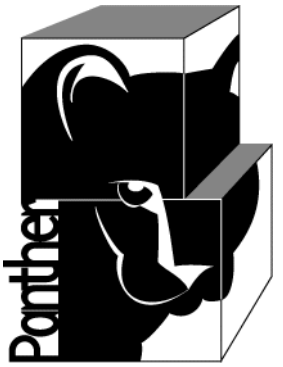
Contents:



Panther

for IBM WebSphere Developer's Studio

Prolifics

Release 5.51

Document 0404
March 2017

Copyright

Send suggestions and comments regarding this document to:

Technical Publications Manager
Prolifics, Inc.
24025 Park Sorrento, Suite 405
Calabasas, CA 91302

http://prolifics.com
support@prolifics.com
(800) 458-3313

Contents:

6. Deploying Enterprise JavaBeans in WebSphere

7. Building Client Screens

8. Deploying Your Application

A. Utilities

B. Sample Applications

C. Adding C Functions

D. Deploying Enterprise JavaBeans in WebSphere 3.5

Index

About This Document

Panther for IBM WebSphere Developer's Studio is aimed primarily at developers who are building Enterprise JavaBeans. It also contains information for system and project administrators who are responsible for setting up a Panther development environment and deploying a Panther application. This guide assumes that you have already installed Panther WebSphere Edition and WebSphere Application Server on your system.

This guide contains the following information:

- An overview of Panther WebSphere system architecture.
- How to setup the WebSphere Application Server for development.
- How to setup the development environment for Panther WebSphere applications.
- How to build Enterprise JavaBeans in the Panther editor.
- How to build client screens that call EJBs in the Panther editor.
- How to deploy your Panther Enterprise JavaBeans on the WebSphere Application Server.
- Description of command-line utilities that can help you develop and manage a Panther application.
- Description of the sample Panther WebSphere application.
- How to add C functions for your Enterprise JavaBeans.

Related Information

- For more information about WebSphere, refer to the IBM site at http://www.ibm.com/.
- For more information about Java, refer to http://www.java.com/.

Documentation Website

The Panther documentation website includes manuals in HTML and PDF formats and the Java API documentation in Javadoc format. The website enables you to search the HTML files for both the manuals and the Java API.

Panther product documentation is available on the Prolifics corporate website at http://docs.prolifics.com/panther/. The documentation is also distributed in the `docs` directory of Panther distributions.

How to Print the Document

You can print a copy of this document from a web browser, one file at a time, by using the File→Print option on your web browser.

A PDF version of this document is available from the Panther library page of the documentation website. You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe website at https://get.adobe.com/reader/otherversions/.

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
Ctrl+Tab	Indicates that you must press two or more keys simultaneously. Initial capitalization indicates a physical key.
italics	Indicates emphasis or book titles.
UPPERCASE TEXT	Indicates Panther logical keys. *Example*: XMIT
boldface text	Indicates terms defined in the glossary.

`monospace text`	Indicates code samples, commands and their options, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <smdefs.h>`
	`chmod u+w *`
	`/usr/prolifics`
	`prolifics.ini`

`monospace italic text`	Identifies variables in code representing the information you supply. *Example*: `String` *`expr`*

Convention	Item

`MONOSPACE UPPERCASE TEXT`	Indicates environment variables, logical operators, SQL keywords, mnemonics, or Panther constants.
	Examples:
	`CLASSPATH`
	`OR`

`{ }`	Indicates a set of choices in a syntax line. One of the items should be selected. The braces themselves should never be typed.
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `formlib [-v] library-name [file-list]...`
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `formlib [-v] library-name [file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

Contact Us!

Your feedback on the Panther documentation is important to us. Send us e-mail at support@prolifics.com if you have questions or comments. In your e-mail message, please indicate that you are using the documentation for Panther 5.50.

If you have any questions about this version of Panther, or if you have problems installing and running Panther, contact Customer Support via:

- Email at support@prolifics.com
- Prolifics website at http://profapps.prolifics.com

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address and phone number
- Your company name and company address
- Your machine type
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

CHAPTER

1 Overview

Panther for IBM WebSphere adds the following capabilities to Panther:

- Building Enterprise JavaBeans (EJBs) in the Panther editor
- Building client screens that call EJBs
- Deploying Panther-built EJBs on IBM's WebSphere Application Server

Panther makes it easy to create EJBs by producing the necessary files and interfaces from your method and property definitions. Panther for IBM WebSphere works with IBM's WebSphere product family to use your EJBs in both web and GUI environments. The products work together to fulfill your e-business requirements—building your transactional database applications quickly and deploying them in a secure enterprise environment.

What is an Enterprise JavaBean Application?

Enterprise JavaBeans (EJBs) are server-side components written in Java that perform the business logic of an application in multi-tier distributed applications. Typically, an application using EJBs consists of:

- An EJB server—EJB servers manage system resources—such as processes, threads, memory and network sessions—on behalf of the applications that run

on them. An EJB server must be able to host EJB containers and provide them with these services.

- EJB containers that run on the EJB server—An EJB container manages EJB classes and is responsible for making instances of these EJBs available to the application's clients. The container insulates the EJB from the workings of the underlying EJB server and provides transaction services, resource management and security services to the EJB classes it contains.
- EJBs that run in those containers—An EJB is a reusable component that implements a business task. It consists of a Java class, home and remote interfaces, a deployment descriptor and any environment settings needed for deployment. Since the EJB container handles the interaction with the server, the EJB focuses only on the business logic of the application.
- EJB-enabled clients—At runtime, clients invoke methods on EJBs to perform their server-side processing.

Note: For more information about EJBs, refer to the EJB site at:

`http://www.oracle.com/technetwork/java/javaee/ejb/index.html`

What is WebSphere Application Server?

The WebSphere Application Server provides a full-featured distributed application environment, including:

- *Multi-tier, distributed applications*. Multi-tier applications expand the client/server model so that the client interacts with the user; the application server contains the business logic; and the resource manager stores the data. Each tier can consist of multiple machines, working together as a single system.
- *Server-side components*. By implementing your business logic using components, you can reuse components in multiple applications, without additional programming.

- *Multi-threaded applications*. Using threads instead of processes improves application performance.
- *Database connection pooling*. One of the major benefits of multi-tier applications is that the application server maintains the database connections, not the application clients, reducing the number of concurrent connections.
- *Scalability.* The number of users for web-based applications can grow quickly and your deployment strategy must plan for this growth.
- *Security.* By defining your security model, you determine which users can access EJBs.
- *Web application administration*. Website usage logging and analysis tools are available. There is also an optional performance pack to handle caching, load balancing, and file replication for high-volume websites.
- *HTTP Web Server.* An Apache-based HTTP server is available, if one is not already in place.
- *Java technology support*. Java technologies provide an object-oriented and platform-independent environment utilizing open standards for the following application programming interfaces:
  - Enterprise JavaBeans—Server-side components.
  - Java Naming and Directory Interface (JNDI)—Access to naming and directory services; look up existing EJBs and interact with directories.
  - Java Messaging Service (JMS)—Asynchronous communications.
  - Remote Message Invocation (RMI)—Remote interfaces for Java-to-Java communications; analogous to a remote procedure call.
  - Java Transaction API (JTA)—Transaction demarcation API.
  - Java Transaction Service (JTS)—Transaction processing system API used within EJBs.
  - Java Interface Definition Language (JIDL)—Interfaces for connection to CORBA objects and applications.
  - Java Servlets—Java extension to HTTP servers for generating web content; an alternative to CGI.
  - JavaMail—Protocol-independent framework for mail and messaging systems.

How Do Panther and WebSphere Work Together?

In a Panther/WebSphere application, you get the benefit of Panther's development tools working together with IBM's WebSphere Application Server deployment capabilities.

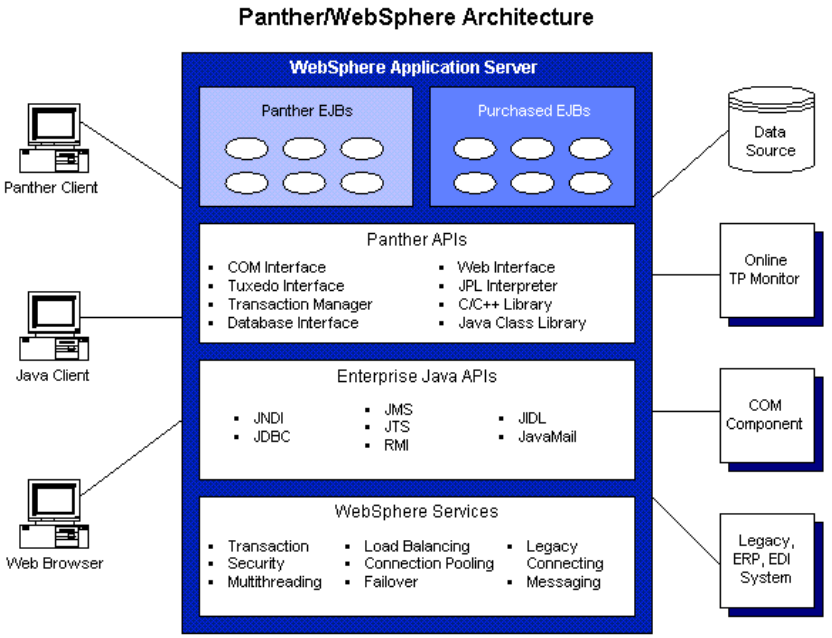


Figure 1-1 The Panther/WebSphere environment combines Panther and WebSphere technologies allowing access to different types of clients and data sources.

Panther's development environment accelerates the creation of application objects—EJBs, client screens— needed in multi-tier, component-based applications. WebSphere Application Server provides the tools for deploying your application on a system designed to manage high-volume, web-based applications.

In the Panther/WebSphere environment, WebSphere Application Server is the EJB server and provides the EJB container—the runtime context for the bean.

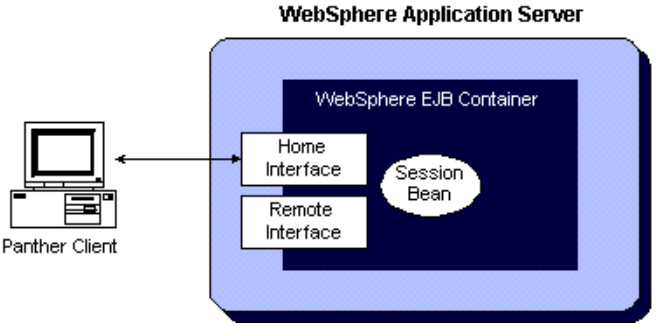


Figure 1-2 An EJB client uses the home interface to create the bean before calling the bean's methods, which are described in the remote interface.

Use the Panther editor to build EJBs—session beans that implement business tasks—and to build client screens that call EJBs. Each instance of a session bean is associated with a particular client and is created and destroyed by that client.

Building a Panther/WebSphere Application

Subsequent chapters describe each of the following tasks that are part of designing and building a Panther/WebSphere application. The following flowchart illustrates the tasks involved in this process.

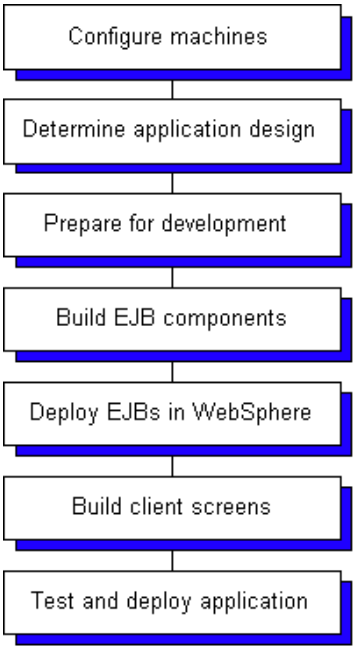


Figure 1-3 Process of developing a Panther/WebSphere application.

CHAPTER

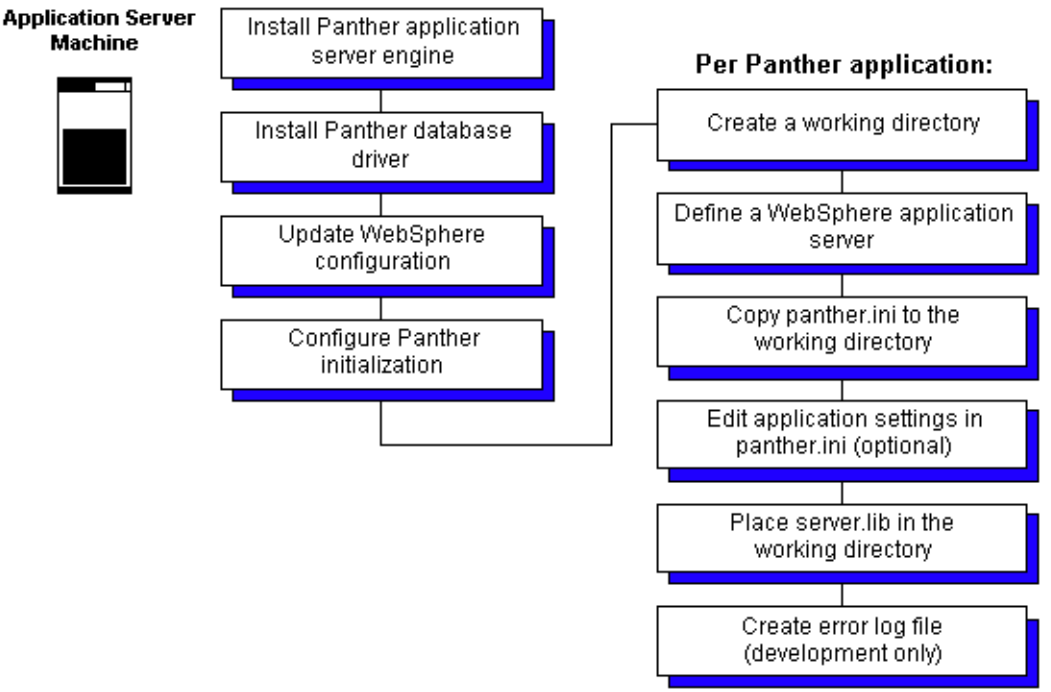
2 Configuring Machines

This chapter describes the process for configuring the machines that work with your Panther software components:

- How to Set Up the Application Server Engine (page 2-1)
- How to Set Up the Development Client (page 2-8)
- How to Set Up the Web Application Broker (page 2-13)

How to Set Up the Application Server Engine

The following flowchart illustrates the process for setting up the Panther application server engine on a machine running WebSphere Application Server:



Installing Panther Software

The *Panther for IBM WebSphere Installation Guide* describes the process for installing Panther software on application server machines for UNIX and Windows platforms.

How to Install Panther Software

1. Install the application server engine on the machine.
2. Add the Panther database driver for the selected database to the Panther installation.
3. Specify the location of the Panther installation in the `SMBASE` environment variable.

Updating Your WebSphere Configuration

Your application server machine should already have installed:

- WebSphere Application Server
- A current JDK version

After installing Panther, certain settings in WebSphere must be changed to recognize the Panther software.

How to Update Your WebSphere Configuration

1. Specify the location of the JDK installation in the `JAVA_HOME` environment variable.
2. Include `$JAVA_HOME/bin` in the `PATH` environment variable.
3. Specify the location of the WebSphere installation in the `WAS_HOME` environment variable.

   On Windows systems, a typical setting is `C:\WebSphere\AppServer`.
4. Include `$WAS_HOME/bin` in the `PATH` environment variable.
5. In the WebSphere Administrative Console, set the `classpath` to include `$SMBASE/config/pro5.jar` and `$SMBASE/servlet/proweb.jar`. For Windows systems, the installer performs this step.

   (`$SMBASE/config/pro5.jar` contains Panther's Java classes. `$SMBASE/servlet/proweb.jar` contains Panther's Java servlet classes.)
6. Specify the location of the Panther shared libraries. (For Windows systems, the installer performs this step.)
   - For Oracle Solaris/Linux, set `LD_LIBRARY_PATH`:

     ```
     LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SMBASE/lib
     ```
   - For AIX, set `LIBPATH`:

     ```
     LIBPATH=$LIBPATH:$SMBASE/lib
     ```
   - For HP-UX, set `SHLIB_PATH`:

     ```
     SHLIB_PATH=$SHLIB_PATH:$SMBASE/lib
     ```

- For Windows, the installer copies the DLLs to the Windows system directory, for example, `C:\Winnt\System`.

On UNIX systems, the shared libraries (`PanDmEJB.so`, `PanSmEJB.so` and `PanTmEJB.so`) are located in the `lib` directory.

On Windows systems, the DLLs (`PanDmEJB.dll`, `PanSmEJB.dll` and `PanTmEJB.dll`) are located in the `bin directory`.

7. Include `$SMBASE/util` in the `PATH` environment variable. This utilities directory provides access to Panther's license and `makeejb` utilities.

   For Windows, the installer copies the license DLL (`lmgr325.dll`) to the Windows system directory.

Configuring Panther Initialization

Each Panther/WebSphere application has an initialization file, `panther.ini`. The default file, located in the `config` directory, contains three sections:

- [`EJB Global`] – Specify settings common to all Panther/WebSphere applications. For more information, refer to page 2-5, "EJB Global Settings."
- [`EJB Class`] – Specify settings for a particular EJB class, which would include all deployed EJBs in that server process. For more information, refer to page 2-5, "EJB Class Settings."
- [`databases`] – Specify which databases are installed and, if it is more than one, specify which database is the default. The database settings are applied as part of the global initialization. For more information, refer to page 2-6, "Database Settings."

How to Edit Global Panther Settings

1. Open `$SMBASE/config/panther.ini`. Since all Panther/WebSphere applications must share the same global settings, edit the default file.
2. If necessary, update the settings in [`EJB Global`] to new values.
3. In the [`databases`] section, enter the database settings for your application.

Note: For Windows, the installer updates the global settings in `panther.ini`.

EJB Global Settings

The `EJB Global` section of `panther.ini` contains the following configuration settings, listed alphabetically.

`LM_LICENSE_FILE`
: Specify the location of the Panther license file. By default, the license file is located in `$SMBASE/licenses/license.dat`. (Mandatory)

`SMBASE`
: Specify the location of the Panther installation. (Mandatory)

`SMINITJPL`
: Specify the name of the JPL file loading shared libraries at startup. (This JPL file *cannot* include other JPL commands, only calls to `sm_slib_load` and `sm_slib_install`. For more information, refer to Appendix C, "Adding C Functions.")

`SMTPCLIENT`
: For Tuxedo applications, specify whether Tuxedo connectivity will be enabled in the application. If unset, the shared libraries needed for Tuxedo will not be loaded. If Tuxedo libraries are needed, set this variable to `native` or `workstation`, for native or workstation clients respectively.

`SMTPINIT`
: For Tuxedo applications, specify the default arguments to the `client_init` command.

`SMTPJIF`
: For Tuxedo application calls, specify the name of the JIF file in the Tuxedo application.

`SMVARS`
: Specify the location of the `SMVARS` file.

EJB Class Settings

The `EJB Class` section of `panther.ini` contains the following configuration settings, listed alphabetically.

`SMFLIBS`
: Specify the Panther application libraries to open on application startup. The default setting is to open `server.lib`, located in the same directory as `panther.ini`. (Mandatory)

SMINITJPL
: Specify the name of the JPL file to run at application startup.

SMMSGS
: Specify the location of the message file.

SMPATH
: Specify the search path for your Panther application files.

Database Settings

The databases section of `panther.ini` specifies which database drivers to load with your Panther applications. List the database drivers to load in the `installed` section, by database keyword:

```
installed=DB2_6
```

If more than one database driver is needed, you must also specify the default driver. For example:

```
default=DB2_6
```

```
installed=DB2_6, oracle8iProC
```

Creating an Application Server

WebSphere documentation recommends that you create a new application server for each application. Within that application server, you must also create a container for your EJBs. (In WebSphere, the name of the default application server is `server1`.)

How to Create an Application Server

1. Start the WebSphere server.
2. Start the WebSphere Administrative Console.
3. Create a new directory on the server machine for your Panther/WebSphere application. Set this directory to have write permissions.
4. Specify the directory created in Step 3 as the Working Directory in the Process Definition property of the application server.
5. Add Panther as a shared library.

6. Add the Panther library to the class loader.
7. Install the application.
8. Add the deployed jar file to the `CLASSPATH`.

Copying Files to the Application Server

The application server's Working Directory must contain the files necessary to deploy Panther-built EJBs:

- `panther.ini`
- A Panther application library containing the service components, such as `server.lib`

How to Set Up the Panther Application Files

1. Go to the application server's working directory.
2. From `$SMBASE/config`, copy `panther.ini` and check the following settings:
   - Check that `SMBASE` is set to the Panther installation directory.
   - Check that `LM_LICENSE_FILE` is set to the Panther license file.
   - In [`EJB Class`], specify the Panther application libraries you need to open in `SMFLIBS`. By default, it is set to open `server.lib`.
   - Edit any other settings in [`EJB Class`] that differ from the settings in [`EJB Global`].
3. In the Panther editor, if your development workstation has WebSphere installed, specify this directory in the EJB section of the Component Interface window. (As a result, the EJBs Java files are generated in this directory.)
4. After you build service components in the Panther editor, create or copy the Panther application library containing the service components to this directory. By default, this library is named `server.lib`.

Creating a Server Log File

You can send server messages to a log file. Because of performance considerations, this is not suggested for application deployment, only for application testing.

How to Activate Server Message Logging

1. Place a file named `server.log` in the application server's working directory.
2. Messages are automatically written to this file when a component is created and destroyed and when an error message is generated.
3. Write additional messages to this file using the JPL `log` command or its C equivalent `sm_log`.

Sample Error Log

The following sample log file illustrates some server messages:

```
Mon Jun 21 22:06:30 2016: Component cCustomers
    Created.
Mon Jun 21 22:06:44 2016: Component cCustomers, Method GetCustomer
    Searching on S
Mon Jun 21 22:06:50 2016: Component cCustomers
    Destroyed.
Mon Jun 21 22:07:03 2016: Component cCustomers
    Created.
Mon Jun 21 22:08:07 2016: Component cCustomers
    Destroyed.
```

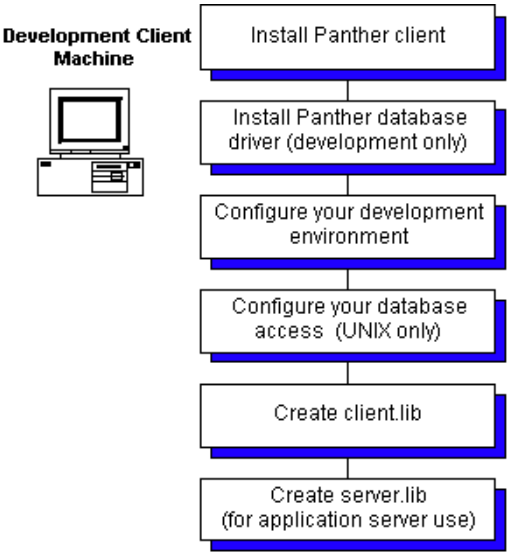
How to Set Up the Development Client

There are three type of development clients:

- Native – WebSphere Application Server is installed on your development machine.

- Local – You log into a machine that is running WebSphere Application Server.
- Remote – You have no direct access to WebSphere Application Server.

The following flowchart illustrates the process for setting up development clients:



Installing Panther Software

The *Panther for IBM WebSphere Installation Guide* describes the process for installing Panther software for development clients and database drivers on UNIX and Windows platforms.

Installing Panther client software allows you to run the Panther editor where you build your client screens and EJBs and store them in Panther application libraries.

Installing a database driver allows your development clients running the editor to make direct connections to the database. Direct connections are required for creating a repository and are also helpful for testing purposes.

Configuring Your Panther Client Environment

As part of the installation process, certain environment settings must be specified.

How to Configure Your Panther Client Environment

1. For Java compilation and testing:
   - Specify the location of the JDK in the `JAVA_HOME` environment variable.
   - Include `$JAVA_HOME/bin` in the `PATH` environment variable.
2. Set `SMJVMOPT` with the settings for the JVM.

   A UNIX/Linux example:

```
export
SMJVMOPT="\"-Dcom.ibm.CORBA.ConfigURL=$WAS_HOME/properties/sas.client.props\"

\"-Dcom.ibm.SOAP.ConfigURL=$WAS_HOME/properties/soap.client.props\"

-Dcom.ibm.CORBA.RasManager=com.ibm.websphere.ras.WsOrbRasManager

-Dwas.install.root=\"$WAS_HOME/\"

-Djava.ext.dirs=\"$WAS_HOME/java/jre/lib/ext:$WAS_HOME/java/lib:$WAS_HOME/classes:$
WAS_HOME/lib:$WAS_HOME/lib/ext:$WAS_HOME/web/help:$WAS_HOME/deploytool/itp/plugins/
com.ibm.etools.ejbdeploy/runtime:/opt/mqm/java/lib:/opt/wemps/java/lib\"

-Djava.security.auth.login.config=\"$WAS_HOME/properties/wsjaas_client.conf\"

-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContentFactory

-Xbootclasspath/p:\"$WAS_HOME/java/jre/lib/ext/ibmorb.jar:$WAS_HOME/java/jre/lib/ex
t/ibmext.jar\""
```

   A Windows example:

```
set SMJVMOPT="\"-Dcom.ibm.CORBA.ConfigURL=%WAS_HOME%\properties\sas.client.props\"

\"-Dcom.ibm.SOAP.ConfigURL=%WAS_HOME%\properties\soap.client.props\"

-Dcom.ibm.CORBA.RasManager=com.ibm.websphere.ras.WsOrbRasManager

-Dwas.install.root=\"%WAS_HOME%\\"

-Djava.ext.dirs=\"%WAS_HOME%\java\jre\lib\ext;%WAS_HOME%\java\lib;%WAS_HOME%\classe
s;%WAS_HOME%\lib;%WAS_HOME%\lib\ext;%WAS_HOME%\web\help;%WAS_HOME%\deploytool\itp\p
lugins\com.ibm.etools.ejbdeploy\runtime;\opt\mqm\java\lib;\opt\wemps\java\lib\"

-Djava.security.auth.login.config=\"%WAS_HOME%\properties\wsjaas_client.conf\"

-Djava.naming.factory.initial=com.ibm.websphere.naming.WsnInitialContentFactory

-Xbootclasspath/p:\"%WAS_HOME%\java\jre\lib\ext\ibmorb.jar;%WAS_HOME%\java\jre\lib\
ext\ibmext.jar\""
```

3. Set the server name and port number in SMPROVIDERURL of the server running WebSphere Application Server, as in:

   corbaname:iiop:*machine*:2809#cell/nodes/*machine*/servers/*server*

4. Set SMBASE to the location of the Panther installation. For Windows platforms, this is set in prol5w32.ini. For UNIX platforms, this is set in the environment.

5. Set SMFLIBS to the Panther application libraries needed to save your client screens and service components. The default names of these libraries are client.lib and server.lib. For Windows platforms, this is set in prol5w32.ini. For UNIX platforms, this is set in the environment.

6. In order to test EJBs, you need to add the following to the CLASSPATH setting:

   - The beans deployed jar file name(s) and location
   - $SMBASE/config/pro5.jar
   - $WAS_HOME/properties
   - $WAS_HOME/lib/bootstrap.jar
   - $WAS_HOME/lib/j2ee.jar
   - $WAS_HOME/lib/lmproxy.jar
   - $WAS_HOME/lib/urlprotocols.jar
   - $WAS_HOME/lib/namingclient.jar
   - Files containing your Java event handlers (optional)

7. For HP-UX, set SMJAVALIBRARY and LD_PRELOAD. For example:

   - export SMJAVALIBRARY=$JAVA_HOME/jre/lib/PA_RISC/libjava.sl
   - export LD_PRELOAD=$JAVA_HOME/jre/lib/PA_RISC/libjava.sl

   Note: If WebSphere is to be started and stopped in the same environment, make sure LD_PRELOAD is set to nothing before issuing the command.

   For more information on client runtime configuration, refer to page 8-3, "How to Set Up the Development Client."

Configuring Your Panther Database Drivers - UNIX

On UNIX systems, in order for the Panther editor to connect to the database during development, you must configure database access in prol5unix.ini.

How to Configure Your Panther Database Drivers on UNIX

1. Open `$SMBASE/config/prol5unix.ini`.
2. Add the database keyword to the `Installed=` line. Panther for IBM WebSphere has the following database keywords available on UNIX:

Keyword	Description
DB2_6	DB2
oracle8iProC	Oracle 8i for ProC
oracle8iOCI	Oracle 8i for OCI

3. If you install more than one database driver, you must also update the `default=` line and specify the default driver.
4. Set `SMPATH` to include the directory with the `prol5unix.ini` location, by default `$SMBASE/config`.

Creating Application Libraries

The Panther distribution includes two application libraries, `client.lib` and `server.lib` in the `config` directory. However, it is recommended that you create your application libraries outside of your Panther distribution.

If you are a native or local development client, you can create the application libraries in the application server's working directory.

How to Create Client Application Libraries

1. In the Panther editor, choose File→New→Create Library.
2. Enter its name, for example `client.lib`.

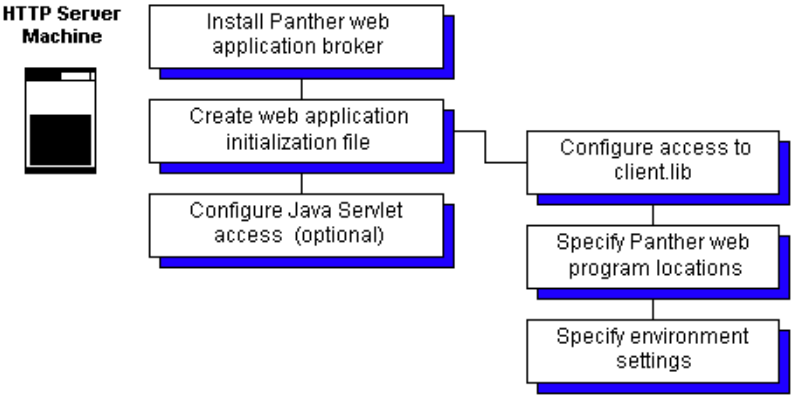
How to Create Server Application Libraries

1. In the Panther editor, choose File→New→Create Library.

2. Enter its name, for example `server.lib`.

How to Set Up the Web Application Broker

The following flowchart illustrates the process for setting up HTTP servers.



Installing Panther Software

The *Panther for IBM WebSphere Installation Guide* describes the process for installing Panther software for HTTP server machines on UNIX and Windows platforms.

The HTTP server for most development configurations is on the same machine as WebSphere Application Server. Therefore, the WebSphere files will already be available. In this case, install the Panther web application broker in the same location as the Panther application server engine.

Installing WebSphere

If WebSphere is not on the HTTP server machine, you must install WebSphere Developer's Client Files.

Copying Client Libraries

The Panther web application broker acts as an application client. Therefore, it will need access to the client application libraries, such as `client.lib`. If WebSphere Application Server and the HTTP server are on the same machine, use the application server's working directory for the client files.

How to Set Up the Application Libraries for the Web

1. Go to the application server's working directory.
2. Copy `client.lib` to this directory.

Creating a Web Initialization File

Panther web applications have an initialization file containing the web application's environment settings and file locations. In that initialization file, you need to specify the location of your application files, of your Panther web application broker executables, and the environment settings for the web application.

This initialization file specifies the runtime environment for the Panther web application broker; settings are not read from the environment.

Note: Web initialization files must contain the full path name for Panther, WebSphere and Java installations. Environment variables cannot be expanded.

How to Create a Panther Web Initialization File

1. Run the Web Setup Manager, Panther's web application utility. For more information, refer to Appendix B, “Web Setup Manager,”in the *Web Development Guide*.

2. (UNIX only) Under Shared Library Path on the Environment Settings window, set `LD_LIBRARY_PATH` to include `$SMBASE/lib`, `$WAS_HOME/bin`, and the JVM shared libraries.

3. Under Path on the Environment Settings window, set `PATH` to include:

   - *`PantherInstallDir`* `/util`
   - *`JavaInstallDir`*`/bin`
   - *`WebSphereInstallDir`*`/bin`
   - Other directories needed by your web application, such as database directories

   Note: This PATH setting is independent of any environment PATH setting so you must specify any directories needed by your Panther web application and by a WebSphere client.

4. On the Additional Environment Variables window:

   - Set `JAVA_HOME` to the location of the JDK installation.
   - Set `SMJVMOPT` with the options for the JVM.
   - Set `WAS_HOME` to the WebSphere installation.
   - Set `SMPROVIDERURL` to the server machine's host name and port number in the following format: `iiop://`*`hostName:portNumber`*.
   - Set `CLASSPATH` to include:
     - `$WAS_HOME/properties`
     - `$WAS_HOME/lib/bootstrap.jar`
     - `$WAS_HOME/lib/j2ee.jar`
     - `$WAS_HOME/lib/lmproxy.jar`
     - `$WAS_HOME/lib/urlprotocols.jar`
     - `$WAS_HOME/lib/namingclient.jar`
     - *`PantherInstallDir`*`/config/pro5.jar`
     - *`PantherInstallDir`*`/servlet/proweb.jar` (for Java servlets)
     - After creating EJBs, `EJBDirectory/deployedEJB.jar`
   - Set the database environment variables that are needed.

5. If you need to edit or view the file, Panther places the web initialization files in `~proweb/ini` on UNIX servers and the Windows directory (for example, `C:\Winnt`) for Windows servers.

Configuring Java Servlet Access

Instead of using the CGI protocol to access a web application, you can configure the web application to run as a Java servlet. For this configuration, WebSphere Application Server needs to know the location of the Panther classes for Java servlets.

If WebSphere Application Server and the HTTP server are on the same machine, this was done as part of the application server configuration (see Step 4.).

If the HTTP server is on a separate machine, you will need to update the machine's `CLASSPATH` variable.

How to Update Your Java Servlet Configuration

1. On separate HTTP servers, add the name and location of the Panther Java classes, `proweb.jar`, to the `CLASSPATH` variable for WebSphere. For example:

   UNIX `CLASSPATH=/usr/panther/servlet/proweb.jar`
   Windows `CLASSPATH=C:\Panther\servlet\proweb.jar`

2. Add the name and location of the Panther Java classes, `proweb.jar`, to the `CLASSPATH` variable for Panther's web application broker. (In the previous section, see Step 4.)

CHAPTER

3 Designing the Application

Design Considerations for Panther/WebSphere

In addition to the standard application design tasks, consider the following tasks and issues for your Panther/WebSphere application:

- *Build your database schema.*

  Having a working database schema speeds the development process. The resulting repository based on that schema contains information identifying database tables and columns and defining table relationships. Your client screens and service components built from that repository contain the necessary information for database access.

- *Plan how to use components.*

  You can, for example, use components to implement all of your database access and business logic or to implement repeated tasks in a portion of your application.

  There are also different methods of organizing components. One way is to have one group of components that correspond to database tables and another group of components that implement business logic. A component in the business

logic group calls the database components it needs to complete a task. The database component group can focus on implementing any database rules while the business logic group focuses on the business rules.

- *Plan how components will access the database.*
  - To write your own SQL statements, we recommend that the names of fields on your service component (in the Identity property section) match the database column names. However, if this is not possible, an alias command is available for mapping the two objects. For more information about mapping database columns to fields, refer to "Targets for a SELECT Statement" on page 29-3 in *Application Development Guide*.
  - To use the transaction manager, fields on the service component must contain information in the Database section of the Properties window and the service component itself must contain table view widgets for database table information and link widgets for database table relationships. For basic information about transaction manager processing, refer to Chapter 31, "Building a Transaction Manager Screen," in *Application Development Guide*.
- *Define what type of application clients you will need.*

  Panther/WebSphere applications can be accessed by GUI clients or by web browsers. Knowing the type of access can help you design your client screens more easily.
- *Decide if you need Panther-generated reports.*

  Panther-built EJBs are not able to access Panther's report generation feature; only runtime client executables can generate reports. To do so, the client will need a direct database connection.

Assembling the Project Team

A Panther/WebSphere development team can encompass the following roles:

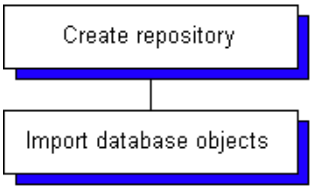
- *Business analyst*—Defines the business needs of the application.

- *WebSphere Application Server administrator*—Provides deployment and management tools and a runtime infrastructure for EJBs.
- *Database administrator*—Coordinates access to databases and other data resources.
- *Enterprise bean developer*—Implements business logic in EJBs and generates their Java classes and interfaces.
- *Application assembler*— Creates applications from existing EJBs and deploys those EJBs in web and GUI environments.
- *Web designer*—Creates HTML pages that can later encapsulate EJB components.

CHAPTER

4 Preparing for Development

This chapter describes the following tasks needed to prepare for development.



Creating a Repository

You can use the visual object repository to help build your application objects. Through inheritance, it can give a consistent interface to your client screens and service components.

For the steps to create a repository, refer to “Creating a Repository” on page 2-20 in *Using the Editors*.

Importing Database Definitions

After creating a repository, you can import your database definitions. The import process creates a repository entry for each database table and fields on each entry for the database columns. You can add additional repository entries during development to store your own application templates.

For information on importing database objects to a repository, refer to “Populating a Repository with Database Objects” on page 2-25 in *Using the Editors*.

For more information on using a repository, refer to Chapter 11, “Creating and Using a Repository,” in *Application Development Guide.*

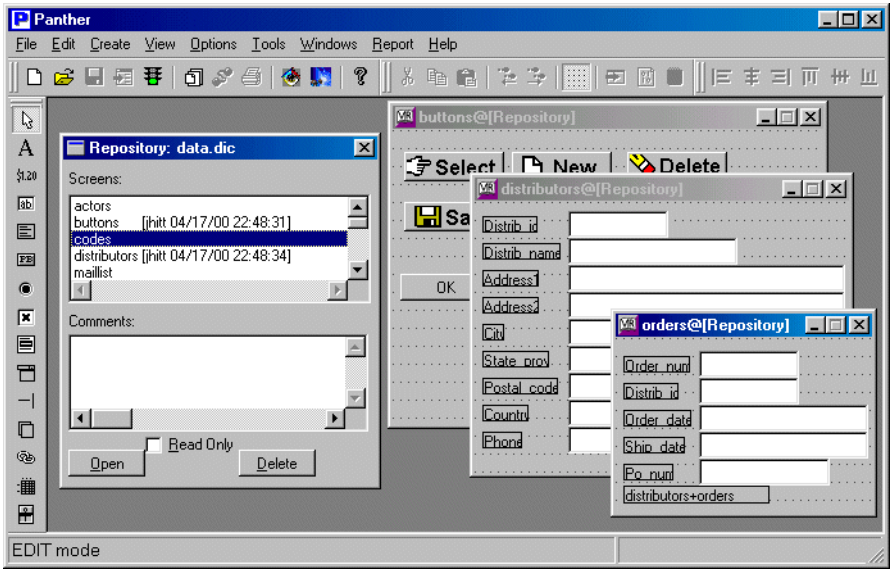
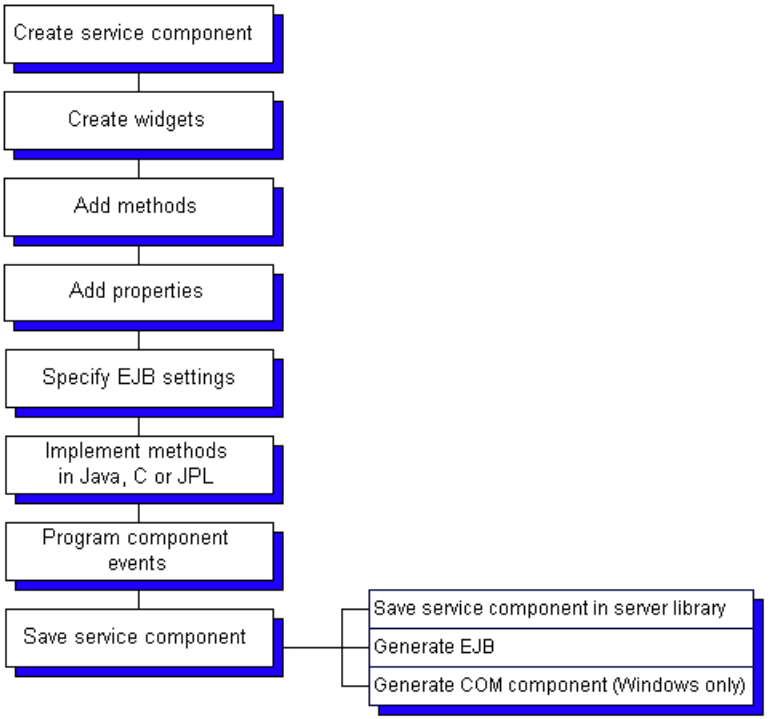


Figure 4-1 The visual object repository can contain entries imported from your database definitions as well as your own application objects.

CHAPTER

5 Building Enterprise JavaBeans

This chapter describes the process for building Enterprise JavaBeans in Panther.

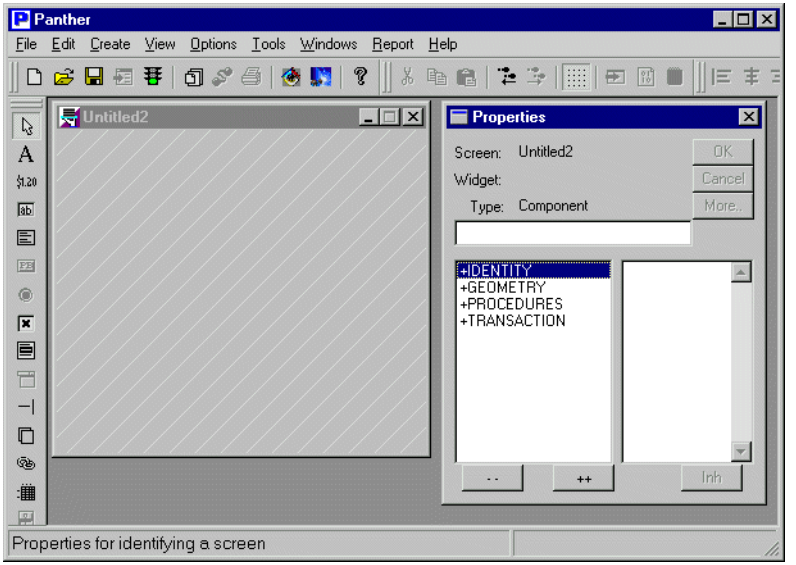


Creating Service Components

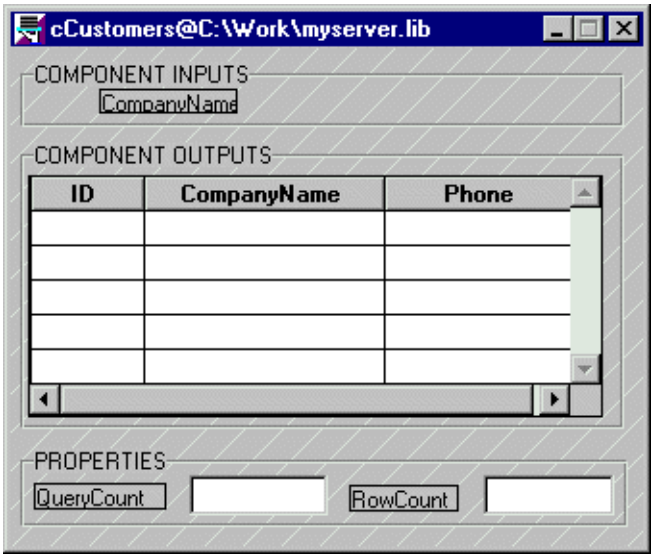
Service components perform the server-side processing in component-based applications. In Panther, building a service component saves a Panther service component in an application library and generates the EJB and/or COM component associated with that service component.

How to Create Service Components

1. In the editor, choose File→New→Service Component, which displays a blank service component.



2. Create the widgets (from the Create menu or from the repository) that will be needed for method parameters and properties.

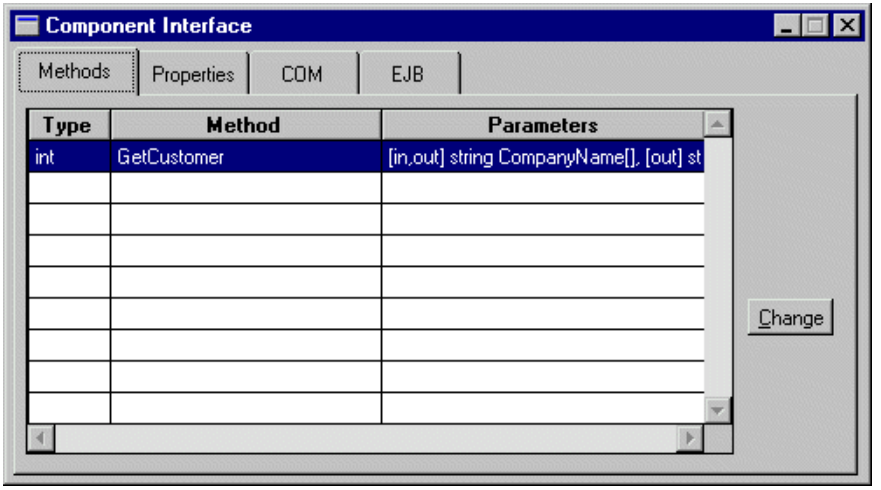


Defining the Component Interface

Components must have a defined interface. This interface consists of properties and methods supported by the component, together with information about those properties and methods, such as data types, parameter lists, and so on. This interface definition is the public face of your component; all interactions with application clients is by means of the properties and methods defined in this interface.

How to Define the Component Interface

1. Choose View→Component Interface. The Component Interface window displays.



2. To add methods, select the Methods tab and choose Change. For more information on adding methods, refer to page 5-5, “Adding a New Method.”

3. To add properties, select the Properties tab and choose Change. For more information on adding properties, refer to page 5-10, “How to Add a Property.”

4. To specify your EJB settings, select the EJB tab. For more information on EJB settings, refer to page 5-12, “The Component Interface Window - EJB Section.”

The Component Interface Window - Methods Section

The Methods section of the Component Interface window lists the methods currently defined for the component. Each row in the grid corresponds to a method. The first column shows the method's return type, the second the method's name, and the third the method's parameters, prefixed by their kind and type.

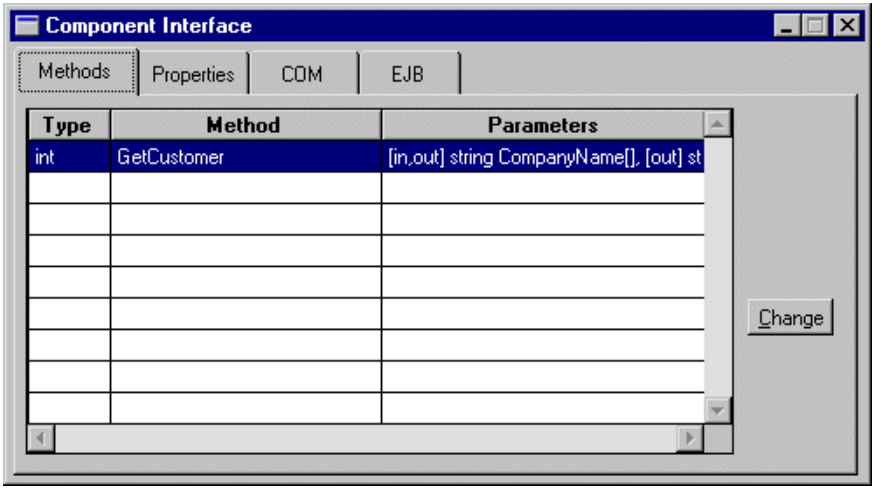


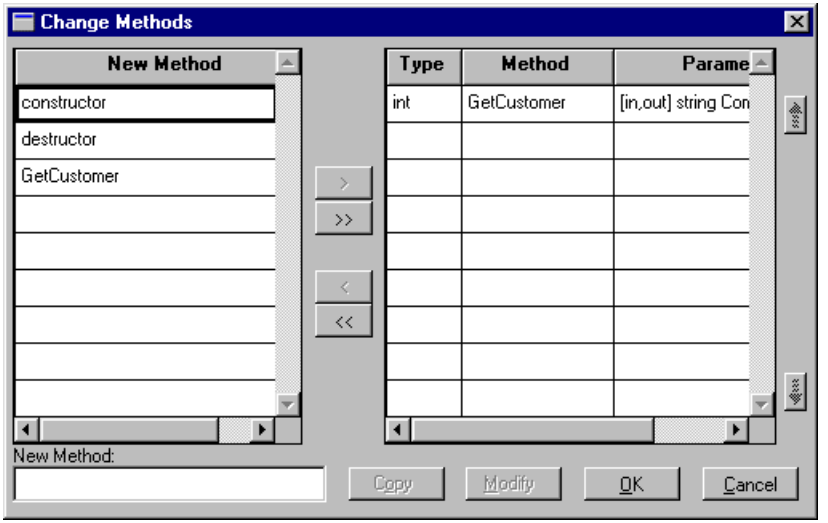
Figure 5-1 The Methods window allows you to view and define the component's methods.

Methods correspond to the actions to be performed by the component. The code to implement the service component's methods can be written in JPL, in C, or in Java. The procedure or function must be in scope at runtime, and the name must match the method name. For information on implementing methods, refer to page 5-16, "Implementing Methods."

Adding a New Method

Choosing Change on the Methods section displays the Change Methods window where you can add a new method, copy an existing method, or edit a method. On the left, the New Methods column displays the JPL procedure names defined in the service component's JPL Procedures property. On the right, the grid contains the current list of methods, their types, and parameters.

Note: You must specify at least one method in order to generate an EJB.



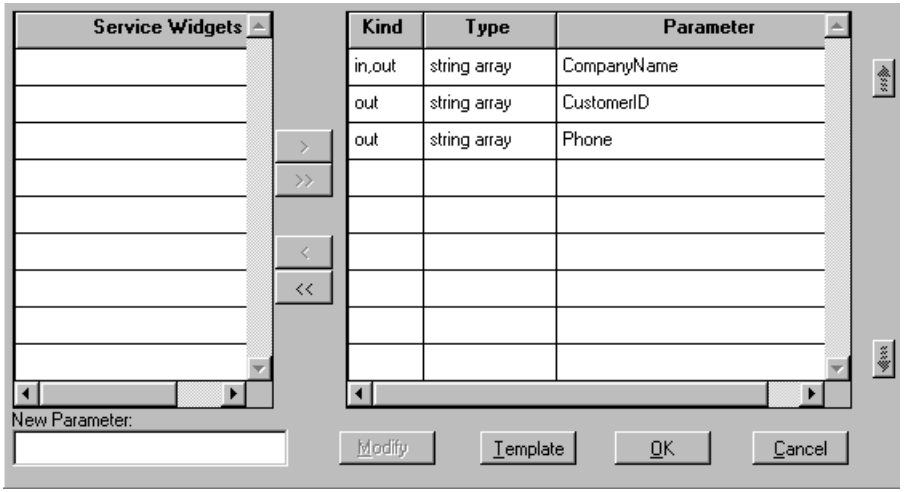
How to Add a Method

1. In the New Methods column, highlight the method names and use the arrow buttons to add methods to the grid.
2. Alternatively, type the method name in the New Method field, and choose OK. (This name does not need to be listed in the New Methods column.)
3. To add (or edit) the method's parameters, highlight the method in the grid on the right. Choose Modify (or double-click) to open the Change Parameters window.
4. Specify the method's return type at the top of the Change Parameters window. The return type for the method can be Void, String, Int, Bool, Double, or Object. The JPL return value for the method will be converted to the appropriate type and returned to the client.

Note: The method's return value cannot be an array. If this functionality is needed, the data should be returned in a parameter.

Specifying the Parameters

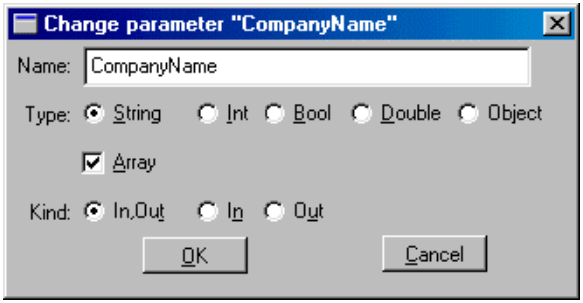
On the Change Parameters window, the Service Widgets column to the left displays the names of the fields on the service component. On the right, the grid contains the current list of parameters, their kind, and type. The order of the parameters is important; it is the order clients will use when calling the method.



How to Add Parameters

1. In the Service Widgets column, highlight the names and use the arrow buttons to add parameters to the grid. If the selected widget is an array, the array specification is selected automatically for the parameter.
2. Alternatively, type the parameter name in the New Parameter field, and choose OK. (This name does not need to be listed in the Service Widgets column.)
3. Choose Modify (or double click on a parameter name) to open the Change Parameter window, which contains four fields:
   - Name—The parameter's name
   - Type—String, Int, Bool, Double, or Object
   - An Array check box
   - Kind—In/Out, In, or Out

4. Choose the desired options for each parameter.



How to Generate the Method's JPL Procedure

- On the Change Parameters window, choose Template to generate a JPL procedure for the method and its parameters in the clipboard. This generated JPL template can then be pasted into the JPL procedures or a file.

  For example, the Template option generated the following JPL for the `GetCustomer` method:

```
proc GetCustomer
{
receive_args (CompanyName)

return_args (CompanyName, CustomerID, Phone)
}
```

The Component Interface Window - Properties Section

The Properties section in the Component Interface window displays a grid that shows the properties supported by the component. Here you can add, modify or delete a component's properties. Often, properties are defined to contain information about the application state.

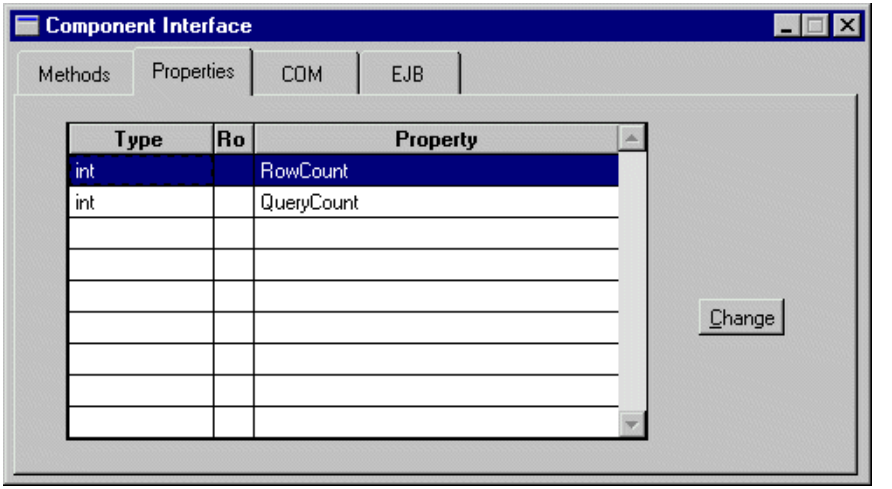
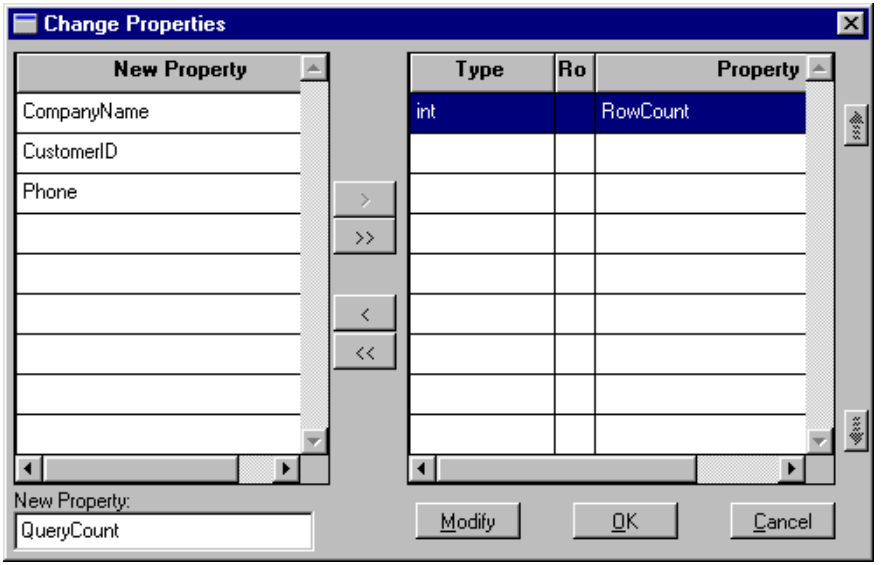


Figure 5-2 The Properties card allows you to add, modify, and delete a component's properties.

Each row in the grid corresponds to a property. The grid's columns correspond to the attributes of each property: Type, Read-only (Ro) and Property Name. The order in which the properties appear in this grid is irrelevant to the functionality of your component.

How to Add a Property

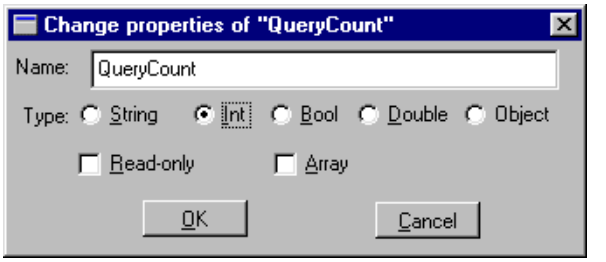
Choosing Change opens the Change Properties window where you can add new properties or modify current properties. The New Property column displays the widget names found in the service component. You do not have to choose one of the names on this list. You can enter the name of a global JPL variable or the name of a field that you will create later.



1. In the New Property column, highlight the widget names and use the arrow buttons to add properties to the grid.

2. Alternatively, you can enter the name of the property in the New Property field and choose OK.

3. Highlight the property name in the grid and choose Modify to display the Change Properties window.



4. The Change Properties window consists of the following fields:

   - Name—The name clients will use when accessing the property. The name must correspond to a field or global JPL variable accessible when the component is open at runtime. This field or variable implements the property.
   - Type—A property can be specified as String, Int, Bool, Double, or Object. The value in the field or JPL variable that implements the property is converted to this type when a client requests the value of the property, and handed to the client in the form specified.
   - Read-only—If a property is marked Read-only, clients can get its value but not set it.
   - Array—A property can have an array of values.

How Panther Implements EJB Properties

The specification for EJBs, unlike other types of components, does not contain support for reading and setting properties. In order to provide this support, the interface generated for the EJB contains a set method and get method for each property defined in the interface.

The Component Interface Window - EJB Section

In the EJB section of the Component Interface window, you can specify:

- The directory in which to generate the EJB Java files
- The settings to be included in the EJB deployment descriptor

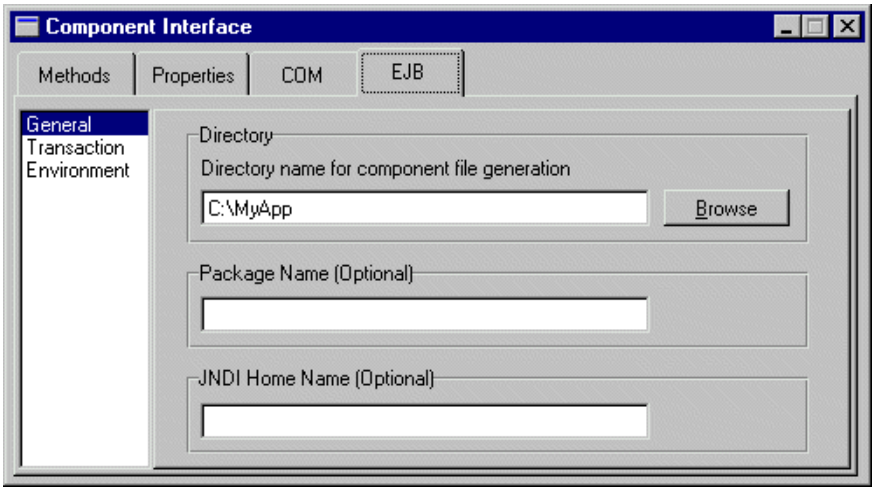
Specifying General Settings

Select General to specify the following:

Directory
: Enter the directory to contain the generated Java files (or Java jar file). If WebSphere is installed on your machine, you can choose the application server's working directory.

Package Name
: Enter a Java package name for the EJB bean class, home interface and remote interface. Typically, the same package name is used for an application or a subset of an application. If you leave this blank, it defaults to the component name with "EJB" appended.

JNDI Home Name
: Enter the name to be associated with the EJB on the JNDI server. If you leave the field blank, the JNDI Home Name defaults to the service component name. If you enter a value, it must be specified as the component name in the client's `sm_obj_create` calls.



Specifying Transaction Settings

On the EJB section of the Component Interface, select Transaction to specify the transaction attributes for the bean. If you write your own JDBC, you can set transaction attributes for the methods in a bean.

Transaction Attributes

Set the Transaction Attribute to include in the EJB deployment descriptor. The possible values are: `BEAN_MANAGED`, `MANDATORY`, `NOT_SUPPORTED`, `REQUIRES_NEW`, `REQUIRED`, and `SUPPORTS`. For more information on each setting, refer to page 5-14, “Transaction Attribute Settings.”

Isolation Level

Set the Isolation Level to include in the EJB deployment descriptor. The possible values are: `SERIALIZABLE`, `REPEATABLE_READ`, `READ_COMMITTED`, and `READ_UNCOMMITTED`. For more information on each setting, refer to page 5-14, “Isolation Level Settings.”

Transaction Attribute Settings

If you write your own JDBC, you can set the Transaction Attribute to the following values. Each setting is briefly described below. For more information, refer to the *IBM WebSphere Documentation*.

- `BEAN_MANAGED`—Indicator that the bean is an active participant in transactions and can call methods to explicitly manage transaction boundaries.
- `MANDATORY`—The container always invokes the bean methods within the transaction context associated with the client.
- `NOT_SUPPORTED`—The container invokes bean methods without a transaction context.
- `REQUIRES_NEW`—The container always invokes the bean method within a new transaction context, regardless of whether the method is invoked within an existing transaction context.
- `REQUIRED`—The container invokes the bean method within a transaction context. If a method is invoked outside a transaction context, the container creates a new transaction context and invokes the bean method from within that context.
- `SUPPORTS`—The container invokes the bean method within a transaction context if the client invokes the bean method within a transaction. If the client invokes the bean method without a transaction context, the container invokes the bean method without a transaction context. (Default)

Note: Panther database drivers support only this option in the current version.

Isolation Level Settings

If you write your own JDBC, you can set the Isolation Level, which determines how isolated one transaction is from another, to the following values. Each setting is briefly described below. For more information, refer to the *IBM WebSphere Documentation.*

- `SERIALIZABLE`—Prohibits dirty reads, nonrepeatable reads, and phantom reads.
  - Dirty reads—An uncommitted "update" (A transaction reads a database row containing uncommitted changes from another transaction.)

- Nonrepeatable reads—A committed "update" (One transaction reads a row, a second transaction changes the same row so that when the first transaction rereads the row, it gets a different value.)
- Phantom reads—An uncommitted "insert" (One transaction reads rows satisfying a `SQL WHERE` clause, a second transaction inserts a row satisfying the same `WHERE` clause, and the first transaction applies the same `WHERE` condition and gets the row inserted by the second transaction.)

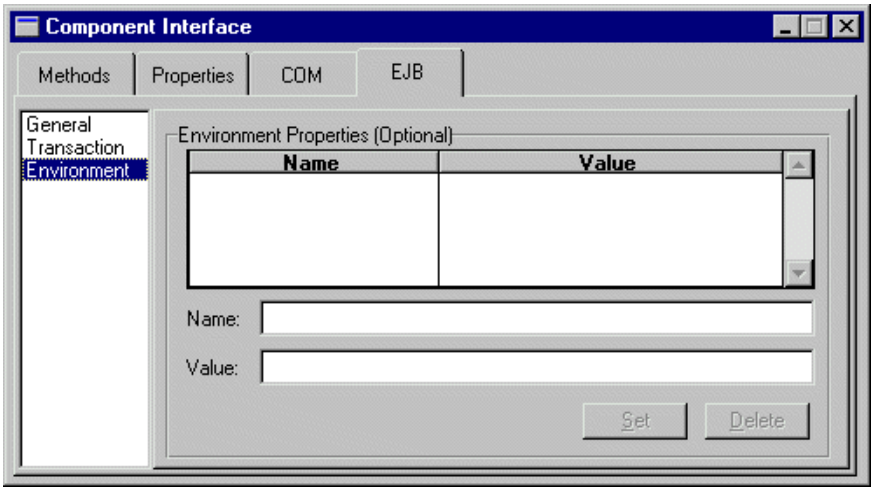
- `REPEATABLE_READ`—Prohibits dirty reads and nonrepeatable reads, but allows phantom reads.
- `READ_COMMITTED`—Prohibits dirty reads, but allows nonrepeatable reads and phantom reads.
- `READ_UNCOMMITTED`—Allows dirty reads, nonrepeatable reads, and phantom reads. (Default)

Note: Panther database drivers support only this option in the current version.

How to Specify Environmental Settings

Select Environment to specify user-defined environment variables and their values for use in Java business logic that implements the bean's methods. These variables are available only at the bean's scope; they are unavailable to the application's environment.

- Enter the Java environment property in the Name field, enter its value in the Value field, and press Set.



Implementing Methods

A method is a piece of work that a client can request the component to perform. The component implements each of its methods by means of a function (written in C, JPL or Java) that has the same name as the method's name.

Implementing Methods in JPL

To implement a method in JPL, the JPL procedure must be in scope when the service component is open at runtime. The simplest way to do this is to use the Template option on the Change Methods window to write a JPL procedure to the clipboard and then paste that procedure in the screen-level JPL (under Procedures⟶JPL Procedures). However, the procedure could also be written in a separate JPL module and made available with `include` or `public` commands.

Receiving a Method's Parameters

A JPL procedure that implements a component's method gets no parameters passed directly to it. To gain access to the method's `in` and `in/out` parameters, as sent by the client making the method request, use the `receive_args` command. For example, a sample JPL implementation of a component's method would be the following procedure:

```
proc my_method()
{
    vars id, name
    receive_args (id, name, address, city, state, phone) ...
}
```

The JPL command `receive_args` is followed by a list of targets. The values of the `in` and `in/out` parameters from the method call are placed in these target locations. The `out` parameters are skipped, therefore if a given method call has two `out` parameters, the first and third, only the values of the second, fourth and following parameters would be copied into the target list for the `receive_args` command.

The items in the target list must have a valid JPL syntax. In the example above, the first two items on the target list are variables declared locally in the procedure. For the target list to be valid, the latter four items on the target list must correspond to fields on the screen, to JPL global variables, or to JPL variables declared in the screen's unnamed JPL. Since any valid JPL syntax can be used, you could copy an incoming parameter into a property by using the property API.

Sending a Method's Parameters

The client that made the request for a method is expecting values passed back to it in the `in/out` and `out` parameters. It also expects a return value for the method as a whole, and it might also be checking for error codes, to determine whether some error has occurred while attempting to perform the method. To send these values back to the clients code like the following would be used:

```
proc my_method()
{
    . . .
    return_args (id, name)
    raise_exception -2
    return 0
}
```

Note: For information on calling methods from a client screen, refer to page 7-5, "Accessing the Component's Methods."

The `return_args` command works similarly to the `receive_args` command. The values of the JPL variables in the list will be written to the `in/out` and `out` parameters of the method, based on the order of the parameters in the method's definition.

Sending an Error Code

The `raise_exception` command is used to send an error code back to the client. The client's error handler can then decide what to do based on the value sent.

Sending the Return Value

The JPL return command is used to provide the return value for the method.

For return types other than integer or object, you must declare the return type in the `proc` statement, for example:

```
string proc s_method()
```

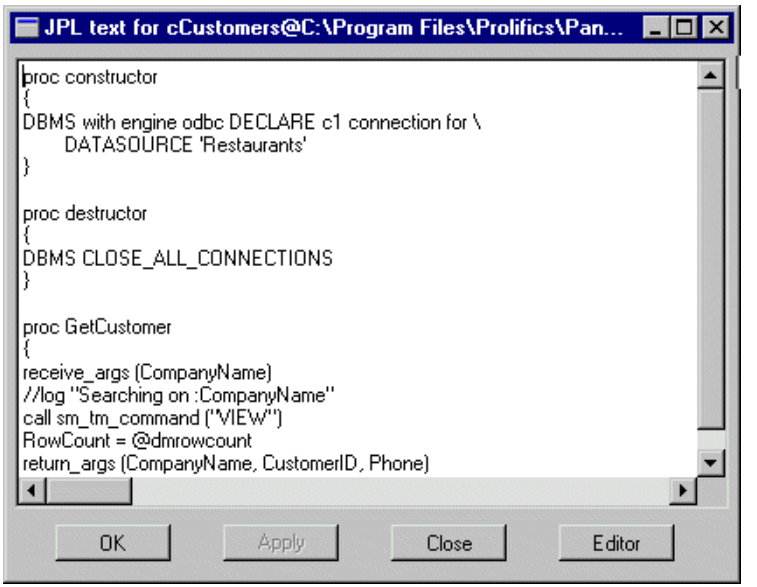


Figure 5-3 This sample screen-level JPL module implements the component's methods.

JPL Variables

JPL variables declared in the component's unnamed procedure are available only in the component-level JPL; they will not be in scope for widget-level JPL or for calls from the client. To increase the scope of the variables, use the JPL `global` command.

Implementing Methods in C

In C, rather than the JPL commands `receive_args`, `return_args`, and `raise_exception`, you would use the following functions:

- `sm_receive_args (char *)`
- `sm_return_args (char *)`
- `sm_raise_exception (int, char *)`

As in JPL, the functions `sm_receive_args` and `sm_return_args` take a list of Panther variables as a single string. The variables can be comma or space-delimited and can include the use of the property API syntax to refer to properties of fields on the component.

To make your C code accessible at runtime, your code must be packaged in a shared library and loaded at application startup.

Note: For more information on calling your C functions in EJBs, refer to Appendix C, "Adding C Functions."

Implementing Methods in Java

To implement a method in Java, the service component must have a Java Tag property set to correspond to a class which implements `ScreenHandler`. This class should have a method whose return type corresponds to the return type specified for the method (such as `int`, `double`, `boolean`, `string` or `object`) and which takes two parameters, a `ScreenInterface` and a `WSFunctionsInterface`.

The `WSFunctionsInterface` contains the following methods:

- `get_bean`

  ```
  public native PantherSessionBean get_bean()
  ```

- `log`

  ```
  public native int log(String message)
  ```

- `raise_exception`

  ```
  public native void raise_exception(String message)
  ```

- `receive_args`

  ```
  public native int receive_args(String args)
  ```

- `return_args`

  ```
  public native int return_args(String args)
  ```

Calling Other Enterprise JavaBeans

An EJB can also create other EJBs and, in a Panther application, open Panther screens. Each component operates in a different context; each component also has its own form stack. This means that the property API cannot be used to pass information between EJBs; instead, you must use the public methods of the beans.

Programming Component Events

For service components, the Procedures category (in the Properties window) contains properties for two events:

- Entry Function—Occurs when the component is instantiated
- Exit Function—Occurs when the component is destroyed

Entry and Exit Functions can be written in JPL, Java or C/C++. Typically, these functions are used to open and close database connections.

For JPL, enter a procedure name in the Entry and/or Exit Function properties and then access the JPL Procedures property to enter the JPL commands.

For Java, enter a Java Tag (under Identity), which is the name of the Java event handler for the entry and/or exit events. For more information on using Panther's Java event handlers, refer to Chapter 21, “Java Event Handlers and Objects,” in *Application Development Guide*.

For C/C++, the function named in the Entry and/or Exit Function properties must be packaged in a shared library and loaded at application startup. For more information on calling your C functions in EJBs, refer to Appendix C, “Adding C Functions.”

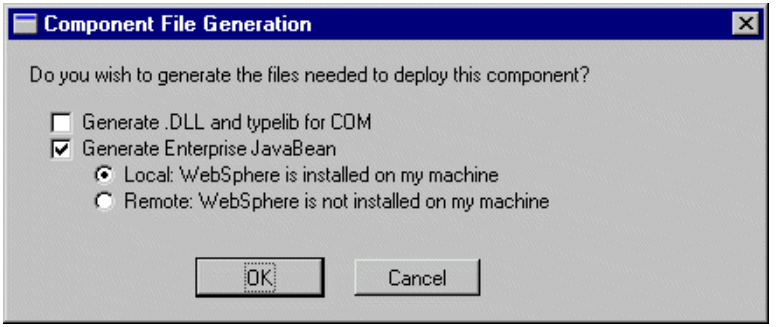
Generating Enterprise JavaBeans

An Panther-built EJB consists of:

- A Panther service component located in a Panther application library.
- A Java class, home and remote interfaces, a deployment descriptor and environment properties packaged in a jar file.

How to Save a Service Component and Generate an EJB

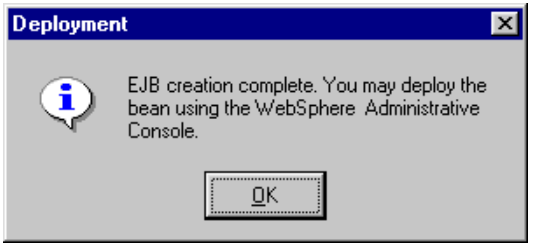
1. Create an application library in which to save your service components, such as `server.lib`, if one does not already exist.
2. If WebSphere is installed on your machine, set the Directory field (on the EJB section of the Component Interface window) to the directory which will contain the initial jar for the generated EJBs.
3. Choose File⟶save As⟶Library Member. Enter a name, and select the application library to save it in, generally `server.lib`.
4. The screen displays the following Component File Generation window.



5. Select your options and choose OK.

- To generate a COM component (32 bit Windows only), select Generate DLL and Typelib for COM. The DLL and typelib are placed in the directory specified on the COM section of the Component Interface.
- To generate an EJB, select Generate Enterprise JavaBean, which places the files in the directory specified in the General setting of the EJB section of the Component Interface.
- Select Local if WebSphere is installed on your machine.
- Select Remote if WebSphere is not available.

6. If you select Local:
   - The Java files for the EJB are generated to the Directory specified on the Component Interface window. (For a list of affected files, see page 5-24, "Description of the Java Files.")
   - The files are compiled and placed in a jar.



   For the next step, deploying the EJB, refer to Chapter 6, "Deploying Enterprise JavaBeans in WebSphere."

7. If you select Remote, the screen displays the following message:

You need to:

- Copy the application library (such as `server.lib`) to the application server's working directory.
- In the working directory, run `makeejb` on the application library. This generates the Java files, compiles the files in a jar file.
- Start IBM's Administration Console in order to deploy the bean. This process is described in Chapter 6, “Deploying Enterprise JavaBeans in WebSphere.”

Description of the Java Files

Generating the EJB results in the following Java files:

- The Java class file, which implements the EJB interface.
- The home interface, which describes how an application client (or another EJB) creates the EJB from its container.
- The remote interface, which describes the bean's methods. An application client or another EJB calls methods defined in the remote interface to perform the business logic implemented by the bean.
- The helper class, which helps clients create a bean and handles any `out` parameters specified in the interface.
- The XML deployment descriptor.

In addition to the Java files, generating the EJB also produces a batch file that is used to prepare the EJB for deployment. In Windows, this files ends with a `.bat` extension and in UNIX `.sh`.

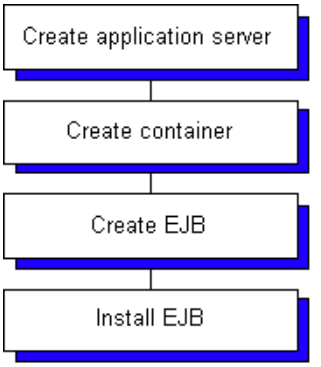
Sample Enterprise JavaBeans

The Panther distribution contains sample EJBs. For more information about the EJB samples, refer to Appendix B, “Sample Applications.”

CHAPTER

6 Deploying Enterprise JavaBeans in WebSphere

This chapter describes the process for deploying Enterprise JavaBeans in WebSphere Application Server.



Installing Enterprise JavaBeans

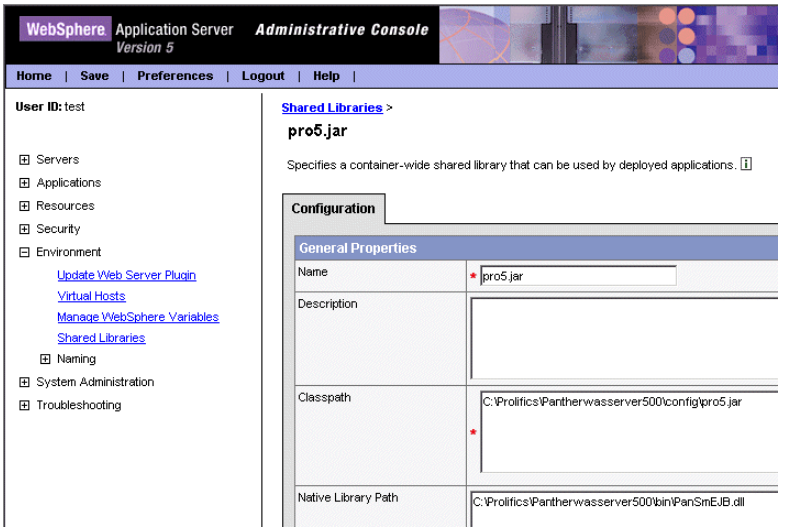
How to Install an EJB in WebSphere

1. Start the WebSphere Administrative Console.
2. Create an application server (if one does not already exist for the application).

   For the steps to create an application server and a container, refer to page 2-6, "Creating an Application Server."
3. Add a shared library for `pro5.jar` and set the Classpath and Native Library Path properties for this jar file.
   - In the tree menu on the left, select Environment→Shared Libraries.
   - In the main window, choose New and enter the name `pro5.jar`.
   - For the Classpath property, specify the location of `pro5.jar`. (The jar file is located in the Panther server installation.) For example:

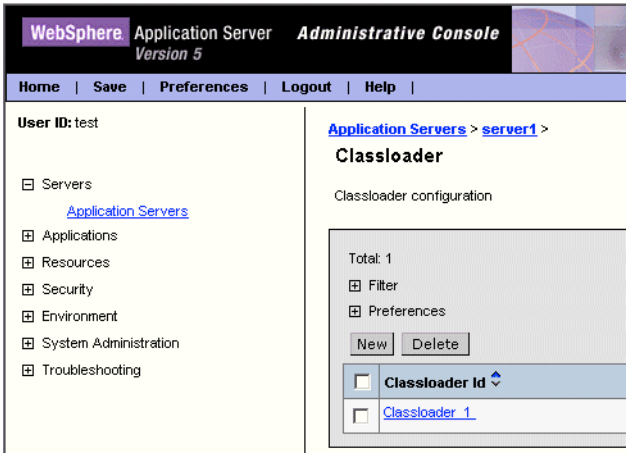
     ```
     C:\Panther\Pantherwasserver500\config\pro5.jar
     ```
   - For the Native Library Path property, specify the location of the `PanSmEJB` shared library (in the Panther server installation). For example:

     ```
     C:\Panther\Pantherwasserver500\bin\PanSmEJB.dll
     ```
   - Press Apply and OK to save these settings. When complete, the Administrative Console lists the shared library.



4. Add the library to the classloader so that `pro5.jar` will be loaded when the application server starts.

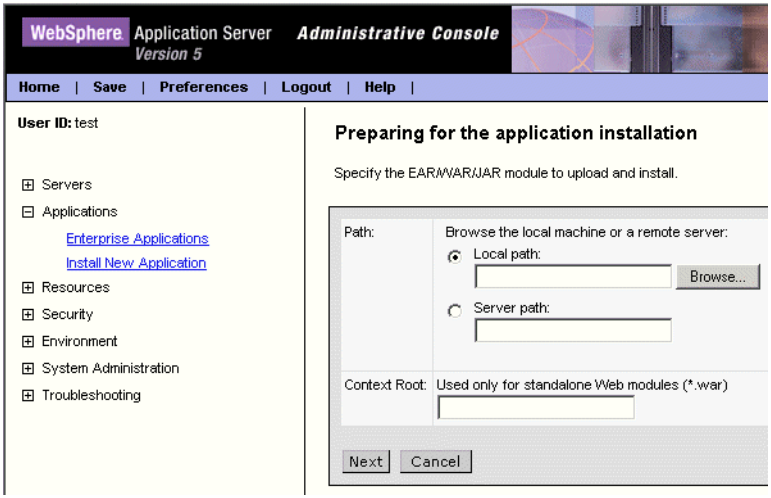
   - In the tree menu on the left, select Servers→Application Servers→*ServerName*.
   - In the main window, choose the server and select classloader to specify this property.



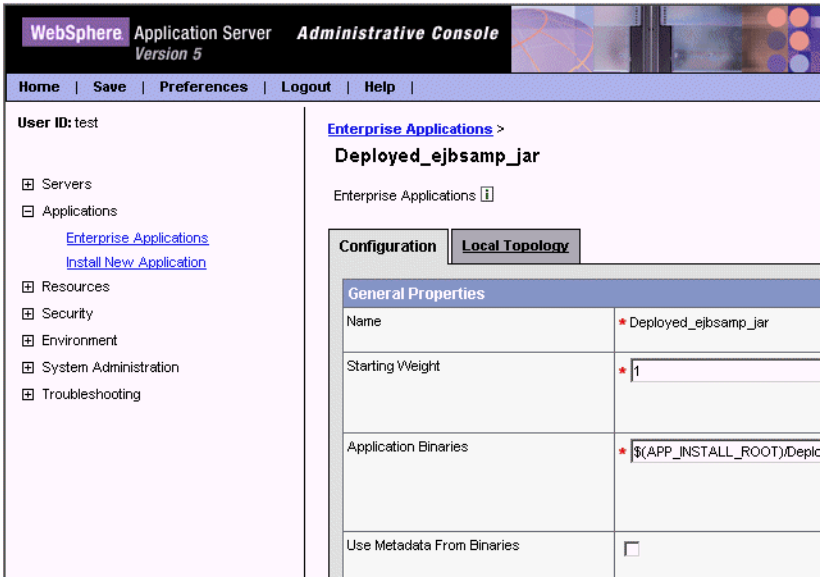
   - In the classloader, choose New→Libraries and select `pro5.jar`.

- Press Save.

5. Install the application.
   - In the tree menu on the left, select Applications→Install New Application.
   - In the main window, choose the jar file you want to deploy and install.
   - Press Next.
   - Enter any binding and mapping information if required. Then press Next.
   - Enter any installation options. Then press Next.



6. Creating the application generates a deployed jar file which should be added to the `CLASSPATH` in your environment.

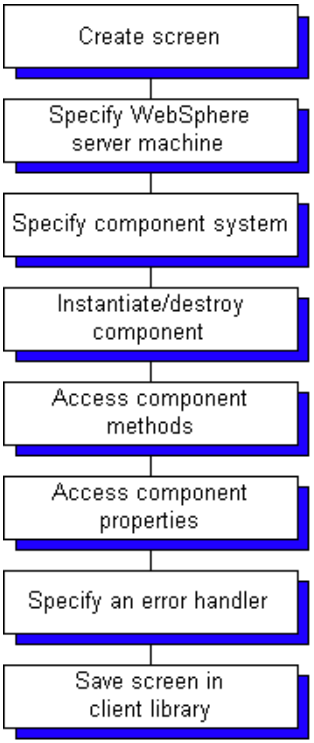


7. Copy the `panther.ini` and `server.lib` into the application server's working directory. (This directory was created as parts of the steps to install and configure Panther for IBM WebSphere in Chapter 2.)

CHAPTER

7 Building Client Screens

This chapter describes how client screens in your Panther application can access Enterprise JavaBeans. A series of C functions, also callable in JPL and through Panther's Java interface, are available for interacting with all Panther service components.



Creating Client Screens

Client screens are the part of the application accessed by users. They can include fields for data, push buttons for performing actions, selection groups for assembling options, option menus for listing choices. These and other user interface options are described in *Using the Editors*.

Refer to "Creating a New Screen" on page 7-2 in *Using the Editors* for steps on creating a new screen.

Specifying the WebSphere Server

You need to specify the machine running WebSphere. At runtime, the value is derived from the `provider_url` application property. If the `SMPROVIDERURL` environment variable is set, this property is initialized with this value. However, you can also set this property programmatically. List the server machine's host name and port number (`iiop://`*`hostName`*`:`*`portNumber`*).

If you specify the value in JPL, remember to use double colons. For example, the following JPL command would set the property back to its default value:

```
@app()->provider_url="iiop:://localhost::900"
```

Specifying the Component System

Before instantiating a component, you must first specify which type of service components are currently in use with the `current_component_system` property: `PV_SERVER_EJB` for EJBs or `PV_SERVER_COM` for COM components.

Instantiating an EJB

After the component system is specified, you instantiate a component by calling `sm_obj_create`. It takes a string parameter, the name of the component.

This function returns an object ID for the specified component or `PR_E_OBJECT` if it fails. Using this object ID, you can access the component's properties and methods.

Sample Component Horoscope

In the horoscope sample, when a user submits the screen, the following JPL procedure specifies the component system and instantiates the EJB:

```
proc submit()
{
    vars id,res

    @app()->current_component_system = PV_SERVER_EJB
    id = sm_obj_create("fortuneEJB")
    if (id<=0)
    {
        msg emsg "Can't create EJB"
        return
    }
    res = sm_obj_call \
        (id,"newFortune",birthdayText,resultTextBox)
    if (res<0)
        msg emsg "Method failed"

    call sm_obj_delete_id(id)
    return
}
```

Destroying EJB Components

After invoking and working with the methods or properties of a component, you should destroy the component by calling `sm_obj_delete_id`. This function takes one parameter: the object ID for the component you wish to destroy. Otherwise, the component will continue to exist until the application terminates (or goes from test to edit mode).

Accessing the Component's Methods

To access EJB methods, you need to know the component's parameters and call the function `sm_obj_call`. The syntax in JPL is as follows:

```
ret = sm_obj_call (objid, methodName, parm1, parm2, ....)
```

The function's first parameter *`objid`* is the object ID of the component whose method you wish to use. The second parameter *`methodName`* is the name of the method you are calling. The rest of the parameters (*`parm1, parm2, ....`*) are a comma-separated list of the parameters to the method itself.

Specifying the Method's Parameters

EJBs can take three kinds of parameters: `in` parameters, `out` parameters and `in/out` parameters. Parameters can be passed as literal strings or using the property API syntax. For `out` and `in/out` parameters, Panther assigns the returning values to the variables, fields or properties originally specified.

Sample Component Employee

For example, a component called `Employee` supports a method called `NewEmployee`, which takes three parameters in the following order:

- An `out` parameter–`EmpId`, which places its return value in a field on the client screen.
- An `in/out` parameter–`Emanate`, which derives its input value from a field on the client screen.
- An `in` parameter–`StartDate`, which derives its input value from a field on the client screen.

You can invoke `NewEmployee` method with the following JPL:

```
vars id, ret
@app()->current_component_system=PV_SERVER_EJB
```

```
id = sm_obj_create ("Employee")

ret = sm_obj_call (id, "NewEmployee", \
    EmpId, Emanate, StartDate)
```

In addition to the out `parameters`, this method call returns a value. `ret` contains the return value for the method.

Accessing the Component's Properties

Properties in EJBs can contain application state information. You can use the `sm_obj_set_property` and `sm_obj_get_property` functions to access properties. The following example sets a property on the component associated with the `id` variable:

```
ret = sm_obj_set_property(id, "PropName", "PropSetting")
```

Designating an Error Handler

You can define an error handler for method invocations, such as in the following example:

```
call sm_obj_onerror ("ErrorHandlerName")
```

The string passed to `sm_obj_onerror` is the name of a function that you want to designate as the error handler. If an operation (method call, property access, or object invocation) generates a negative exception code, the error handler function is called. The specified function is passed three parameters: the error number in decimal format, the error number in hexadecimal format, and a description of the error.

(On the server side, methods can also return exception codes using the JPL verb `raise_exception` or its C equivalent `sm_raise_exception`.)

Writing a Java Event Handler

Instead of using JPL, you can use Java to implement the client processing. In the screen's Java Tag property, enter the name of the screen's Java event handler.

For more information about Java event handlers, refer to Chapter 21, “Java Event Handlers and Objects,” in the *Application Development Guide*.

Sample Java Event Handler: Client Screen

In the following sample which instantiates and destroys the component, the client screen calling this event handler has a Java Tag of `ClientScreen`.

```
import com.prolifics.jni.*;

public class ClientScreen extends ScreenHandlerAdapter{

    public void screenEntry(ScreenInterface s, int context){
           FieldInterface id = s.getField("id");
       FieldInterface id1=s.getField("id1");
       CFunctionsInterface cFuncs = s.getCFunctions();
       ApplicationInterface appface=s.getApplication();
       ScreenInterface tscr=appface.getScreen();
    int a=appface.set_int
           (Constants.PR_CURRENT_COMPONENT_SYSTEM,
            Constants.PV_SERVER_EJB);
    id1.itofield(a);
    id.itofield(cFuncs.sm_obj_create("cStrings"));
    }

    public void screenExit(ScreenInterface s, int context){
       CFunctionsInterface cFuncs = s.getCFunctions();
       FieldInterface id = s.getField("id");
       cFuncs.sm_obj_delete_id(id.intval());
```

Sample Java Event Handler: Push Button

On this client screen, a button event handler for the Search push button calls the method and gets the number of returned rows. The push button calling this event handler has a Java Tag of `SearchButtonHandler`.

```
import com.prolifics.jni.*;

public class SearchButtonHandler extends ButtonHandlerAdapter{

    public int buttonValidate
           (FieldInterface f, int item, int context){
       ScreenInterface s = f.getScreen();
       FieldInterface idField = s.getField("id");
       FieldInterface rowField = s.getField("RowCount");
       FieldInterface searchField = s.getField("search");
       FieldInterface
           companyNameField = s.getField("CompanyName");
       CFunctionsInterface cFuncs = f.getCFunctions();

       companyNameField.putfield(1,searchField.getfield());
       int id = idField.intval();
       String i = cFuncs.sm_obj_call("(" + id + ",
           'GetCustomer',CompanyName,CustomerID,Phone)");
       String st = cFuncs.sm_obj_get_property
           ( id, "RowCount");
       rowField.putfield(st);
       return id;
   }
}
```

Saving Client Screens

Client screens are stored in an application library available to the client executable. The default name of the application library is `client.lib`.

Refer to “Saving an Application Component” on page 2-16 in *Using the Editors* for information about saving screens to application libraries.

Sample Client Screen

Although simple in appearance, this screen contains the fields and push buttons needed to operate the client part of an application and illustrates the concepts described in this chapter.

However, apart from the component system specification, nothing in the client interface indicates that it is calling an Enterprise JavaBean. In fact, the same code could be used to call a COM component. The component system specification determines which type of component gets instantiated.

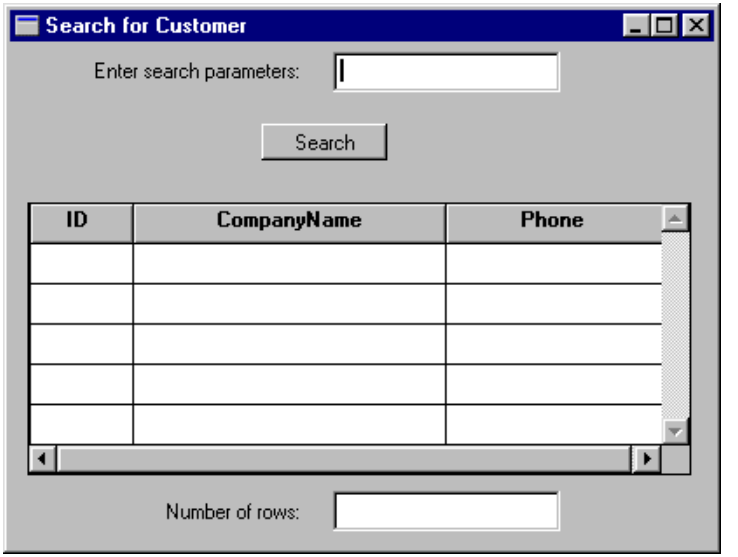


Figure 7-1 This client screen allows users to look up company names and telephone numbers.

Sample Component: Customer

The unnamed JPL procedure creates the variable which will hold the object ID of the EJB. During screen entry, the EJB is instantiated.

```
vars id

proc enter
{
@app()->current_component_system=PV_SERVER_EJB
id = sm_obj_create("Customers")
}
```

On exiting the screen, the EJB is destroyed.

```
proc exit
{
call sm_obj_delete_id(id)
}
```

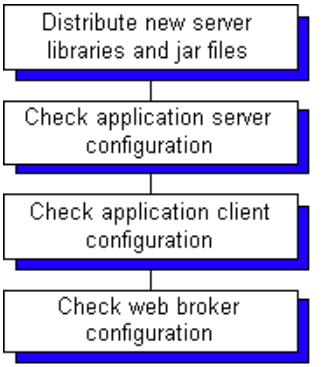
This JPL procedure calls the EJB's `GetCustomer` method and gets the value of the `RowCount` property:

```
proc do_search
{
vars error
CompanyName[1] = search
error = sm_obj_call (id, "GetCustomer", \
    CompanyName, CustomerID, Phone)
rowcount = sm_obj_get_property (id, "RowCount")
}
```

The Panther distribution contains additional EJB samples. For a full description, refer to Appendix B, "Sample Applications."

CHAPTER

8 Deploying Your Application



Packaging the EJBs

As you develop and test EJBs in your application, the EJBs can be in separate jar files. When you are ready to deploy your application, you will want to package a group of EJBs in a single jar file.

How to Package EJBs

1. Delete the test versions of the EJBs from WebSphere Application Server.
2. Run the `makeejb` utility on an application library. This utility generates the EJBs, compiles them, and packages them in a deployable jar file.
3. Re-install and redeploy the EJBs in WebSphere Application Server.

Configuring the Application Server

The section "How to Set Up the Application Server Engine" on page 2-1 describes the process for setting up an application server machine. This process is summarized here as a runtime checklist.

At runtime, the application server must have:

- `$SMBASE/config/pro5.jar` and `$SMBASE/servlet/proweb.jar` included in its `classpath` setting. This can be set in the WebSphere Administrative Console.
- The Panther shared libraries must be in the system's `PATH`. For Windows, the Panther installer places them in the Windows system directory. For UNIX, they must be added to `LD_LIBRARY_PATH` or `LIBPATH`.
- The EJBs must be deployed and installed in WebSphere Application Server.
- The following files are in the application server's working directory:
  - `panther.ini`, the Panther initialization file, specifies the Panther installation location, license information, the application libraries for the server, database shared library information, and other Panther settings.
  - The application library containing the Panther service components. The default setting opens `server.lib`. To access another application library, it must be specified in `panther.ini`.

- The Panther-generated jar file is also in this directory, but is no longer needed once the EJB has been deployed and installed in WebSphere.

Setting the Number of Application Servers

A Panther/WebSphere application consists of a group of Panther-built EJBs deployed in WebSphere Application Server. These EJBs can share an application server process, a container, a Java package, a Panther application library, a jar file, and a `panther.ini` file with its global and class settings.

If needed, you can distribute the application among server processes. All Panther-built EJBs must have the same global settings in `panther.ini` and access the same database drivers.

Configuring Runtime Clients

The section “How to Set Up the Development Client” on page 2-8 describes the process for setting up development clients. This process is summarized here for runtime clients.

The runtime client must have:

- *`deployedEJB`*`.jar`
- Panther application client library (for example, `client.lib`)
- WebSphere Developer's Client files
- Oracle's JVM (available on the WebSphere software CD)
- Oracle's JDK/JVM setting for `SMJAVALIBRARY` (for example: `%JAVA_HOME%\jre\bin\classic\jvm.dll`)
- Settings for the JVM in `SMJVMOPT`

- CLASSPATH in the Environment section of prol5w32.ini; prol5w64.ini or prol5unix.ini with:
  - $SMBASE/config/pro5.jar
  - $WAS_HOME/properties
  - $WAS_HOME/lib/bootstrap.jar
  - $WAS_HOME/lib/j2ee.jar
  - $WAS_HOME/lib/lmproxy.jar
  - $WAS_HOME/lib/urlprotocols.jar
  - *deployedEJB*.jar
  - Java event handlers for the client screens (if applicable)
- provider_url with the server machine's host name and port number (iiop://*hostName*:*portNumber*), which can be initialized from SMPROVIDERURL

Configuring the Web Application Broker

The section "How to Set Up the Web Application Broker" on page 2-13 describes the process for setting up an HTTP server machine. This process is summarized here as a runtime checklist.

At runtime, the HTTP server must have:

- *deployedEJB*.jar
- Panther application client library (for example, client.lib)
- WebSphere Developer's Client files
- Oracle's JVM
- Panther web initialization file (*myApp*.ini) located in ~proweb/ini on UNIX servers and the Windows directory (for example, C:\Winnt) for Windows servers with:

- `JAVA_HOME`
- `WAS_HOME`
- `PATH` with:
  - *`PantherInstallDir`*`/util`
  - *`JavaInstallDir`*`/bin`
  - *`WebSphereInstallDir`*`/bin`
- (UNIX only) `LD_LIBRARY_PATH` or `LIBPATH` with:
  - *`PantherInstallDir`*`/lib`
  - *`WebSphereInstallDir`*`/bin`
  - JVM shared libraries
- `CLASSPATH` with:
  - *`PantherInstallDir`*`/config/pro5.jar`
  - *`PantherInstallDir`*`/servlet/proweb.jar` (for Java servlets)
  - *`DeployedEJBDirectory/deployedEJB`*`.jar`
  - `$WAS_HOME/properties`
  - `$WAS_HOME/lib/bootstrap.jar`
  - `$WAS_HOME/lib/j2ee.jar`
  - `$WAS_HOME/lib/lmproxy.jar`
  - `$WAS_HOME/lib/urlprotocols.jar`
- `SMPROVIDERURL` with the server machine's host name and port number (`iiop://`*`hostName:portNumber`*)
- `SMJVMOPT` with the JVM's options.

Note: Web initialization files must contain the full pathname for Panther, WebSphere and Java installations. Environment variables cannot be expanded.

APPENDIX

A Utilities

This chapter describes command-line utilities that can help you develop and manage a Panther WebSphere application. Utility descriptions are organized into the following sections, as applicable:

- Utility name and brief description.
- Syntax line and argument descriptions.
- Description of the utility.

To get a command-line description of a utility's available arguments and command options, type the utility's name with the `-h` switch. For example:

```
makeejb -h
```

makeejb

Generate the service component's EJB and associated files

```
makeejb [-inv] library [library...]
```

`-i`	Generate one jar file per component.
`-n`	Do not run the generated deployment scripts.
`-v`	Display information in verbose mode. Lists service components being processed.
`library`	Name of library

Description

The `makeejb` utility generates a Java class and a deployment descriptor for each service component in a Panther application library into a single jar. With the resulting jar, it runs the deployment scripts to prepare the EJBs for deployment in Web Sphere.

APPENDIX

B Sample Applications

The distribution includes some EJB samples developed for the Panther for IBM WebSphere product. The samples are located at:

PantherInstallDir/samples/ejb

- *Banner Ad Server (Rotating Banner Ads)*—Use an Enterprise JavaBean to configure a banner in your web application.
- *Horoscope (Fortune Teller)*—See what's in store for you in business and in love.
- *Panther Store (Shopping Cart)*—See an eStore application in action.

How to Install the EJB Samples

Follow the instructions listed in:

PantherInstallDir/samples/ejb/ejbinstall.htm

Warning: Before starting the EJB Gallery server, you must first stop the Web Gallery server. Otherwise, you get the message “internal Windows error.”

Using a Web Banner

Feature:

Provide simple EJB that can be called from any client.

Description:

Users display a simple web page containing a button that submits the screen. Each time the screen is submitted with this push button, a new random ad appears on the screen. The ad consists of an image, a URL that will be invoked when the ad is clicked, and alternative text for the ad. All of this information comes from the AdServer component, which has a grid containing rows of information for possible ads.

Language(s) Used:

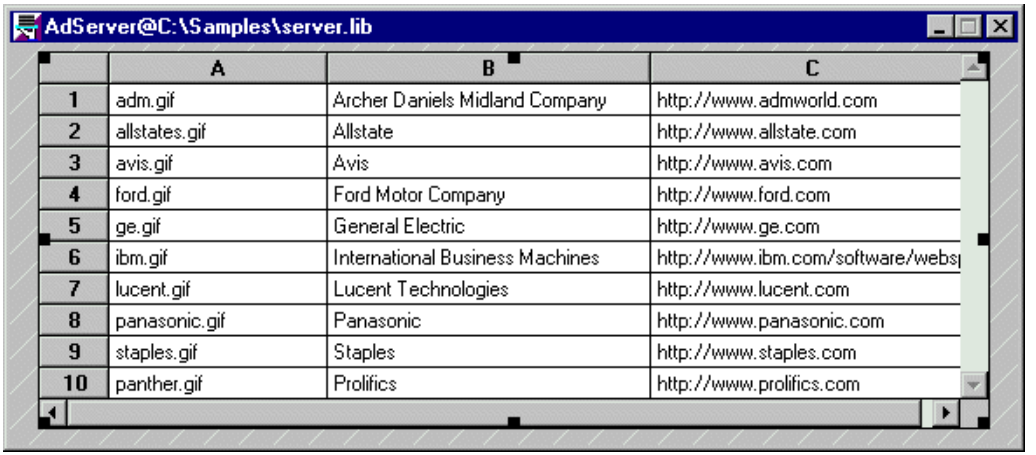
JPL

Client Screen:

Name: `adserver.scr`

Component:

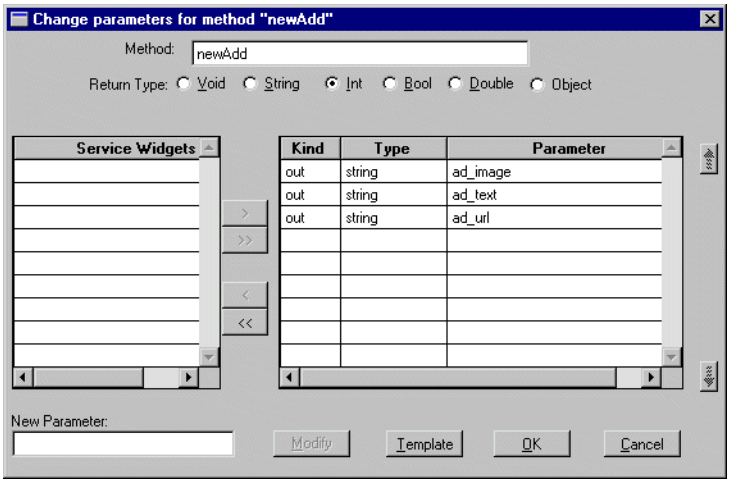
Name: `AdServer`



	A	B	C
1	adm.gif	Archer Daniels Midland Company	http://www.admworld.com
2	allstates.gif	Allstate	http://www.allstate.com
3	avis.gif	Avis	http://www.avis.com
4	ford.gif	Ford Motor Company	http://www.ford.com
5	ge.gif	General Electric	http://www.ge.com
6	ibm.gif	International Business Machines	http://www.ibm.com/software/websp
7	lucent.gif	Lucent Technologies	http://www.lucent.com
8	panasonic.gif	Panasonic	http://www.panasonic.com
9	staples.gif	Staples	http://www.staples.com
10	panther.gif	Prolifics	http://www.prolifics.com

Component Methods:

`newadd()` returns three arguments: `ad_image`, `ad_text`, and `ad_url`.



Finding Your Horoscope

Feature:

Use of Java inside client and EJB

Description:

Users display a screen where they can enter their birthday, and then click on a push button to see their horoscope.

Language(s) Used:

Java

Client Screen:

Name: `horoscope.scr`

`callfortune_pb`: Push button that invokes `newfortune()` method.

Java Files: `FortuneClient.java`, `FortuneClient.class` (in `$SMBASE/samples/ejb/ejbclient.lib`)

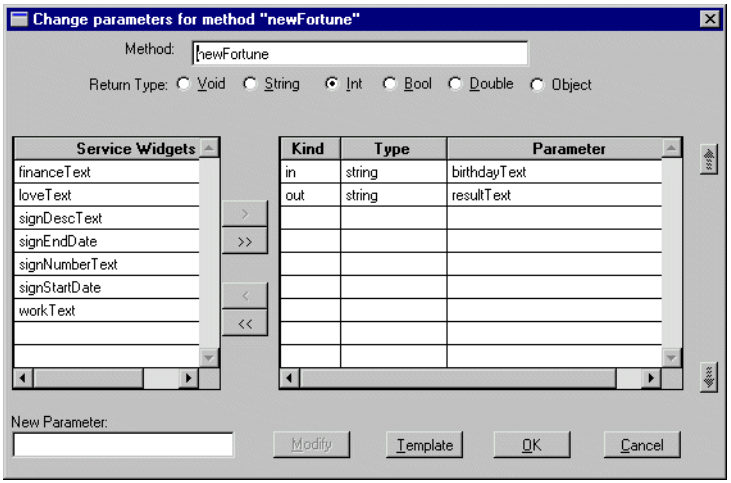
Component:

Name: `Fortune`

Java Files: `FortuneLogic.java`, `FortuneLogic.class` (in `$SMBASE/samples/ejb/server.lib`)

Component Methods:

`newfortune`()- This method is passed a date, and returns a text string (255 characters) containing a fortune, based on the Astrological Sign of the date.



Running the eStore Application

Feature:

Provide a sample EJB that makes database calls.

Description:

Users see a list of items. They can click on the items to add them to their shopping cart. When they press "Checkout," it takes them to the checkout screen, where they can enter their customer information.

Language(s) Used:

JPL

Client Screens:

`fillcart.scr`, `viewcart.scr`, `checkout.scr`

Components:

`Carts`, `Store`

APPENDIX

C Adding C Functions

This appendix discusses how to make C functions that you write available to EJBs.

How to Add C Functions for EJB Access

1. Compile the C functions into a shared library (`.dll` or `.so`).
2. Create a JPL file with calls to `sm_slib_load` and `sm_slib_install` to install and load the new shared library.
3. Save the JPL file in a Panther application library accessible to the WebSphere application server process, such as `server.lib`. (To see what application libraries are available, check the `SMFLIBS` setting in `panther.ini`.)
4. Edit `panther.ini` and add the JPL file to the `SMINITJPL` setting in the `EJB Globals` section.

   Note: `SMINITJPL` in `panther.ini` must only be used to call `sm_slib_load` and `sm_slib_install`. All other JPL commands are unavailable.
5. Make the shared library available to the system.

   For UNIX, add the shared library to `LD_LIBRARY_PATH` (Solaris) or `LIBPATH` (AIX).

   For Windows, add the DLL to the working directory, the Windows system directory, or the `PATH`.

6. Your C functions are added to the global funclist whenever EJBs initialize Panther on the application server.

Building DLL's using MSVC 6 on Windows

1. Create a new project. Select File→New. Select Project Tab→Win32 Dynamic-Link Library.
2. Enter the name of your DLL for the project name.
3. Write your C code in the `.cpp` file created with your project in the Source Files. (This is in the FileView tab on the left frame of MSVC.)
4. Include the correct header files such as `smmach.h` and `smproto.h`. The include statements for these header files have to be added to the `StdAfx.h` file in Header Files in the File View frame.
5. Now the include directory for these header files have to be added to the project. Select Tools→Options→Directories tab. Make sure that Show Directories For is set to `Include files`. (You can either double click in the window to add a new entry, or click on the yellow box.)
6. You should also include the directory for your lib files. This is the same as step 5, except for Show Directories For should be set to `Library files`.
7. Now you should specify the .lib files you need to include in the project. Select Project→Settings→Link Tab→Object/Library Files. Add the names of your libraries here. You may also need to add the files themselves to the project. Select Project→Add to Project→Files.
8. Create a `.def` file to export your functions. The `.def` file should just contain the Keyword `EXPORT` and then the name of your functions following it. Add the `.def` file to the project. Same as previous step to add lib files to project.
9. Necessary compile and linking options must be specified. Select Project→Settings→C/C++ tab. Add /`DWIN32_EJB` to Project Options.
10. Create the DLL by selecting Build→Build ***`DLLName`***.

Building Shared Libraries (.so) on UNIX

.so -- shared libs, statements in {} have to be modified.

1. Write your C code.
2. Write a `.def` file to export your functions.
3. Compile with following options:

```
-lpthread -D_REENTRANT -D__EXTENSIONS__ -G
-L{path_to_panther_lib_directory} -l{library_file}
-I{include_other_files_such_as_.def_file} -D{PLAT_NAME}  -o
{my_.so_file}
```

An example compile line for UNIX is:

```
acc -lpthread -D_REENTRANT -D__EXTENSIONS__ -G
 -L/usr/prolifics/prlwassv425/lib  -lPanSmEJB
 -Imy_dll.def -DSOLARIS2_6 -o libMyShared.so my_shared.c
```

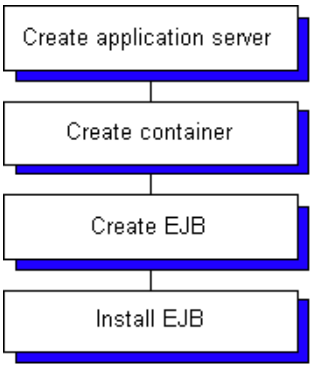
Here is a very simple `.def` file for the function `isit_there`.

```
my_dll.def:
EXPORT
isit_there
```

APPENDIX

D Deploying Enterprise JavaBeans in WebSphere 3.5

This chapter describes the process for deploying Enterprise JavaBeans in WebSphere Application Server 3.5.

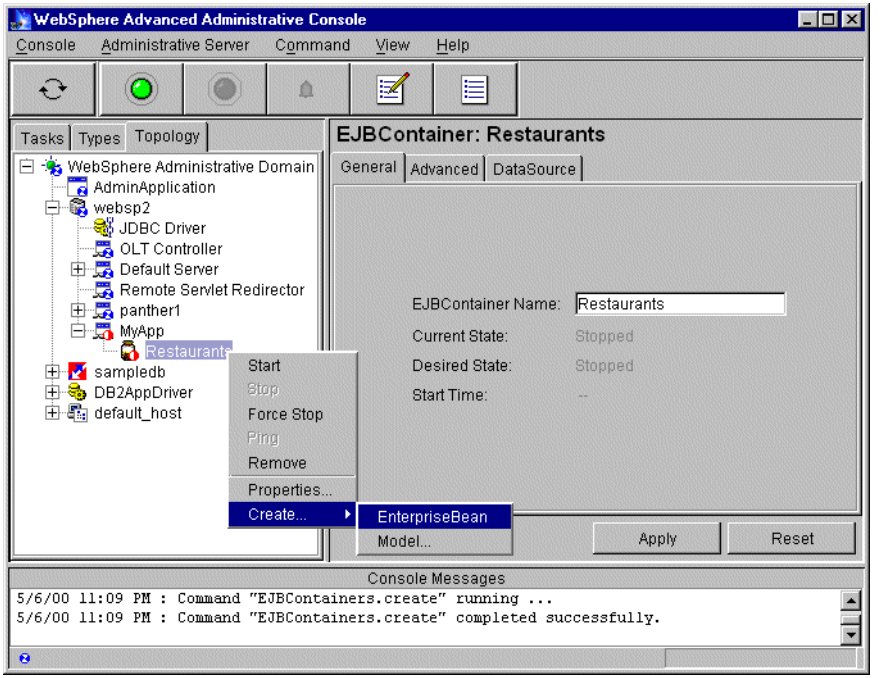


Installing Enterprise JavaBeans

How to Install an EJB in WebSphere

1. Start the WebSphere Administrative Console.
2. Create an application server (if one does not already exist for the application).

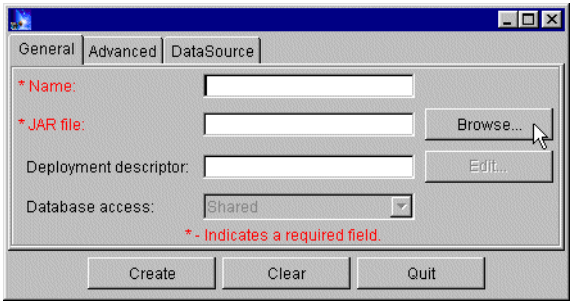
   For the steps to create an application server and a container, refer to page 2-6, "Creating an Application Server."
3. Create an EJB container (if one does not already exist).
4. With the container selected, prepare to deploy the EJB you created in Panther by right-clicking and choosing Create→Enterprise Bean.

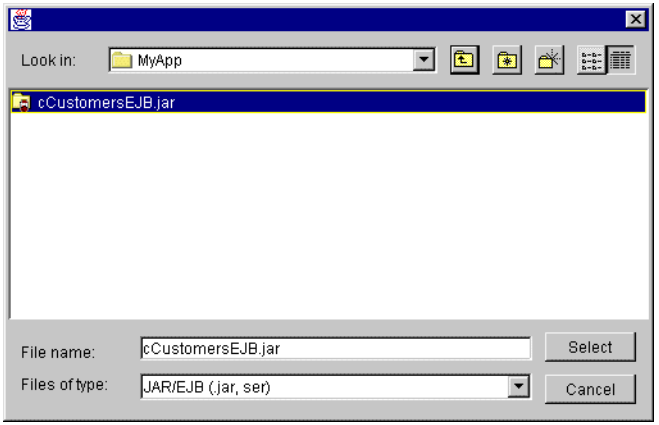


5. Choose Browse to select the jar file containing the deployable bean. (By selecting the jar file, WebSphere automatically fills in the Name field.)

   For local and native clients, the jar file is located in the directory specified in the Directory field of the EJB section of the Component Interface window.

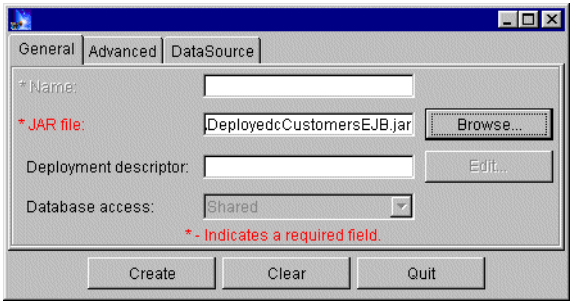
   For remote clients, the jar file is located where you ran the `makeejb` utility on the server's application library.



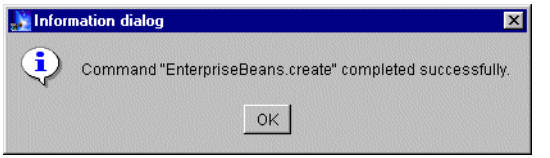


6. After selecting the jar file, WebSphere displays deployment messages for the jar and its associated beans. Choose Yes in response to the messages. WebSphere converts the Panther-generated jar file into a deployedBean jar file.

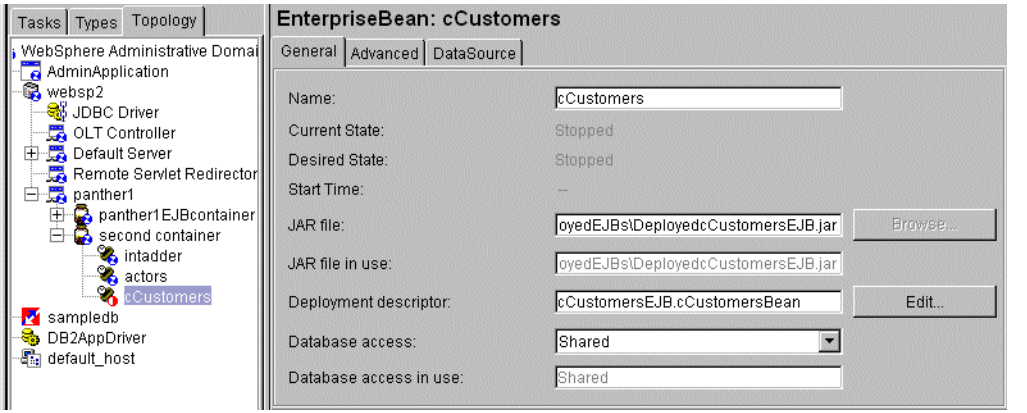
7. To install the EJB, choose Create.



8. If the bean installs successfully, WebSphere displays the following message. Choose OK.



9. The EJB is now installed in WebSphere. You need to start the bean for it to be available. For information on how to use this bean in a runtime application, refer to Chapter 8, “Deploying Your Application.”



Changes to the Jar File

As part of its EJB generation, Panther creates a jar file containing the EJB's Java files, deployment descriptor, and environment settings needed by the bean.

When you deploy and install the EJB in WebSphere Administrative Console, it creates a jar file in its Deployed EJBs directory (typically `$WAS_HOME/deployedEJBs`). This changes the name of the jar file, prepending `Deployed` to the jar file name.

Index

I

J

M

P

R

S

T

W