:clearupdatesite eclipsedir="myplugins" />
```

copypluginstoupdatesite

NOTE This task does not refer to copying an NSF update site, this task is for copying to an update site on the filesystem

After building plugins, sometimes you want to put them in an update site on the filesystem. This task will copy the plugins to your target update site, and also generate the site xml.

Properties

builtPluginsZip

the zip file of the built plugins after headless build

Attributes

eclipsedir

The root folder that contains the plugins and features directory. I call it eclipse directory because often you will find it in the structure ../something/**eclipse**/plugins and ../something/**eclipse**/features

Example

```
<bxp:copypluginstoupdatesite eclipsedir="my/target/updatesite/eclipse" />
```

Designer Related Tasks

startDesigner

CAUTION Using this task to start designer will cause designer to shutdown at the end of the ant build process. Due the the process being a sub-process. I need to do this a better way such as the powershell script way but I haven’t got around to it yet. I actually in real life just leave designer running all the time so I don’t need to start/close it.

Before building an NSF with the Headless Designer Plugin, you need to make sure designer is running. You can use the **checkDesignerRunning** task to check, and if it is not running you can use this **startDesigner** task.

Properties

notesProgDir
: The Notes program directory on this machine

Example

```
<bxp:startDesigner />
```

buildnsf

The buildnsf task is used to build an ODP into an NSF. To do this it connects to the Designer Headless Server, and instructs it to build. If the NSF that is specified does not yet exist, it will be created. You

Attributes

project
: This is the .project file that is in the root directory of the On-Disk Project

projectname
: When the on disk project is imported it must be given a 'name'. This name needs to be unique, so you can specify it here. Note: Do not put your ODP under the Notes Workspace.

server
: This is the server of the NSF that you are building

nsf

This is the filepath of the nsf that your building with

port

This specifies the port that your Designer Headless Server is running on (Default 8282)

failonerror

If Errors are found against the nsf after building, fail the task

Example

```
<target name="testbuildnsf" description="Build an NSF">
    <bxp:buildnsf project="D:\\workspaces\\testworkpace\\WahWah\\.project"
projectname="FromAnt" nsf="stageing\\fromant2.nsf">
    </bxp:buildnsf>
</target>
```

```
<target name="testbuildnsf2" description="Build an NSF">
    <bxp:buildnsf ondiskproject="V:\\eclipse-kepler\\runtime-
DominoDesigner\\DoraHeadless\\.project" targetfilename="fromant2.nsf" port="8283">
    </bxp:buildnsf>
</target>
```

closedesigner

This task connects to the Headless Designer Plugin and instructs it to shutdown.

Attributes

port

The port that the Headless Designer Plugin is running on. Default = 8282

Example

```
<bxp:closedesigner />
```

markersreport

This task connects to the Headless Designer Plugin, and requests a report of all the problem markse (e.g. Error, Info, Warning etc.) for a particular NSF.

If you like , you can cause the build to fail if there are errors present on the NSF, to prevent further tasks like deployment.

Attributes

ondiskproject

The On Disk Project file (.project) of the NSF that you want a report for.

failonerror

When set to true (Default) the build will fail if errors are present on the NSF

port

The port which Headless Designer Plugin is running on (default = 8282)

Example

```
<bxp:markersreport ondiskproject="C:\\workspaces\\neon-64\\runtime-
HeadlessPlugin\\DeleteMe\\.project">
</bxp:markersreport>
```

checkdesignerrunning

CAUTION Windows only!

This task checks if the nlnotes.exe process is running, if so it sets a property **designer.running**. You can then use this property to decide whether to take some other action.

Example

```
<bxp:checkDesignerRunning />
<fail if="designer.running" message="Designer was running" />
```

updateDesignerLink

NOTE I am not sure this is a great way to deploy plugins so I haven’t detailed this part yet.

This task is used to set up an update site that will always be loaded when Designer Is started up. This is only used when you are deploying plugins to Domino Designer using the sneaky 'permanent update site' method.

Properties

notesProgDir

The Notes Program Directory on this machine

Attributes

updateSiteLabel

This is a short code to be used as a name for the update site. It will be used in a filename so it should be something simple e.g. 'extlib'

updateSiteDir

This is the Directory that contains the update site. It is expected to have a subdirectory called **eclipse**, and within that subdirectory should be the site.xml, **features** directory and **plugins** directory

Example

```
<bxp:updateDesignerLink updateSiteLabel="extlib"
updateSiteDir="C:\\Projects\\extlib\updatesite" />
```

initdesignersite

NOTE I am not sure this is a great way to deploy plugins so I haven’t detailed this part yet.

This task is used to set up an update site that will always be loaded when Designer Is started up. This is only used when you are deploying plugins to Domino Designer using the sneaky 'permanent update site' method.

This task is just like the **updateDesignerLink** task, however it will create the update site folder structure within the framework directory of the Notes Installation.

You can then copy plugins to this update site, and they will be loaded by designer (if you have configured platform.xml using the **checkplatformxml** task)

Properties

notesProgDir

The Notes Program Directory on this machine

Attributes

updateSiteLabel

This is a short code to be used as a name for the update site. It will be used in a filename so it should be something simple e.g. 'extlib'

Example

```
<bxp:updateDesignerLink updateSiteLabel="extlib"
updateSiteDir="C:\\Projects\\extlib\updatesite" />
```

checkplatformxml

This checks that the IBM Notes Installation’s platform.xml file is configured to automatically load plugins from the filesystem. The task will fail if the file is not configured properly.

Properties

notesProgDir

The Notes Program Directory on this machine

Example

```
        <!-- Make sure notesDataDir property is set -->
        <property name="notesDataDir" location="H:\Notes\Data"/>

        <bxp:checkPlatformXml/>
```

configuredynamicplugins

This task configures IBM Notes' **platform.xml** file so that IBM Notes will load plugins from the filesystem without the need to 'approve' them via the User Interface.

Usually you need to go through the whole provisioning process for plugin updates, but this task will put IBM Notes into a configuration that bypasses this.

Properties

notesDataDir

the location of IBM Notes Data directory on this machine

Example

```
        <!-- Make sure notesDataDir property is set -->
        <property name="notesDataDir" location="H:\Notes\Data"/>

        <bxp:configureDynamicPlugins />
```

Server Related Ant Tasks

These tasks are designed to interact with the domino server by issuing commands to the console.

controlhttp

The controlhttp task is used to control a Domino Server’s http tasks. It ultimately just issues a console command to the domino console.

Attributes

server

The server that you are controlling

action

start|stop|restart

Example

```
<target name="teststarthttp">
    <bxp:controlhttp server="Domino02" action="start" />
</target>
```

```
<target name="teststophttp">
    <bxp:controlhttp server="Domino02" action="stop" />
</target>
```

```
<target name="testrestarthttp">
    <bxp:controlhttp server="Domino02" action="restart" />
</target>
```

maintenancewarning

The maintenance warning task issues a console command to the Domino OSGi console which tells current users that there will be a maintenance outage in a certain amount of time

> The Maintenance warning relies on another plugin that has not yet been added to BuildXPages so this task wont be useful until that plugin is added to this project

Attributes

server

The server that you are controlling

minutes

in how many minutes time is the outage going to occur

Example

```
<target name="testmaintenancewarn">
    <bxp:maintenancewarning server="CameronWS" minutes="20" />
</target>
```

```
<target name="testmwcancel">
    <bxp:maintenancewarning server="CameronWS" cancelwarning="true" />
</target>
```

loaddesign

Issues a 'load design' console command for a server

Attributes

server

The server that you are controlling

directory

which directory should have it’s designs refreshed

Example

```
<target name="testloaddesign">
    <bxp:loaddesign server="CameronWS" directory="Cameron\Jogs\JobHub" />
</target>
```

Having Trouble?

If you have found a bug, can’t understand something, want some advice then. Submit an Issue using the Project’s Github Issues page and I will get back to you!

Feedback Welcome!

Over the past couple of years, lots of trial, error and research has gone into getting this project to where it is (for my own benefit!)

I recently decided to spend some further time to clean it all up and write documentation so it is easily available to the rest of the community. It has taken about a month to get all this done, so if this actually helps anyone else I would love to hear about it so I know it was worth the effort!

If this project helps you, if you have suggestions anything just send me a tweet at @gregorbyte or put through an issue in the Project’s Github Issues page