


OpenOffice NetBeans Integration

Kay Koll

Using a new OpenOffice.org plug-in module to create, build, deploy and debug OpenOffice.org/StarOffice extensions and client applications



fice.org



Automating office tasks, implementing document-based workflows or building corporate solutions are common tasks that can benefit from using the API of an Office suite. OpenOffice.org with its millions of users, and

Java with its huge community have several similarities; both, for example, are multiplatform and open source. But a tool to combine both worlds was missing. That's where the new OpenOffice.org plugin module for NetBeans comes into the game.

In this article we will show you how to use the new module to build Java components which extend OpenOffice.org's general functionality; we also create a new Calc function and a client application that accesses OOo features. (StarOffice won't be mentioned explicitly in the article, but everything shown here works for StarOffice as well.)

Overview

The OpenOffice.org plugin module for NetBeans provides four wizards which let you create general Add-Ons, Calc Add-ins, Components and

Client Applications:

- An **Add-On** is widely available and not limited to a certain document type. It can also implement its own toolbars and menus. Add-ons are typically used for implementing new features that are directly accessible to users.
- A **Calc Add-In** implements a new Calc function for the Function Autopilot in spreadsheet documents. A Calc function requires a different set of attributes than those covered by the general Add-On wizard – like function parameter definitions.
- **Components** can be used to extend the OpenOffice.org API and be accessed through scripting languages. Components can also extend OOo's charting functionality.
- Finally, external **Client Applications** can use OpenOffice.org's

 wiki.services.openoffice.org/wiki/OpenOffice_NetBeans_Integration

The project's homepage on the OpenOffice.org Wiki

functionality to create, convert, print or manipulate documents. Thus, OpenOffice.org can be used as powerful rendering and printing engine within a larger solution.

The new wizards create NetBeans projects with all necessary configurations, such as links to the OpenOffice.org Java libraries. They also define build targets, perform remote debugging setup and generate Java code skeletons, among other operations.

Requirements, installation and configuration

To use the NetBeans OOo plugin module you need NetBeans 5.5 or newer, and either OpenOffice.org 2.0.4 or StarOffice 8 PU4; also needed is the OpenOffice.org SDK 2.0.4 or newer.

The wizards are provided in a common NetBeans module that can be installed and updated via the Update Center. Just download the file `api.openoffice.org/Projects/NetBeansIntegration/org-openoffice-extensions.nbm` and use `Run>Update Center>Install Manually Downloaded Modules (.nbm Files)`¹.

Setup is straightforward. There are just two configuration items available: the paths to the OpenOffice.org installation and to the SDK. These are accessible during installation and also via the `Tools >Options>Miscellaneous` dialog.

Creating OOo extensions

All four extension types are packed as OOo packages. This package format was introduced in OpenOffice.org 2.0.4 and uses the file extension `.OXT`. A corresponding MIME type is registered in OpenOffice.org, which enables users to install extensions simply by double clicking.

Add-Ons

The `File>New Project` dialog provides a new `OpenOffice.org` category that opens the Add-On wizard. Start the wizard, and enter “myAddOn” for both the project and add-on names. Set the Java package to “org.openoffice”; provide the project folder, and check the `Create Menu` and

`Create Toolbar` checkboxes.


Click `Next` and you’ll be able to specify the add-on commands. Each command can have an icon assigned to it on the toolbar. Ideally, you should provide four different icons (two different sizes in two contrast levels), but it’s possible to use the same image for all icon types. If you do this OpenOffice.org will scale the image accordingly.

You also need to enter a display name for the command. This name is used for the menu and toolbars and can be different from the command name. Note that only the display name can be translated: the wizard lets you define different locales, but the command name is the same for all locales.

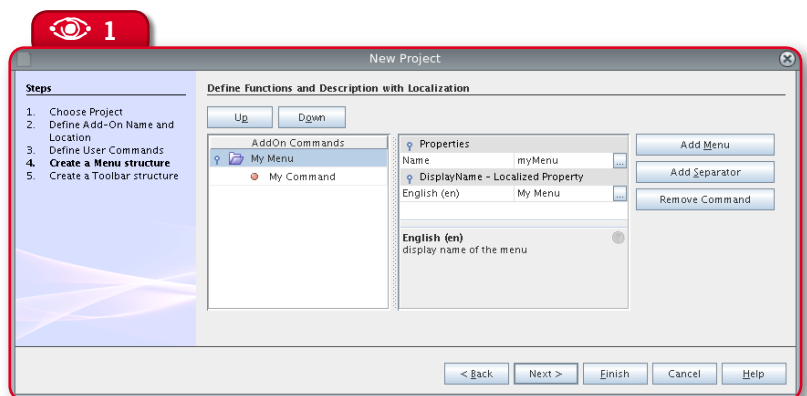
In the next step you define the menu structure. This is optional, as not all add-ons have their own menus. **Figure 1** shows the menu definition wizard page.

The next step is also optional: the definition of a toolbar to call the add-on’s commands (see **Figure 2**). The icons in the toolbar preview should look familiar, as they were specified in step three. Set the names as “myAddOn”, define the icons and set the category.

After you click `Finish`, the wizard creates two configuration files and a Java class

 **Figure 1**
Defining the add-on’s menu structure.

¹ By the time you read this, it’s possible that the module will show up in the Update Center, making the manual download unnecessary.



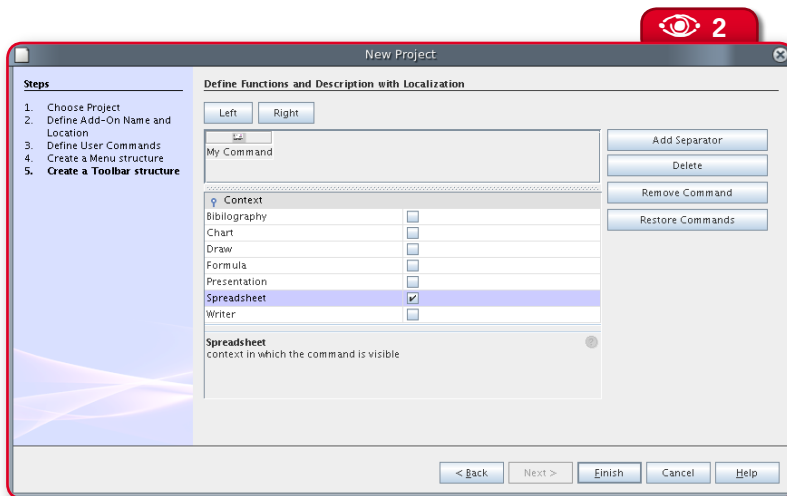


Figure 2
Defining the
toolbar for the
add-on.

`dispatch()` method each time the protocol handler routes commands. The fully implemented `dispatch()` method, which adds the “Hello World” message to the current document is shown in **Listing 2**. Note that only the lines in bold are new; the rest were part of the skeleton. The code basically determines the current document and then adds text to it. Notice also that the `aURL` parameter uses the

skeleton. The `AddOns.xcu` configuration file includes the add-on parameters, and `ProtocolHandler.xcu` defines the protocol handler configuration. Protocol handlers are part of OpenOffice.org dispatch framework; they bind user-interface controls, such as menu or toolbar items, to the functionality of OpenOffice.org. Everything reachable through the user interface is described by a command URL and corresponding parameters.

The structure of the `ProtocolHandler.xcu` file defines a namespace for the add-on (`org.openoffice.myaddon`, for our example). All commands defined by the same add-on use this namespace. See **Listing 1**.

The Java code skeleton looks more complicated than it really is. Most of the methods are necessary only for OpenOffice.org internal implementation reasons and don't need to be changed at all.

OpenOffice.org calls the

OpenOffice.org specific `com.sun.star.util.URL` class instead of the `java.net.URL` class.

Since all commands fired by the user interface are passed through the `dispatch()` method, it's necessary to filter explicitly for the namespace `org.openoffice.myaddon`, which represents the commands of our add-on. Finally, `myAddOn` is the command that's fired when the user calls the add-on via the toolbar or menu.

Our add-on is now ready to deploy. Right click the project name and choose *Deploy Office Extension* from the context menu. NetBeans compiles all necessary files, creates an OpenOffice.org extension package file, and deploys it. Depending on the setup of the add-on, a new top-level menu and/or a toolbar are displayed in

Listing 1. Excerpt from the `ProtocolHandler.xcu` configuration file.

```
<node oor:name="HandlerSet">
  <node oor:name="org.openoffice.myAddOn" oor:op="replace">
    <prop oor:name="Protocols" oor:type="oor:string-list">
      <value>org.openoffice.myaddon:*</value>
    </prop>
  </node>
</node>
```

Listing 2. `dispatch()` method example.

```
public void dispatch( URL aURL, PropertyValue[] aArguments )
{
    if ( aURL.Protocol.compareTo("org.openoffice.myaddon:") == 0 ) {
        if ( aURL.Path.compareTo("myAddOn") == 0 ) {
            XTextDocument xDoc = (XTextDocument) UnoRuntime.queryInterface(
                XTextDocument.class, m_xFrame.getController().getModel());
            xDoc.getText().setString("Hello World");
            return;
        }
    }
}
```

OpenOffice.org. For our example both should show up (see the new menu in **Figure 3**).

Developing more complex add-ons will probably require an extensive debugging session. However, as the add-ons run in OpenOffice.org's JVM, NetBeans' built-in debugger won't work. We need remote debugging.

The OoO plugin module adds a command to the project's context menu for starting a remote debugging session, which means that a manual setup of the client JVM or the remote debugger is not required. To use this feature, set a breakpoint within the `dispatch()` method, then call *Debug Extension in Target Office* as displayed in **Figure 4**. An OpenOffice.org instance will start automatically. Choose *My Command* from the add-on's menu and the debugger will stop at the breakpoint.

Calc Add-Ins

Let's now see how to create a Calc Add-In extension, which implements OpenOffice.org Calc functions. These functions are tightly integrated with the Calc application, so users will not recognize the differences between a standard function and one provided by an add-in; there are no new menus, toolbars or other evidences of an extension.

The Calc Add-In wizard is also located in the *StarOffice/OpenOffice.org* category of the *File>New Project* dialog. Start the wizard and enter a name and location for the NetBeans project. We'll use "myAddIn" for both names and "org.openoffice" again as the package name; also make sure *Create backward compatible Calc Add-In* is unchecked.

Click *Next* to enter the name and parameters of the Calc functions.

The definition of a Calc Add-In function requires specifying the following parameters:

- The name of the Java method which implements the Calc function.
- The data type of the result of the new function.
- The exception the Java

implementation throws in case of errors. An additional dialog provides access to all available exceptions. (This property is optional.)

- The Category where the function is listed within Calc's Function Wizard.
- The function's display name. This can be different from the name of the corresponding Java method.

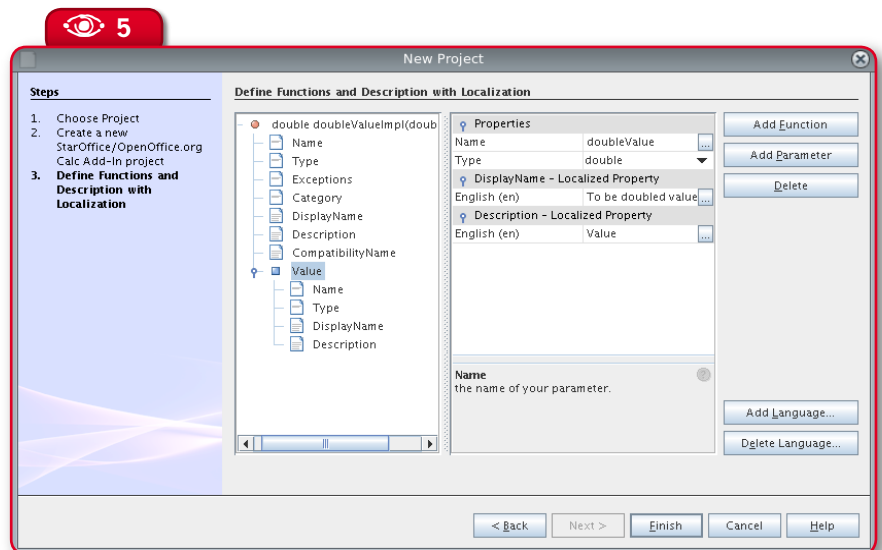
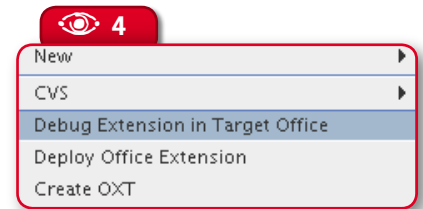
Figure 5 shows how the wizard presents these parameters.

There are some additional parameters, which are all localizable. Calc functions are usually localized, having different names for each language; for example, the function **Sum()** in the English-language Calc is

Figure 3
The new add-on menu.

Figure 4
Remote debugging context menu item.

Figure 5
Defining a function and its parameters in the Calc Add-In wizard.



named **Summe()** in German Calc releases.

The *Description* parameter indicates the purpose of the function, and is displayed in Calc's Function Wizard. *Compatibility Name* is necessary to deal with Microsoft Excel integration; it's optional and usually not necessary. Finally we have the optional parameters of the Calc function. They require a specification of the data type, the implementation name, and the displayed name and description.

To create our simple Calc Add-In, set Name to "doubleValueImpl" and Type to *double*; leave the Exceptions section empty; set Category to "Add-In" and Displayed Name to "doubleValue". Change Displayed Description to "Simple Calc Add-In: Doubles the given value", and set Compatibility Name to "doubleValue". Then provide the following values for the first parameter:

- Name – "doubleValue"
- Type – double
- Displayed Name – "Value"
- Displayed Description – "Value to be doubled"

Click *Finish* and several add-in-related files will be created. The most important is the Java class for the add-in implementation. There's also the configuration file *CalcAddin.xcu*, which holds the add-in's parameters. Functions exported by the add-in need to be defined in a new interface. The function names in this interface, together with the add-in's service name, are used internally to identify an add-in function. The

myAdd-In.idl and *XmyAddIn.idl* files define this service and the interface. They are used by tools and compilers available in the OpenOffice.org SDK, which build Java source and header files; but this process is hidden by the OOo plugin module.

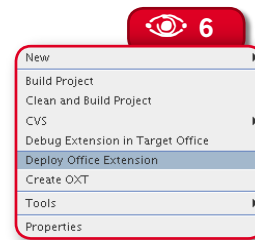
Most of the initial skeleton code need not be changed. Our **doubleValueImpl()** method is called by the add-in and provides the implementation of its functionality. The implementation is really simple; it just doubles all values given by the user:


```
public double doubleValueImpl(double doubleValue) {
    return doubleValue * 2;
}
```

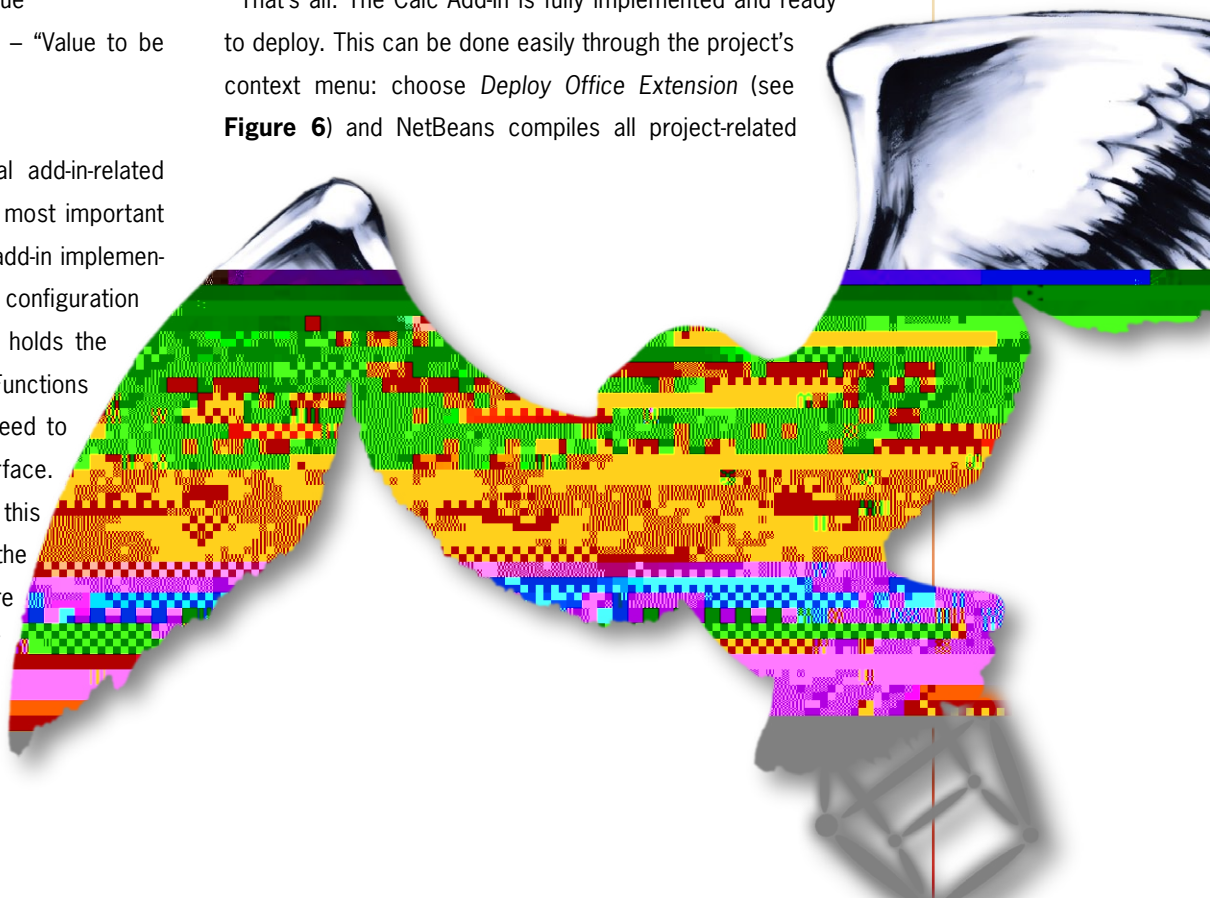
NetBeans will complain that the **XmyAddIn** interface is missing, which is true so far. The reason is that the interface is defined in a UNO IDL (Interface Definition Language) file and not as a Java class. The plugin will create Java code based on this IDL file as well as other add-in related services and interfaces automatically when the project is compiled.

 **UNO (Universal Network Objects)** is OpenOffice.org's component technology.

That's all. The Calc Add-In is fully implemented and ready to deploy. This can be done easily through the project's context menu: choose *Deploy Office Extension* (see **Figure 6**) and NetBeans compiles all project-related



 **Figure 6**
Deployment context menu



Download
page of
the latest
release of the
OpenOffice.org
SDK

download.openoffice.org/sdk.html

files, builds an OXT extension package and installs it in OpenOffice.org.

You can test the new add-in by starting a Calc instance and creating a new spreadsheet document. Then call the Calc's Function wizard, where you'll see the new function listed under the Add-In category. To verify that our new function works, enter in any cell the formula "`=doublevalue(3)`". As expected, Calc will produce 6 as the result.

Client Applications

A client application is an external solution that makes use of the OpenOffice.org functionality instead of extending it. It can use OOO to convert or process any document supported by the office suite.

The Client Application wizard is also available in a OpenOffice.org New Project category. You just need to start the wizard and enter the name and location of the NetBeans project; no further settings are required. Click *Finish*, and the wizard creates the Java skeleton.

As a client application is not an integrated part of OpenOffice.org, most of the configuration and IDL files are not necessary. The NetBeans project consists of just a Java class and classpath settings for OpenOffice.org Java libraries.

You'll notice that the code skeleton for a client application is quite small in comparison to the add-in and add-on skeletons. It is basically not much different from the Java class code generated by the general Java Class wizard. The implementation of the `main()` method contains a single line of code (besides exception handling):

```
XComponentContext xContext = Bootstrap.bootstrap();
```

The generated class works as a client of an OpenOffice.org pro-

cess, with OpenOffice.org acting as a server with its own component context. The client program initializes the Universal Network Objects technology (UNO) and gets the component context from the OOO process. This initialization process establishes a pipe connection to a running OpenOffice.org process (starting a new process if necessary) and returns the remote component context.

The `getServiceManager()` method from the component context obtains the remote service manager from the OpenOffice.org process, which allows access to the complete office functionality available through the API:

```
XMultiComponentFactory xMCF =  
    xContext.getServiceManager();
```

Having the service manager, we can obtain the OpenOffice.org Desktop, which handles application windows and lets you load and create documents. The `com.sun.star.frame.Desktop` service represents this Desktop:

```
XDesktop xDesktop = (XDesktop) UnoRuntime.  
queryInterface(XDesktop.class,  
    xMCF.createInstanceWithContext(  
        "com.sun.star.frame.Desktop", xContext));
```

Listing 3. Creating a text document.


```
XComponentLoader xComponentLoader =  
    (XComponentLoader) UnoRuntime.queryInterface(  
        XComponentLoader.class, xDesktop);  
  
PropertyValues xEmptyArgs[] = new PropertyValues[0];  
  
XComponent xComponent =  
    xComponentLoader.loadComponentFromURL(  
        "private:factory/swriter",  
        "_blank", 0, xEmptyArgs);  
  
XTextDocument xTextDocument =  
    (XTextDocument) UnoRuntime.queryInterface(  
        XTextDocument.class, xComponent);
```



Now we have an instance of the Desktop without a document; but a text document is necessary to display our greeting. The **XComponentLoader** interface exports the **loadComponentFromURL()** method to load and create a document. See it in use in **Listing 3**. The *private:factory/swriter* URL creates a new text document.

The new document will show a cursor waiting for input. This input has to come from the client application. The OOo API uses a text cursor abstraction to add text to the document, represented by the **XTextCursor** interface:

```
XText xText = xTextDocument.getText();
XTextCursor xTextCursor =
    (XTextCursor) xText.createTextCursor();
```

 The blinking cursor in the document and **XTextCursor** are independent of each other. OpenOffice.org Writer uses MVC to separate the content/model from the view. The text cursor is the view in this context, and the cursors created by **createTextCursor()** are the model. You can create several models for text cursors.

Finally, the method **insertString()** adds the message to the document:

```
xText.insertString( xTextCursor, "Hello World", false );
```

Components

OpenOffice.org can be extended by Components. These are shared libraries or JAR files with the ability to instantiate



objects that can integrate themselves into the UNO environment. A Component can access existing features of OpenOffice.org, and be used from within the office suite through the object communication mechanisms provided by UNO. In fact, the add-ons and add-ins described before are nothing more than specialized UNO components.

Components created by the Component wizard do not require access to a menu or to toolbars, nor do they extend the Calc function repository. They can be used to implement new interfaces and services. This flexibility and power makes it impossible to create a simple “Hello” component, and creating a fully working Component would go beyond the scope of this article. There are many excellent articles and documents available which describe the creation of new OpenOffice.org interfaces and services. Specifically, we refer you to Chapter 4 of the OpenOffice.org Developer Guide, which is a good source of examples.

Conclusions

In the past, writing components to integrate with OpenOffice.org required an extensive setup of the NetBeans infrastructure, with steep learning curves. Everything was documented somewhere but putting this information together took far too much effort. This has changed with the new OpenOffice.org plugin module we’ve covered in this article. The module takes care of integration chores and lets developers concentrate on the implementation of their extensions. Also, the module’s remote debugging capabilities make it much easier and faster to debug applications based on OpenOffice.org. If you need to integrate with OpenOffice.org or StarOffice, give it a try!

The next releases of the plugin will integrate Java more closely into the OpenOffice.org scripting framework, and will let you use Java for typical scripting related tasks, combining the advantages of an integrated scripting language with the power of NetBeans and Java technology. ☸



Kay Koll
(kay.koll@sun.com) is responsible for the technical marketing of StarOffice/ OpenOffice.org. He has been working in various positions for StarOffice since 1995. Kay lives in Hamburg, Germany.

