

# QUICK START GUIDE

Quick Start Guide Build 1. Software Version 3.03+.  
For iSeries 400, Windows 95+ and XP Edition.

**Exclusive Distribution Rights:  
Business Computer Design International, Inc.**

950 York Road  
Hinsdale IL, 60521 USA  
Phone: 630-986-0800  
Fax: 630-986-0926  
E-Mail: [sales@bcdsoftware.com](mailto:sales@bcdsoftware.com)  
Web: [www.bcdsoftware.com](http://www.bcdsoftware.com)

**For Technical Support, Contact:**  
ExcelSystems Software Development Inc.  
Phone: 250-655-1766  
Fax: 250-655-1733  
E-Mail: [excel@excelsystems.com](mailto:excel@excelsystems.com)  
Web: [www.excelsystems.com](http://www.excelsystems.com), [www.progenwebsmart.com](http://www.progenwebsmart.com)

## **ProGen *WebSmart* JSE**

**The PC-Based iSeries 400 Web Application Development Tool!**

**Copyright Notice**

ProGen WebSmart OE (Original Edition) is a Trademark of ESDI. Program © 2000 - 2003 ExcelSystems Software Development Inc. (ESDI). All rights reserved. Exclusive distribution rights Business Computer Design International, Inc. (BCDII)

ProGen WebSmart JSE 3.03 Quick Start Guide written and produced by ESDI using Microsoft Word 2002. Copyright © 2000-2003. ESDI. All rights reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without written consent from ESDI.

**System Requirements**

ProGen WebSmart JSE requires QShell on this iSeries and a minimum of V4R5M0 of the OS/400 operating system. You also need to have TCP/IP and the OS/400 FTP server configured correctly. WebSmart JSE will initially occupy approximately 30 MB on your iSeries 400, and about twice that during the install, and until the install libs are removed from your system. Minimum PC O/S is Windows 95, with IE 4 installed. Minimum processor speed: Pentium 200 (Note: it will run fine on slower machines, just slower!)

**Acknowledgments**

Throughout this manual, reference is made to several trademarks: ProGen WebSmart OE, WebSmart, WebSmart OE, WebSmart JSE and Exodus are trademarks of ESDI. IBM, AS/400, iSeries 400 and OS/400 are trademarks of International Business Machines. Windows, Windows 95 and Windows NT are registered trademarks of Microsoft Corporation. Acrobat is a registered trademark of Adobe. All other trademarks are acknowledged as the properties of their respective owners.

Printed in Canada

**Build 2 PDF: March 17, 2003.**

## Table of contents

Overview .....	4
WebSmart JSE servlet deployment scenarios .....	5
The Servlet life cycle.....	7
The WebSmart JSE Servlet Creation Process .....	8
Changes to the IDE to support WebSmart JSE.....	10
Minor PML syntax changes .....	15
Function calls within function calls .....	15
The syntax of the runtask() function .....	15
Testing for equality .....	16
New program fields.....	16
The WebSmart JSE WAS .....	19
Product Updates .....	21
Extending the PML .....	21
Creating your own WebSmart JSE functions.....	22
Using the WSStruct class.....	24
Using the IBM AS/400 Toolbox for Java .....	24
Applying updates and the WSUser class.....	24
Class reloading .....	26
Class reloading on the iSeries .....	26
Library list issues .....	27
Performance issues.....	27
WebSmart JSE Advantages.....	28
Porting WebSmart JSE Servlets to non-iSeries systems.....	28
Accessing non-iSeries data.....	30
Connecting to a JDBC data source.....	30
Repository issues.....	33
Using pgmf_lasterror and pgmf_lasterrtx .....	33
WebSmart JSE Server Side Configuration.....	34
WebSphere Configuration.....	34
The Servlet Environment .....	34
User Profiles.....	34
The Class Path.....	34
File locations .....	36
Apache/Tomcat Configuration.....	36
Starting the configuration and administration forms .....	36
User Profiles.....	37
Configuring Apache to send requests to a Tomcat worker instance.....	38
Configuring the Tomcat worker.....	39
Placing your servlet file in the correct location.....	42
Start the HTTP Server.....	42
Start the out-of-process ASF Tomcat server .....	42
Troubleshooting .....	43
Changes in WebSmart function usage in WebSmart JSE.....	45

## Overview

WebSmart Java Servlet Edition (JSE) is an enhanced version of WebSmart that allows you to deploy your WebSmart program definitions using Java Servlet technology. Although it is shipped as part of every copy of WebSmart version 3.0 or higher, a separate security code is required in order to use the product. You can obtain a temporary code for evaluation purposes from the BCD sales office. This guide should be used in conjunction with the WebSmart Reference and User guides, as the concepts and functionality provided by WebSmart are almost identical regardless of the deployment platform used. Any differences between the base product and the JSE version will be outlined here. A section dealing with differences between the functions provided by the RPG and Java versions of the product is included at the end of this guide.

The WebSmart Integrated Development Environment (IDE) executes on any PC using the Microsoft Windows Operating System. This system will be referred to as the “development client” for the remainder of this guide. WebSmart Original Edition (OE) can be used to generate CGI scripts for the IBM iSeries that provide access to DB2 data using a Web browser. These scripts are created using compiled RPG ILE source code which is generated from a WebSmart program definition. WebSmart JSE and WebSmart OE both require an association between the IDE and an IBM iSeries server, which will be referred to as the “development server” for the remainder of this guide. The development server is the machine on which the RPG and Java source code is compiled into executable objects.

The purpose of WebSmart JSE is to reproduce the functionality of existing RPG based WebSmart program definitions using the Java language, and to allow the creation of new definitions. This means that the existing iSeries dependent features of the product have been preserved. These include features such as access to data areas, the ability to call iSeries programs, record level access to DB2 data and the use of validation lists. For this reason the dependency between the iSeries and WebSmart JSE remains.

The most unique feature of Java technology is the portable nature of the executable Java byte code. Java objects will run on any hardware/OS platform that supports a Java virtual machine (VM). This means that WebSmart JSE servlets can be transferred to a variety of platforms. All that is required is the presence of a servlet engine on the non-iSeries system, and network access to an iSeries machine on which the WebSmart WAS is installed. This allows you to separate the Web serving and Servlet execution infrastructure from the iSeries, while still accessing iSeries data using existing program definitions. This has security and performance advantages over the CGI approach provided by WebSmart OE. For the remainder of this guide, the system on which a WebSmart JSE Servlet executes will be referred to as the “execution server”. The implementation of such an approach is discussed later in this guide.

The Java DataBase Connectivity (JDBC) standard allows a Java program to access a wide variety of data bases in a platform independent manner. JDBC defines a set of objects that enable a Java program to access a database using the SQL-92 syntax. Using JDBC, a Java program running on a PC could access an Oracle database running under the Solaris operating system on a Sun SPARC workstation. Another Java program running on an iSeries server could access a MySQL database running under the FreeBSD operating system on a PC. In each of these cases, the platform dependent code required to access the database is provided by a JDBC driver which is loaded when the program runs. WebSmart JSE Servlets have access to this functionality using the same SQL access functions provided by WebSmart OE. This frees your program definitions from reliance on DB2 as the single data source when using SQL access. The record level access functions are restricted to DB2 access on the iSeries. For the remainder of this guide, the system(s) on which the data resides will be referred to as the “database server(s)”.

The following table summarizes the different hardware/software components utilized in the creation and execution of a WebSmart JSE servlet:

<b>Server</b>	<b>Description</b>
Development Client	The Microsoft Windows based system on which the WebSmart IDE executes.
Development Server	The system with which the WebSmart IDE interacts when creating WebSmart JSE Servlets. This is always an iSeries system on which the WebSmart WAS is installed.
Execution Server	The system on which the WebSmart JSE Servlet runs.
Web Server	The system that handles the HTTP requests used to invoke the WebSmart JSE Servlet. Depending on the platform and servlet engine used, this may be a different system from the Execution Server.
Database Server	A system from which data that can be accessed using SQL or the Record Level Access Functions. In the latter case this is always an iSeries system. It is possible to maintain SQL and RLA connections with two different systems at the same time.

### ***WebSmart JSE servlet deployment scenarios***

You may want to execute a Servlet using the IBM HTTP Server (Powered by Apache) and the Tomcat servlet engine on an iSeries machine, and access SQL data using an iSeries DB2 JDBC driver. In this case the Development Server, the Execution Server, the Web Server and the Database Server may all be the same machine. This is illustrated in figure 1:

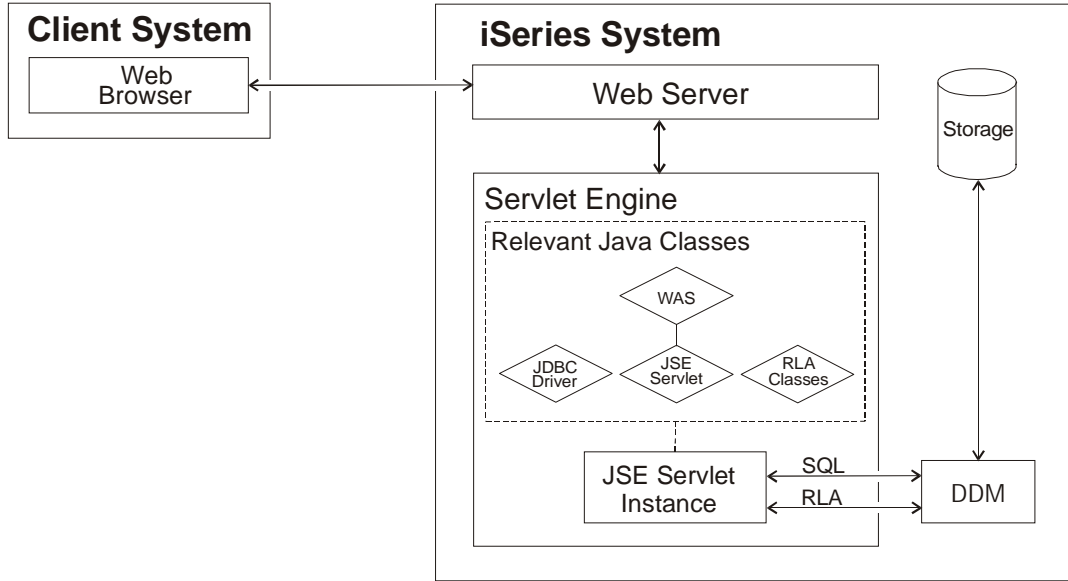


Figure 1

In another situation you may want to execute your WebSmart JSE servlet on a PC, Unix machine or other execution server, but still access iSeries data using SQL or record level access as shown in figure 2:

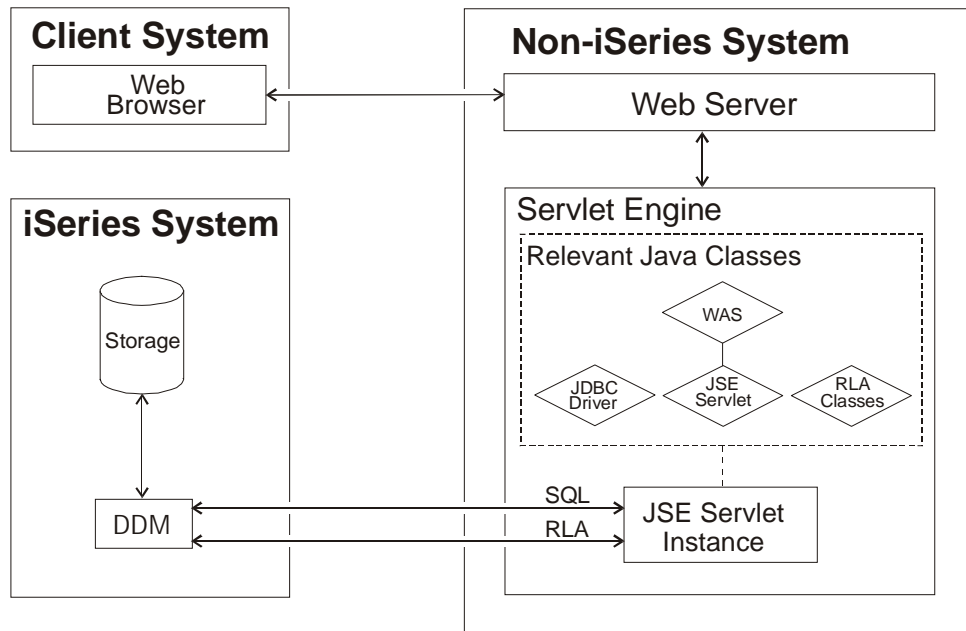


Figure 2

Another permutation of work load distribution may involve using the iSeries as the Web Server, the Execution Server and for record level access, while using a separate machine for SQL data access as shown in figure 3:

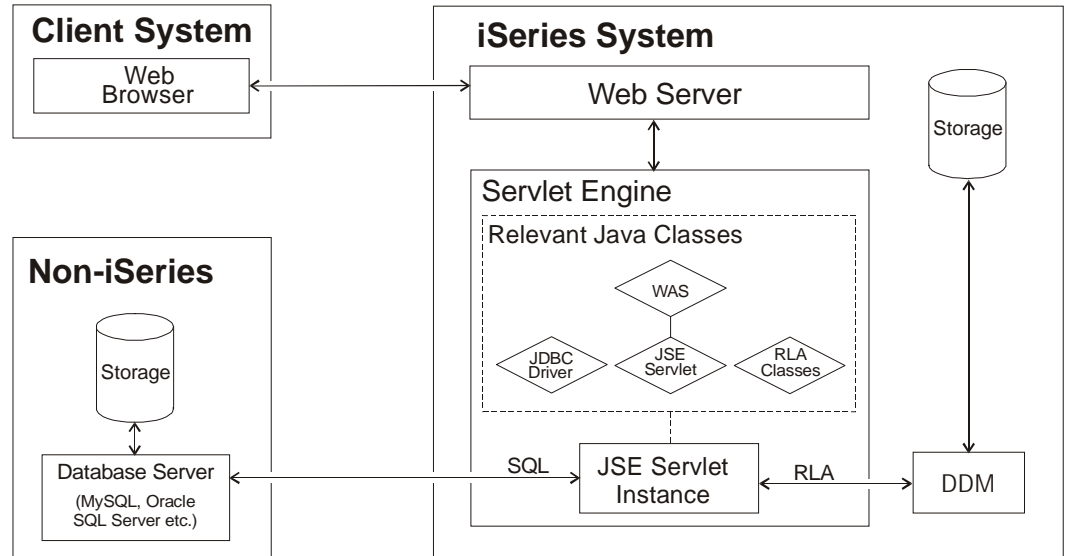


Figure 3

These examples illustrate only three potential solutions, in reality your situation may require another approach that can be easily implemented.

### **The Servlet life cycle**

Servlets are persistent objects that exist within an application server known as a Servlet engine. They handle requests that are issued from an external source. These requests can originate from another Java program, or as a result of a request from a Web Browser. WebSmart JSE Servlets are HTTP servlets, i.e., they service Web browser requests.

Previous versions of WebSmart produced only RPG CGI scripts, which are native iSeries executables. On the iSeries, a CGI script runs using a single thread of execution in a process under the QHTTSPVR subsystem. Using the classic CGI model the HTTP request is serviced using a single entry point, the program runs to completion and the output is returned to the Web browser. The process then terminates and all the data created by that process (the state information) is discarded. Under this single threaded model, all data within the program can be global without risking corruption by other threads. This is a good execution model for PML programs, as all data is global.

Java servlets run using a multi-threaded model in which all global data is shared between threads. This poses a problem when translating between a PML program and a Java servlet. In WebSmart JSE this problem is addressed by storing all global data in a table. Variables in this table can be accessed by name, and the table is stored in a ThreadLocal object. This allows a copy of the table associated with the current thread to be retrieved whenever the value of a PML variable needs to be read or updated. A knowledge of this approach is a great help in understanding the Java code generated from a PML program.

In order to improve performance, servlets perform time consuming operations as infrequently as possible, and preferably only once – at the time the servlet is invoked. Under the most basic Servlet execution model, the Servlet object is loaded into memory when it is first invoked. The Servlet engine allocates any storage the Servlet requires, and invokes its initialization method. The Java VM may also use a Just In Time (JIT) compiler to convert the Java byte code of the Servlet into native machine code, and may create a pool of threads to draw on when servicing requests. In addition to these servlet engine dependent tasks, a WebSmart JSE servlet may also perform the following tasks on startup: the establishment of JDBC connections, the creation of AS/400 Toolbox connections for DB2 record level access and the retrieval of DB2 file formats. This initial start up phase may be quite time consuming, but once the servlet is loaded in memory, subsequent requests will be handled much faster.

### ***The WebSmart JSE Servlet Creation Process***

The source code for JSE Servlets is produced by merging the procedural components of the program definition, represented by the PML segment, with the presentation components, represented by the html segments, into a single Java source file. There is also a single compiled Java program with the extension .class for each WebSmart definition. There is a close correspondence between the structure of the Java source code and that of the PML, although they differ in their syntax. The servlet creation process has the following steps:

1. The formats of any files used by the definition are loaded into the repository at design time, using the Exodus middleware product, which ships with WebSmart. Currently this is only an option for DB2 files residing on the iSeries. You must define work fields that correspond with the fields of tables residing in other databases that will be accessed using SQL.
2. When the “generate” option is selected from the file menu, by clicking the toolbar button or by hitting F6, the generate/compile cycle is started.
3. The program definition is converted into Java source code on the PC, and the source code is transferred to the iSeries development server using FTP. The source code then resides in the iSeries’ IFS.
4. An FTP command is issued which invokes the QSHELL interpreter on the iSeries in order to compile the source code that was uploaded in step 3. The resulting class file is placed in the same IFS directory as the source code.
5. A program is run on the iSeries which returns any compilation error messages to the WebSmart IDE over the same FTP connection that is used to upload the Java source code. If there were no compilation errors the servlet is ready to run.

If you have an instance of the WebSphere application server running on your iSeries, and the IFS directory which contains the newly compiled servlet is in WebSphere’s classpath, you will now be able to run the servlet. This is not the case if you are running the Tomcat servlet engine as each servlet has to be configured as part of a Web Application before it can be run. This configuration process is described in the “Apache/Tomcat Configuration” section of this guide.

Note: If there is a firewall between the development client and the development server, the firewall must be configured to allow FTP traffic to pass between the two machines.

The servlet creation process, and the software and hardware components involved, is illustrated diagrammatically in Figure 4.

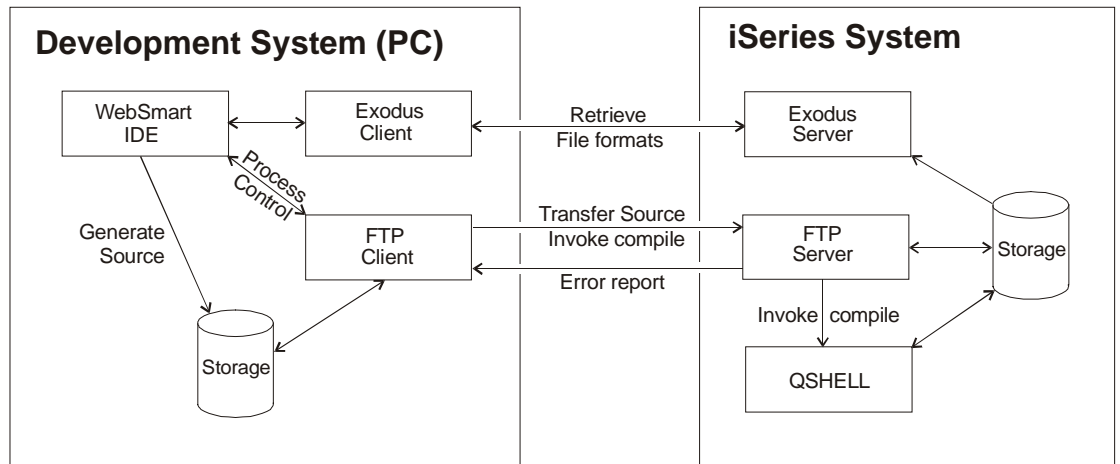


Figure 4

## Changes to the IDE to support WebSmart JSE

Some minor changes have been made to the interface of the IDE for situations where the RPG approach differs from the Java approach. All of these changes are contained in the Attributes dialog, which can be displayed by selecting any entry from the Attributes menu, and the options dialog, which can be displayed by selecting the Tools>Options menu option. We'll deal with the changes to the Attributes dialog first. The Generation options tab now contains a drop down list from which the target platform/language for generation can be chosen, as shown in Figure 5:

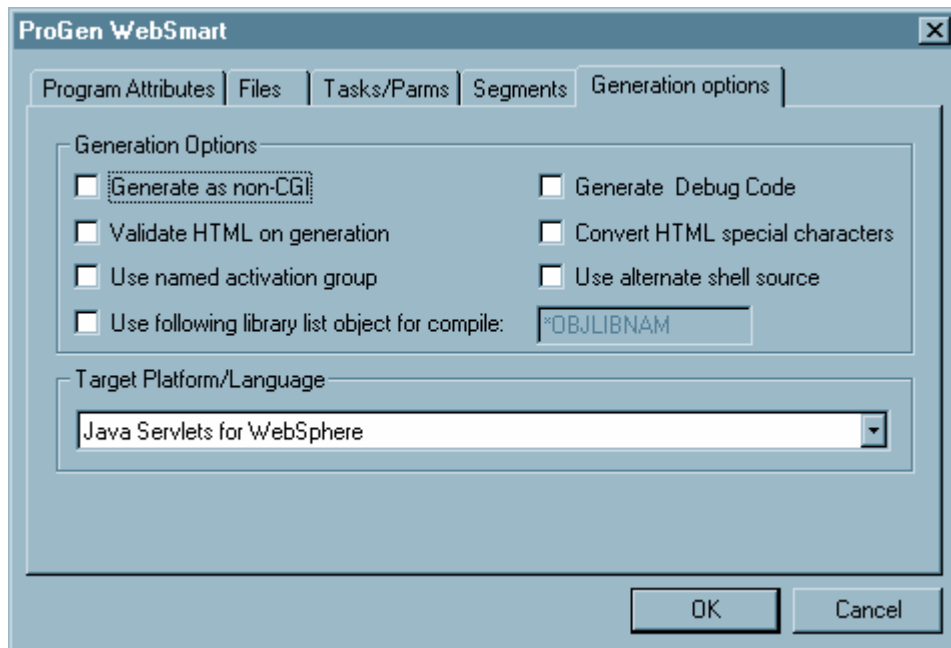


Figure 5: The Generation options tab

The choices are “Java Servlets for WebSphere” and “iSeries ILE RPG Generator”. The value present in this drop down affects the set of fields that are displayed in the Program Attributes tab. When “iSeries ILE RPG Generator” is selected the fields displayed are as shown in figure 6:

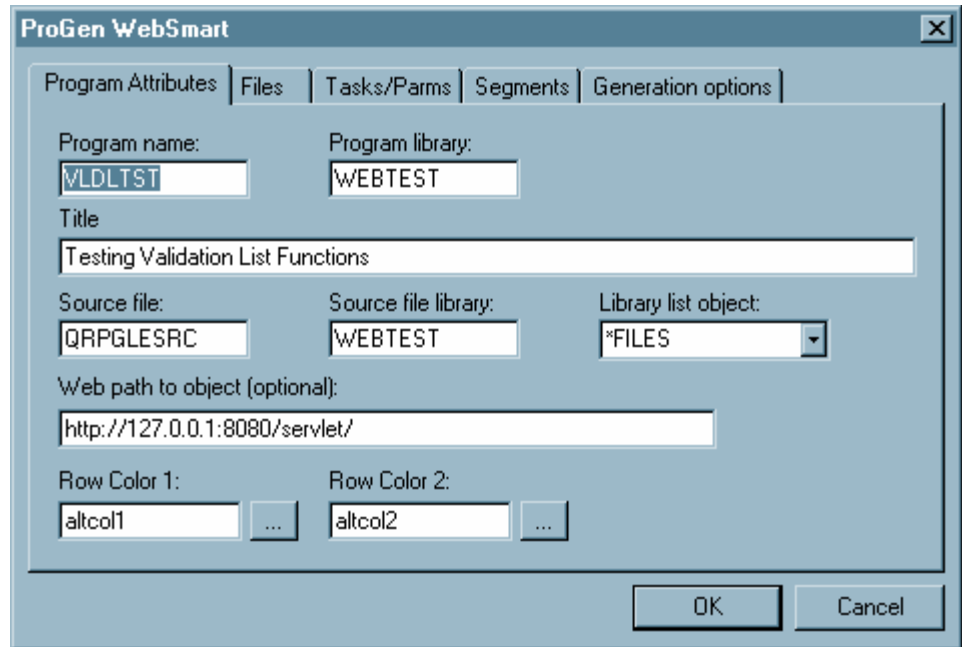


Figure 6: The Program Attributes tab for the ILE RPG platform

When “Java Servlets for WebSphere” is selected the fields are as shown in figure 7:

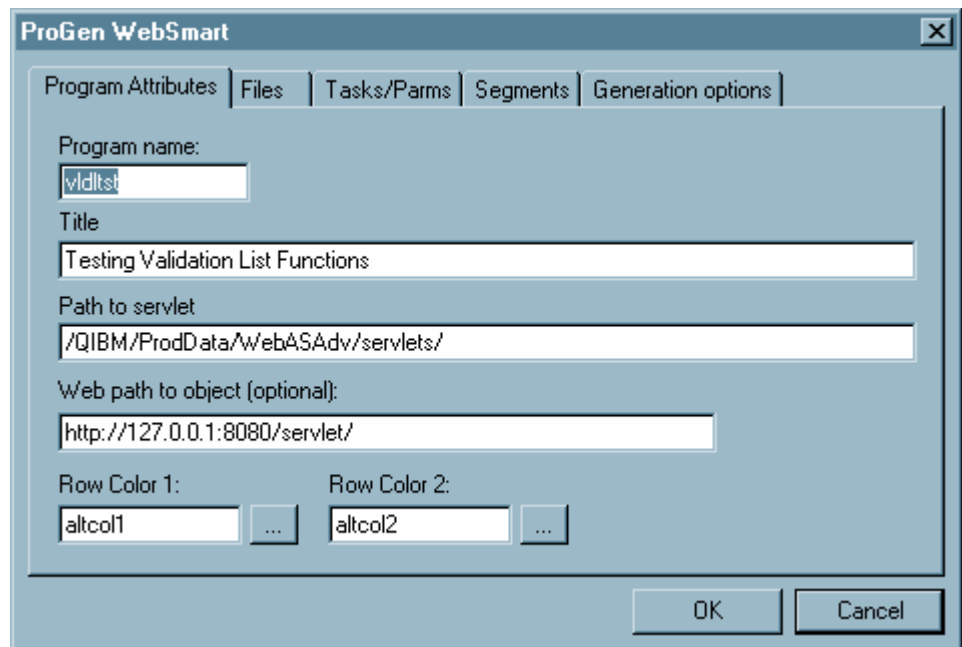


Figure 7: The Program Attributes tab for the Java platform

Note: When you change the target platform, by selecting a different value from the dropdown in the Generation options tab, the contents of the Program Attributes tab

will not be changed unless you first click on the OK button, and then reopen the dialog.

The differences between these two versions of the tab are mostly a result of the different file systems to which the source code and executable code are written. The RPG source and executable code are written to the QSYS file system, so the IDE needs to know which source file to use and which library the file resides in. The executable ILE object can be saved to a different library from that containing the source file, so the program library is prompted for as well.

These three fields are replaced with the single field “Path to Servlet” in figure 7. This field contains the fully qualified path to the IFS directory that will contain both the source code and the executable Java object. The names for the source and executable object files are derived from the program name, so using the example in figure 7 the source file would be called vldtst.java and the compiled program would be called vldtst.class. As the source and executable files are stored in the same IFS directory, there is no need to prompt for a second directory in which to place the executable file.

Note: Java is a case dependent language. This means that the names of all objects, including the Java servlet that you create, must be referenced using the correct case. While the full URL that you enter in the location field of your Web browser to run the Servlet depends on the location of the file in the execution server’s file system, and how the Web server and servlet engine are configured, the name of the servlet must always be referred to using the same case that you entered in the “Program name” field. An example URL to run the servlet named in figure 7 could be as follows:

```
http://127.0.0.1:8080/servlet/vldtst
```

If you entered VldTst instead of vldtst as the last portion of this URL, the servlet engine would not be able to find the definition for the servlet class, and an error would result. To make certain of the case to use in the URL you can check the contents of the Java source file, in this case vldtst.java, for the line that defines the class. This should be as follows:

```
public class vldtst extends WSUser
```

You’ll also note that the “Library list object” dropdown is missing from the “Program Attributes” tab in figure 7. This is because it is not possible to recreate the library list handling approach used in the RPG solution with the AS/400 Toolbox for Java. This issue is discussed in more detail in the “Library List Issues” section.

The remaining Interface changes are contained in the “Options” dialog. A Target Platform dropdown has been added to the “Environment” tab as shown in figure 8. This allows you to select the default platform to use when creating new program definitions:

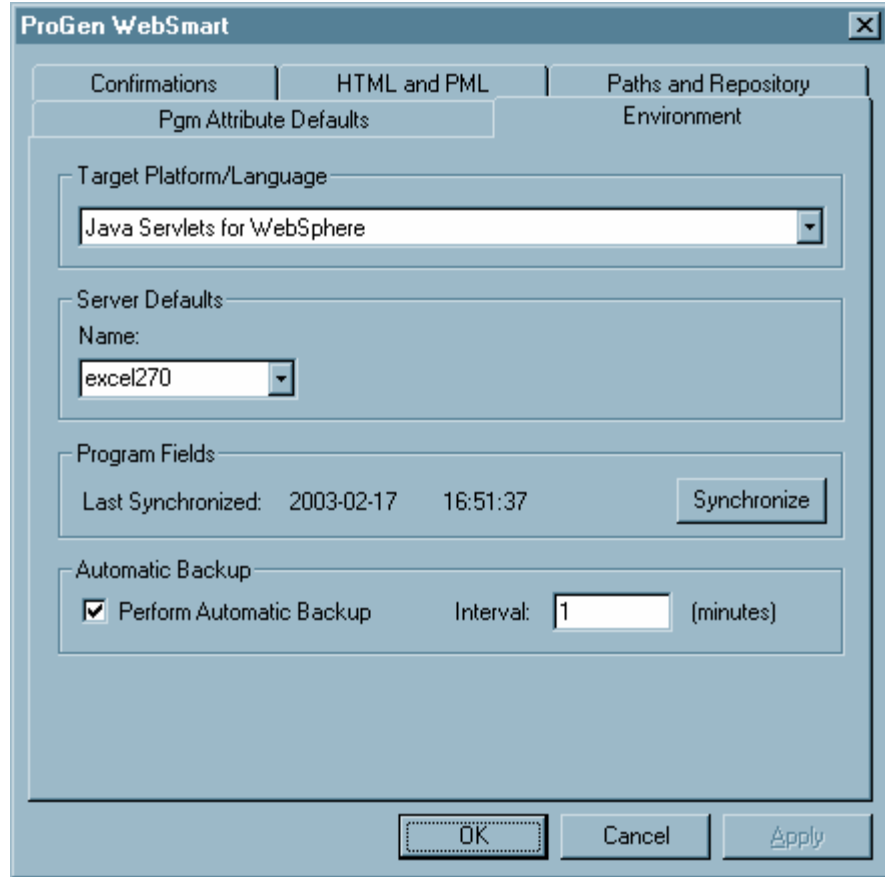


Figure 8: The Environment Tab

The first time you run the WebSmart version 3 IDE, it will prompt you for the value of this field using the dialog shown in figure 9:

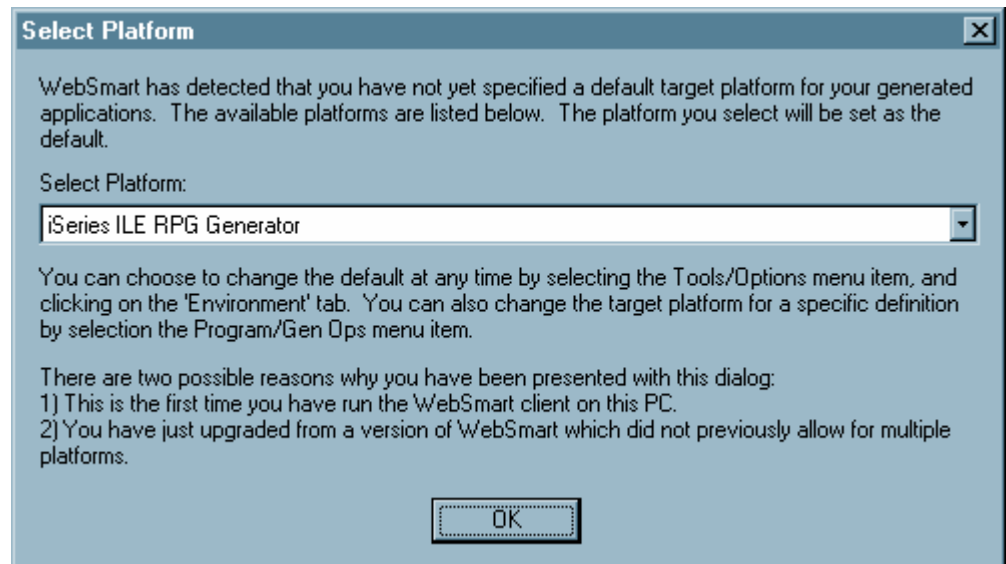


Figure 9: The platform selection dialog

Figure 10 shows the “Paths and Repository” tab. This has been renamed from the “Paths” tab in WebSmart version 2, and contains more changes than those relating specifically to WebSmart JSE. The single field to note in this context is the “Servlet Directory” field. This allows you to define the default value for the “Path to Servlet” field, in the “Program Attributes” tab, that will be used for new definitions.

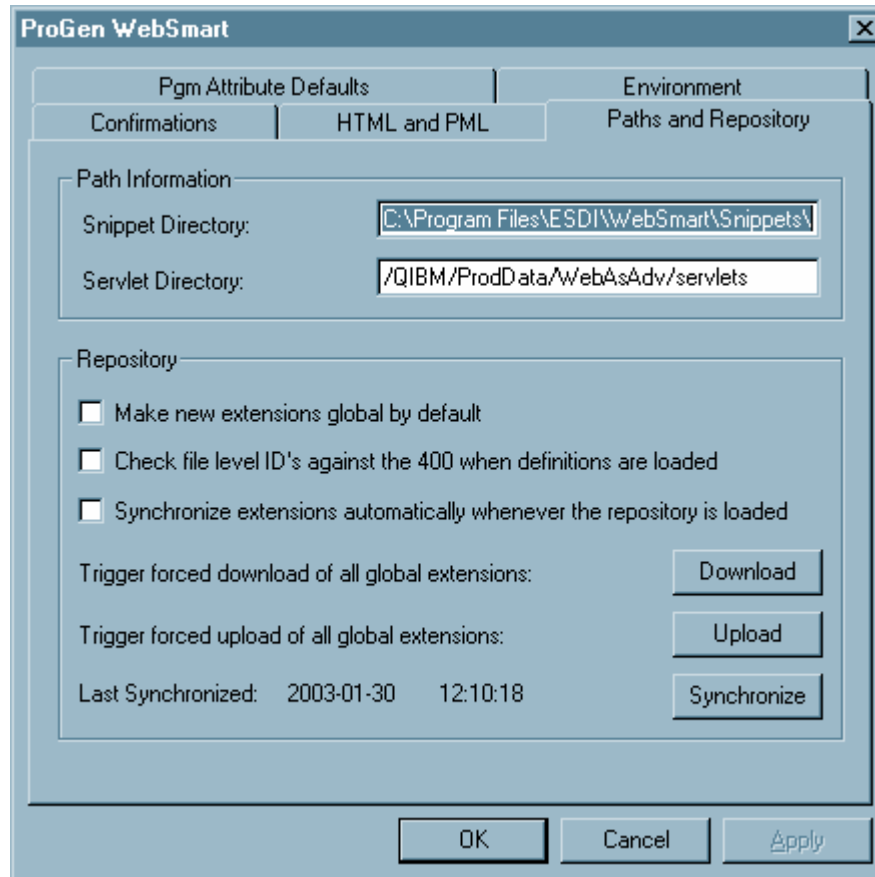


Figure 10: The Paths and Repository tab

## Minor PML syntax changes

Some minor changes have been made to the PML syntax because of the introduction of new features, and incompatibilities between Java and the PML supported in WebSmart version 2.

## Function calls within function calls

PML now supports function calls within function calls. Consider the following example which generates a random number between 0 and 9, and assigns it to an alphanumeric variable:

```
crtfld(aRand, 1, "A", 0, "Alphanumeric random number");  
aRand = numtoalpha(genrand(9));
```

In order to achieve the same result using the RPG solution you would have to produce the following code:

```
crtfld(aRand, 1, "A", 0, "Alphanumeric random number");  
crtfld(nRand, 1, "N", 0, "Numeric random number");  
  
nRand = genrand(9);  
aRand = numtoalpha(nRand);
```

This technique avoids the creation of temporary variables, such as `nRand` above, in many situations. Another situation where this technique is useful involves the display of a numeric value in a Web page using the `wrthtml()` function. Using the definition of `nRand` above:

```
nRand = genrand(9);  
wrthtml("The value of nRand is ", numtoalpha(nRand));
```

While these examples seem trivial, they can significantly reduce the amount of effort involved in producing a program definition, in situations where the technique is applicable. This approach can also aid in the readability of the PML code.

Note: While this functionality is available in WebSmart JSE, it is not transferable to the RPG solution. This means that if you use this approach you will be making your program definitions dependent on the Java platform.

## The syntax of the `runtask()` function

The introduction of the function call within a function call syntax produced an ambiguity in the syntax of the `runtask()` function. In WebSmart version 2 the following line appears in the `main()` function of many of the templates:

```
runtask("DEFAULT", display());
```

This instructs the CGI script generated from the PML to execute the `display()` function when the value of the `TASK` parameter is “DEFAULT” or blank. The second argument has the same syntax as a function call, and the generator is unable to determine whether the value of the argument should contain the output of the `display()` function, or if the name of the function should be associated with the value of the first argument (which is the correct interpretation). For this reason WebSmart version 3 expects the following syntax:

```
runtask("DEFAULT", display);
```

the only difference being that the opening and closing parentheses no longer follow the function name. This is the syntax used by all of the templates that ship with WebSmart version 3. As all the program definitions created using WebSmart version 2 will contain the old syntax, the IDE that ships with version 3 automatically removes the parentheses from the second argument of `runtask()` when an old definition is loaded.

## Testing for equality

The PML provided in WebSmart version 2 allowed you to test for the equality of two values using the following two approaches:

```
if(value1 = value2)
```

and

```
if(value1 == value2)
```

assignment was accomplished using a single approach:

```
variable1 = value1;
```

In most programming languages a distinction is made between the operator used for assignment and the operator used to test for equality. RPG uses the single equals sign in both cases, and determines the function of the operator from the context in which it is used. The PML implementation in WebSmart version 2 allowed you to use either the RPG syntax, or the syntax used by C and Java to test for equality. In order to support the Java syntax, however, WebSmart version 3 requires a distinction between the assignment and equality operators. Assignment is always performed using a single equals sign, and tests for equality are always performed using a double equals sign.

When opening a program definition that was created using WebSmart version 2 in the version 3 IDE, any occurrences of a single equals sign that are used to test for equality are converted to a double equals sign.

## New program fields

Two new program fields have been introduced to aid in the production of platform independent PML code. These are:

1. pgmf\_qpgmnam
2. pgmf\_platform

These are described in the following paragraphs. All versions of WebSmart include a program field called pgmf\_pgmnam which contains the name of the currently executing program. It is common to build hypertext links in an HTML segment that link to the currently executing program. This may also be done in the PML code. Two examples of this can be seen below, firstly the HTML then the PML:

1. `<a href="<field name=*pgmflds.pgmf_pgmnam>.pgm?TASK=display">...`
2. `url = append(pgmf_pgmnam, ".pgm?TASK=display");  
redirect(url);`

In the first example the `<a>` tag is followed by some link text, and then an `</a>` tag which are not shown. Clicking on the link text will run the program with the task "display". In the second example the program is run by redirecting to the url that is built programmatically by appending the contents of the pgmf\_pgmnam variable, the string ".pgm" and the query string.

The difficulty with these two code examples relates to the presence of the ".pgm" string. In each case, accessing the link works fine if the target program is an RPG CGI script running on an iSeries, but if it is a servlet the request will fail. This is because servlets are invoked, by default, using their program names with no extension, and the presence of the ".pgm" string at the end of the URL prevents the servlet engine from finding a matching class file.

A platform independent solution to this is to remove the hard coded ".pgm" strings from all locations in the HTML and PML code, and to substitute the variable pgmf\_qpgmnam for pgmf\_pgmnam. The latter contains a different value depending on the target platform selected in the "Generation options" tab for the program definition. For example, if the program name is "custmnt" and the target platform is set to "iSeries RPG ILE Generator", the variable pgmf\_qpgmnam will contain the string `custmnt.pgm`. If the target platform is "Java Servlets for WebSphere" the variable will contain `custmnt`. Making this change ensures that you can use the same program definition to generate either RPG or Java code.

Performing this step manually would be very time consuming, so the WebSmart version 3 IDE is capable of making the changes for you. There may be situations where you want to override this automatic translation process, so the IDE presents the dialog shown in figure 11 the first time you open a program definition that was created using WebSmart version 2:

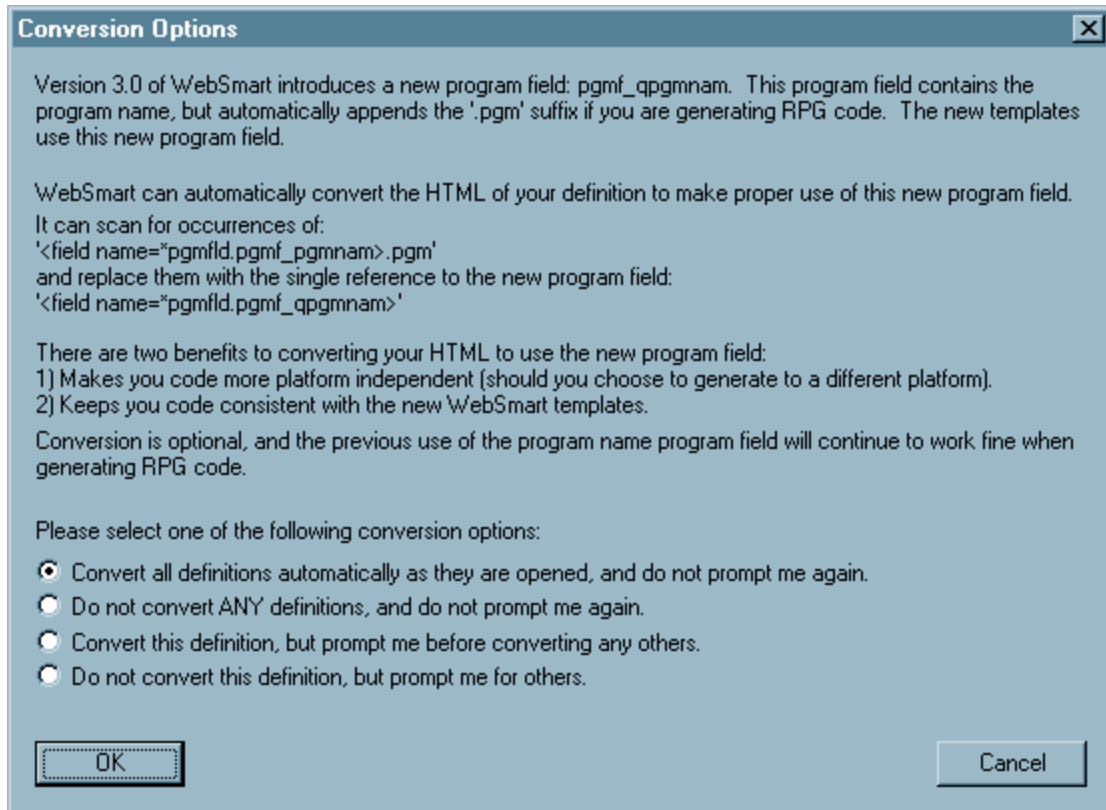


Figure 11: The conversion options dialog

This solution is satisfactory if the program being referenced in the URL is the program that is currently executing. However, there may be times when you want to create links to other programs. In these cases you may not know whether the target program is a CGI script or a Servlet at design time, but you do know that it is using the same target platform as the program that is currently executing. This will be the case when you have a set of program definitions, and you want the entire set to run either on an iSeries as RPG CGI scripts, or as Servlets on an iSeries or other execution server.

In this case you would use the pgmf\_platform variable to determine which execution platform is being used, and build all program names accordingly. For example, a customer maintenance system may consist of three program definitions: custlist, custupd and custdsp which list, update and display customers respectively. The main program is custlist, and the other two programs are called by selecting the links that are created by custlist. These links may be constructed either in HTML segments, or in PML functions. Placing the following code in the main() function of the custlist program, before any wrtseg() or runtask() calls, creates variables corresponding to the two called programs, and assigns values to them in a platform independent manner:

```

crtfld(custupd, 20, "A", 0, "Customer update program name");
crtfld(custdsp, 20, "A", 0, "Customer display program name");

if(pgmf_platform == "servlets")
{
    custupd = "custupd";
    custdsp = "custdsp";
}
else
{
    if(pgmf_platform == "RPG400")
    {
        custupd = "custupd.pgm";
        custdsp = "custdsp.pgm";
    }
}

```

Notice that the `pgmf_platform` variable has the value “servlets” if the generated program is executing as a servlet, and “RPG400” if the program is executing as an RPG CGI script. After this code has executed, a link can be built to run the default task of the `custupd` program in an HTML segment as follows:

```
<a href="<field name=*WRKFLDS.custupd>?TASK=default">...
```

A link can be constructed to redirect to the default task of the `custdsp` program in a PML function as follows:

```
url = append(custdsp, "?TASK=default");
```

Both of these links will access the called program using the correct syntax for the target platform. These examples illustrate one of many potential uses of the `pgmf_platform` variable. The same approach can be used to isolate any code that should only be executed on one of the available platforms.

## The WebSmart JSE WAS

The following two sections describe the structure of the Java libraries used by WebSmart JSE. A good understanding of the Java programming language is required to fully understand these sections.

The WAS for WebSmart JSE takes the form of a class library consisting of two packages:

1. `com.esdi.websmart`
2. `com.esdi.client`

The `websmart` package contains code produced by ExcelSystems, the `client` package contains code produced, or modified, by you. The `com.esdi.client` package is discussed in the following section. These packages consist of a set of uncompressed `.class` files beneath the IFS directory `/esdi/websmart/lib`. The class locations follow

the Java package naming convention of using a domain name with the order of its components reversed. The dots between the components of the package name correspond with the '/' characters in the directory path that references the package, e.g., the websmart package that ships with the initial product release is contained in the directory /esdi/websmart/lib/com/esdi/websmart.

The websmart package is equivalent to the service program used by the ILE RPG version of WebSmart. It contains the following classes:

<b>Class Name</b>	<b>Description</b>
WebSmart	Contains all WebSmart functions. Extends Java's HttpServlet class to provide handling for HTTP requests. Each request invokes a separate thread that executes the code for the class. The WSUser class, which is described in the next section, extends the WebSmart class, and all WebSmart JSE servlets extend the WSUser class. This gives WebSmart JSE servlets access to all methods in the WebSmart class.
WebSmartException	Thrown whenever a WebSmart error occurs.
WSConnectionPool	Provides a JDBC connection pool to improve the performance of SQL Database access.
WSConst	Contains all constant definitions used by the package.
WSDate	The data type representing PML dates.
WSDB2File	Provides support for DB2 Record Level Access.
WSDecimal	The data type representing PML numeric variables.
WSExitException	Thrown whenever a WebSmart JSE servlet needs to exit immediately, e.g., by the execution of the "exit" keyword.
WSFHandle	Represents a PML file handle for stream file access.
WSGlobal	Contains the global PML variables and their values for each HTTP request.
WSKlist	A Key List implementation for DB2 Record Level Access.
WSSmurf	Represents the value of a smurf returned by <code>getsmurf()</code> .
WSString	The data type representing PML alpha fields.
WSStruct	The data type representing PML structures.
WSTime	The data type representing PML time variables.
WSUtil	Contains utility methods used by the package.
WSVariable	An Interface implemented by all WebSmart JSE data types.

These classes are accessible to all JSE servlets running on the iSeries as long as the directory /esdi/websmart/lib is present in the classpath of the servlet engine. See the section "WebSmart JSE Server Side Configuration" for information on configuring WebSphere and Tomcat to run JSE Servlets on the iSeries.

## **Product Updates**

New functionality and bug fixes for WebSmart are installed using product updates. When an update for the ILE RPG version of WebSmart is applied, a new version of the service program containing the bulk of the WebSmart WAS is often installed. Because it is not practical to maintain and test for backward compatibility between successive versions of the product, binding directory changes are made which cause all subsequently generated WebSmart programs to bind with the new version of the WAS. The previous version remains on the system, and all programs that were bound to that service program will continue to access it when they are run. This avoids the need to recompile all existing programs so that they access the new library, while ensuring that any changes made to the new library will not affect the operation of existing programs.

WebSmart JSE updates are subject to the same limitation. When a WebSmart JSE servlet is compiled, it becomes dependent on the class libraries that it references. If these class libraries are changed in any way, for instance by a subsequent update, incompatibilities may result. To avoid this, a different package is created for each update, and all existing packages remain on the system. Packages installed after the initial WebSmart JSE installation have the following naming convention:

`com.esdi.websmart999`

where 999 represents a three figure update number. These numbers start at 001 and are incremented for each update. The IDE that is installed on the development PC as part of the update will generate JSE servlet code that references the new package. Because the websmart package consumes only 150K of disk space, the preservation of previous versions of the WAS does not consume too many resources.

## **Extending the PML**

The WebSmart ILE RPG generator allows the definition of additional WebSmart functions by adding source code to the member UPARSER in the file XL\_WEBSPT/PW\_PSRC. The same kind of functionality is available in WebSmart JSE by modifying the WSUser class. The source code for this class, as installed with the first release of WebSmart JSE, can be found at `/esdi/websmart/lib/com/esdi/client/WSUser.java`. The default installation of this file contains a single method, `countrybx()`. This is one of the standard WebSmart functions, but it is included in this class so that you can customize the list of country names according to your requirements.

Note: For the remainder of this section the distinction between the terms “method” and “function” will be blurred from time to time, as PML functions are implemented using Java methods.

As mentioned in the previous section, all WebSmart JSE servlets extend the WSUser class. This means that any public or protected method in this class can be used by any WebSmart JSE servlet. The WSUser class extends the WebSmart class, which means

that any method in `WSUser` has access to any public or protected method or variable in the WebSmart class. This includes the methods that correspond with all WebSmart functions. You can read the javadoc documentation for all classes in the `websmart` package at:

<http://www.excelsystems.com/websmartjsepml.htm>

This documentation will be of great value if you need to extend the PML, as it provides detailed information about the inner workings of the WebSmart JSE WAS. Further information on extending the PML will be added to this page as it becomes available.

If you are unable to find a resolution to your problem using any of these resources, please call BCD technical support at 250-655-1766.

The most important issue to note when modifying the `WSUser` class is that the Servlet environment is a multi-threaded environment. This means that any instance variables that you define in your `WSUser` class may be accessed by other threads at any time. You should not declare any instance variables unless you control access to them using the `synchronized` keyword, or you are sure that multithreaded access to the data is safe. The best approach is to use local variables that are declared within your functions. This prevents other threads from accessing them.

Any user defined functions already implemented using the ILE RPG version of WebSmart will have to be rewritten in Java as part of the `WSUser` class, if they are to be used in WebSmart JSE servlets. Please refer to the section “Extending the PML” in the WebSmart reference guide for information on making changes to the `prototypes.pml` file on the development client. You will need to add a function prototype for each of your user defined functions to this file before the IDE will recognize them.

## Creating your own WebSmart JSE functions.

You must be able to code in Java to create your own WebSmart JSE functions. The best approach is to treat each new function as a black box. The only data that the function should be aware of are the input parameters, and the value that is returned. It should not need to access any variables in the calling program. The input parameters, and the return value, must be one of the WebSmart JSE data types or `Object`. Each data type, apart from `WSStruct`, is a wrapper for a native java object as shown in the following table:

Data type	Wrapped Class	Description
WSDate	java.lang.Date	The data type representing PML dates.
WSDecimal	java.math.BigDecimal	The data type representing PML numeric fields.
WSString	Java.lang.String	The data type representing PML alpha fields.
WSTime	java.lang.Date	The data type representing PML time variables

These classes are mutable, i.e., their values can be changed by assigning a new instance of the wrapped object to them using the `set()` method. This is necessary in order to support the functionality of variable parameters. Hence, a function that you create can return values in two ways: by changing the value of one of the parameters passed to it, or by creating a single new object of the return type and returning it to the calling program. You can use the package documentation on our Web site to determine how to make use of these data types to your best advantage.

There are some special issues to bear in mind when passing alpha fields as parameters when writing your own functions. You may want to pass an alpha constant as a parameter, in which case the correct Java object type is `java.lang.String`. You may also want to pass the value of a WebSmart JSE alpha variable, or the alpha return value of another function, in the same parameter position, in which case the correct object type is `com.esdi.websmart.WSString`. This would normally cause compilation errors, because each parameter of a function needs to have a single type. Take the following simple function:

```
public void wrth1(String value)
{
    wrthtml("<h1>" + value + "</h1>");
}
```

If this were called in the Servlet as follows: `wrth1("Hello World")`, the servlet code would compile, but if it were called as follows: `wrth1(getparm("message"))`, there would be a compilation error. This is because the `getparm()` function returns a `WSString` object, and not a `String` object. You can approach this in two ways:

1. Define two versions of the `wrth1()` function, one that accepts a `String` object as its argument, and another that accepts a `WSString` object. This would involve duplication of code, which could lead to bugs as the programmer maintaining the code would have to remember to apply modifications to both functions.
2. Create a single version of the method that accepts an `Object` as its argument. Because `String` and `WSString` both extend `Object` you can create a new version of `wrth1()` as shown below:

```
public void wrth1(Object value)
{
    wrthtml("<h1>" + value.toString() + "</h1>");
}
```

This has the added advantage that you can pass any WebSmart object type as the argument and it will always display the String representation of the variable's value. This is because all WebSmart object types provide a `toString()` method.

## Using the **WSStruct** class

`WSStruct` is different from the other four WebSmart JSE data types because there is no corresponding object type in the Java language, and because it represents data in iSeries format. `WSStruct` contains two methods of note:

1. `convertToBytes()` – returns the values of all variables associated with the structure as an array of EBCDIC encoded bytes. This method is used when storing structure values in a transaction or smurf record (using the `transaction` or `smurf` functions) and when calling an iSeries program that accepts a structure as its argument. This ensures that a JSE servlet can call any program on the iSeries, and the data will be passed to and from the called program transparently. It also means that smurfs and transactions that contain structures can be shared between WebSmart JSE Servlets and RPG CGI scripts.
2. `setStructVals()` – allows you to set the values of all variables associated with the structure by passing an array of EBCDIC encoded bytes as the method's single argument.

Because the customized function you are creating won't know which fields are contained in the structure that is passed to it, all it can do is pass it on to another function or use its byte value as an argument to a program call.

## Using the **IBM AS/400 Toolbox for Java**

If your customized functions need to interact with the iSeries in ways other than those provided by existing WebSmart functions, you can use the IBM AS/400 Toolbox for Java to provide additional functionality. The Toolbox class library can be found in the file `jt400.jar` which ships with update `websmart002` or higher. Earlier versions of WebSmart JSE required that you track down the location of this file on your system, as it could be in a number of IFS locations.

The Toolbox for Java provides a comprehensive list of functionality. The programmer's guide can be found at the following address:

<http://publib.boulder.ibm.com/series/v5r2/ic2924/index.htm?info/rzahh/page1.htm>

## Applying updates and the **WSUser** class.

The discussion in the previous section covered the application of updates to the WebSmart JSE WAS. This has an impact on the `WSUser` class, as `WSUser` extends the `WebSmart` class. If you apply an update which creates the `com.esdi.websmart002` package, your `WSUser` class must extend the class `com.esdi.websmart002`. This dependency is necessary if you want to have access to the product enhancements and bug fixes in the new package. The IDE that is installed on the development PC enforces this dependency in the generated code.

This is accomplished for each update by creating a new package called `com.esdi.client999` where 999 is the number of the update. The first three lines of the `WSUser.java` file for the first WebSmart JSE update are as follows:

1. `package com.esdi.client001;`
2. `import com.esdi.websmart001.*;`
3. `public class WSUser extends WebSmart`

The first line provides the name of the package in which the `WSUser.class` resides. This must match the “client” package name that is imported into all WebSmart JSE servlets generated by the IDE. The second line defines the package name that contains the `WebSmart` class, which is the superclass of `WSUser`. The third line defines the class `WSUser` as a subclass of `WebSmart`. The version of the `WebSmart` class that is extended by `WSUser` depends on the import statement on line 2. Each time an update is installed, the first two lines must be modified, and the location of the `WSUser.java` and `WSUser.class` files must be changed. This is accomplished by running the program `XL_WEBLIB/PW_WSUSRUP` which performs the following four steps:

1. It creates a new directory `/esdi/websmart/lib/com/esdi/client999` where 999 is the number of the update being applied.
2. It copies the `WSUser.java` file, which includes your custom functions, from the most recent update directory to the new directory.
3. It changes the import and package statements in the `WSUser.java` file to reflect the new update number.
4. It compiles the program to create the `WSUser.class` file in the new `client999` directory.

After this process has completed, the source file will be at the following location:

```
/esdi/websmart/lib/com/esdi/client999/WSUser.java
```

At this point the new `WSUser.class` file is dependent on the classes in the new `WAS` and will have access to all of the new functionality and bug fixes. It is possible that the code you have added to `WSUser.java` utilizes class libraries that you have written, or third party class libraries. In order to ensure that these are accessible when the `WSUser` code is compiled, and when it runs, please place the classes for these packages under the directory `/esdi/websmart/lib` prior to applying the update. If those classes are contained in a `.jar` file, please use the `jar` program that ships with `QShell` to extract them into the above directory.

If you received Java compilation errors during the update, or the `WSUser.class` is not in the update directory after the update is applied, you can address the problem and recompile the program by entering the following command in an iSeries session:

CALL XL\_WEBLIB/PW\_WSUSRUP '999'

where 999 is the number of the update that is being installed. If the new update already exists, the program merely recompiles the existing WSUser.java file. We advise you to use this program to recompile your WSUser.java file whenever this is required. This avoids the need to setup the CLASSPATH variable required by the QSHELL environment, which can be very lengthy. If you continue to receive compilation errors and you are unable to resolve the problem, please call BCD technical support.

## Class reloading

After making changes to a class, either by recompiling an existing Java source file, or by installing a new version of an existing file, the file must be reloaded by the Servlet engine before the new code can be used. This is the case when an update has been applied to the WebSmart JSE class libraries, so after applying an update you must restart your servlet engine. A description of the issues behind this requirement is given in this section.

A Java VM will cache, in main memory, the classes that have been loaded as a result of a set of program invocations. This avoids the need to load a class each time it is accessed, which significantly improves performance. In a perfect world, the VM would only have to load each class once, and would continue to use that class until the process terminated. However, if the class file is modified it needs to be reloaded for the changes to be reflected. The process of choosing which classes, if any, to reload is not a trivial issue, but the alternative is unacceptable. Imagine a development environment in which you had to reboot your computer every time you made a change to your program. Without a reboot the computer would continue to use the old version of the program. This is the equivalent of stopping and restarting your servlet engine whenever you make a code change. On the iSeries the process of restarting WebSphere or Tomcat can easily take as long as rebooting a PC.

To avoid this behaviour, the class loader must compare the modified date and time of the class which resides in the memory cache with that of the class file on the storage device. This must be done each time the class is accessed, which in itself incurs a performance overhead. For this reason most servlet engines allow you to choose whether the modified date and time of the Servlet's class file should be checked each time it is accessed.

### ***Class reloading on the iSeries***

WebSphere and Tomcat on the iSeries take two different approaches to this issue. WebSphere maintains two separate class paths: the reloadable classpath and the non-reloadable classpath. Any classes placed in IFS directories or Jar files in the reloadable classpath are checked for modifications on a regular basis, although this is not each time they are accessed. You may find that you have to press the reload button on your browser up to four times before changes to your WebSmart JSE servlet are reflected. Classes placed in the non-reloadable classpath are never checked

for modifications, and changes will only be reflected when WebSphere is restarted. This is why we recommend you generate all of your WebSmart JSE servlets to the following directory:

```
/QIBM/ProdData/WebAsAdv/servlets
```

until they are released to production, as this directory is in WebSphere's reloadable classpath.

Tomcat supports a "Reloadable" flag for each servlet, which indicates that the class file should be monitored for updates on a regular basis. (See step 10 in the Apache/Tomcat configuration section of this guide for the context in which this flag is used.) Unfortunately this feature is described as "experimental" in the Tomcat documentation, and is not guaranteed to work. This means that you may have to restart the Tomcat worker each time you make a change to each of your servlets. This is invasive enough when you use Tomcat as a production Servlet engine, but makes it unsuitable as a development platform. If you do not have a WebSphere instance on your iSeries, we advise you to download the WebSmart JSE runtime package from our Web site. This contains a Tomcat installation for the Windows platform that can be used for development, as it does not suffer from the class reloading problem.

## Library list issues

Due to the dependence of WebSmart JSE on the AS/400 Toolbox for Java, it is not possible to provide the same kind of library list support that is available in RPG. Files that are opened using the record level access functions do not use library lists at all, and the library in which each file resides is hard coded at design time. This is not the case for SQL access to DB2 data on the iSeries. The `sqlconnect()` function uses the library list identified in the call to the `setlibl()` function. Each WebSmart JSE program contains the following call to the `setlibl()` function by default:

```
setlibl(" *FILES" );
```

You can modify the library list name to a library list of your choice, and the change will be reflected in SQL access to DB2 data. You can find out how to create and modify library lists, using the EWRKLIBL command, by reading the "Creating and working with library list objects" section in Section 4 of the WebSmart Reference Guide.

## Performance issues

The ILE RPG version of WebSmart creates programs that run as CGI jobs in the QHTTPSVR subsystem. From the Java perspective, the CGI approach is viewed as inefficient as it requires the initiation of a new process each time a CGI program is executed. The creation of processes is a resource intensive task compared with the multi-threaded approach taken by Java. On the iSeries, IBM has largely avoided the process creation bottleneck by maintaining a pool of active processes for each Web

server instance in the QHTTPSVR subsystem. This allows a CGI request to be executed in a process that has already been created and initialized. The pool of processes will grow and contract depending on the number of requests the server receives. The expansion and contraction of the pool does involve a process creation and tear down overhead.

Despite these issues, the execution of Java Servlets on the iSeries does not provide a performance advantage over CGI execution, at least not when the system is receiving a small number of requests. Java's multi-threaded approach may provide better performance when many requests are being received, but this can only be determined by evaluating the performance of an individual application under load, and comparing the two execution platforms. The Apache foundation offers a free performance monitoring tool called JMeter which can be downloaded from the following location:

<http://jakarta.apache.org/jmeter/>

Although this program is written in Java, it can be used to measure HTTP server performance regardless of the kind of content served. All it needs to know is the URL of the resource that is to be accessed, and how many threads of execution should be used to send requests to the server. You may find this tool to be a satisfactory performance monitoring solution when determining the relative performance of WebSmart JSE servlets versus RPG CGI scripts under various load conditions.

## WebSmart JSE Advantages

The two main advantages of WebSmart JSE are:

1. Portability
2. Accessibility to non-iSeries data

These two issues are dealt with in the following sections.

### ***Porting WebSmart JSE Servlets to non-iSeries systems***

Although an iSeries server is required in order to compile WebSmart JSE programs, you can move the resulting executable files (the .class files) to any system that has a Web Application server which supports the servlet standard. Unfortunately, as we do not know which Web server/Web application server combination you will be using, we cannot provide configuration information for your chosen execution platform.

In order to execute the servlet you will also need to copy the class libraries and support files that it requires. The simplest way to accomplish this is by copying the IFS directory /esdi/websmart and all of its contents to the alternate system. This directory also contains all of the graphics, include files, style sheets and JavaScript source files used by the WebSmart templates and any programs that were created using those templates. Ensure that the directory "websmart" is placed under the directory "esdi" which is at the root level of the target system's directory hierarchy. You will need to configure the classpath of your servlet engine to include the jar files

contained in the directory `/esdi/websmart/lib`, or place these files in a special deployment directory, if your servlet engine supports this. The directory `/esdi/websmart/lib` should also be added to the classpath, as this provides access to the `WSUser` class, which is the superclass of all WebSmart JSE servlets.

Certain WebSmart functions require access to an iSeries server even when they are executing on a non-iSeries machine. These functions use the AS/400 Toolbox for Java, regardless of whether the servlet is running on the iSeries or not. The Toolbox classes are contained in the file `jt400.jar` which must be copied from the iSeries to the non-iSeries system. The location of this file on the iSeries varies depending on which version of OS/400 you are using. Under V4R5 it is located in the following directory:

```
/QIBM/ProdData/HTTP/Public/jt400/Lib
```

You can place the file in a location of your choice on the non-iSeries system, after which you must add it to the classpath of the servlet engine.

The functions that are dependent on access to an iSeries system include:

1. Transaction functions
2. Validation list functions
3. Smurf functions
4. Program Calls and some other server side functions
5. Record level access functions

When these functions are executing on the iSeries, the user profile used, when establishing a connection with the server using the Toolbox, is inherited from the process in which the servlet is running. As we cannot rely on consistent user profile handling across all Java execution environments, an alternate method has been supplied by which the system name, user id and password can be provided to the servlet. The file `/esdi/websmart/conf/login.properties` is a java properties file (analogous to a Windows ini file) which contains the following text:

```
default.system=<iSeries IP address or domain name>  
default.userId=<user ID>  
default.password=<password>
```

You can create this file in the directory indicated using any text editing tool, as properties files consist of plain ASCII text. Just substitute the correct values in place of the text between angle brackets shown above. As this file contains the password of an AS400 user profile, you should use the operating system's access control features to protect it.

Finally, you must add a URL mapping from the following address:

```
http://<domain or IP>:<port>/websmart/
```

to the directory /esdi/websmart. Please refer to your Web server's documentation for information on how to accomplish this.

## Accessing non-iSeries data

WebSmart JSE servlets can access AS/400 data in the normal manner using the record level access functions. They can also access data using the SQL functions, but while SQL access may appear to work in a similar manner using Java and ILE RPG, there are some major underlying differences.

RPG on the iSeries uses the SQL/400 syntax, while WebSmart JSE servlets use the Java DataBase Connectivity (JDBC) standard (which implements SQL-92). We have made every effort to ensure that in most cases the SQL/400 statements contained in existing WebSmart definitions will work in both RPG and Java, but sometimes functionality may have been used that is only available under SQL-92. In these cases the programs will have to be modified to work with JDBC.

The WebSmart JSE SQL functions emulate the use of SQL/400 cursors by mapping JDBC result sets to a cursor name, and then handling "fetch" requests on the result set using the embedded SQL/400 syntax. For example, the following statements:

```
selectStmt = "SELECT CCO#, CCUST#, CCUSTN FROM CUSTMST";  
sqlquery(selectStmt, "MC");
```

would execute the select statement, and associate the resulting JDBC result set with the cursor name "MC". Data can then be retrieved from the result set using "FETCH NEXT" or "FETCH RELATIVE" statements that reference the cursor name, for example:

```
sqlexec("FETCH NEXT FROM MC INTO :CCO#, :CCUST#, :CCUSTN");
```

The names of the variables that are to receive the data from the result set are presented using the SQL/400 host variable syntax, i.e., a comma delimited list of variable names, each preceded by a colon (":"). The first argument to `sqlquery()` is passed as-is to the JDBC driver. The string argument to `sqlexec()` is processed to translate between embedded SQL/400 syntax, and JDBC syntax. The main goal of this translation is the assignment of values from a JDBC result set to the corresponding host variables (in the case of the FETCH statement), and the substitution of the host variable values for the variable names (in the case of the UPDATE statement). The WebSmart JSE `sqlexec()` function currently supports the FETCH, UPDATE, DELETE and CLOSE statements. The argument to `sqlexec()` is converted to lower case before processing.

## Connecting to a JDBC data source

When deploying WebSmart programs as ILE RPG CGI scripts, you have the option of using the local iSeries DB2 database or a remote iSeries DB2 database for SQL access. WebSmart JSE gives you the opportunity to connect to any database for

which a corresponding JDBC driver is available. This includes such systems as Microsoft Access, Microsoft SQL Server, MySQL and Oracle. Before accessing any data from a remote system you must first make a connection to that system using either the `sqlconnect()` function or the `jdbconnect()` function.

Using WebSmart JSE you can connect to a DB2 database on an iSeries server using the `sqlconnect()` function, which has the following prototype:

```
VOID sqlconnect(ALPHA rdn, ALPHAFIELD user, ALPHAFIELD password);
```

In this case, the `rdn` variable is interpreted as the domain name or IP address of the remote system. In order to make a JDBC connection to another system, the Java servlet must have access to a JDBC driver. This is a set of Java classes which normally reside in a jar file that is in the classpath of the servlet engine.

`sqlconnect()` allows you to connect to an iSeries machine using the Toolbox JDBC driver which is located in the file `jt400.jar`. To make this available to your non-iSeries server, locate the file `jt400.jar` under the `/QIBM IFS` directory on the iSeries, and copy it to a location on the target system that is in the classpath of the servlet engine. The location of this file varies depending on which version of OS/400 you are using. Under V4R5 it is located in the following two directories:

```
/QIBM/ProdData/HTTP/Public/jt400/Lib  
/QIBM/ProdData/HTTPA/Admin/Pgm
```

It will only be present in the second directory if the Apache web server has been installed on the iSeries. When you connect to a system in this way, the class `com.ibm.as400.access.AS400JDBCDriver` will be loaded, and a pool containing four JDBC connections to the target system will be created. This connection pool vastly improves the performance of any JDBC access to a remote database, because the creation of a JDBC connection is the most time consuming part of the data access sequence. The four data base connections are kept open until the servlet terminates.

The `jdbconnect()` function has the following prototype:

```
VOID jdbconnect(ALPHA driver, ALPHA jdbcurl);
```

A call to this function is required when you are connecting to a non-iSeries database server. The `sqlconnect()` function is able to assume two things based on the fact that it is connecting to an iSeries machine using the Toolbox JDBC driver:

1. The class name of the JDBC driver
2. The format of the JDBC URL

A JDBC URL is analogous to a URL that you would use to access a web page using a web browser. It is generally the only piece of information that a JDBC driver needs in order to establish a connection. Each JDBC driver has a different class name, and may have a totally unique URL format. The following example illustrates how a

connection can be established between a WebSmart JSE servlet, and an Oracle database:

```
url = "jdbc:oracle:thin:SYSTEM/manager@192.168.0.96:1521:JSESID";  
sqlconnect("oracle.jdbc.driver.OracleDriver", url);
```

The url syntax expected by the Oracle JDBC driver is unconventional. It has the following format:

```
<protocol>:<user>/<password>@<system>:<port>:<database identifier>
```

In order for the `jdbcconnect()` call to work, the oracle thin JDBC driver must be in the classpath of the servlet engine. Before you can install it you first have to obtain it. The driver may ship with the product, but if not, try going to the Web site belonging to the database vendor and searching for the term "JDBC". You can also search the Internet using a similar search term, as sometimes third party software developers create JDBC drivers for other vendor's products. Most JDBC drivers are available free of charge, or on a trial basis. Normally the JDBC driver is in a jar file that must be added to the classpath of the servlet engine to make it available to the `jdbcconnect()` function.

## Repository issues

WebSmart JSE version 3 does not support the downloading of JDBC table formats into the repository. This means that you will have to define work fields that correspond to each field you will be referencing in each table. Each work field must have the correct data type and attributes when it is used to read values from the table and save values to the table using the SQL “select” or “update” statements, respectively. For example, the following declarations are required in order to support the SQL calls that follow:

```
crtfld(compnbr, 3, "N", 0, "Company Number");
crtfld(custnbr, 6, "N", 0, "Customer Number");
crtfld(custname, 30, "N", 0, "Customer Name");

selectStmt = "SELECT CCO#, CCUST#, CCUSTN FROM CUSTMST";
sqlquery(selectStmt, "MC");
sqlexec("FETCH NEXT FROM MC INTO :COMPNBR, :CUSTNBR, :CUSTNAME");
```

This restriction will be addressed in a later release of the product.

## Using pgmf\_lasterror and pgmf\_lastertrt

These two program fields contain the number of the most recent program error that occurred, and a textual description of the error. The iSeries has an established method of error reporting which is incompatible with the Java exception handling approach. This means that the Java version of a WebSmart function is not aware of which error number will be returned from the corresponding ILE version of the function under all circumstances. If you have code which uses either of these variables and you need to generate a Java servlet version of the program, please review the behavior of the program.

# WebSmart JSE Server Side Configuration

## *WebSphere Configuration*

This section assumes that you have WebSphere 3.5 or above installed and running on your iSeries, and that you are able to run the example servlets. It is also assumed that you have a workstation configured to run the WebSphere Administrative Console, and that you are able to use this to start and stop server instances and change their attributes. The hints given here refer to the use of WebSphere 3.5.x, which is the last version of the product to be distributed free of charge. WebSphere 3.5.x uses the Java 2 SDK (version 1.2), and this is the release that WebSmart JSE is designed to work with.

## The Servlet Environment

In this section we will illustrate the simplest approach to running servlets under the WebSphere Application Server. WebSphere allows you to configure multiple application server instances. Each instance is associated with a single servlet engine. WebSphere ships with one server instance by default, the “Default Server”, under which you can run WebSmart JSE servlets without making many configuration changes. The documentation provided with the Administrative console describes how to configure additional server instances, how to set up and control individual servlets and how to add WebSphere mediated user authentication and authorization for a servlet, among other things. None of these steps are required in order to run WebSmart JSE servlets under WebSphere.

## User Profiles

WebSphere runs under the QEJBSBS subsystem. Each process in this subsystem executes using the QEJB user profile by default. This profile is created during the WebSphere installation process, and it has no password. In order to use the IBM AS/400 Toolbox for Java, which supports such features as DB2 record level access and OS/400 program calls, you must assign a password to this user profile.

## The Class Path

You must configure classpath associated with the application server prior to running WebSmart JSE Servlets. This gives the Servlets access to the class libraries that they require to run. The classpath of the application server is set using the “Command line arguments” field for the “Default Server” instance. You can set the value of this field as follows:

1. Open the WebSphere Administrative Console and expand the node corresponding to your iSeries. This is normally identified using the domain name of the system.
2. Click on the “Default Server” branch that appears under the system node
3. A tabbed dialog will appear to the right with the title “Application Server: Default Server”, and the “General” tab should be selected by default. If it is not, click on the “General” tab.

4. Locate the “Command line arguments” field, which is the sixth field on this tab
5. It is possible that a classpath has already been configured for the server, if so you will add entries to it. If there is no classpath command line argument, then enter the text “-classpath” following any text that is already there. The classpath attribute must be separated from any other text by at least one space.
6. Following the “-classpath” attribute, add an entry for each of the following libraries (each library should be separated by a colon ‘:’, and there must be no spaces between the library entries):

Library path	Description
/QIBM/ProdData/java400/jt400ntv.jar	Allows servlets to make connections with the local iSeries, using the IBM AS/400 Toolbox for Java, without requiring a user name and password.
/esdi/websmart/lib	Gives access to the WSUser class. The source code for this class can be modified by users to provide functionality that the WebSmart JSE class library does not contain. The source code is provided in the file: /esdi/websmart/lib/com/esdi/client/WSUser.java.
/esdi/websmart/lib/cos.jar	Library containing HTTP file upload routines. These allow a user to transfer a file from their local system to the server using a Web browser and the HyperText Transfer Protocol.
/esdi/websmart/lib/WebSmart.jar	The WebSmart JSE class library. Contains classes that define the data types used by the product, and all WebSmart functions.
/esdi/websmart/lib/bcprov-jdk12-113.jar	The cryptography library that provides the DES algorithm used by the encrypt() and decrypt() functions.

The libraries above do not constitute the entire classpath used by the application server, but they are added to the start of the classpath. Other system libraries and directories are appended to these entries by WebSphere. The complete classpath string, that you must enter, will appear as follows:

```
-classpath /QIBM/ProdData/java400/jt400ntv.jar:/esdi/websmart/lib:
/esdi/websmart/lib/cos.jar:/esdi/websmart/lib/WebSmart.jar:
/esdi/websmart/lib/bcprov-jdk12-113.jar
```

The line breaks that are necessary when printing this information here are not part of the classpath; it should be entered on a single line.

7. Click the Apply button to save the changes
8. The application server instance must be restarted for the changes to take effect. To do this, click on the “Default Server” branch and click the “stop” button on the

tool bar. After receiving a message informing you that the server has been stopped, click on the “start” button. After you have received a message saying that the application server has been successfully started the environment for WebSmart JSE servlets will have been fully configured.

## File locations

As discussed in a previous section, we advise you to generate your WebSmart JSE Servlets to the following IFS directory if you are developing Servlets under WebSphere:

```
/QIBM/ProdData/WebAsAdv/servlets
```

You can put your servlets in any directory as long as that directory is in the servlet engine’s classpath, but changes to the class will probably not be reflected until WebSphere is restarted. The location above is part of WebSphere’s reloadable classpath, so changes made to classes in this location will take effect immediately.

## Apache/Tomcat Configuration

In the previous section we saw how to configure the IBM HTTP Server (Original), and the WebSphere Servlet engine, to process WebSmart JSE Servlet requests. This section describes how to configure the IBM HTTP Server (Powered by Apache) and the Tomcat Servlet engine for the same purpose.

This section assumes that you have the Apache Web server installed and running on your iSeries. As Apache does not ship with a default Servlet engine configuration, we will provide more explicit instructions on the configuration of the server environment than those given for WebSphere. We will still cover the most basic configuration possible under which WebSmart JSE servlets can be run, but there are more issues to deal with in the context of Apache. All the instructions you need to produce a basic Apache/Tomcat configuration can be found at the following URL:

```
http://publib.boulder.ibm.com/pubs/html/iseriess\_http/v4r5/info/rzaie/rzaietomcatopservlets.htm
```

This describes how to configure an out-of-process Tomcat instance. Tomcat can run in the same process as the Apache Web server, but we do not encourage this because of the user profile issues discussed later in this section. The above document, with WebSmart JSE specific annotations, has been reproduced here. The annotations cover additional information that you may find useful, and deal with some of the pitfalls involved in the configuration process.

## Starting the configuration and administration forms

Unlike the WebSphere Application Server, Tomcat has no administrative console. All configuration is accomplished using a Web based interface that is accessed using the following URL:

`http://<server IP address or domain>:2001/`

These pages are accessed by clicking on the “IBM HTTP Server for AS/400” link on the page that is displayed as a result of accessing the URL above. There are some differences between OS/400 V4R5 and later versions of the OS at this point. You can access the configuration pages under V5R1 or later by selecting “Configuration and Administration” link. V4R5 also includes a link with the same text, but the one you need to use for Apache is as follows:

NEW! Updated Configuration and Administration for HTTP servers (original and powered by Apache) NEW!

You are now in a position to start following the configuration instructions.

NOTE: Before attempting any of these configuration steps you should ensure that the Apache Web server instance that you will be using to provide access to WebSmart JSE servlets has been stopped. This is because the server needs to be restarted in order to load the configuration changes you make here. Once you have accessed the configuration pages, you can stop the server by following these steps:

1. Click the Administration tab.
2. Click Manage HTTP servers under the General Server Administration heading.
3. Select your server configuration from the manage HTTP Servers table.
4. Click Stop.

## User Profiles

When using the configuration pages, you must always sign in using the same user profile. You will not be able to view or edit any Apache or Tomcat server settings that you previously defined unless you sign in to the configuration pages using the same user profile that was used to create them. You must make authority changes to two objects before you will be allowed to save the changes you have made, the profile must have:

- \*JOBCTL authority
- \*ALL authority to the file QUSRSYS/QATMHASFT
- \*CHANGE authority to the library object QUSRSYS

The main reason that we encourage you to use an out-of-process server instance is that you then have complete and easy control over the user profile under which the server executes. An active WebSmart JSE servlet will use the servlet engine’s user profile when making connections to the iSeries for program calls and record level access etc. Using a separate user profile in this way allows you to configure the authorities associated with the profile, so that the executing servlet can access the data that it needs. For this reason you will not accept the default for “Server userid” (which is QTMHHTTP) as suggested in the configuration instructions, but will enter a profile of your choice.

## Configuring Apache to send requests to a Tomcat worker instance

1. Start the Configuration and Administration forms.
2. Click the Configuration tab.
3. Select your HTTP Server (powered by Apache) configuration from the server menu.
4. Click your server's global settings.
5. Click ASF Tomcat Setup task under the Dynamic Content heading.
6. Do the following:

Enable servlets for this HTTP Server	Select to enable servlets.
Workers definition file	Accept the default workers definition file.

7. Click Next.
8. Do the following:

Enable an "in-process" servlet engine	Do not select this option.
Enable "out-of-process" servlet engine connections	Select to enable out-of-process.

9. Click Add.
10. Do the following:

NOTE: The Web server needs to know which URLs (or mount points) are to be mapped to the servlet engine, and what the parameters of the servlet engine are. You set this up by configuring a "worker" (or servlet engine instance), to which you give a name, an IP address and a port number. The port number must not be used by any other service on the host system, and must certainly not be the port of the Web server that you are linking with the servlet engine. The port number is the port that your out-of-process Tomcat worker will be listening to for requests from the Apache web server. The mount points are defined last. Any requests received by the web server that match any of the mount points will be handed to the servlet engine for processing. Mount points can be constants, e.g., /calc, or they can be patterns, e.g., /calc/\*. The worker configuration itself does not exist until you have configured it using the ASF Tomcat server configuration web interface (described in the next section).

Worker name	Enter the name of the out-of-process ASF Tomcat server. For example, mytomcat
Worker type	Select Binary (AJP13).
Hostname:Port	Enter localhost and a port number if your out-of-process ASF Tomcat server is on the same system. For example, localhost:8009. Enter the name of the remote host and the port number if your out-of-process ASF Tomcat server is not on your local iSeries. For example, IBMiSeries:8009.

11. Click Continue.
12. Click Next.
13. Click Add.
14. Do the following

URL (Mount point)	Enter the name of the URL mount point. For example, /calc. This will be the portion of the URL following the host address and port number, that will be used to access your first WebSmart JSE servlet.
ASF Tomcat worker	Accept the default value.

15. Click Next.
16. Click Finish.
17. Click OK.

## Configuring the Tomcat worker

1. Click the ASF Tomcat tab.
2. Click Create ASF Tomcat Server under Engine Administration.
3. Enter your worker name in the ASF Tomcat server name field. This is the name of the ASF Tomcat server. For example, mytomcat.
4. Click Next.
5. Do the following:

Server userid	Do not accept the default value (QTMHHTTP). Enter the user profile under which you would like your WebSmart JSE servlets to run, as described under the heading “User Profiles”.
Java version (JDK)	Accept the default value.
ASF Tomcat home	Accept the default value.
Java classpath entries	You will need to add several classpath entries to give the WebSmart JSE servlet access to the Java class libraries that it needs in order to execute. These are shown below.

Library path	Description
/QIBM/ProdData/java400/jt400ntv.jar	Allows servlets to make connections with the local iSeries, using the IBM AS/400 Toolbox for Java, without requiring a user name and password.
/esdi/websmart/lib	Gives access to the WSUser class. The source code for this class can be modified by users to provide functionality that the WebSmart JSE class library does not contain. The source code is provided in the file:

	/esdi/websmart/lib/com/esdi/client/WSUser.java.
/esdi/websmart/lib/cos.jar	Library containing HTTP file upload routines. These allow a user to transfer a file from their local system to the server using a Web browser and the HyperText Transfer Protocol.
/esdi/websmart/lib/WebSmart.jar	The WebSmart JSE class library, or WAS. Contains classes that define the data types used by the product, and all WebSmart functions.
/esdi/websmart/lib/bcprov-jdk12-113.jar	The cryptography library that provides the DES algorithm used by the encrypt() and decrypt() functions.

6. Click Next.
7. Do the following:

IP address	Accept the default value.
Port	Enter the port number you provided when you configured the worker process in the Apache worker configuration section
Server type	Select Binary (AJP13).

8. Click Next.
9. Click Add.
10. Do the following:

NOTE: You will need to add an application context for each servlet that will be running under the servlet engine instance. To do this you need to provide a URL path and an Application base directory. The servlet engine will run under the IFS directory:

/ASFTomcat/<servlet engine name>

beneath this will be a base directory for each web application. Normally the base directory consists of the path “webapps/<web application name>”. Each web application has the following directory hierarchy (as an extension of the servlet engine’s base directory shown above):

webapps/<web application name>/WEB-INF/classes

The classes directory is where you put your class files. In the WebSmart JSE context, this will be the value of the “Path to servlet” field in the “Program attributes” tab of the IDE.

URL path	Enter the name of your URL path For example, /tstcfg.
----------	---

Application base directory	Enter the name of your application base directory. For example, <code>webapps/tstcfg</code> .
Reloadable	Select this if you want to be able to change your servlet and have it automatically loaded when it is next accessed. Select this check box if the servlet is under development, as you will be reloading it every time you recompile.

11. Click Continue.
12. Click Configure.
13. Click Add.
14. Do the following:

NOTE/REVIEW: Clicking the add button allows you to associate a class file with the base directory for the web application. It also allows you to associate a URL pattern with that class file. The best way to describe the relationship between the URL entered at the browser, and the URL pattern associated with a class file is by providing an example:

To run a web application using the URL `http://192.168.0.100:3002/tstcfg/cfg` you need to perform the following steps:

1. Set up an application context as shown in step 10. This instructs the servlet engine to associate any URLs passed from the web server which start with the string `"/tstcfg"` with the base directory `"webapps/tstcfg"`. All classes for this application must be in the directory `webapps/tstcfg/WEB-INF/classes`. The http server must be configured to support a worker which will catch all URLs matching the pattern `/tstcfg/*`, and which will pass them to the servlet engine. The wild card on the end is there to ensure that the string `"/tstcfg/cfg"` constitutes a match, and that the request is processed properly.
2. Click the configure button next to the web application entry to associate a class with the URL (this is step 12). You enter the class name and a URL pattern. The URL pattern is the portion of the URL following the part that matches the web application. For example, the web server is set up to pass all URLs matching `/tstcfg/*` to the servlet engine. The engine matches the string `"/tstcfg"` against all known web applications and finds one that matches. It then looks for an entry for a class with a URL pattern that matches the remainder of the URL entered, in this case `"/cfg"`. If it finds a match it will load and run the class file indicated.

Servlet classname	Enter the name of your servlet classname. For example, <code>CalculatorExample</code> (without the <code>.class</code> extension).
URL patterns	Enter the portion of the servlet's address that will follow the URL path for the web application, e.g., if the URL used to access the servlet will be <code>/tstcfg/cfg</code> , the URL pattern will be <code>/cfg</code> .
Startup load sequence	Accept the default.

15. Click OK.
16. Click Next.
17. Click Finish.
18. Click OK.

## Placing your servlet file in the correct location

At this point you can either copy an existing servlet .class file from another location into the IFS directory:

```
/ASFTomcat/<servlet engine name>/webapps/<web app name>/WEB-INF/classes
```

(in this case, the name of the class must be the same as that defined in step 14 above), or you can set the value of the “Path to servlet” field in the “Program attributes” tab of the WebSmart JSE IDE to the above path, and then generate the WebSmart definition. In this case the name of the program must be the same as the name of the class file defined in step 14 above. Remember also that in Java the names of classes are case dependent, so if you defined the class name `tstcfg.class` in step 14, and the class file is actually named `TstCfg.class`, an error will occur when attempting to load the class.

## Start the HTTP Server

1. Click the Administration tab.
2. Click Manage HTTP servers under the General Server Administration heading.
3. Select your server configuration from the manage HTTP Servers table.
4. Click Start.

## Start the out-of-process ASF Tomcat server

1. Click the ASF Tomcat tab.
2. Click Manage ASF Tomcat Servers under the Engine Administration heading.
3. Select your ASF Tomcat server.
4. Click Start.

After you start the Tomcat server, a process is created under the QSYSWRK subsystem which has the same name as the Tomcat instance you defined. Depending on the capabilities of your system this process may take several minutes to initialize. You can monitor its progress using the `WRKACTJOB` command. After the process has stopped consuming large amounts of CPU, you can attempt to run your servlet.

## Troubleshooting

### *DDM Problems*

If you get the error “A remote host refused an attempted connect operation” or “java.net.ConnectException: Connection refused: connect” this is probably because the DDM server is not running on your iSeries. You can start the DDM server by issuing the following command:

```
STRTCPSVR SERVER(*DDM)
```

### *Performance*

In order to improve performance, servlets perform time consuming operations as infrequently as possible, and preferably only once – at the time the servlet is invoked. This step may include the creation of threads, the establishment of database connections, the creation of AS/400 Toolbox connections for DB2 record level access, the retrieval of DB2 file formats and the compilation of the Java byte code into Power PC machine code instructions. This means that the first time you run a servlet after the application server has been restarted, or after the class file has been modified, it may take a few seconds before you receive the servlet’s output in your web browser window. Subsequent servlet invocations will be much faster.

### *Configuration changes*

WebSphere and Apache cache a lot of configuration information in order to improve performance. This means that an application server instance normally needs to be restarted when configuration changes are made. In the WebSphere context, restarting the application server instance does not always cause the configuration changes to take effect, and you have to restart the QEJBSBS subsystem. The subsystem can be stopped from the WebSphere Administrative console, but it can only be restarted from a command line. The following two commands will achieve a restart:

```
ENDSBS QEJBSBS OPTION(*IMMED)
```

```
STRSBS QEJB/QEJBSBS
```

Once the subsystem has restarted you can start the Administrative Console to ensure that the application server instance(s) are operating correctly. It is very important to confirm that the server instances are working properly if you have made configuration changes, but you are not sure if the changes you made were correct. You may assume that you have made a mistake when all that is needed is a full restart of the WebSphere subsystem.

Some configuration changes, e.g., the creation of virtual hosts, require that both the Web Server and the WebSphere subsystem are restarted. Sometimes the order in which these two components are restarted is important also. The rule of thumb is: if

you have followed the configuration instructions it is likely that the configuration is correct, and all that is required is to recycle the Web Server, WebSphere or both.

*The Tomcat Instance starts, but the process terminates prematurely*

If the servlet engine fails to start properly you must check the output spool files for the user profile under which the engine is supposed to run. Tomcat on the iSeries is very picky about the authorities assigned to certain files and directories, and may not start because some of the libraries in the classpath have \*PUBLIC write access.

*General Problems running servlets under Apache*

You must also ensure that the access rights to the directories and files under your Web application base directory are sufficiently open for the Tomcat instance to access them. Remember that you configured the Tomcat instance to use a particular user profile, and this is the profile that should be given increased access. Various problems will occur if you don't do this. You may not get any error messages indicating that this is the problem.

After making any configuration changes to the Apache/Tomcat installation, you must ensure that both the web server and the servlet engine are restarted. In our experience this is necessary even if the only change made was to alter the access rights to a file or directory.

## Changes in WebSmart function usage in WebSmart JSE

The following lists outline the differences in function support between WebSmart OE and WebSmart JSE.

WebSmart OE functions which are not available in WebSmart JSE:

1. The Index functions
2. The graphic functions
3. `callb()`
4. `dclentplist()`
5. `persist()`

Differences in functionality between WebSmart OE and WebSmart JSE functions:

1. Data encrypted by JSE servlets can't be unencrypted using RPG programs and vice versa.
2. Programs written to process stream file data may behave inconsistently if deployed to a different language than that under which they were developed.
3. The `setlibl()` function only affects SQL access to DB2 data using the JDBC driver included in the IBM AS/400 Toolbox for Java. Record level access functions use the Library/File name combination to determine which file to access.
4. The `sqlexec()` function currently supports only the FETCH, UPDATE, DELETE and CLOSE statements.
5. The `sqlquery()` function supports the SQL-92 syntax and not the SQL/400 syntax, so you may discover some incompatibilities when moving between RPG and Java.
6. You cannot store the contents of a structure in a cookie using WebSmart JSE.

Any program definition can be made platform independent by conditionally executing the platform dependent code based on the value of the `pgmf_platform` program field.