

Combining kbmMW with Java

for kbmMW v. 1.03+

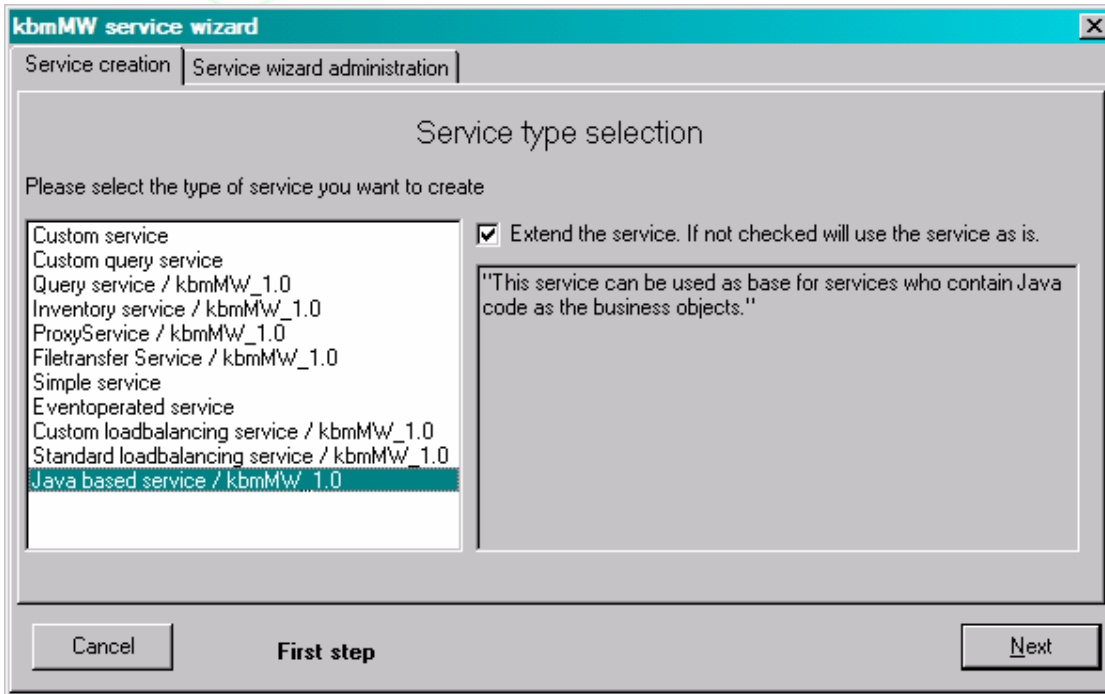
kbmMW includes a custom service, `TkbmMWCustomJavaService`, which makes kbmMW support Java services on the server.

A Java service is not much different from any other kbmMW service except that the service is written in Java instead of Delphi or C++.

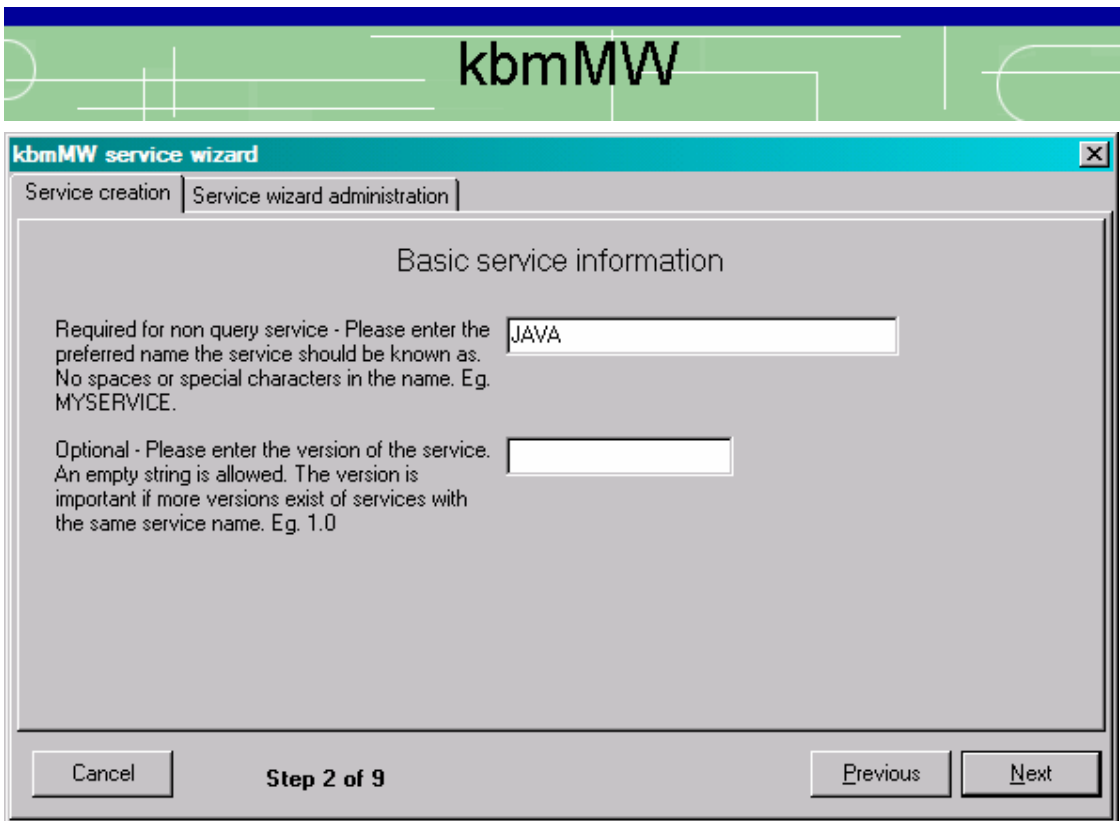
What the purpose of this then? By supporting Java in kbmMW, your clients have easy access to EJB and RMI based application servers. Since the kbmMW Java service can coexists with any other native kbmMW services and the Java service operates just like any other kbmMW based service, the clients will never get to know or need to know that Java code has run.

Creating the Java service

First create new Java service using the kbmMW service wizard:

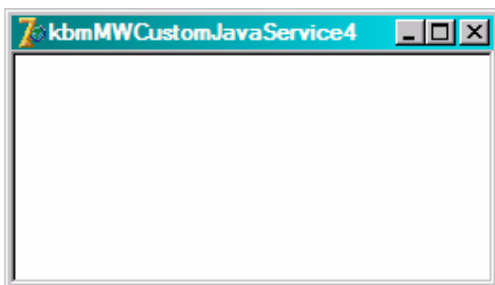


As with any other custom service, give it a preferred name and fill in any other optional service informations.



You don't need to add any service functions to the service, since that functionality is already in place in the `TkbmMWCustomJavaService` base class. You can however as usual choose to add your own extra native functionality in the same service if you want to by overriding `ProcessRequest` and remember to call inherited for functions not covered by your custom native code.

Finish the service creation and you will end up with an empty service datamodule as usual.



The datamodule do not contain Java specific to setup as such. Instead all special Java settings must be given while registering the Java service.



Registering the Java service

Just like with any other kbmMW services, the Java service needs to be registered to the TkbmMWServer. At the same time several Java specific options can be set:

```
var
  sd:TkbmMWCustomServiceDefinition;
begin
  sd:=kbnMWServer1.RegisterService(TJavaService, false);
  with sd as TkbmMWJavaServiceDefinition do
  begin
    LibraryFileName:='c:\Programmer\Java\j2re1.4.1_02\bin\client\jvm.dll';
    JNIVersion:=mwjni_1_4;
    ClassPath:='.';
    MainClass:='Demo';
    MainFunction:='ProcessRequest';
    Debug:=true;
    Verbose:=true;
    JIT:=false;
    CallbackEnabled:=true;
  end;
end;
```

Here you can see that the custom service definition returned by RegisterService can be casted to a Java specific version of it. This gives access to several Java specific settings that usually should be set.

LibraryFileName should point to the jvm.dll (J2RE 1.2 and newer) or javai.dll (J2RE 1.1) file on Windows. On Linux it should point to the shared object library jvm.so or javai.so.

JNIVersion should be set to the version of the Java Native Interface used. Its usually the same as the Java runtime environment version.

ClassPath should be set to the path that should be searched for *.class files. You can, in normal Java tradition, also include names of .jar files. All entries should be separated by ; on Windows and : on Linux.

MainClass should be the name of the class containing the service function to call.

MainFunction should be the name of the service function within the MainClass. The service function must be declared as

```
public static Object MainFunction(String Func, Object [] Args)
```

Debug can be set to true to enable some Java debug (through the vfprintf Java interface) to the Delphi/BCB/Kylix debug console (event view). Notice that anything printed via System.out do not show up in the debug. Instead this must be redirected to a file to be visible for the developer.

Verbose can be set to true to make the debug extra verbose.

JIT can be set to false (default true) to disable Just in Time compilation. Thus the Java code will be interpreted instead of compiled to native code on the fly.

CallbackEnabled can be set to true (default false) to allow the Java code to issue requests to the kbmMW based server the like the Java code is a client.

kbmMW supports marshalling the following variant values to Java objects and back:

Variant type	Java type
varBoolean	java.lang.Boolean
varShortInt, varByte	java.lang.Byte
varDouble varCurrency	java.lang.Double
varSingle	java.lang.Float
varSmallint, varWord	java.lang.Short
varInteger	java.lang.Integer
varLongWord varInt64	java.lang.Long
varDate	java.util.Date
varEmpty, varNull	java.lang.Void
varString	java.lang.String
varArray	Array of Object

Example of a simple Java service adding two integers.

```
import java.io.*;
import java.lang.Integer;
import java.lang.String;
import kbmMW.*;

public class Demo {

    // Notice that you can choose other names instead of Demo and ProcessRequest, but that the
    // function must be static, return an Object and accept a string and an array of objects.
    public static Object ProcessRequest(String Func, kbmMWClientIdentity Ident, Object [] Args) {

        try {
            // Redirect stdout and stderr to files in the root of the C drive.
            // Allows the developer to add debug internally in the Java application.
            System.setOut(new PrintStream(new java.io.FileOutputStream("c:\\demo_out.txt")));
            System.setErr(new PrintStream(new java.io.FileOutputStream("c:\\demo_err.txt")));

            // Show the function called.
            System.out.println("Func "+Func+" called");

            // Show the client information.
            System.out.println("Username="+Ident.Username);
            System.out.println("Password="+Ident.Password);
            System.out.println("Token="+Ident.Token);
            System.out.println("Location="+Ident.ClientLocation);
            System.out.println("RemoteLocation="+Ident.RemoteLocation);
            System.out.println("ID="+new Long(Ident.ID).toString());
            System.out.println("StateID="+new Integer(Ident.StateID).toString());
            if (Ident.Data == null)
                System.out.println("Data=<none>");
            else
                System.out.println("Data="+Ident.Data.toString());

            // Extract the 2 arguments (one should probably test that the number of arguments match
            // before fetching them ☺ )
            Integer i1,i2;
            i1=new Integer(Args[0].toString());
            i2=new Integer(Args[1].toString());

            // Request a service function on the server without arguments.
            kbmMWClientIdentity ci=new kbmMWClientIdentity();
            ci.Username="User";
            ci.Password="Password";
            Object o1=kbmMWRequest.kbmMWRequest("INVENTORY","KBMMW_1.0","LIST",ci);
            System.out.println("Callback returned: "+(String)o1);

            // Return a new Integer object which is an addition of the two arguments.
            return(new Integer(i1.intValue()+i2.intValue()));
        }
        catch (Exception e) {
            // This will go to the redirected files.
            e.printStackTrace();
            return(null);
        }
    }
}
```

Now compile the Java code, and copy its .class file to the place specified by the ClassPath (in this sample into the same directory as where the kbmMW server is run from). Also make sure that the kbmMW package containing the kbmMWRequest.class and kbmMWClientIdentity.class files is available as a sub directory to the place where this samples class file has been placed. Other placement is also possible, but require you to add the place to the ClassPath.

You can have multiple business functions in the same class/main function by checking the Func string and operate accordingly to its setting.

Multiple classes can be added by simply registering the kbmMW based Java service multiple times and then set its service description to point on other main class, main function and classpath.

Client side

This sample allows a client to call the service in a normal way:

```
var
  n:variant;
begin
  kbmMWSimpleClient.ClientLocation:='A Java client';
  n:=kbmMWSimpleClient1.Request('JAVA','','ADD',[10,20]);
  ShowMessage('Result is '+inttostr(n));
end;
```

Additional information

The Java service must accept and return Objects, not primitive types.
Thus instead of returning 5, return new Integer(5).

Arrays can be returned like this:

```
public class Demo {  
    public static Object ProcessRequest(String Func, kbmMWClientIdentity Ident, Object [] Args) {  
        Integer n[];  
  
        try {  
            System.setOut(new PrintStream(new java.io.FileOutputStream("c:\\demo_out.txt")));  
            System.setErr(new PrintStream(new java.io.FileOutputStream("c:\\demo_err.txt")));  
            System.out.println("Func "+Func+" called");  
  
            n=new Integer[2];  
            n[0]=new Integer(10);  
            n[1]=new Integer(50);  
            return(n);  
        }  
        catch (Exception e) {  
            e.printStackTrace();  
            return(null);  
        }  
    }  
}
```

In which case the client will now get a variant array as result from the call.

Happy Java'ing

Kim Madsen
Components4Developers