Developer's Guide

Netscape Extension Builder

Version 4.0, Limited Release

 Recycled and Recyclable Paper

Version 4.0

Part Number 151-07655-00

Printed in the United States of America. 00 99 98 5 4 3 2 1

Netscape Communications Corporation, 501 East Middlefield Road, Mountain View, CA 94043

Contents

Introduction 11
Syntax Conventions 11
Programming Language Conventions 11
Operating System Conventions 12

Chapter 1 About Netscape Application Server Extensions 13
What is a Netscape Application Server Extension? 13
About the NAS Application Model 14
How Extensions Relate to NAS Application Model 15
Why Build an Extension? 16
Integrate Existing Functionality 16
Integrate Third-Party Solutions 16
Implement Shared Services 17
About Building a Netscape Application Server Extension 18
Define the API 18
Design the Interfaces and Coclasses 19
Generate IDL Files 20
Generate Extension Source Code 20
Write the Implementation Code 21
Compile the Extension Source Code 21
Deploy the Extension 21
Write the Application Component 22

Chapter 2 About the Netscape Extension Builder 23
What is the Netscape Extension Builder? 23
Components of Netscape Extension Builder 24
Netscape Extension Builder Designer 24
KIDL Compiler 25
Netscape Extension Builder Runtime Features 25
Extension Make Harness 27

Chapter 3 Introduction to Netscape's Interface Definition Language 29
What is Netscape's Interface Definition Language? 29
About KIDL 30
How KIDL Corresponds to the Parts of a Netscape Extension 30
Extension Module 32
Access Module 33
Service Module 35

Chapter 4 Designing a Netscape Extension 39
About Netscape Extension Builder Designer 40
Overview of the Graphical User Interface (GUI) 40
How to Use Netscape Extension Builder Designer 43
Extension Design Flow Using Netscape Extension Builder Designer 43
Starting Netscape Extension Builder Designer 46
Creating an Extension Module 47
Creating an Access Module 48
Defining a Global Constant 49
Creating a Service Module 50
Creating an Access or Service Module by Importing an IDL File 52
Designing an Interface 53
Defining a Method 54
Designing a Coclass 57
Implementing an Interface 59

Chapter 5 Generating Output Files 67
About Generating Output Files 67
Output From Netscape Extension Builder Designer 68
Source Code Output From Running gmake 71
Generating Files from the Designer 76
Generating Source Code Using gmake 77
Source Code for a Java Extension 77
Source Code for a C++ Extension 79

Diagram of Extension Components 80

Chapter 6 Completing Method Stubs 83
Steps to Completing Method Stubs 83
Locating Stub Files 84
Locating Interface Code 85
Adding Implementation Code 87
Adding a Creation Method 88
Extending the Class Definition 88
Using Netscape Application Server Services 89
Using Extensions From Applications 90
Determining Accessor Names 90
Adding Accessor Code to Application Components 91
Linking an Application Component 92

Chapter 7 Using Template Streaming 93
Introduction to Template Streaming 93
Steps for Enabling Template Streaming 94
Example of Template Streaming 95
Setting the Template Streaming Decoration 95
Completing Stubs to Support Template Streaming 96
Template Streaming in C++ Extensions 97
Template Streaming in Java Extensions 99
Creating a Template 102
Testing an Application 103
The CZoo.java File 104

Chapter 8 Managing State and Session Information 109
About State and Session Management 109
Using Application State in an Extension 110
Creating the Root Node in a C++ Extension 111
Creating the Root Node in a Java Extension 111
Setting and Managing State Data 112

Using Sessions in an Extension 112
Local Versus Distributed State and Session Data 113
Using a Distributed Session or State 114
Using a Local Session or State 114
Example Code Walkthrough 115
Implement the IState2 Interface 116
Implement the ISession2 Interface 121

Chapter 9 Using Object Pools 125
Introduction to Object Pooling 126
Summary of Object Pooling Tasks 126
Key Concepts 127
Process Overview for Object Pooling 128
Example of Object Pooling 130
Inside a Pool-Enabled Extension 133
Creating a Connection 133
Calling a Method on a Virtual Connection 134
The Introspection Interface 135
The Object Evaluation Interface 136
Inside the Object Pool Manager 137
Matching a Pooled Object 138
Creating a Pooled Object 139
Replacing a Pooled Object 139
Queuing a Request 140
Returning a FAILURE State 140
Adding Decorations for Object Pooling 140
Creating Pooled Objects 141
Designating Objects That Can Be Pooled 144
Changing a Pooled Object's Lifetime 145
Passing a Pooled Object 147
Completing Stubs for Pooled-Object Creation Methods 150
Completing Stubs for Object Evaluation Methods 152
Implementing MatchObject( ) 152
Implementing CreateObject( ) 155

Implementing StealObject( ) 157
Implementing ReleaseObject( ) 158
Implementing InitObject( ) 159
Implementing UninitObject( ) 161
Implementing GetHint( ) 162
Reference List of Object Pooling Decorations 162

Chapter 10 Compiling the Extension Source Code 167
About Compiling the Source Code 167
Introduction to Editing Makefiles 168
Editing Makefiles on Solaris 168
Editing Makefiles for C++ Extensions 169
Editing Makefiles for Java Extensions 171
Editing Makefiles On Windows NT 172
Editing Makefiles for C++ Extensions 172
Editing Makefiles for Java Extensions 173
Configuring the make Harness 174
The make Harness 175
Inside the IDL Directory 175
Inside the cpp Directory 176
Inside the Java Directory 177

Chapter 11 Deploying and Managing a Netscape Extension 179
Deploying a Netscape Extension 179
Verifying Extension Configuration 182
Disabling an Extension 184
Adjusting Object Pooling Configurations 184

Chapter 12 Example Extension: HelloWorld 187
About the Example 187
Design the Extension 188
Open Netscape Extension Builder Designer 188
Create an Access Module 188
Create a Service Module 189
Generate IDL Files 189

Complete the Extension 189
Build the Application 190
Test the Application 192

Appendix A C++ Helper Functions 193
GXContextCreateDataConn( ) 193
GXContextCreateDataConnSet( ) 196
GXContextCreateHierQuery( ) 197
GXContextCreateMailbox( ) 199
GXContextCreateQuery( ) 200
GXContextCreateTrans( ) 201
GXContextDestroySession( ) 203
GXContextGetAppEvent( ) 204
GXContextGetObject( ) 205
GXGetStateTreeRoot( ) 205
GXContextIsAuthorized( ) 207
GXContextLoadHierQuery( ) 209
GXContextLoadQuery( ) 213
GXContextLog( ) 215
GXContextNewRequest( ) 216
GXContextNewRequestAsync( ) 218

Appendix B Java Helper Static Methods 223
CreateDataConn( ) 223
CreateDataConnSet( ) 227
CreateHierQuery( ) 228
CreateMailbox( ) 230
CreateQuery( ) 231
CreateTrans( ) 233
DestroySession( ) 234
GetAppEvent( ) 235
GetObject( ) 236
GetStateTreeRoot( ) 236
IsAuthorized( ) 237
LoadHierQuery( ) 239

LoadQuery( ) 242
Log( ) 244
NewRequest( ) 245
NewRequestAsync( ) 248

Appendix C Java Class Decorations 253

Appendix D Reserved Words 255

Appendix E The ConnManager.cpp File 257

Glossary 273

Index 281

Introduction

The *Developer's Guide* explains how to build extensions, which are services residing in Netscape Application Server. Extensions enable legacy systems, third-party solutions, and shared services to be integrated into Netscape Application Server applications without the need to rebuild the original functionality.

Developer's Guide is broken into the following parts:

- Syntax Conventions
- Programming Language Conventions
- Operating System Conventions

Syntax Conventions

Symbol	Indicates
[ ]	An optional item
< >	A term that you substitute with an actual value
…	Place holder for commands not listed in an example

Programming Language Conventions

You can build an extension using either Java or C++. Some tasks are significantly different depending on the language used. In this case, the topics are usually presented in separate sections.

For other tasks, the basic approach is the same for both languages, but there are slight differences in syntax. When this is the case, text is marked as applying specifically to Java or C++. For example, paragraphs that distinguish between C++ and Java implementation of accessors would read as follows:

C++ The global function is KET_Get_<service_module> and the function signature is found in <NEB_ROOTDIR>\extensions\include\access_<service_module>.h.

Java The static method is get<service_module> in class access_<service_module>.

Operating System Conventions

You can build extensions on Unix or Windows NT. Differences between the two operating systems are explained as needed. In many cases, the only difference is the specification of pathnames, which require a forward slash (/) on Unix and a backward slash (\) on Windows NT. *Developer's Guide* uses the following conventions for pathnames on either operating system:

Pathname	Indicates
<NEB_ROOTDIR>	Directory in which you installed NEB.
<NAS_ROOTDIR>	Directory in which you installed NAS.

The Windows NT naming convention is used when describing files that are equivalent on both operating systems—that is, when the only difference is the pathname separators. For example:

```
<NEB_ROOTDIR>\my_extension\cpp\CClass.cpp
```

Chapter

1

About Netscape Application Server Extensions

This chapter describes Netscape Application Server extensions, which provide an important service in applications by allowing Java application components to access third-party solutions, legacy systems and shared services.

The following topics are included in this chapter:

- What is a Netscape Application Server Extension?
- About the NAS Application Model
- Why Build an Extension?
- About Building a Netscape Application Server Extension

What is a Netscape Application Server Extension?

A Netscape Application Server (NAS) extension provides an important service in Netscape Application Server applications by allowing Java application components to access third-party solutions, legacy systems and shared services. A Netscape Application Server extension allows these technologies, which do not typically conform to the Netscape Application Server architecture, to be integrated into Netscape Application Server applications without rebuilding the

original functionality. In addition, Netscape Application Server extensions provide a bridge between different architectures, allowing older technology to take advantage of new technology, often adding performance and reliability to existing applications.

A Netscape Application Server extension is similar to the Netscape Application Server application services built into the server, such as the Data Access Service, the State and Session Management Service and the Transaction Management Service. Both Netscape Application Server extensions and Netscape Application Server services are accessed by Java application components through exposed interfaces. The main difference between a Netscape Application Server extension and the Netscape Application Server services is that you must define what the extension does whereas the services are defined for you.

Netscape Application Server extensions can be written in C++ or Java. Java extensions can be accessed from Java clients, which are application components written in Java. Similarly, C++ extensions can be accessed from C++ clients, which are AppLogic objects written in C++. Java clients can also access C++ extensions, provided you enable the Java Access Layer for the C++ extensions. More details about choosing the extension language and using the Java Access Layer are provided in later sections of this guide.

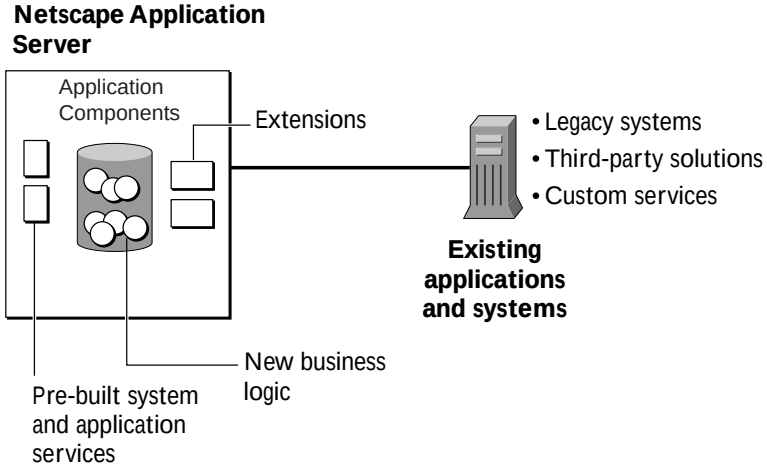
About the NAS Application Model

The NAS application model is the conceptual division of a software application into functional components. For Java applications, the model incorporates standard components based on Java technologies such as servlets, JavaServer Pages (JSP), Enterprise JavaBeans (EJB) and any other Java class.

If an application is written C++ or if it will run in the NAS 2.x environment, application components include AppLogics. An AppLogic is a set of programming instructions, written in C++ that perform a well-defined, modular task within an application.

How Extensions Relate to NAS Application Model

A Netscape Application Server extension is a middle-layer service that resides within Netscape Application Server. An extension fits between components of an application and the piece of software code you are integrating into the Netscape Application Server application. Application components call into the extension using the exposed interfaces and corresponding method calls. These method calls then map into the appropriate part of the extended technology. The following diagram illustrates the location of the Netscape Application Server extension relative to the application components (the circles in the center core of Netscape Application Server) and the existing technology, whether a third-party or legacy solution.



The illustration depicts the legacy or third-party technology as separate from Netscape Application Server. This is because such technology is outside of the Netscape Application Server context. The Netscape Application Server context is where services are loaded into Netscape Application Server so the server can interact with those services.

Access to the third-party or legacy technology is provided through the Netscape Application Server extension, which, like the pre-built Netscape Application Server services, is a part of Netscape Application Server context. A Netscape Application Server extension becomes part of the Netscape Application Server context through a manager class, which is explained later in this guide.

Why Build an Extension?

Building a Netscape Application Server extension is useful for quickly and efficiently integrating existing functionality and reusable services into Netscape Application Server so you can provide that functionality or service to users through a high performance Netscape Application Server application. A Netscape Application Server extension enables you to extend the functionality of Netscape Application Server to do the following:

- Integrate Existing Functionality
- Integrate Third-Party Solutions
- Implement Shared Services

Integrate Existing Functionality

Netscape Application Server extensions provide a means of integrating legacy code and existing client/server solutions into a new Netscape Application Server application. This can save you tremendous time and money because you do not have to re-create these solutions when building a new Netscape Application Server application.

For example, you might have a CICS/mainframe application providing services to your company's employees. This application is running on expensive hardware and required many engineering hours to build. The investment in this application is not trivial and warrants concern about migrating to a new application server technology. To preserve this investment, you can build a Netscape Application Server extension that provides the layer that allows application components to call into the legacy code.

Integrate Third-Party Solutions

Third party companies have developed many of the necessary technologies used in Web applications, such as credit card billing, membership and order-fulfillment services, and so on. Rather than re-create these technologies within Netscape Application Server application components, you can encapsulate

access to this functionality within a Netscape Application Server extension and allow the application components to call into the third-party technology through the extension.

For example, a typical electronic-commerce application consists of many technologies such as credit card billing, customer fulfillment, order verification and tracking, and database access and record retrieval. For such an application, Netscape Application Server provides the best technology for user access, verification of whether a product is available (through database access), and customer order tracking. Netscape Application Server also provides many performance enhancing functionalities such as load balancing and ease of scalability.

For the credit card billing and customer fulfillment services, this functionality already exists. Therefore, it makes more sense to extend that functionality to the Netscape Application Server application, rather than build it again.

Implement Shared Services

Building a Netscape Application Server extension is also useful for implementing core functionality as a service that is accessible to multiple applications and across multiple servers.

The following list highlights key features that make extensions more appropriate than application components for implementing services:

- Extensions are multi-threaded, whereas application components are single threaded. Therefore, multiple clients can access the same instance of an extension at the same time.
- Extensions are persistent, living in the server upon startup and until shutdown. This gives you the opportunity do expensive initiation operations at server startup instead of at application component initialization, when the client is waiting for a reply.
- Extensions provide a programmatic interface to clients, which are typically application components but could be other extensions. Application components rely on a ValList object, a list of value/key pairs, to pass information to clients. While great for HTML access, a ValList does not perform as well as a programmatic method call.

- Extensions have access to object pooling, an important value-added feature that is not easily accessible to application components. Object pooling provides shared use of expensive resources, such as connections, and is one of several Netscape Extension Builder runtime features that are pre-built for easy implementation in your extension. The Netscape Extension Builder runtime features are discussed in a later section.

Extensions should be used as base services that are independent of any presentation logic, such as input validation or calls to the Netscape Application Server Template Engine. Use application components as the presentation layer between the client and the service or services you are implementing in an extension.

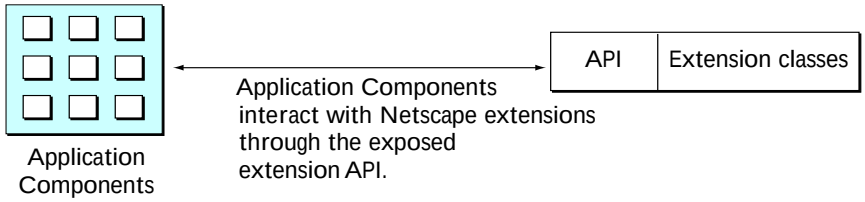
About Building a Netscape Application Server Extension

This section provides an overview of the process of building a Netscape Application Server extension. The details of these procedures follow in the subsequent sections of this guide.

Define the API

The first step in creating a Netscape Application Server extension is to define the API that will be available to the Netscape Application Server application components and, possibly, other extensions. The interfaces of this extension API are the access points between the application components and the technology being integrated.

For example, the interfaces in legacy and third-party technologies that are exposed to clients should be redefined in the Netscape Application Server extension. By doing so, you make that technology accessible to application components, and/or other extensions, as depicted in the following illustration:



Also depicted in the illustration are the extension classes, called coclasses at design time, that compose the remainder of the extension. Coclasses implement the interfaces you define.

Design the Interfaces and Coclasses

The next step in creating a Netscape Application Server extension is to design the interfaces that compose the extension's API as well as design the coclasses that become the extension classes.

Interface and coclass design can be encapsulated by the following procedural overview:

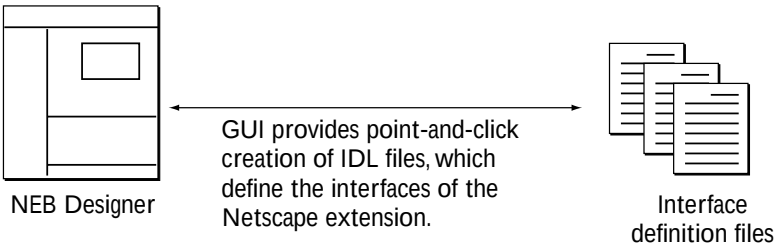
1. Start by defining the name of the interfaces and the names of the methods and parameters exposed by each interface.
2. Define the names of the coclasses that will become the extension classes.
3. Specify the interface or interfaces that each coclass implements.

The interfaces and coclasses are defined in code written with the Netscape Application Server Interface Definition Language. For more information about this language and how it is used to define the components of a Netscape Application Server extension, see Chapter 3, "Introduction to Netscape's Interface Definition Language."

Netscape Extension Builder contains a GUI tool, Netscape Extension Builder Designer, to generate this interface code for you based on selections you make in the designer. You do not need to know how to write code with Netscape Application Server IDL. Designing the interfaces and coclasses with Netscape Extension Builder Designer is described in Chapter 4, "Designing a Netscape Extension."

Generate IDL Files

After you design the extension, you are ready to generate the IDL files that will later become the source code for the extension. Netscape Extension Builder Designer contains an IDL file generator that automatically writes the IDL for the extension and produces the necessary files for completing the extension. The following illustration shows how Netscape Extension Builder Designer generates interface and coclass IDL files:



Generate Extension Source Code

Next, you generate source code from the IDL files. Netscape Extension Builder Designer's file generator creates a make harness that you use to run the KIDL Compiler, included in Netscape Extension Builder. Running the KIDL Compiler takes the IDL files generated by Netscape Extension Builder Designer and translates them into source code and method stub files and places those files in their appropriate directories. The method stub files are where you will write the code that defines what the method does.

Also generated from the interface and coclass IDL files are value-added classes provided by Netscape Extension Builder, called runtime features, which you might have enabled for your extension. More information about the runtime features is provided in a later section. The source code files and directories and runtime feature classes are discussed in more detail in Chapter 5, "Generating Output Files."

Write the Implementation Code

After you generate the source code, you write the implementation code in the method stub files. The implementation code provides the functionality to link calls made by Netscape Application Server application components, or other extensions, to the technology you are extending. Typically, this is the only code you have to write when building a Netscape Application Server extension. Information about completing method stubs is covered in more detail in Chapter 6, “Completing Method Stubs.”

Compile the Extension Source Code

After you write the implementation code, use the make harness to compile the completed method stubs and other source-code files. Compiling the extension source code is done using the makefiles created by Netscape Extension Builder Designer as well as a standard C++ or Java compiler.

The makefiles keep track of the source code files that compose your extension, and this information is fed into the compiler at the appropriate time to complete the extension. You do not have to keep track of each and every file and manually compile them. More information about compiling the extension source code is provided in Chapter 10, “Compiling the Extension Source Code.”

Deploy the Extension

Once you have completed and compiled the method stubs and source-code files, you are ready to deploy the extension. Deploying the extension loads the extension run-time files onto Netscape Application Server or servers, registers the extension on those servers, and enables the extended technology to be used in Netscape Application Server applications.

Extensions can be built on Windows NT or on Unix. Java extensions built on one operating system can be deployed to a Netscape Application Server on another operating system. C++ extensions can be designed on Windows NT, but the project file must be moved to Unix for subsequent code generation, compilation, and deployment.

Procedural information about deploying extensions is provided in Chapter 11, “Deploying and Managing a Netscape Extension.”

Write the Application Component

To access the extended technology, Netscape Application Server application components must be written to call into the extension, using the exposed extension interfaces you designed. As appropriate, the application components call into the extension through the methods contained in those interfaces. At this level, therefore, developing the application components that call into the extension is no different than developing application components that use, for example, the data access engine.

Chapter

2

About the Netscape Extension Builder

This chapter describes the Netscape Extension Builder, a collection of tools and services provided so that you can quickly design, build and deploy Netscape Application Server Extensions.

The following topics are included in this chapter:

- What is the Netscape Extension Builder?
- Components of Netscape Extension Builder

What is the Netscape Extension Builder?

Netscape Extension Builder allows you to create Netscape Application Server extensions which extend the functionality of the Netscape Application Server to integrate existing client/server and third-party solutions. For more information about extensions, see “What is a Netscape Application Server Extension?” on page 13.

This section describes the components of Netscape Extension Builder and what services each component provides to you as you design, build and deploy a Netscape extension.

Components of Netscape Extension Builder

Netscape Extension Builder is composed of the following components:

- Netscape Extension Builder Designer
- KIDL Compiler
- Netscape Extension Builder Runtime Features
- Extension Make Harness

Netscape Extension Builder works in conjunction with Netscape Application Builder to allow fast development and deployment of Netscape extensions. You use Netscape Extension Builder for development of the Netscape extensions, whereas Netscape Application Builder is used for deploying the completed extension.

More information about deploying Netscape extensions is provided in Chapter 11, “Deploying and Managing a Netscape Extension.”

Netscape Extension Builder Designer

Netscape Extension Builder Designer is a graphical user interface (GUI) tool that guides you through the process of defining the interfaces and coclasses that compose an extension. Interfaces and coclasses are specified using Netscape’s Interface Definition Language, called KIDL. For more information about KIDL, see Chapter 3, “Introduction to Netscape’s Interface Definition Language.”

Netscape Extension Builder Designer generates the IDL for you from the choices you make in the tool. The output of Netscape Extension Builder Designer is a set of IDL files, so named because they contain IDL code and have a .idl suffix.

Netscape Extension Builder Designer also generates a source code directory tree and makefiles, which are used to compile the IDL and extension (.cpp and .java) files. For more information about using Netscape Extension Builder Designer, see Chapter 4, “Designing a Netscape Extension.”

KIDL Compiler

KIDL Compiler is a source code generation and pre-compilation tool. It takes IDL input from either Netscape Extension Builder Designer or from hand-written IDL and generates source code within a source code tree as dictated by makefiles. The makefiles are generated by Netscape Extension Builder Designer at the same time as the IDL files.

KIDL Compiler uses the makefiles as compilation guidelines. The makefiles keep track of all the files that compose a particular extension. The compiler generates the appropriate files and directories based on the contents of the makefile. For example, depending on values in the makefile, KIDL Compiler produces either C++ code or Java code tree structures.

KIDL Compiler also generates the method stubs in which you write the implementation code that defines the functionality of your extension. Once completed, the method stubs link the Netscape extension to the technology being extended. More information about method stubs is provided in a later section of this guide.

Netscape Extension Builder Runtime Features

Netscape Extension Builder also includes several value-added features that enrich the extension during runtime. These features, called Netscape Extension Builder Runtime Features, save you from writing the code for the value-added functionality.

Netscape Extension Builder Runtime Features provide solutions to scalability and performance issues commonly associated with building high-performance applications that integrate third-party and legacy technology. Netscape Extension Builder Runtime Features include the following:

- Method Locking
- Template Streaming
- Extension State and Session Management
- Object Pooling

You specify that you want to use these features in Netscape Extension Builder Designer as you are designing your extension. For some of the features, you must write some additional implementation code when you are completing the method stub files. The procedural information about doing this is explained in Chapter 6, “Completing Method Stubs.”

During runtime, the Netscape Extension Builder Runtime Features are implemented in the Netscape Extension Builder Runtime Layer, which provides the “under-the hood” processing and functionality for those features.

When you design and build an extension, consider whether you need to use these runtime features.

Method Locking

Method locking provides thread safety for single-threaded extensions. Most legacy systems are built on technology that is not multi-threaded. Integrating this technology into the Netscape Application Server environment, which is multi-threaded, can create thread safety issues.

Method locking eases the task of integrating non-thread safe technologies into a Netscape Application Server application by ensuring that only one call can be made to a locked method. If you are integrating single threaded functionality into Netscape Application Server, you probably want to use method locking.

Method locking is usually used in conjunction with Netscape Application Server’s multi-process/single-threaded (MP/ST) configuration to provide increased multiple client support of a single threaded service. Configuring an MP/ST environment requires collaboration with the Netscape Application Server system administrator.

Enabling method locking is covered in more detail in Chapter 4, “Designing a Netscape Extension.”

Template Streaming

Template streaming enhances the performance of applications that use extensions. Extensions enabled for template streaming allow the Netscape Application Server Template Engine to stream results obtained by the extension back to the client. Template streaming is useful for extensions that return large amounts of data or that take a significant amount of time to process results.

For more information on template streaming and how to enable this feature, see Chapter 7, “Using Template Streaming.”

Extension State and Session Management

Extension state and session management maintains user session and application state information for the extended technology. You most likely want to enable this feature when your legacy solution tracks user information on a user by user basis, or the legacy solution is part of an application that is distributed across multiple servers.

More information about enabling extension state and session management is provided in Chapter 8, “Managing State and Session Information.”

Object Pooling

Object pooling alleviates bottlenecks in application performance by allowing clients to share limited-resource objects, thereby reducing the time applications must wait while physical objects are created and destroyed.

Use object pooling if your extension may cause a performance bottleneck. To enable object pooling, you use Netscape Extension Builder Designer and then complete method stubs of a Netscape-defined interface.

For more information about object pooling and how to enable this feature, see Chapter 9, “Using Object Pools.”

Extension Make Harness

The extension make harness is an important tool included in Netscape Extension Builder for compiling the extension source code files. The make harness consists of several files that refer to all of the extension source code files, such as completed method stubs and .cpp or .java files created by Netscape Extension Builder Designer.

The make harness uses the makefiles and a standard C++ compiler to compile the generated source code and the completed method stubs into the runtime extension. Running the design-time files through the make harness is the last

process required to complete the extension. More information about compiling the extension source code is provided in Chapter 10, “Compiling the Extension Source Code.”

Chapter

3

Introduction to Netscape's Interface Definition Language

This chapter describes Netscape's Interface Definition Language, which is used to define the interface specifications and class descriptions of a Netscape extension.

The following topics are included in this chapter:

- What is Netscape's Interface Definition Language?
- About KIDL
- How KIDL Corresponds to the Parts of a Netscape Extension

What is Netscape's Interface Definition Language?

Netscape's Interface Definition Language, called KIDL, is the language used to define the interface specifications and class descriptions of a Netscape extension. An interface specifies what a client can do with an object. A class description, or coclass, specifies the definition of a class by noting the interfaces the class will implement.

KIDL, similar to related industry IDL, does not specify the implementation details of a class (or, once instantiated, an object). For example, implementation details are the operating system on which an object is running or the language in which an object is written. Since interfaces and classes can be defined without regard to these types of details, the objects instantiated from KIDL defined interfaces and classes can inter-operate with each other regardless of implementation differences.

KIDL encompasses theories and practices of CORBA IDL and Microsoft COM IDL. If you are not familiar with IDL at all, you should read an introductory book on the subject that describes both CORBA and COM interface design languages.

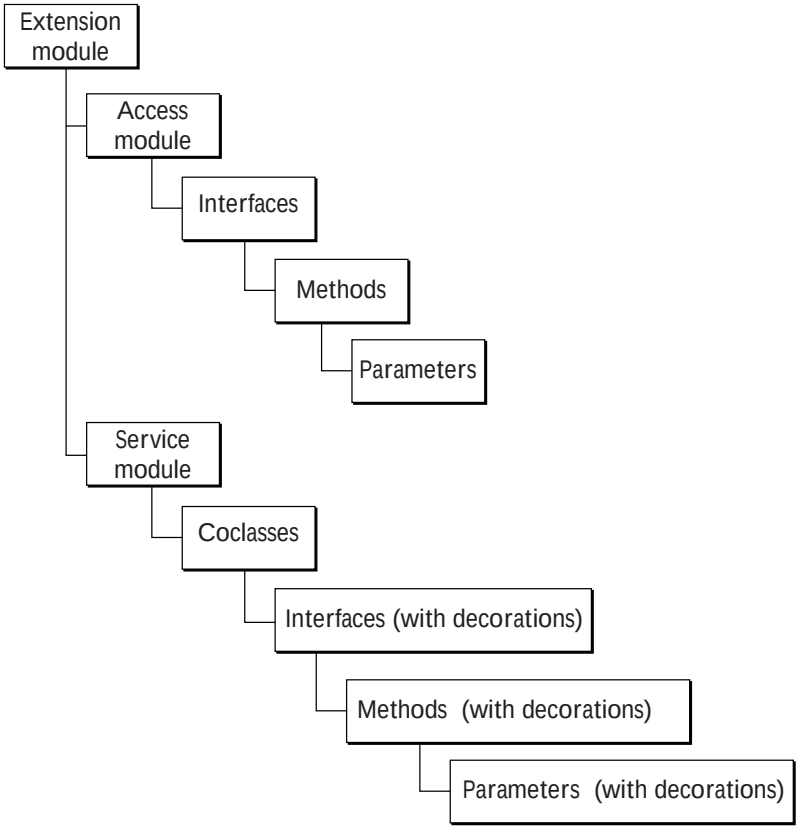
About KIDL

KIDL allows you to develop interfaces and classes that can link disparate and proprietary code to application components. KIDL allows the inter-operation of application components and Netscape extension objects, regardless of how the extension objects are implemented. Without KIDL, much more work would be required to provide application components access to legacy and third-party technology.

In addition to providing a gateway for the integration of third-party and legacy functionality, KIDL is used to generate repetitive code. By supporting the use of implementation hints, called decorations, the KIDL Compiler can generate specific method stubs and value-added code, saving development time once spent writing similar code over and over again. KIDL's various uses are described further in the next section, "How KIDL Corresponds to the Parts of a Netscape Extension."

How KIDL Corresponds to the Parts of a Netscape Extension

A Netscape extension is designed and built in parts, as depicted in the following illustration:



The top three levels are modules, or logical groupings, of an extension and the components within an extension. You design extensions within these modules, which are described in the following sections:

- Extension Module
- Access Module
- Service Module

Each of the above modules corresponds to individual IDL files: one file for the extension, one file for each access module, and one file for each service module. KIDL is used in different ways depending on the module of the

extension you are designing. The IDL files are then used as input to the KIDL Compiler, which creates the extension source code and method stubs. These concepts and procedures are described in later sections.

Extension Module

The extension module represents the extension as a whole. It is a grouping of the interfaces and services that compose the extension. The extension module contains the access module and the service module, and everything those modules contain.

The IDL file for the extension module is very simple. Below is an example of the IDL code as it appears in an actual extension module IDL file.

```
//
// This file is generated by KIDL - do not edit
//
global [package(testextension), extension_language(java_code),
extension_name(TestExtension_EXT)]

package extension;

typedef  long time_t;
typedef  unsigned int size_t;
typedef  HRESULT SCODE;
typedef  _GUID GUID;
typedef  GUID IID;
typedef  GUID CLSID;
typedef  IID REFIID;
typedef  CLSID REFCLSID;
typedef  GUID REFGUID;

#include "KIVA.idl"
#include "myiext.idl"
```

This file is named TestExtension_EXT.idl, as denoted by the extension_name decoration in the first lines of code.

Note the `# include` statements at the bottom of the IDL file. The first includes the Netscape Application Server access module IDL file, which contains the support interfaces necessary for inter-operating with the Netscape Application Server. The second `# include` statement refers to the extension access module IDL file, which describes the interfaces created for the extension.

The service modules are included in the extension by referring to the extension module using a `# include`, as described later.

The entire content of the extension IDL file is generated by the Netscape Extension Builder Designer, so you don't have to write any of it.

Access Module

The access module is a grouping of the interfaces used by application components to access Netscape extension objects. The interfaces within the access module have no implementation details and are simply the specifications of the operations a client, by invoking the methods exposed by the interfaces, can perform when interacting with an object.

For example, you might define an IQueryAccount interface with a GetBalance( ) method. This method probably contains several parameters. Within the access module, the IDL code would specify that an IQueryAccount interface was going to exist, as well as the GetBalance( ) method and the parameters of that method. Basically, the access module specifies the names of the interfaces, methods, and parameters available to the application components. Below is an example of the IDL code as it appears in an access module IDL file.

```
//
// This file is generated by KIDL - do not edit
//
[package(testextension)]
module myiext
{
[local, object, uuid(90BB7EB0-518F-11D1-A1AA-006008293C54)]
interface IExtMain : IGXObject
{
  HRESULT GetMyName(
    [out] LPSTR pName,
    [in] unsigned long nameSize);
```

```
  HRESULT CreateFirst(
    [out] IExtFirstInterface** ppFirst);
  HRESULT CreateSecond(
    [out] IExtSecondInterface** ppSecond);
};
[local, object, uuid(B0399FF0-518F-11D1-A1AA-006008293C54)]
interface IExtFirstInterface : IGXObject
{
  HRESULT DeliverTo(
    [in] LPSTR deliverWhat,
    [in] LPSTR toWhom);
  HRESULT TurnUp(
    [in] DWORD increment,
    [out] IExtSecondInterface** ppNewGuage);
  HRESULT SetFilmSize(
    [in] DWORD filmSize);
  HRESULT GetFilmSize(
    [out] DWORD* pFilmSize);
};
[local, object, uuid(D94AC9F0-518F-11D1-A1AA-006008293C54)]
interface IExtSecondInterface : IGXObject
{
  HRESULT GetDescription(
    [out] LPSTR pDesc,
    [in] unsigned long descSize);
  HRESULT SetHighWaterMark(
    [in] DWORD newMark);
 };
}
```

In this example, the file is named myIExt.idl. There are three interfaces defined in this access module: the IExtMain, the IExtFirstInterface and the IExtSecondInterface. These are specified on the lines that begin with the keyword `interface`. Each interface is a client of (denoted by the colon) the IGXObject interface, from which all interfaces are derived.

Within each interface are the specifications for the methods that clients can use to invoke operations from the object or objects that will implement these interfaces. Methods are typically denoted with the `HRESULT` keyword. In the parentheses after each method are the parameters of the method along with a scope notation (denoted in the square brackets) of whether it is an `in` or `out` parameter.

Service Module

The service module is a grouping of coclasses. A coclass is a design specification of a class and becomes a class after the service module IDL file is compiled.

In the service module, coclasses typically implement one or more of the interfaces defined in the access module. When implementing those interfaces, you can add decorations to the interfaces, methods and parameters. Decorations describe a portion of the implementation details for how a coclass becomes a class. For example, a decoration can provide information to the KIDL Compiler that a certain interface is going to have a Java access layer that supports calls from Java application components.

Implementation details are not specified in the access module to simplify extension building and to support multiple interface implementation.

Multiple interface implementation allows for different implementations of the same interface. It is easier to assign decorations on a coclass basis, rather than define two interfaces, which would appear identical to the application component layer.

For example, you could define two coclasses that implement the same interface, but implement that interface in different ways. Within the service module you might have a Java version coclass called myExtJavaService as well as a C++ version coclass called myExtC++Service. Each coclass would implement the same interface. However, the implementation of the interface in the Java service would use a Java Access Layer decoration, whereas the C++ coclass implementation would not use the Java Access Layer decoration. This example is depicted in the following illustration:

```
TestExtension module
    myIExt access module
       myInterface
          doSomething()
             inParameter
    myExt service module
       myExtJavaService
          myInterface (with Java wrapper decoration)
             doSomething() (with Java wrapper decoration)
                inParameter (with Java wrapper decoration)
       myExtC++Service
          myInterface (without Java wrapper decoration)
             doSomething() (without Java wrapper decoration)
                inParameter (without Java wrapper decoration)
```

Below is an example of the IDL code as it appears in a service module IDL file, named myext.idl. Note how the first line of code includes the extension module.

```
//
// This file is generated by KIDL - do not edit
//
#include "TestExtension_EXT.idl"
[package(testextension.myext)]
module myext
{
[uuid(72c1c580-9f31-11d1-a1bb-006008293c54), wrapper_uuid(72c1c581-
9f31-11d1-a1bb-006008293c54)]
 coclass CExtModule
 {
  interface IGXModule
 {
    Init( pObj);
    Uninit( pObj);
 }
  interface IExtModule
  {
    GetMyName([size_is(nameSize)] pName, [default_value(512)] nameSize);

CreateFirst([java_class(testextension.myext.CExtFirstClass),
   cpp_class(CExtFirstClass)] ppFirst);
 CreateSecond([java_class(testextension.myext.CExtSecondClass),
```

```
   cpp_class(CExtSecondClass)] ppSecond);
  }
 };
[uuid(72c1c582-9f31-11d1-a1bb-006008293c54), wrapper_uuid(72c1c583-
9f31-11d1-a1bb-006008293c54)]
 coclass CExtFirstClass
 {
    interface IExtFirstInterface
  {
    DeliverTo( deliverWhat,  toWhom);
   TurnUp( increment, [java_class(testextension.myext.CExtSecondClass),
   cpp_class(CExtSecondClass)] ppNewGuage);
    IgnoreThis( pIgnore);
    SetFilmSize( filmSize);
    GetFilmSize( pFilmSize);
  }
 };
[uuid(72c1c584-9f31-11d1-a1bb-006008293c54), wrapper_uuid(72c1c585-
9f31-11d1-a1bb-006008293c54)]
 coclass CExtSecondClass
 {
    interface IExtSecondInterface
  {
GetDescription([size_is(descSize)] pDesc, [default_value(512)]
descSize);
    SetHighWaterMark( newMark);
  }
 };
}
```

The definitions of the coclasses, denoted by the `coclass` keyword, include the interfaces, methods and parameters each coclass implements. Around all of these definitions are decorations, which are contained in square brackets [ ]. Notice how the decorations provide implementation details. Decorating the components of a coclass is covered later in Chapter 4, “Designing a Netscape Extension.”

Chapter

4

Designing a Netscape Extension

This chapter describes Netscape Extension Builder Designer, a tool used to design the components of a Netscape Application Server extension.

The following topics are included in this chapter:

- About Netscape Extension Builder Designer
- How to Use Netscape Extension Builder Designer
- Starting Netscape Extension Builder Designer
- Creating an Extension Module
- Creating an Access Module
- Creating a Service Module
- Creating an Access or Service Module by Importing an IDL File
- Designing an Interface
- Designing a Coclass

About Netscape Extension Builder Designer

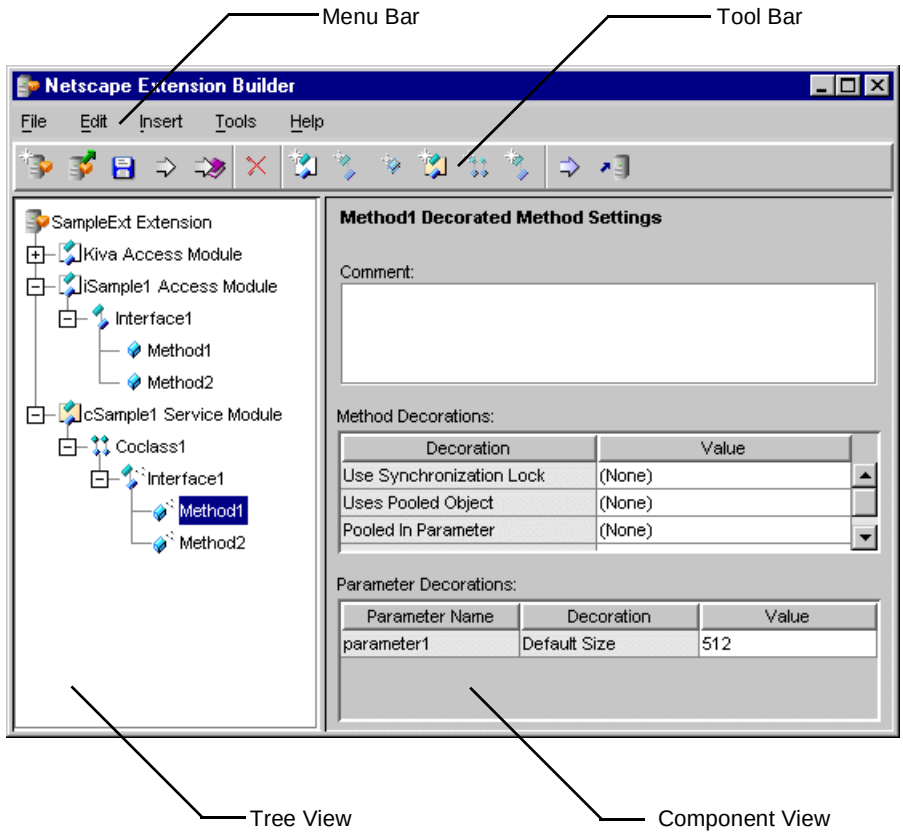
Netscape Extension Builder Designer is a Graphical User Interface (GUI) tool you use to rapidly design the interfaces and coclasses that compose an extension. These interfaces and coclasses are specified using KIDL, and they link Netscape Application Server application components to legacy and third-party functionality.

You define the extension interfaces and coclasses by making selections within Netscape Extension Builder Designer. When you are finished, you generate IDL files in which Netscape Extension Builder Designer translates your GUI selections to KIDL syntax. Netscape Extension Builder Designer also generates a source code tree and makefiles, which are used to compile the IDL and extension class (.cpp and .java) files. These files are compiled using the KIDL Compiler.

Overview of the Graphical User Interface (GUI)

The GUI of Netscape Extension Builder Designer is divided into several parts. The most prominent portions of the GUI are the tree view, the component view, the menu bar and the tool bar.

The following illustration introduces the main parts of Netscape Extension Builder Designer:



Tree View

The tree view displays the component hierarchy of the extension tree. The components of the extension are displayed as they relate to each other. For example, methods are displayed within the interfaces to which they belong, and the interfaces are displayed within the access module to which they belong, and so on.

Component View

The component view provides information about a component selected in the tree view. The component view is where you specify the attributes of a component and perform most of your design work.

The component view is context sensitive according to the item selected in the tree view. For example, when a method is selected, the component view displays GUI controls to allow you to define parameters for that method. If an interface is selected, the GUI controls are relevant only to interfaces.

Menu Bar

The menu bar provides standard features, such as file management, as well as access to the various actions you might perform while designing the components of a Netscape extension. The menu bar has the following menu items, as described in the following table:

Menu Items	Description
File	Contains actions specific to managing files, such as open, close, save and import.
Edit	Contains actions specific to editing particular extension components.
Insert	Contains actions specific to creating the different components of an extension. Some of the items in this menu are disabled at various times depending on which component is selected in the tree view.
Tools	Provides access to the generation and deployment functionality of Netscape Extension Builder Designer. These are used when you are ready to generate IDL files or deploy the extension to a Netscape Application Server.
Help	Provides access to HTML help topics specific to using Netscape Extension Builder Designer and building Netscape extensions.

Tool Bar

The tool bar provides access to the common actions you might perform when designing a Netscape extension. For example, in addition to the standard items such as opening and saving a file, there are icons for importing IDL files, creating new modules, interfaces, and coclasses, generating IDL files and deploying the extension to a server.

Certain menu and tool bar items are enabled or disabled as appropriate for the design of an extension component. For example, the button used to add a new method is only enabled when you have an interface or method selected in the tree view.

How to Use Netscape Extension Builder Designer

Before using Netscape Extension Builder Designer, you should have determined the names of the interfaces (API) needed to integrate the third-party or legacy technology into a Netscape Application Server application. The interfaces that need to be exposed for a Netscape Application Server application are the interfaces you will design using Netscape Extension Builder Designer.

Extension Design Flow Using Netscape Extension Builder Designer

There is no right or wrong design flow for building a Netscape extension using Netscape Extension Builder Designer. The following, however, is a methodical approach to designing a complete extension from start to finish:

1. Start Netscape Extension Builder Designer
2. Create an Extension Module
3. Create the Access and Service Modules
4. Design the Interfaces

5. Design the Coclasses
6. Generate the IDL Files
7. Complete the Extension Method Stubs
8. Compile the Extension Source Code
9. Test and Deploy the Extension

Aside from completing the extension method stubs and compiling the extension source code, all of the above steps are completed using Netscape Extension Builder Designer. Information about completing the method stubs and compiling the source code is provided in a subsequent section of this guide.

This design flow assumes you are creating a new extension. For extensions that are work in progress, you might be at any stage of this design flow.

In addition, each of the steps above is a high-level procedure usually consisting of several to many sub-procedures that must be completed before the overall procedure is complete. For example, when designing a coclass, you might need to assign certain decorations to the interfaces, methods, or parameters within the coclass. Assigning decorations to these components is a sub-procedure of designing a coclass.

Start Netscape Extension Builder Designer

The first step in using Netscape Extension Builder Designer is to start it. For more information about starting Netscape Extension Builder Designer, see “Starting Netscape Extension Builder Designer” on page 46.

Create an Extension Module

Once you start Netscape Extension Builder Designer, the next step is to create the extension module, which represents the extension as a whole. The extension module contains the access modules and the service modules that compose the entire extension. The procedural information about creating an extension module is covered in “Creating an Extension Module” on page 47.

Create the Access and Service Modules

After you create the extension module, the next logical step is to create the access and service modules. An access module groups related interfaces of the extension, and a service module groups related coclasses of the extension. Once you create these modules, you have the base structure around which the extension takes form. More information about creating the access and service modules is provided in later sections of this guide.

Design the Interfaces

After creating an access module, you are ready to design the interfaces used by application components or other extensions to interact with the technology being extended. Designing interfaces consists of naming the interface, defining the methods that the interface contains, and defining the parameters of each of those methods. Procedural information about designing interfaces is provided in “Designing an Interface” on page 53.

Design the Coclasses

After creating a service module and designing the interfaces, you are ready to design the coclasses of the extension. Coclasses contain information for how the extension classes are implemented. For example, the designation of whether an extension supports access by both Java and C++ clients is made in a coclass. Coclasses become skeleton classes after you generate and compile the IDL files generated by Netscape Extension Builder Designer.

Each coclass implements one or more interfaces defined in the access modules. You determine the interfaces that are implemented by a coclass as well as provide the implementation details, called decorations, for those interfaces (and the interface’s methods and parameters). Procedural information about designing coclasses is covered in “Designing a Coclass” on page 57.

Generate the IDL Files

Once you have defined your interfaces and extension classes, you are ready to generate the IDL files. The IDL files compose most of the source code for the extension. These files are then compiled using the KIDL Compiler. Netscape Extension Builder Designer quickly generates IDL files. Generating IDL files is covered in a later section in this guide.

Complete the Extension Method Stubs

The KIDL Compiler converts IDL files and generates extension source code and method stubs. The method stubs are incomplete source code files. You need to complete the method stubs before compiling the source code and deploying your extension. Information about completing the method stubs is provided in Chapter 6, “Completing Method Stubs.”

Compile the Extension Source Code

Once you complete the method stubs, you are ready to compile the extension source code to create the extension runtime libraries and classes. Compiling the source code requires the use of the Netscape Extension Builder-generated makefiles. Procedural information about compiling the extension source code is provided in Chapter 10, “Compiling the Extension Source Code.”

Test and Deploy the Extension

The final step is to test and deploy the extension. You can use Netscape Extension Builder Designer to quickly deploy your extension to host Netscape Application Servers. Information about deploying extensions is provided in Chapter 11, “Deploying and Managing a Netscape Extension.”

Starting Netscape Extension Builder Designer

Start Netscape Extension Builder Designer to begin building your Netscape extension.

To start Netscape Extension Builder Designer

1. In Windows NT, choose Start - Programs - Netscape Extension Toolkit - Netscape Extension Builder Designer.
2. In UNIX, from a shell prompt, type `neb`.

   Netscape Extension Builder Designer starts.

Creating an Extension Module

The extension module groups the components that compose an extension, as described in the following list:

- The Netscape Application Server access module, which contains special Netscape Application Server interfaces necessary for all extensions
- Your extension's access module or modules, which contain the interfaces available to application components
- Your extension's service module or modules, which contain the coclasses that define the implementation of the extension

When you create an extension module, a sample extension is loaded into Netscape Extension Builder Designer. The sample extension contains a sample access module that has one interface with two methods. The sample extension also contains a sample service module that has a coclass that implements the interface defined in the sample access module. The sample extension also contains the required Netscape Application Server access module.

The sample extension is designed to guide you in creating an extension. The sample components provide a framework for how an extension is organized. You can use this framework to build the rest of your extension.

The sample access and service modules are meant to be edited to reflect your particular extension. This means you can rename the modules, and their components, and add components, as necessary, to these modules. Only the Netscape Application Server Access module is not meant to be modified.

To create an extension module

1. If you are working on a current extension project, choose File - New.

   A new extension module, called Sample, appears in Netscape Extension Builder Designer. This sample provides a skeleton for the main components of an extension.

   The sample extension is the default extension loaded into Netscape Extension Builder Designer at startup. For this reason, choosing File - New is only necessary when you are currently working on another extension project.

2. In the Tree view, rename the sample access and service modules for your extension.

3. As appropriate, rename the sample interfaces and coclasses, including methods and parameters, for your extension.

4. In the Component view, specify the appropriate decorations for this extension, as described in the following table. Decorations applied at the extension level apply to all components of the extension.

Decoration	Description
Package Name	Specifies the package name for extensions written in Java and extensions written in C++ with a Java access layer. The package name should reflect the name of the extension.
Language	Specifies the language in which the extension is written; either C++ or Java.
Create a Lock	Only available for C++ extensions. This decoration enables method locking for extensions that must run in single-threaded mode. When on, methods within the extension can only be used by one client at a time.
Java Access Layer	Only available for C++ extensions. This decoration allows Java clients to call into the C++ extension.

5. When you are finished, choose File - Save.

   At the prompt, save the file as .gxp

   You are now ready to create the additional components of your extension.

Creating an Access Module

Create an access module when you need to separate and group extension interfaces. An access module provides a way to logically group similar interfaces. For example, an extension might have several interfaces used to connect to and query a backend server. The extension might also have interfaces that provide access to business logic. It makes sense, therefore, to group the connection interfaces in one access module, and the business interfaces in another access module.

For extensions with a small set of interfaces, you most likely need only one access module. Therefore, it is most convenient to rename the sample access module initially loaded into Netscape Extension Builder Designer to a name that better describes your extension.

Use the following procedure to create an additional access module when necessary:

To create an access module

1. When creating a new extension, first rename the access module named iSample to better describe your extension. Most extensions only need one access module.
2. For extensions that require more than one access module, choose Insert - Access Module or click the “Create a new Access Module” tool-bar icon.

   A new access module is created.
3. Change the name of the NewAccessModule to a name representative of your extension.
4. To describe this access module, enter a comment in the Comment text box.

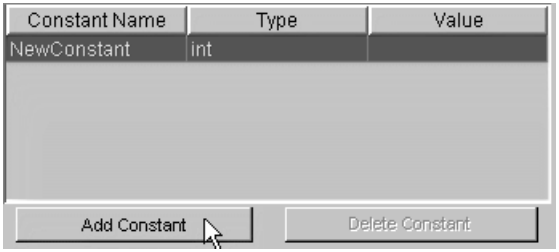
   This comment will appear in the IDL file that corresponds to this access module.
5. If this access module is going to support global constants, add them in the constant table at the bottom of the Component view, as described in the next section.“Defining a Global Constant” on page 49.
6. Choose File - Save to save your changes.

Defining a Global Constant

Netscape Extension Builder Designer allows you to define global constants within an access module. Constants defined in an access module can be used in application and extension code without re-declaring the constants in those code modules.

To define a global constant

1. In the Component view under the constant table, click Add Constant.



2. Under Constant Name, specify a name for the constant.
3. Under Type, select a data type from the list box.
4. Under Value, specify a default value for the constant.
5. For additional constants, click Add Constant and repeat steps 2 through 4.
6. When you are finished, choose File - Save.

Creating a Service Module

For new extensions, it is most convenient to rename the service module named cSample initially loaded into Netscape Extension Builder Designer to a name representative of your extension.

Use the following procedure to create an additional service module when necessary:

To create a service module

1. If you have not already done so, rename the service module named cSample to a name representative of your extension.
2. If you need additional service modules in your extension, choose Insert - Service Module or click the tool-bar icon.

A new service module is created.

3. In the Component view, change the service module named NewServiceModule to a name representative of your extension.

4. To describe this service module, enter a comment in the Comment text box.

   This comment will appear in the IDL file that corresponds to this service module.

5. In the decoration table, specify the appropriate decorations for this service module, as described in the following table. These decorations are only available if you are developing this extension in C++.

Decoration	Description
Java Access Layer	Specifies whether the methods in this service module are accessible to Java clients. Default is same as designation made at extension module. If the extension does not use a Java access layer and you turn on the Java layer for a service module, you will be prompted to enable a Java layer for the entire extension or just the service module. If you specify access for just this module, methods in other modules cannot be accessed by Java clients, unless you specify otherwise within those modules. Note that this designation will turn the Java Access Layer to ON for the extension, however, only this service module is actually enabled to serve requests from Java clients. If you specify to enable the Java access layer for the extension, all methods in the extension can be accessed by Java clients.
Create a Lock	Specifies that the methods in this service module are to be used in single-threaded mode. When accessed by a client, the method is locked to other client calls until finished with current client request.

6. Choose File - Save to save your changes.

Creating an Access or Service Module by Importing an IDL File

You can also create access and service modules by importing already composed IDL files. An IDL file is usually a text file and, as long as the code is written in KIDL syntax, the file can be read by Netscape Extension Builder Designer.

Netscape Extension Builder Designer cannot import CORBA or Microsoft IDL files. Such files must be re-written using the KIDL syntax.

There are two ways you can import IDL files, as described in the following table:

Type of Import	Description
Import IDL	Imports IDL in editable format, allowing you to change the definitions defined in the file.
Import IDL as Library	Imports the IDL format by referencing the IDL file. The referenced IDL file becomes a `#include` in the IDL generated by Netscape Extension Builder Designer. The Netscape Application Server Access Module is an example of IDL imported as a library. The definitions for the interfaces in the Netscape Application Server Access Module are #includes in the corresponding IDL file.

Creating an Access Module

When creating an access module, you would either import IDL directly or as a library, depending on what your extension was going to do with the interfaces defined in the files. For example, if you need to add parameters to the methods of the interfaces, import the IDL file directly. If you were not going to make changes to the definitions, import the IDL file as a library.

Creating a Service Module

When creating a service module, you will always want to import the IDL files directly so that you can provide the implementation details for your extension.

To create an access or service module by importing an IDL file

1. Open the extension project into which you want to import the IDL file.
2. Choose File - Import IDL or, for an access module, Insert - Import IDL as Library.
3. In the Select IDL File dialog box, navigate to the directory where the IDL file resides.

   You can enter that directory directly or you can browse to find that directory.
4. Select the file and click Open.

   The selected file is imported in Netscape Extension Builder Designer and the contents of that file become accessible through the GUI.

   Any changes made within Netscape Extension Builder Designer are not saved to the source IDL file. Such changes are saved in the IDL file generated by Netscape Extension Builder Designer.
5. Continue with the appropriate steps to finish your extension.

Designing an Interface

Once you have created your access and service modules, you are ready to design the extension interfaces.

Designing an interface entails creating the interface and defining the methods that compose the interface. This task can only be done in the access module.

To design an interface

1. First, if working in a new extension, rename the sample interface of the sample extension already loaded into Netscape Extension Builder Designer.
2. If there is more than one access module in the extension, in the Tree view select the access module to which the new interface will belong.
3. Choose Insert - Interface or click the “Create a new Interface” tool-bar icon.

A new interface, named NewInterface, is created in the selected access module. The new interface can be seen in the Tree view and it becomes the currently selected extension component.

4. In the Component view, change the NewInterface name to match one of your predefined interfaces.

5. Specify the interface from which this interface inherits its basic and necessary functionality.

   Most interfaces should inherit from the default IGXObject interface.

6. To describe the interface, enter a comment in the Comment text box.

   This comment will appear in the IDL and source code files that correspond to the access module that contains this interface.

7. To create additional interfaces, repeat steps 3 through 6.

8. When you are finished, choose File - Save.

   You should now define the methods that compose the interface or interfaces you have created, as described in the next section. “Defining a Method” on page 54.

Defining a Method

Once you have created the interfaces for your extension, you must define the methods that these interfaces expose.

To define a method

1. First, if working in a new extension, rename the sample methods loaded into Netscape Extension Builder Designer as a default.

2. To add new methods, in the Tree view select the interface to which the new method will belong.

3. Choose Insert - Method, or click the “Create a new Method” tool-bar icon.

   A new method, named NewMethod, is created in the selected interface. The new method can be seen in the Tree view and it becomes the currently selected extension component.

4. In the Component view, change the new method named NewMethod to reflect a name of an actual method for this interface.

   An interface cannot have multiple methods with the same name.

5. To describe the method, enter a comment in the Comment text box.

   This comment will appear in the IDL and source code files that correspond to this method.

6. To create additional methods for this interface, repeat steps 3 through 5.

7. When you are finished, choose File - Save.

   You must now define the parameters used by the methods you have created, as described in the next section. “Defining a Parameter” on page 55.

Defining a Parameter

As you create the methods exposed by the interface or interfaces of your extension, you need to define the parameters that are used by those methods.

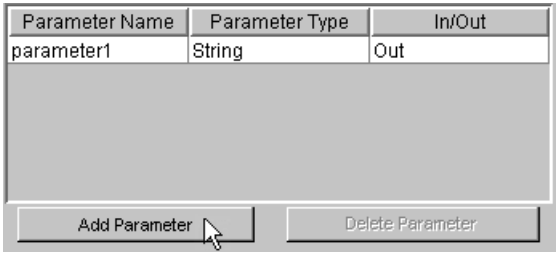
To define a parameter

1. First, if you have not already done so, rename the sample parameters loaded into Netscape Extension Builder Designer as a default.

2. From the Tree view, select the method for which you want to define a parameter.

3. To edit an existing parameter, use the Component view and make your changes in the Parameter table.

The bottom-half of the Component view contains a parameter table to allow you to define the parameters for the selected method.



4. To add a new parameter, click Add Parameter.

   A new parameter is created.

5. Under Parameter Name, change the new parameter named NewParameter to reflect a unique parameter name for this method.

   A method cannot have multiple parameters with the same name.

6. Under Parameter Type, specify the data type for the parameter.

7. Under In/Out, specify whether the parameter is an In or Out parameter.

   It is recommended that methods have only one Out parameter for possibly providing future access to the extension by Java clients. It is mandatory that methods have only one Out parameter for Java extensions and C++ extensions with the Java access layer. This is a Java constraint.

8. To create additional parameters for the selected method, repeat steps 4 through 7.

9. When you are finished, choose File - Save.

   You are now ready to implement the interfaces, including methods and parameters, by designing coclasses, as described in the next section. “Designing a Coclass” on page 57.

Designing a Coclass

A coclass is the design-time representation of a Netscape extension class. A coclass becomes an extension class after the IDL files are compiled using the KIDL Compiler.

Coclasses are designed in the service module so as to separate interface design from interface implementation. The purpose of designing a coclass is to add implementation details, called decorations, to the interfaces you defined in the access module or modules.

Designing a coclass entails creating the coclass, specifying the interface or interfaces that coclass implements, and providing the appropriate decorations for the coclass, interfaces, methods, and parameters.

Typically, you will have more than one coclass in a service module.

To design a coclass

1. If you have more than one service module, select the module to which you want to add a coclass.
2. Choose Insert - CoClass or click the “Create a New CoClass” tool-bar icon.

3. In the Component view, change the new coclass named NewClass to a name more appropriate for your extension.

   A coclass name is the same as a class name and should reflect the function of the service. For example, if the class is a connection manager class, the coclass name might be ConnManager.
4. If necessary, under Super Class, specify the super coclass this coclass subclasses.

   The super class must be defined in the same service module as the coclass that subclasses the super class.
5. To describe the coclass, enter a comment in the Comment text box.

This comment will appear in the IDL and source code files that correspond to this coclass.

6. In the decoration table, specify the appropriate decorations for this coclass, as described by the following table:

Decoration	Description
Manager Class	Specifies whether this class is the manager class for the service module. A manager class loads the extension into the Netscape Application Server context. There can be only one manager coclass in a service module. A manager class provides extension persistence- allowing the extension to be loaded into Netscape Application Server during bootstrapping. Manager coclasses cannot have a super class.
Template Streaming	Specifies whether the results obtained by the methods of this class are handled by the NAS Template Engine so as to stream the results back to the user. For more information about working with template streaming, see Chapter 7, “Using Template Streaming.”
Java Access Layer	Specifies whether the methods in this coclass are accessible to Java clients. The default setting is the same as the designation made at extension and/or service module. The option to specify the Java access layer here allows you to have more control if there are multiple coclasses within the service module. If the extension or service module does not use a Java access layer and you enable it for the coclass, you will be prompted to specify whether to enable Java access for the entire extension, the entire service module, or just this coclass. If you specify just this coclass, methods in other modules and coclasses cannot be accessed by Java clients, unless you specify otherwise within those modules and coclasses.
Create a Lock	Specifies that the methods of this module are to be used in single-threaded mode. When accessed by a client, the method is locked to other client calls until finished with current client request.

Decoration	Description
Poolable Object	Specifies whether this object can be pooled. Pooled objects are shared between clients so as to minimize the time clients wait to be served. For more information about object pooling and the object pooling decorations described in this section, see Chapter 9, "Using Object Pools."
Introspection Interface	Only available when Poolable Object is set to Yes. This decoration specifies the interface that defines the methods used by the object pool manager to query the virtual and physical objects for match criteria.

7. When you are finished, choose File - Save.

Implementing an Interface

Once you have defined your coclass or coclasses, you are ready to implement the appropriate interface or interfaces for that part of the extension.

Coclasses can implement one or more interfaces, and interfaces can be implemented by one or more coclasses. A coclass's implementation of an interface is independent of another coclass's implementation of the same interface. This is useful for providing different functionality for different clients through the same interfaces. This modular functionality is dictated by the implementation code of each coclass.

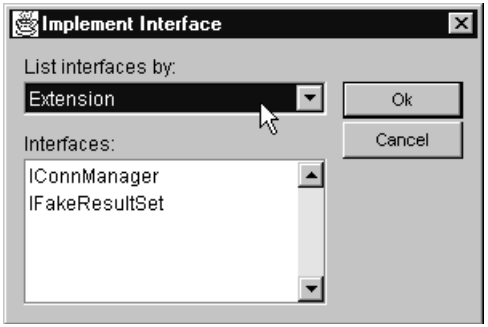
For example, you might have an ICoolService interface that provides a ProvideService( ) method. The type of service the method provides, however, might vary in discreet ways depending on the type of client that is using the interface. Two or more coclasses would implement the ICoolService interface unique to the type of client that would be accessing the ProvideService( ) method.

To implement an interface

1. In the Tree view, select the coclass that will implement the interface you want.

2. Choose Insert - Implement Interface or click the "Add interface(s) for the CoClass to implement" tool-bar icon.

The Implement Interface dialog box appears.



3. From the "List interfaces by" list box, select the view from which you want to select the interface to implement, as described by the following table:

List Interfaces By	Description
Extension	Lists interfaces defined in the current extension only.
System	Lists interfaces available on the machine on which you are working.
All	Lists interfaces defined in all extensions and files on the machine on which you are working.

4. Under Interfaces, select the interface or interfaces you want the selected coclass to implement and click OK.

The selected interface or interfaces appear in the Tree view under the selected coclass after you click OK. The first interface will be the current selection.

5. To describe the interface, enter a comment in the Comment text box.

This comment will appear in the IDL and source code files that correspond to this implementation of the interface.

6. If necessary, change the Java Access Layer decoration to provide or not provide Java clients access to methods.

   If you turn the Java Access Layer off, all methods within this interface will not be accessible to Java clients.

   If you turn the Java access layer on, and the Java layer was off for the coclass, service module, and/or extension, you will be prompted to specify to which level you want to turn on the access layer. The following table describes the effects of the different choices:

Level of Java Access	Description
Extension	Enables Java clients to access *all* methods in the extension.
Service Module	Enables Java clients to access only the interface methods implemented in this coclass. Note that selecting this level of access will turn on the Java Access Layer for the extension, but it will not enable Java access for the extension. Java clients will only be able to access methods within the service module.
Coclass	Enables Java clients to access just the methods implemented in the coclass in which this interface is implemented. Note that selecting this level of access will turn on the Java Access Layer for the service module and the extension, but it will not enable Java access for the service module or extension. Java clients will only be able to access methods within this coclass.
Interface only	Enables Java clients to access just the methods within this interface. Note that selecting this level of access will turn on the Java Access Layer for the coclass, service module and the extension, but it will not enable Java access for all the methods in the coclass, service module or extension. Java clients will only be able to access the methods within this interface.

7. When you are finished, choose File - Save.

Implementing a Method

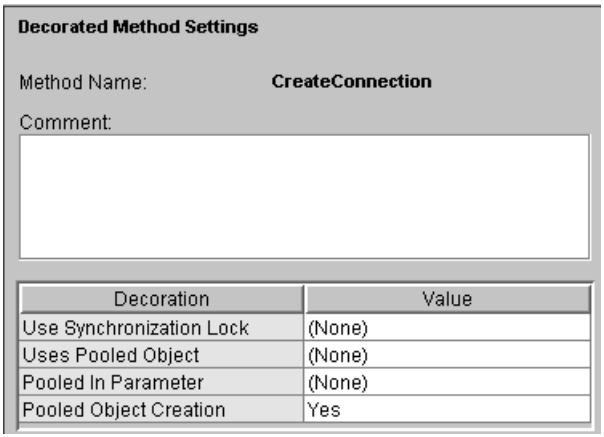
The final step in designing a coclass is to implement the methods in each interface. Methods provide the functionality of the extension and are the integral link between application components and the extension.

Netscape Extension Builder Designer allows you to specify appropriate decorations for individual methods. The decorations provide the first step in specifying what each method does. Once you have finished decorating the methods in Netscape Extension Builder Designer, you need to write the implementation code in the generated methods stubs. Methods stubs are part of the output of the KIDL Compiler and are described in further detail in Chapter 6, “Completing Method Stubs.”

To implement a method

1. In the Tree view, open the coclass and interface containing the methods you want to implement.
2. Select the method to which you will assign decorations.
3. To describe this method’s implementation, enter a comment in the Comment box.

   This comment will appear in the IDL and source code files that correspond to this method’s implementation.
4. In the method decoration table, assign decorations as appropriate for this method.



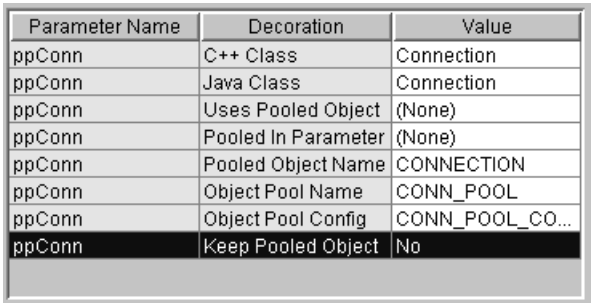
The following table describes the available method decorations:

Decoration	Description
Use Synchronization Lock	Specifies to which level, if at all, usage of this method is locked to one client at a time. If object locking is enabled at the extension, service, or coclass levels, you can specify which one of those levels this method synchronizes its usage lock.
Uses Pooled Object	Specify the value of the pooled_object_name decoration that is specified on the Out parameter of the pooled coclass being passed through this method's In parameter. Use of this decoration is only necessary when you need to extend the reservation time of the pooled object to the called object at the time of this method call. For more information, see "Passing a Pooled Object" on page 147.
Pooled In Parameter	Enter the name of this method's In parameter that holds the pooled object being passed to the called object. For methods, you must use this decoration if you use the Uses Pooled Object decoration.
Pooled Object Creation	Defines this method as one that creates a pooled object through its Out parameter.

Decoration	Description
Keep Pooled Object	Specifies whether the virtual object will keep the real object bound (not allowing other matches) for the life of the virtual object. Establishes a persistent connection between the virtual and real object.
Pooled No Reuse	Specifies whether the physical object will be destroyed when the virtual object returns the physical object to the object pool. This is useful for methods known to alter the state of an object in an irreversible and/or undeterminable way, making future use of the physical object unattractive.

For more information about the concepts of object pooling, see Chapter 9, "Using Object Pools."

5. In the parameter decoration table, specify the decorations of each method's Out parameter.



Parameter Name	Decoration	Value
ppConn	C++ Class	Connection
ppConn	Java Class	Connection
ppConn	Uses Pooled Object	(None)
ppConn	Pooled In Parameter	(None)
ppConn	Pooled Object Name	CONNECTION
ppConn	Object Pool Name	CONN_POOL
ppConn	Object Pool Config	CONN_POOL_CO...
ppConn	Keep Pooled Object	No

The following table describes the available parameter decorations. More information about the use of the object pooling decorations is explained in Chapter 9, "Using Object Pools."

Out Parameter Decoration	Description
C++ Class	Identifies the coclass in the Netscape Extension Builder runtime layer that wraps the extension coclass. If the value of C++ Class is Connection, _Connection, which is pre-defined in the runtime layer, becomes the wrapper coclass. Use this decoration only for Out parameters that implement an interface and use Netscape Extension Builder Runtime Features. Make sure the coclass you specify here exists in your service module and implements the correct interface.
Java Class	This decoration is mandatory for C++ extensions that use a Java access layer. For such extensions, specify the Java coclass used to wrap the C++ extension coclass therefore making the instance of that class (the extension object) available to Java clients. If Java Class is Connection, <packagename>.<servicename>.connection, which is pre-defined in the runtime layer, becomes the wrapper coclass. If this Out parameter implements a Netscape Application Server interface, use the correct value of the decoration as specified in Appendix C, “Java Class Decorations.” If this Out parameter implements an interface you defined, use a coclass defined in this service module and verify that it implements the correct interface. For Java extensions, Java Class identifies the coclass in the Netscape Extension Builder runtime layer that wraps the extension coclass. If the value of Java Class is Connection, <packagename>._<servicename>._Connection, which is pre-defined in the runtime layer, becomes the wrapper coclass. Use this decoration only for Out parameters that implement an interface and use Netscape Extension Builder Runtime Features. Make sure the coclass you specify here exists in your service module and implements the correct interface.
Uses Pooled Object	Specify whether a pooled object is to be passed through this Out parameter. If an In parameter is specified in the Pooled In Parameter decoration, that parameter's corresponding object is passed to this Out parameter. Otherwise, this method's object is passed.
Pooled In Parameter	(Optional) Enter the name of the In parameter that holds the pooled object when you are passing a pooled object from the In parameter to the Out parameter.

Out Parameter Decoration	Description
Pooled Object Name	Enter the name of the pooled object. This is a name you determine so as to make it easier to refer to the pooled object. This decoration is required if the Object Pool Name is specified.
Object Pool Name	Specify the name of the pool that will contain the pooled object being passed through this Out parameter. This decoration is required if the Pooled Object Name is specified.
Object Pool Config	(Optional) Enter a key name that specifies this pool in the Netscape Application Server registry. If null, the extension uses Object Pool Name as the default value.
Keep Pooled Object	Specifies the length of binding time between a virtual object and the physical, pooled object represented by this Out parameter. If set to No, the virtual object binds to a physical object only for the duration of a method call on the virtual object. If set to Yes, the virtual object created binds itself immediately to a physical object and reserves the physical object for the life of the virtual object. This is useful when the physical object maintains state information necessary for future method calls on the virtual object.

6. When you are finished, choose File - Save.

Chapter

5

Generating Output Files

This chapter describes the different types of output files that are generated after an extension is designed.

The following topics are included in this chapter:

- About Generating Output Files
- Generating Files from the Designer
- Generating Source Code Using gmake
- Diagram of Extension Components

About Generating Output Files

You generate output files after you finish designing an extension. This introductory section presents the different types of output produced.

Generating output files involves two main steps:

1. In Netscape Extension Builder Designer, run the Generate IDL command.

   This step is described in “Generating Files from the Designer” on page 76.

2. From the command line, run the make utility.

This step is described in “Generating Source Code Using gmake” on page 77.

Output From Netscape Extension Builder Designer

When you run the Generate IDL command, Netscape Extension Builder Designer places all resulting files and subdirectories in the directory you specify. This is the extension’s top-level directory.

For your convenience, it is recommended that you name an extension’s top-level directory using the following format:

```
<NEB_ROOTDIR>\extensions\<myextension>
```

<NEB_ROOTDIR> is the root directory of your Netscape Extension Builder installation, and <myextension> is the name of the extension you designed in Netscape Extension Builder Designer.

For example, if you installed Netscape Extension Builder in C:\neb and you designed an extension named SchoolDistrict, you might specify the following top-level directory:

```
C:\neb\extensions\SchoolDistrict
```

When the Generate IDL command finishes, the top-level directory contains the following types of output:

- Netscape Application Server Project File and Registration File
- An idl directory containing IDL Files
- make Harness
- Source Code Directory Structure

Netscape Application Server Project File and Registration File

A Netscape Application Server project file lists the components of an extension and is used to deploy an extension to a remote Netscape Application Server. Every extension has one Netscape Application Server project file, whose name has the following format:

myextension.gxp

For example:

SchoolDistrict.gxp

A Netscape Application Server registration file assigns a GUID to each service in the extension. This assignment enables an extension to be registered with Netscape Application Server. Every extension has one Netscape Application Server registration file, whose name has the following format:

myextension.gxr

For example:

SchoolDistrict.gxr

IDL Files

During generation, each module you created in Netscape Extension Builder Designer is stored as a separate IDL file whose name has a .idl suffix. Access modules and service modules are stored in slightly different locations.

Access modules are stored in the idl directory, and an additional file, <myextension>_ext.idl, is also created. This file contains #include directives to include all of the individual access modules.

Service modules are stored in subdirectories of the idl directory. Each subdirectory contains a single IDL file whose basename matches the directory containing it.

For example, if your extension contains two access modules and three service modules, the idl directory would contain the following IDL files:

```
<myextension>_ext.idl
<access_module1>.idl
<access_module2>.idl
<service_module1>\<service_module1>.idl
```

```
<service_module2>\<service_module2>.idl
<service_module3>\<service_module3>.idl
```

make Harness

A make harness is a set of related makefiles—one file per directory—within a directory tree. Makefiles are named GNUmakefile.sun on Solaris, or GNUmakefile.nt on Windows NT.

Makefiles appear in the top-level directory and in all child directories. The top-level makefile contains instructions to traverse the directory tree and call the other makefiles. Subordinate makefiles, in turn, traverse into other directories to affect makefiles or source files there. As a result, the makefiles are programmatically connected, and they provide a "harness" for extension development.

See Chapter 10, "Compiling the Extension Source Code"for more information about the make harness or about using it to compile extensions.

Source Code Directory Structure

To prepare for later code generation, Netscape Extension Builder Designer creates a source code directory tree. Initially, each directory in the tree is empty except for a makefile. After code generation, the directory tree contains source files, either Java or C++.

The extension language you specify in Netscape Extension Builder Designer determines which directories are created.

Extension Type	Directories Created
C++ extension	cpp
Java extension	java
C++ extension with Java Access Layer	cpp and java

Source Code Output From Running gmake

Running the gmake command generates an extension's source code. This source code can be grouped into the following main categories:

- Method Stubs
- Netscape Extension Builder Runtime Layer
- Java Access Layer
- Public Interfaces
- Accessors

Method stubs are the only source code you must modify. For more information, see Chapter 6, "Completing Method Stubs."

Output files produced by the make utility reside in <NEB_ROOTDIR> which denote the root directories of Netscape Extension Builder or Netscape Application Builder, respectively. To describe output locations, the following additional variables are used to denote directories:

<myextension>	The extension name you specified in Netscape Extension Builder Designer.
<service_module>	The name of a service module you defined in Netscape Extension Builder Designer. Each service module will have a corresponding directory.
<package>	The Java package for this extension, expressed as a directory hierarchy. For example, package com.kivasoft.hello translates into directory com\kivasoft\hello.

Method Stubs

After creating the extension module, access modules, and service modules, the corresponding IDL files are used as input to the KIDL Compiler. The KIDL Compiler compiles the IDL files and generates the extension source code and method stubs. Method stubs reside in your extension classes.

A method stub is framework code that you must complete. By completing this code, you tie the extension to the functionality being integrated or extended. The method stub is where you write the code that defines what the methods and parameters actually do.

For example, suppose in Netscape Extension Builder Designer you created an ISetValue interface with a SetTime( ) method. At this point, all you have defined is that the interface and method exists, that there are certain parameters associated with the method, and that all of these methods and parameters use certain decorations, such as the Java Access Layer and Template Streaming.

Now you must define what the SetTime( ) method does, and that is what you do in the method stub. The functionality you implement in method stubs is mapped to the functionality in the object being extended or integrated. This mapping lets AppLogics call into the object.

Location of Method Stubs

For a C++ extension, method stubs are found in .cpp files in the following directories:

```
<NEB_ROOTDIR>\<myextension>\cpp\<service_module>
```

For a Java extension, method stubs are found in .java files in the following directories:

```
<NEB_ROOTDIR>\<myextension>\java\<package>\<service_module>
```

Netscape Extension Builder Runtime Layer

Based on the decorations you added in Netscape Extension Builder Designer, some of the generated code implements Netscape Extension Builder runtime features such as template streaming, method locking, and object pooling. This code, collectively known as the Netscape Extension Builder Runtime Layer, resides in directories whose names start with an underscore (_). There is no need to edit any code in these directories.

The Netscape Extension Builder Runtime Layer provides the “under-the-hood” processing and functionality for the Netscape Extension Builder Runtime Features. The runtime feature classes reside in the Netscape Extension Builder Runtime Layer and are denoted by the underscore prefix. For example, _YourRuntimeClass.cpp is a C++ feature class that corresponds to the coclass called YourRuntimeClass, which you defined in Netscape Extension Builder Designer.

If a method call to an extension class, for example YourRuntimeClass, uses a Netscape Extension Builder Runtime Feature, then this class is first routed through a corresponding runtime feature class, _YourRuntimeClass. The runtime layer matches the called class with the feature class by using the value you specify in the C++ Class or Java Class decoration. This decoration is made on the appropriate method in Netscape Extension Builder Designer.

The Netscape Extension Builder Runtime Layer also supports the accessor class created by the accessor method. The accessor class, also denoted by the underscore prefix, provides access to a service in the extension. There is typically one access class for each service in an extension. If no runtime features are used in an extension, the method call to the accessor class is the only call to go through the runtime layer. For more information, see “Using Extensions From Applications” on page 90.

Location of Netscape Extension Builder Runtime Layer

For a C++ extension, code for the Netscape Extension Builder Runtime Layer is found in .cpp files in the following directories:

```
<NEB_ROOTDIR>\<myextension>\cpp\_<service_module>
```

For a Java extension, code for the Netscape Extension Builder Runtime Layer is found in .java files in the following directories:

```
<NEB_ROOTDIR>\<myextension>\java\<package>\_<service_module>
```

Java Access Layer

The Java Access Layer is the set of code that lets Java clients call into a C++ extension. This code is generated only if you set the Java Access Layer decoration to On.

For a C++ extension with a Java Access Layer, the make command generates additional files, none of which require editing.

The directory

```
<NEB_ROOTDIR>\<myextension>\java\<package>\<service_module>
```

contains Java native methods and Java classes with native method signatures. Both types of files enable Java code to communicate with the C++ implementation of a service module.

Java native methods are written in C++ and use a naming scheme of native_<class>.cpp, where <class> is one of the classes in the service module. Java classes with native method signatures use a naming scheme of <class>.java.

Java native methods are compiled into a library located in the directory:

```
<NAS_ROOTDIR>\APPS\bin
```

The library uses a naming scheme of

Windows `jx2<service_module>.dll`

Unix `libjx2<service_module>.so`

Java classes with native method signatures are compiled into .class files located in the directory:

```
<NAS_ROOTDIR>\APPS\<package>\<service_module>
```

Public Interfaces

Public interfaces are what application components use to communicate with an extension. For a C++ extension, public interfaces are .h files found in the following directory:

```
<NAS_ROOTDIR>\APPS\extensions\include
```

For a Java extension, public interfaces are found in .java files and their corresponding .class file. The .java files reside in the following directory:

```
<NEB_ROOTDIR>\<myextension>\java\<package>
```

whereas the .class files reside in the following directory:

```
<NAS_ROOTDIR>\APPS\<package>
```

Accessors

Accessors enable application components to access a particular service in an extension. For more information, see “Using Extensions From Applications” on page 90.

C++ Accessors

For a C++ extension, accessors are global functions. C++ accessor code includes .cpp implementation files, .h header files, and .lib libraries. Accessor implementation files are located in the following directory:

```
<NEB_ROOTDIR>\<myextension>\cpp\accessors
```

You do not need to modify nor reference the accessor implementation files.

Accessor header files use the naming convention access_<service_module>.h, and they are located in the following directory:

```
<NAS_ROOTDIR>\APPS\extensions\include
```

A C++ developer must add #include directives to the C++ code in order to use accessors. Accessor libraries are compiled into an axs<myextension>.lib file located in the following directory:

```
<NAS_ROOTDIR>\extensions\lib
```

A C++ developer must link this library when compiling the AppLogic application component.

Java Accessors

For a Java extension, accessors are static methods in an accessor class. Java accessor code includes .java files and .class files.

Accessor .java files are in:

```
<NEB_ROOTDIR>\<myextension>\java\<extension_package>
```

You do not need to modify nor reference accessor .java files.

Accessor .class files use the naming convention access_<service_module>.java, and they are located in:

```
<NAS_ROOTDIR>\APPS\<package>
```

A Java developer must modify the Java code by importing the package containing the accessor class.

Generating Files from the Designer

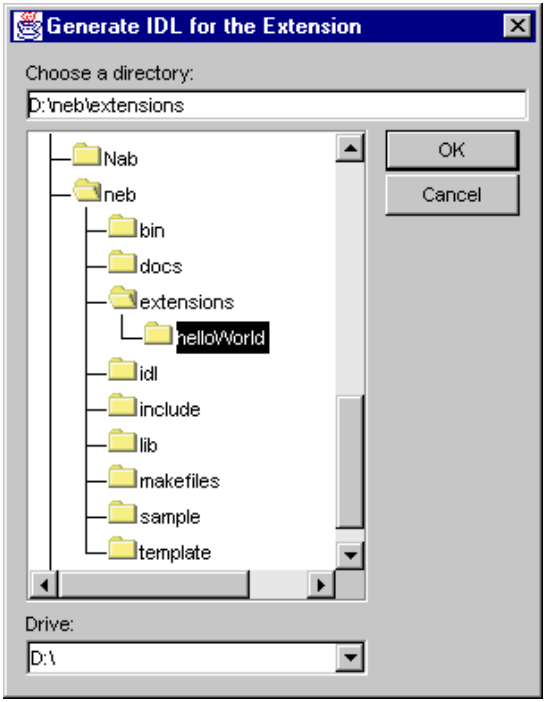
To generate files from Netscape Extension Builder Designer

1. Select the Generate IDL command from the Tools menu.

2. Enter the name of the top-level directory to contain the output.

   On Solaris, you can enter absolute and relative pathnames. However, environment variables and the tilde (~) character are not interpreted, so do not use them.

   On Windows NT, you can also open directories and choose them from a directory window, as shown in the following figure:



Generating Source Code Using gmake

After you run the Generate IDL command from Netscape Extension Builder Designer, run the gmake utility from your extension's top-level directory:

Windows `gmake -f GNUmakefile.nt`

Solaris `gmake -f GNUmakefile.sun`

Running gmake performs the following actions:

- Invokes the KIDL Compiler, which translates IDL files into source code.
- Compiles the source code.
- Copies the code to appropriate locations in the Netscape Application Server root directory.

The next two sections summarize the source code found in an extension's top-level directory. Topics are organized by the type of extension:

- Source Code for a Java Extension
- Source Code for a C++ Extension

Source Code for a Java Extension

For a Java extension, the top-level directory contains a java directory, which in turn contains a series of subdirectories corresponding to your Java package name. For example, if your Java package is com.mycompany.SchoolDistrict, then the top-level directory has the following structure:

```
java\com\mycompany\SchoolDistrict
```

The Java package hierarchy contains the following subdirectories:

Subdirectory	Description
<service_module1>, <service_module2>, ..., <service_modulen>	For each service module you define, Netscape Extension Builder Designer creates a directory with that module's name. After code generation, these directories contain files whose method stubs you need to complete.
gen.<service_module1>, gen.<service_module2>, ..., gen.<service_modulen>	Each service module directory has a corresponding working directory, whose name uses a prefix of *gen*, for *generated*. As you redesign your extension and regenerate source code, the updated code is stored in the *gen*. directories in order to preserve manual changes you may have made in the service module directories. After you determine that the *gen*. directories contain correctly updated files, copy them to the corresponding location in the service module directory.
_<service_module1>, _<service_module2>, ..., _<service_modulen>	For each service module you define, Netscape Extension Builder Designer creates a directory with that module's name and preceded by an underscore (_). After code generation, these directories contain source code for implementing Netscape Extension Builder's runtime features, such as method locking and object pooling. Do not modify any files in these directories.
types	Contains constants that are automatically defined in Netscape Extension Builder Designer.

Source Code for a C++ Extension

For a C++ extension, the top-level directory contains a cpp directory, which in turn contains the following subdirectories:

Subdirectory	Description
<service_module1>, <service_module2>, ..., <service_modulen>	For each service module you define, Netscape Extension Builder Designer creates a directory with that module's name. After code generation, these directories contain files whose method stubs you need to complete.
gen.<service_module1>, gen.<service_module2>, ..., gen.<service_modulen>	Each service module directory has a corresponding working directory, whose name uses a prefix of *gen.*, for *generated*. As you redesign your extension and regenerate source code, the updated code is stored in the *gen.* directories in order to preserve manual changes you may have made in the service module directories. After you determine that the *gen.* directories contain correctly updated files, copy them to the corresponding location in the service module directory.
_<service_module1>, _<service_module2>, ..., _<service_modulen>	For each service module you define, Netscape Extension Builder Designer creates a directory with that module's name and preceded by an underscore (_). After code generation, these directories contain source code for implementing Netscape Extension Builder's runtime features, such as method locking and object pooling. Do not modify any files in these directories.
accessor	This directory contains accessor functions, which are used to enable AppLogics to access an extension's services. Do not modify any files in this directory. For information on accessors, see "Using Extensions From Applications" on page 90.

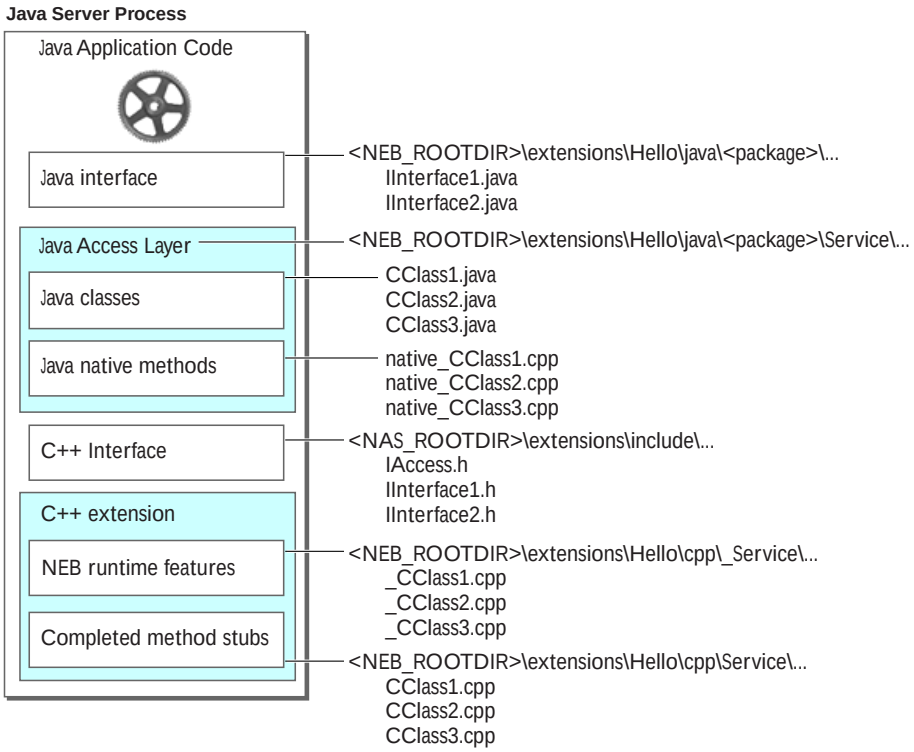
For a C++ extension with a Java Access Layer, the top-level directory contains both a cpp directory and a java directory. The structure of the cpp directory is the same as for a non-wrapped C++ extension.

The Java package hierarchy contains the following subdirectories:

Subdirectory	Description
<service_module1>, <service_module2>, ..., <service_modulen>	For each service module you define, Netscape Extension Builder Designer creates a directory with that module's name. After code generation, these directories contain Java native methods, which act as the "glue" between Java and C++. Do not modify any files in these directories.
types	Contains constants that are automatically defined in Netscape Extension Builder Designer.

Diagram of Extension Components

The following diagram is an example of how extension components are represented by source code. The extension is named Hello and is written in C++ with a Java Access Layer. Assume also that the extension consists of one access module and one service module, named Access and Service, respectively.


Java Server Process
Java Application Code
Java interface
<NEB_ROOTDIR>\extensions\Hello\java\<package>\...
IInterface1.java
IInterface2.java
Java Access Layer
<NEB_ROOTDIR>\extensions\Hello\java\<package>\Service\...
Java classes
CClass1.java
CClass2.java
CClass3.java
Java native methods
native_CClass1.cpp
native_CClass2.cpp
native_CClass3.cpp
C++ Interface
<NAS_ROOTDIR>\extensions\include\...
IAccess.h
IInterface1.h
IInterface2.h
C++ extension
NEB runtime features
<NEB_ROOTDIR>\extensions\Hello\cpp_Service\...
_CClass1.cpp
_CClass2.cpp
_CClass3.cpp
Completed method stubs
<NEB_ROOTDIR>\extensions\Hello\cpp\Service\...
CClass1.cpp
CClass2.cpp
CClass3.cpp

Chapter

6

Completing Method Stubs

This chapter describes steps you must follow to complete a method stub.

The following topics are included in this chapter:

- Steps to Completing Method Stubs
- Locating Stub Files
- Locating Interface Code
- Adding Implementation Code
- Extending the Class Definition
- Using Netscape Application Server Services
- Using Extensions From Applications

Steps to Completing Method Stubs

To complete a method stub, perform the following general procedure:

1. Locate a stub file.
2. Locate the interface code in the stub file.

3. Add implementation code to the methods defined in each interface.
4. If necessary, extend the class definition.
5. Repeat Steps 1 through 4 for each stub file.

Locating Stub Files

When you run the `gmake` command, each coclass you specified in Netscape Extension Builder Designer is code-generated into a separate file, and each file contains one or more method stubs, according to how you designed the extension. You must find these files in the appropriate cpp or java subdirectory.

For example, in the SchoolDistrict extension, assume the following specifications from the design phase:

- The top-level directory is C:\neb\extensions\SchoolDistrict.
- A service module named EmployeeClasses contains three coclasses: CTeacher, CPrincipal, and CEmployeeMgr.
- The Java package is MySchool.

After code generation, the stub files have the following locations and names:

If the extension is C++ based

```
C:\neb\extensions\SchoolDistrict\cpp\EmployeeClasses\
   CTeacher.cpp
   CPrincipal.cpp
   CEmployeeMgr.cpp
```

If the extension is Java based

```
C:\neb\extensions\SchoolDistrict\java\MySchool\EmployeeClasses\
   CTeacher.java
   CPrincipal.java
   CEmployeeMgr.java
```

After locating the stub files, the next step is to edit them. It doesn't matter which file you edit first. To edit the files, locate the interfaces in which the method stubs are defined.

Locating Interface Code

Using your preferred text editor, open a stub file and locate the interfaces. For example, the CTeacher class implements two interfaces: IEmployee and ITeacher.

C++ When you open the CTeacher.cpp file, you see code similar to the following. Note the line that reads “Stub method bodies for interface”. Within this interface block are the method stubs you must fill out.

```
// Stub method bodies for interface: ITeacher
HRESULT
CTeacher::GetClassSize(
      /* [out] */ int *pClassSize)
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   GXASSERT(FALSE, GXASSERT_WARNING, "No implementation for
   CTeacher::GetClassSize");
   return hr;
}
HRESULT
CTeacher::SetClassSize(
      /* [in] */ int classSize)
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   GXASSERT(FALSE, GXASSERT_WARNING, "No implementation for
   CTeacher::SetClassSize");
   return hr;
}
// Stub method bodies for interface: IEmployee
HRESULT
CTeacher::GetName(
      /* [out] */ LPSTR pName,
      /* [in] */ int _sizepName)
```

```
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   GXASSERT(FALSE, GXASSERT_WARNING, "No implementation for
   CTeacher::GetName");
   return hr;
}
```

Java When you open the CTeacher.java file, you see code similar to the following. Note the line that reads “Method bodies for interface”. Within this interface block are the method stubs you must fill out.

```
public class CTeacher
   implements com.schooldistrict.ITeacher,
com.schooldistrict.IEmployee, com.kivasoft.ITemplateData
{
   public com.schooldistrict.EmployeeServices.CEmployeeMgr m_Module;
   public CTeacher(com.schooldistrict.EmployeeServices.CEmployeeMgr
   module)
   {
      m_Module=module;
   }
   public void finalize()
   {
   }
   // Method bodies for interface: ITeacher
   public int getClassSize()
   {
      // ****
      // **** Provide your implementation here
      // ****
      System.out.println(
      "No implementation for CTeacher::getClassSize");
      return 0;
   }
   public int setClassSize(
      int classSize)
   {
      // ****
      // **** Provide your implementation here
```

```
    // ****
    System.out.println(
    "No implementation for CTeacher::setClassSize");
    return 0;
}
// Method bodies for interface: IEmployee
public java.lang.String getName()
{
    // ****
    // **** Provide your implementation here
    // ****
    System.out.println(
    "No implementation for CTeacher::getName");
    return null;
}
```

Adding Implementation Code

How you fill out method stubs depends on the requirements of your extension. You may need to perform one or more of the following coding tasks:

- Supply implementation code for *user-defined* methods; that is, for the methods you defined explicitly in Netscape Extension Builder Designer.

  An example of completing a user-defined method stub is provided in the section immediately following, “Adding a Creation Method” on page 88.

- Supply implementation code for methods in *Netscape Application Server-defined* interfaces.

  Netscape Application Server-defined interfaces will appear in stub files if you added decorations for certain runtime features, such as template streaming or object pooling. In addition, to manage state and session information, you must use the corresponding Netscape Application Server interfaces, which are not automatically code-generated.

  Procedures for completing method stubs for runtime features are described in their own chapters on template streaming, state and session management, and object pooling, later in this guide.

- Use helper functions or static methods to access Netscape Application Server services.

  For an overview of the code to supply for using Netscape Application Server services, see “Using Netscape Application Server Services” on page 89. Reference material on this code is provided in Appendix A, “C++ Helper Functions” and Appendix B, “Java Helper Static Methods.”

Adding a Creation Method

As an example in C++, suppose you defined CreateSampleObject( ), a method in the ISampleManager interface. If the CSampleManager coclass implements the ISampleManager interface, then code-generation would produce a file named CSampleManager.cpp. In this file, you might implement the CreateSampleObject( ) method using the following code:

```
CSampleManager::CreateSampleObject(
  /* [out] */ ISampleObject **ppSampleObject)
{
 HRESULT hr=GXE_SUCCESS;

 // Create an instance of a class that supports the
 //  ISampleObject interface.  Netscape Extension Builder gives us a
 // class that
 //  corresponds to the CSampleObject coclass in Netscape Extension
 // Builder Designer.
 // The C++ class is also called CSampleObject.  It can be
 // found in this directory in
 // CSampleObject.h / CSampleObject.cpp

 *ppSampleObject=new CSampleObject(this);

 return hr;
}
```

Extending the Class Definition

Depending on what you add to a method stub, you may need to extend class definitions in any of the following ways:

- Add method signatures
- Define a member variable that is referenced by a method
- Modify inheritance definitions

To extend a class definition, edit the appropriate file. For example, in the SchoolDistrict extension, you extend the definition of the CTeacher class by editing CTeacher.h (for C++ extensions). In Java extensions, the class definition and the method stubs are in the same file, CTeacher.java for example.

Using Netscape Application Server Services

To use Netscape Application Server services from an extension, you add one or more helper function calls to a C++ method stub. For Java access to Netscape Application Server services, you add one or more helper static method calls to your Java method stub. The helper code you add depends on the service you want to use, as shown in the following table:

Service	C++ Helper Function	Java Helper Static Method
database access	GXContextCreateDataConn, GXContextCreateDataConnSet, GXContextCreateHierQuery, GXContextCreateQuery, GXContextCreateTrans, GXContextLoadHierQuery, GXContextLoadQuery	GXContext.CreateDataConn, GXContext.CreateDataConnSet, GXContext.CreateHierQuery, GXContext.CreateQuery, GXContext.CreateTrans, GXContext.LoadHierQuery, GXContext.LoadQuery
mail	GXContextCreateMailbox	GXContext.CreateMailbox
events	GXContextGetAppEvent	GXContext.GetAppEvent
sessions	GXContextDestroySession, GXGetStateTreeRoot	GXContext.DestroySession, GXContext.GetStateTreeRoot
utilities	GXContextGetObject, GXContextLog	GXContext.GetObject, GXContext.Log
security	GXContextIsAuthorized	GXContext.IsAuthorized
Application Component invocation	GXContextNewRequest, GXContextNewRequestAsync	GXContext.NewRequest, GXContext.NewRequestAsync

For information on using Netscape Application Server services, see Appendix A, “C++ Helper Functions” or Appendix B, “Java Helper Static Methods.”

Using Extensions From Applications

To use an extension, application code calls one or more *accessors*. An accessor is code that gets a reference to a service’s manager class. By referencing the manager class, an application is able to access all other functionality in that service.

You need not define accessors. Netscape Extension Builder creates them automatically during code generation. However, you must modify application code by adding the appropriate accessor calls. A C++ accessor is a global function, whereas a Java accessor is a static method in an accessor class.

To use an extension from an application

1. Determine the names of all accessors the application will invoke.
2. In the application, add the code to call the accessors.
3. If necessary, modify the application makefile before linking the application.

Determining Accessor Names

An application communicates with each extension service through a separate accessor. For example, an application that uses five services must contain five accessor calls.

By convention, an accessor derives its name from the service it provides access to. The naming scheme has the following format:

C++ The global function is KET_Get_<service_module> and the function signature is found in <NAS_ROOTDIR>\apps\extensions\include\access_<service_module>.h.

Java The static method is get<service_module> in class access_<service_module>, in the extension package <package_name>.

Adding Accessor Code to Application Components

C++ To add accessor code to a C++ AppLogic application component:

1. Include one or more accessor header files in the #include section. For example, if the service module is named EmployeeService, add the following include directive:

   ```
   #include "access_EmployeeService.h"
   ```

2. In the Execute method, call the necessary accessor functions. Each accessor function returns a pointer to an interface. Use this pointer to call a method of that interface. For example:

   ```
   MyAppLogic::Execute()
   {
   IEmployeeMgr *pInterface=NULL;
   if(((hr=KET_Get_EmployeeService(m_pContext, &pInterface,
         NULL)) ==GXE_SUCCESS)&&pInterface)
   {
       // successfully got extension
       pInterface->GetMyName();
   }
   ...

   }
   ```

Java To add accessor code to a Java servlet, call a static method against the accessor class. For example:

```
public class MyApplicationComponent extends HttpServlet {

 public void service(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException
 {
 // access servlet's underlying AppLogic instance to use accessor
 HttpServletRequest2 req2;
 req2 = (HttpServletRequest2) req;  // legal cast within NAS

 AppLogic al = req2.getAppLogic();
 com.kivasoft.IContext alContext;
 alContext = al.context;
```

```
 // Here's where the accessor is used
 IEmployeeMgr eMgr;
 emgr = hello=access_EmployeeService.getEmployeeService(alContext,
null, al);

 if (eMgr != null) {
     eMgr.getMyName();
 }

 ...

 }
}
```

Linking an Application Component

For C++ extensions, Netscape Extension Builder compiles an accessor library. When you link a C++ AppLogic application component, you must link with this library, which has the following location:

Windows `<NAS_ROOTDIR>\APPS\extensions\lib\axs<extension>.lib`

Solaris `<NAS_ROOTDIR>/APPS/bin/libaxs<extension>.so`

For Java extensions, Netscape Extension Builder compiles accessor classes and places them in a directory that is already listed in the CLASSPATH variable. As a result, you need not modify your Java makefile.

On Solaris, be sure to modify CLASSPATH to build any application components to have the Netscape Application Server jar files and <NAS_ROOTDIR>/APPS.

Chapter

7

Using Template Streaming

This chapter describes template streaming, which enhances the performance of applications that use extensions.

The following topics are included in this chapter:

- Introduction to Template Streaming
- Setting the Template Streaming Decoration
- Completing Stubs to Support Template Streaming
- Creating a Template
- Testing an Application
- The CZoo.java File

Introduction to Template Streaming

Template streaming enhances the performance of applications that use extensions. Extensions enabled for template streaming allow the Netscape Template Engine to stream results obtained by the extension back to the client.

The template engine supports both HTML and GXML streaming for extensions. This means you can stream results back to non-HTML clients, such as OCL clients.

Template streaming is useful for extensions that return large amounts of data or that take a significant amount of time to process results. Streaming returns results to clients in groups of rows, allowing the client to process the data in chunks before all of the processing is actually completed.

Steps for Enabling Template Streaming

To enable extension objects to be easily streamed using Netscape's Template Engine, perform the following main tasks:

1. In Netscape Extension Builder Designer, set the Template Streaming decoration in any coclasses for which you want to stream results.

   This step is described in "Setting the Template Streaming Decoration" on page 95.

2. After generating source code, complete method stubs in the appropriate interface: either ITemplateData for Java extensions, or IGXTemplateData for C++ extensions.

   This step is described in "Completing Stubs to Support Template Streaming" on page 96. For reference information on these interfaces, see the Java or C++ edition of *Netscape Application Server Foundation Class Reference*.

3. Create an HTML or GXML template to use with an application component.

   This step is described in "Creating a Template" on page 102.

4. Test an application that uses the extension.

   This step is described in "Testing an Application" on page 103.

Example of Template Streaming

To demonstrate the preceding steps, a complete example is provided in the Netscape Extension Builder sample directory. The zoo subdirectory contains a Java extension that demonstrates the use of template streaming.

The sample application demonstrates the creation of objects, in this case animals. The AddAnimal application component enables you to create animals, one a time, along with associated data. The ListAnimals application component enables you to display the list of animals you created. Because this list is potentially large, the ListAnimals application component relies on the extension's template streaming to display the results immediately.

This example is relatively simple, in order to demonstrate the techniques involved. A more complex example might involve database access to obtain data about each animal. In that case, template streaming would be particularly useful because an end user would not need to wait for completion of all database accesses before seeing results.

Setting the Template Streaming Decoration

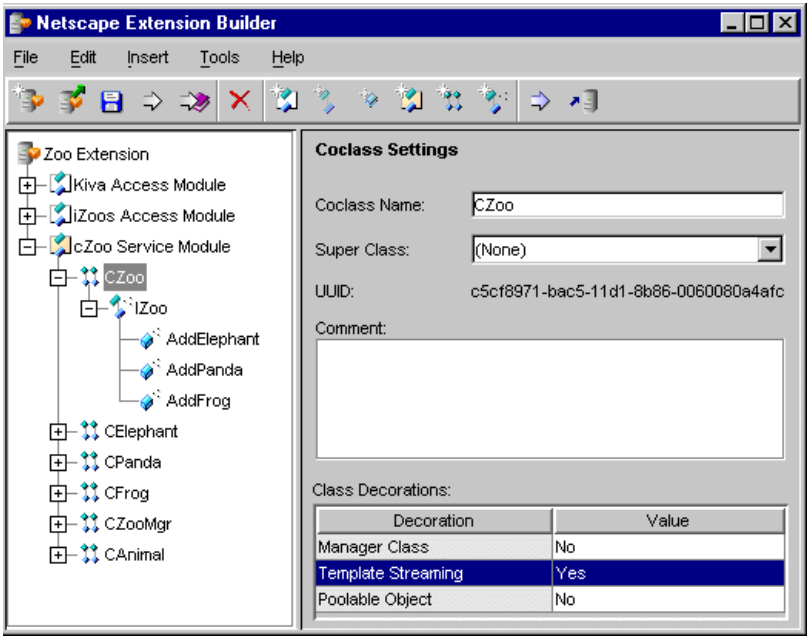
Template streaming is useful for coclasses that represent lists of objects whose data might be streamed to the browser.

In Netscape Extension Builder Designer:

1. Highlight a coclass for which you want to enable template streaming.
2. In the component view, set the Template Streaming decoration to *Yes*.

   This indicates that the results obtained by methods of this class are handled by the Netscape Template Engine, and the results will be streamed back to the user.
3. Finish designing the extension, and generate IDL files as you would for any extension.

For example, if you open the zoo.gxp project file in Netscape Extension Builder Designer and expand the tree view, a window would appear like the one shown in the following figure. In this example, CZoo is the only coclass with Template Streaming enabled.



Completing Stubs to Support Template Streaming

When the KIDL Compiler detects Template Streaming decorations, it generates source code for a particular Netscape Application Server-defined interface: either IGXTemplateData in a C++ extension or ITemplateData in a Java extension. The generated code appears in the stub files corresponding to coclasses you decorated, and you must complete the method stubs to provide the appropriate streaming implementation.

The methods in the IGXTemplateData or ITemplateData interface are used to iterate through a set of data. In applications, these interfaces are used by EvalOutput( ) or evalOutput( ) to stream data to the browser or OCL client.

Template Streaming in C++ Extensions

In Netscape Extension Builder Designer, suppose you enabled template streaming on a coclass named CTeacher. Then subsequent code generation would produce a corresponding C++ stub file, CTeacher.cpp.

Before you complete the method stubs of IGXTemplateData, this interface would appear as in the following code fragment. For an example of completed method stubs, see “Template Streaming in Java Extensions” on page 99.

```
// Stub method bodies for interface: IGXTemplateData
HRESULT
CTeacher::IsEmpty(
      /* [in] */ LPSTR group,
      /* [out] */ BOOL *empty)
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   GXASSERT(FALSE, GXASSERT_WARNING,
   "No implementation for CTeacher::IsEmpty");
   return hr;
}
HRESULT
CTeacher::MoveNext(
      /* [in] */ LPSTR group)
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   GXASSERT(FALSE, GXASSERT_WARNING,
   "No implementation for CTeacher::MoveNext");
   return hr;
}
HRESULT
CTeacher::GetValue(
      /* [in] */ LPSTR szExpr,
      /* [out] */ IGXBuffer **ppBuff)
```

```
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   GXASSERT(FALSE, GXASSERT_WARNING,
   "No implementation for CTeacher::GetValue");
   return hr;
}
HRESULT
CTeacher::SetHint(
      /* [in] */ LPSTR group,
      /* [in] */ DWORD flags,
      /* [in] */ unsigned long max,
      /* [in] */ IGXValList *pVal)
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   GXASSERT(FALSE, GXASSERT_WARNING,
   "No implementation for CTeacher::SetHint");
   return hr;
}
```

Note the three method stubs in the interface: IsEmpty( ), MoveNext( ), and GetValue( ). First, the Netscape template engine makes a callback to IsEmpty( ) to determine whether any data was returned. Next, MoveNext( ) is called to move to the next block of data in the result set, until the end is reached. Finally, for each row in the result set, GetValue( ) is called to retrieve field values.

Any class that supports streaming can be used as a parameter to EvalOutput( ). For example:

```
 IMyInterface *pIf;

 //
 // Get a pointer to IMyInterface
 //

 HRESULT hr=GXE_FAIL;
```

```
IGXTemplateData *pData=NULL;

// Navigate to the IGXTemplateData interface
if(((hr=pIf->QueryInterface(IID_IGXTemplateData,
(LPVOID*)&pData))==GXE_SUCCESS)&&pData)
{
 // Stream back to the client. The template engine
 // will use the IGXTemplateData method callbacks
 // to accomplish streaming

 EvalOutput("mypage.html", pData, NULL, NULL, NULL);
 pData->Release();
}
```

Template Streaming in Java Extensions

In addition to completing method stubs for your user-defined interfaces, you must locate the ITemplateData interface code and complete the method stubs defined in this interface.

Any class that supports template streaming, by implementing the ITemplateData interface, can be used as a parameter to evalOutput( ). For example:

```
public int execute()
{
   IZoo zoo = getZoo();

   return evalOutput("zoo\Templates\ZooListing.html",
       (ITemplateData) zoo, null, null, null);
}
```

In the Zoo sample extension, CZoo is the only coclass that uses the Template Streaming decoration. This coclass is in the cZoo service module, so the stub file you want to edit is:

```
zoo\java\zoo\cZoo\CZoo.java
```

As an example of completing method stubs for template streaming, you might modify the Zoo extension using the following procedure. The complete implementation is shown in “The CZoo.java File” on page 104.

1. Open CZoo.java.

   Note the following code near the top of the file:

```
public class CZoo
   implements zoo.IZoo, com.kivasoft.ITemplateData
```

2. Locate the beginning of the method bodies for ITemplateData.

   The isEmpty( ) method appears first. You might complete it as shown in the following sample code:

```
// Method bodies for interface: ITemplateData
   public boolean isEmpty(
      java.lang.String group)
   {
      // ignore group parameter, not using it
      return !mAnimalEnumeration.hasMoreElements();
   }
```

   The isEmpty( ) method determines whether there are any more elements in the result set. You can complete this method in several ways. The example code uses an enumeration.

3. Locate the next method stub, moveNext( ), which iterates through the result set.

   You might complete it as shown in the following sample code:

```
public int moveNext(
      java.lang.String group)
   {
      // ignore group parameter, not using it

      int retVal = -1; // if empty

      if (mAnimalEnumeration.hasMoreElements()) {
         mCurrentAnimal = (IAnimal)
            mAnimalEnumeration.nextElement();
         retVal = 0;
```

```
      }

      return retVal;
   }
```

4. Locate the next method stub, getValue( ), which dynamically retrieves the specified values.

   The szExpr string is the current tag from the template as EvalOutput( ) processes the template. If the EvalOutput( ) method understands the tag, it returns the corresponding string data from the selected animal object, or from the zoo object itself.

   Note also that getValue( ) must return an IBuffer value, instead of directly returning a string value.

   You might complete getValue( ) as shown in the following sample code:

```
public com.kivasoft.IBuffer getValue(
      java.lang.String szExpr)
   {
      String outString = "Error animal";

      System.out.println("getValueString: " + szExpr);
      if (mCurrentAnimal != null && szExpr != null) {

         if (szExpr.equals("AnimalName")) {
            outString = mCurrentAnimal.getName();
         } else if (szExpr.equals("AnimalType")) {

            if (mCurrentAnimal instanceof IElephant) {
               outString = "Elephant";
            } else if (mCurrentAnimal instanceof IPanda) {
               outString = "Panda";
            } else if (mCurrentAnimal instanceof IFrog) {
               outString = "Frog";
            }

         } else if (szExpr.equals("AnimalId")) {
            outString = mCurrentAnimal.getId() + "";
         } else if (szExpr.equals("AnimalData")) {

            if (mCurrentAnimal instanceof IElephant) {
```

```
                IElephant elephant = (IElephant)
                   mCurrentAnimal;
                outString = elephant.getTrunkSize() + "feet";
             } else if (mCurrentAnimal instanceof IPanda) {
                IPanda panda = (IPanda) mCurrentAnimal;
                int type = panda.getType();
                if (type == 0) {
                   outString = "Giant Panda";
                } else {
                   outString = "Red Panda";
                }
             } else if (mCurrentAnimal instanceof IFrog) {
                IFrog frog = (IFrog) mCurrentAnimal;
                outString = frog.getColor();
             }

          } else if (szExpr.equals("ZooName")) {
             outString = getName();
          }
       }

       IBuffer buffer = GX.CreateBufferFromString(outString);
       return buffer;
    }
```

5. Ignore the setHint( ) method by having it return 0.

Creating a Template

In addition to completing method stubs, you must create a template to display the results. The template is used by the application component that calls the extension.

In the Zoo extension, the sample template file is:

```
zoo\Templates\ZooListing.html
```

This HTML file contains the following text:

```
<HTML><HEAD><TITLE>Zoo Listing</TITLE></HEAD>
<BODY bgcolor="#FFFFFF" text="#000000" link="#CC0000" vlink="#FF3300"
alink="#003366" basefont=3>
```

```
<H1>A list of the animals currently in the zoo:</H1>

<TABLE BORDER=1 CELLPADDING=2 CELLSPACING=0>
  <TR><TH> animal name
 </TH><TH> animal type
 </TH><TH> animal id
 </TH><TH> animal data
 </TH></TR>
    %gx type=tile id=Zoo MAX=100%
    <TR ALIGN=RIGHT>
    <TD>%gx type=cell id="AnimalName"%%/gx%
</TD>
    <TD>%gx type=cell id="AnimalType"%%/gx%
</TD>
    <TD>%gx type=cell id="AnimalId"%%/gx%
</TD>
    <TD>%gx type=cell id="AnimalData"%%/gx%
</TD>
    </TR>
    %/gx%
</TABLE>
<br>

<a href="/ZooAdmin/index.html">
back to main page
</a>

</BODY>
</HTML>
```

Testing an Application

After the extension and template are completed, you can test the new functionality in an application.

Note Before tesing the new functionality in the application, be sure that you have stopped NAS and NES before compiling the extension and have re-started them before loading the test page in the browser.

To test the new functionality in an application, perform the following steps:

1. From your web browser, load a test page that uses the application. Point your browser to http://<my_host_name>/extensions, and select the Zoo example. The web page that tests the Zoo extension is:

   ```
   zoo\applogic\docs\ZooAdmin\index.html
   ```

   This page offers two choices. You can either add animals or list animals. Each choice relies on a separate application object for processing. The application component that displays the list is the one whose functionality was extended by the Zoo extension.

2. To add animals, click “Add animals” or load the AddAnimal.html page. You can add arbitrary values.

3. When you decide you are ready to list the animals, return to index.html and click “List animals.”

   This invokes an application component whose functionality has been extended with template streaming. A streamed result set is returned and displayed using the ZooListing.html template.

The CZoo.java File

The CZoo.java file contains completed method stubs for a user-defined interface named IZoo, as well as the Netscape Application Server-defined ITemplateData interface.

```
//
//   This file is initially generated by KIDL - Edit as
//   necessary to complete
//
package zoo.cZoo;
import com.kivasoft.*;
import com.kivasoft.types.*;
import com.kivasoft.util.*;
import zoo.*;
import java.util.*;
public class CZoo
   implements zoo.IZoo, com.kivasoft.ITemplateData
```

```
{
   public zoo.cZoo.CZooMgr m_Module;
   protected String mName = null;
   protected Vector mAnimalList = null;
   protected Enumeration mAnimalEnumeration = null;
   protected IAnimal mCurrentAnimal = null;
   public CZoo(zoo.cZoo.CZooMgr module, String name)
   {
      m_Module=module;
      mName = name;
      mAnimalList = new Vector();
      reset();
   }
   public void finalize()
   {
   }
   public String getName()
   {
      return mName;
   }
   public void reset()
   {
      mAnimalEnumeration = mAnimalList.elements();
      if (mAnimalEnumeration.hasMoreElements()) {
         mCurrentAnimal = (IAnimal)
            mAnimalEnumeration.nextElement();
      }
   }
   // Method bodies for interface: IZoo
   public zoo.IElephant addElephant(
      java.lang.String name,
      int id,
      double trunkSize)
   {
      IElephant elephant = new CElephant(m_Module, name, id,
         trunkSize);
      mAnimalList.addElement(elephant);
      return elephant;
   }
   public zoo.IPanda addPanda(
      java.lang.String name,
      int id,
```

```
   int type)
{
   IPanda panda = new CPanda(m_Module, name, id, type);
   mAnimalList.addElement(panda);
   return panda;
}
public zoo.IFrog addFrog(
   java.lang.String name,
   int id,
   java.lang.String color)
{
   IFrog frog = new CFrog(m_Module, name, id, color);
   mAnimalList.addElement(frog);
   return frog;
}
// Method bodies for interface: ITemplateData
public boolean isEmpty(
   java.lang.String group)
{
   // ignore group parameter, not using it
   return !mAnimalEnumeration.hasMoreElements();
}
public int moveNext(
   java.lang.String group)
{
   // ignore group parameter, not using it

   int retVal = -1;
   if (mAnimalEnumeration.hasMoreElements()) {
      mCurrentAnimal = (IAnimal)
         mAnimalEnumeration.nextElement();
      retVal = 0;
   }
   return retVal;
}
public com.kivasoft.IBuffer getValue(
   java.lang.String szExpr)
{
   String outString = "Error animal";

   System.out.println("getValueString: " + szExpr);
   if (mCurrentAnimal != null && szExpr != null) {
```

```
      if (szExpr.equals("AnimalName")) {
         outString = mCurrentAnimal.getName();
      } else if (szExpr.equals("AnimalType")) {

         if (mCurrentAnimal instanceof IElephant) {
            outString = "Elephant";
         } else if (mCurrentAnimal instanceof IPanda) {
            outString = "Panda";
         } else if (mCurrentAnimal instanceof IFrog) {
            outString = "Frog";
         }
      } else if (szExpr.equals("AnimalId")) {
         outString = mCurrentAnimal.getId() + "";
      } else if (szExpr.equals("AnimalData")) {
         if (mCurrentAnimal instanceof IElephant) {
            IElephant elephant = (IElephant)
               mCurrentAnimal;
            outString = elephant.getTrunkSize() + "feet";
         } else if (mCurrentAnimal instanceof IPanda) {
            IPanda panda = (IPanda) mCurrentAnimal;
            int type = panda.getType();
            if (type == 0) {
               outString = "Giant Panda";
            } else {
               outString = "Red Panda";
            }
         } else if (mCurrentAnimal instanceof IFrog) {
            IFrog frog = (IFrog) mCurrentAnimal;
            outString = frog.getColor();
         }
      } else if (szExpr.equals("ZooName")) {
         outString = getName();
      }
   }
   IBuffer buffer = GX.CreateBufferFromString(outString);
   return buffer;
}
// We're not making use of this callback
public int setHint(
   java.lang.String group,
   int flags,
```

```
        int max,
        com.kivasoft.IValList pVal)
    {
        return 0;
    }
}
```

Chapter

8

Managing State and Session Information

This chapter describes extension state and session management, which maintains application state and user session information for extended technology.

The following topics are included in this chapter:

- About State and Session Management
- Using Application State in an Extension
- Using Sessions in an Extension
- Local Versus Distributed State and Session Data
- Example Code Walkthrough

About State and Session Management

Application state and user session information is maintained by extension state and session management for the extended technology. You most likely want to enable this feature when your legacy solution tracks user information on a user-by-user basis, or the legacy solution is part of an application that is distributed across multiple servers.

Managing state and session information is supported by Netscape extensions as well as by applications, but the usage is slightly different for extensions. This chapter presents only those differences.

When enabling an extension to use state and session data, there are no decorations to add in Netscape Extension Builder Designer. You implement the code directly, after the source code is generated.

Extension state and session management is provided through the following Netscape Application Server interfaces:

Java • IState2 and ISession2

C++ • IGXState2 and IGXSession2

As you are filling in the method stubs of an extension, you can use these interfaces to enable an extension to store the user session and/or application state information.

For information about managing user sessions and application state from applications, see the *Programmer's Guide*.

Using Application State in an Extension

Application state management in extensions works the same as it does in application components, except for the method you use to access the state interface. The state interface is IGXState2 for C++ extensions and IState2 for Java extensions.

State data is useful if, for example, you want to track the total number of users of your extension.

To use application state:

1. Create the root node of the state tree.
2. Set and manage the state data.

Creating the Root Node in a C++ Extension

To create the root node of the state tree, use the GXGetStateTreeRoot( ) helper function. The following code example shows how to create a state tree and a child node:

```
HRESULT hr;

hr = GXGetStateTreeRoot(m_pContext, GXSTATE_DISTRIB|GXSTATE_PERSISTENT,
  "Grammy", &m_pStateRoot);

if (hr == NOERROR && m_pStateRoot)
{
   IGXState2 *pState = NOERROR;
   hr = m_pStateRoot->GetStateChild("Best Female Vocal",
      &pState);
   if (hr != NOERROR || !pState)
   {
     hr = m_pStateRoot->CreateStateChild("Best Female Vocal",
         0, GXSTATE_DISTRIB|GXSTATE_PERSISTENT, &pState);
```

For more information on GXGetStateTreeRoot( ), see Appendix A, “C++ Helper Functions.”

Creating the Root Node in a Java Extension

To create the root node of the state tree, use the GetStateTreeRoot( ) helper static method in the GXContext class. The following code example shows how to create a state tree and a child node:

```
IState2 tree = GXContext.GetStateTreeRoot(m_Context,
   GXSTATE.GXSTATE_DISTRIB|GXSTATE.GXSTATE_PERSISTENT,
   "Grammy");

if (tree!=null)
{
   IState2 child = tree.getStateChild("Best Female Vocal");
```

```
if (child == null)
{
   child = tree.createStateChild("Best Female Vocal", 0,
     GXSTATE.GXSTATE_DISTRIB|GXSTATE.GXSTATE_PERSISTENT);
```

For more information on GetStateTreeRoot( ), see Appendix B, “Java Helper Static Methods.”

Setting and Managing State Data

To set and manage the state data, use methods defined either in the IGXState2 interface for C++ extensions or in the IState2 interface for Java extensions. In C++ extensions, for example, use CreateStateChild( ) to create child nodes, and use SetStateContents( ) to place state information into the node. Similarly, for Java extensions use createStateChild( ) and setStateContents( ).

The string parameter in these methods cannot exceed 31 characters.

Using Sessions in an Extension

User sessions work the same in an extension as they do in an application component, except for the method you use to access the session interface. The session interface is IGXSession2 for C++ extensions and ISession2 in Java extensions.

Internally, session data for extensions and applications is stored in separate memory spaces; however, extension session data has the same session ID as that of its calling application. Therefore, in order for session management to work in an extension, the application must have previously created an application session.

To use sessions in an extension, use the thread session method as described in the following paragraphs. This method is implemented by the Manager class in each of your extension services.

C++ In C++ extensions, use the GetThreadSession( ) method, which has the following syntax:

```
HRESULT GetThreadSession(
   DWORD sessionType,
```

```
   IGXSession2 **ppSession);
```

sessionType is either GXSESSION_LOCAL or GXSESSION_DISTRIB, indicating that the session is either local or distributed. ppSession is a pointer to the created IGXSession2 object.

The following code example accesses a local session:

```
IGXSession2* pSession2;
mgr->GetThreadSession(GXSESSION_LOCAL, &pSession2)
```

Java In Java extensions, use the getThreadSession( ) method, which has the following syntax:

```
public ISession2 getThreadSession(
   int sessionType)
```

sessionType is either GXSESSION.GXSESSION_LOCAL or GXSESSION.GXSESSION_DISTRIB, indicating that the session is either local or distributed

The following code example accesses a distributed session:

```
ISession2 session2=
mgr.getThreadSession(GXSESSION.GXSESSION_DISTRIB)
```

Local Versus Distributed State and Session Data

State and session objects can be local or distributed. An extension can have local and distributed state/session objects at the same time.

Distributed data is replicated on multiple servers and can be accessed from any of those servers. By contrast, local data can be accessed from only one server.

Use distributed data whenever possible, for two main reasons. First, Java extensions do not support local state or session data, so you are limited to using C++ extensions.

Second, local state or session data requires the use of sticky load balancing. Applications must be "sticky" if they access extensions that rely on local state or session data. A sticky application is one that always returns to the same process for the same user. Sticky load balancing is described in more detail in *Administration Guide*.

Using a Distributed Session or State

Use a distributed session or state if the data to be put into the session or state can be completely described using any combination of the following methods:

C++ SetValString( ), SetValInt( ), SetValBlob( ) in the IGXValList interface.

Java setValString( ), setValInt( ), setValBlob( ) in the IValList interface.

In other words, data can be distributed if you can represent it as either a string, an integer, or a blob.

For C++ extensions, session data is stored in an IGXValList object. To get this data, use:

```
IGXSession2.GetSessionData()
```

For Java extensions, session data is stored in an IValList object. To get this data, use:

```
ISession2.getSessionData()
```

Using a Local Session or State

Local state or session data is supported only for C++ extensions. Use a local session or state if the data does not meet the criteria for distributed session or state. For example, objects that rely on physical connections, such as database or result-set objects, cannot be placed into a distributed session or state.

To place data into a local session, derive the data object from the IGXObject interface. The IGXObject interface allows the data in a session to be destroyed at the same time that the session is destroyed. A session is destroyed whenever it times out.

If the data to be placed in the local session has implemented the IGXObject interface, place the data directly into an IGXValList.

However, if the data does not implement IGXObject, perform the following steps to place the data in a local session:

1. Use the GX_LOCALSESSION_CLASS_DECL macro to define a helper class that will go in the session. For example:

```
GX_LOCALSESSION_CLASS_DECL(MyClassName);
```

2. Use the GX_LOCALSESSION_CLASS_INST macro to create an instance of the helper class, which will be used to wrap the instance of your user-defined class. For example:

```
GXObject* pWrapObject=GX_LOCALSESSION_CLASS_INST (MyClassName,
pMyHeapObjectInstance);
```

3. Put pWrapObject in the session's IGXValList object as follows:

```
GXVAL val;
val.vt = GXVT_UNKNOWN;
val.v.punkVal=pWrapObject;
pValList->SetVal("MySessionData",&val);
pSession->SaveSession(0);
```

4. Use the GX_LOCALSESSION_CLASS_UNWRAP macro to retrieve your original object. For example:

```
IGXValList *pSessVlist=session->GetSessionData();
GXVAL val;
if((hr=pSessVList->GetVal("MYSESSIONDATA", &val))==
    GXE_SUCCESS)
{
   GXObject *pWrappedObject=val.u.punkVal;
   MyClassName* pMyHeapObjectInstance=
   GX_LOCALSESSION_CLASS_UNWRAP(MyClassName, pWrappedObject);
}
```

Example Code Walkthrough

In the Netscape Extension Builder sample directory is a subdirectory named Auction, which contains a Java extension that demonstrates the use of state and session management. The Auction extension is a complete example.

The sample application simulates an online auction. On the auction home page you can perform any of the following tasks:

- List merchandise available for bidding, with the current bid prices. This task relies on application state data, because bid prices come from all users of the application.
- List your own bids. This task relies on session data, because every bid you make during the current login session must be tracked.
- Submit a bid. This task requires updating of state and session information, because your new bids must appear on both lists.

To manage state and session data, perform the following steps:

1. Implement the IState2 Interface

   You add this code in the service module's manager class.

2. Implement the ISession2 Interface

   You add this code in the class representing a user.

Implement the IState2 Interface

The manager class is CAuctionModule and is represented by this file:

```
sample\auction\java\Auction\myext\CAuctionModule.java
```

The complete contents of this file, including additional comments, are as follows:

```
//
//  This file is initially generated by KIDL - Edit as
//  necessary to complete
//

package Auction.myext;

import com.kivasoft.*;
import com.kivasoft.util.*;
import com.kivasoft.types.*;
import Auction.*;
```

```
public class CAuctionModule extends
  com.kivasoft.bind.BinderBase
  implements com.kivasoft.IModule, Auction.IAuctionModule
{
  public com.kivasoft.IContext m_Context;
  public static final String m_extensionNodeName =
     "NASExtensionNode";
  public static final String m_auctionNodeName =
     "AuctionExtensionNode";

  private IState2 m_auctionState;
  private IAuctionPool m_auctionPool;
  public CAuctionModule()
  {
    super();
    m_auctionState = null;
    m_auctionPool = null;
  }
  public void finalize()
  {
    super.finalize();

    //***
    //*** Add your own code to the destructor here.
    //***
  }

 // use init to set up the state information

 public synchronized int init(com.kivasoft.IObject obj) {
    int res;

    res = super.init(obj);

   com.kivasoft.util.Util.dictionaryPut(obj, "Auction.IAuctionModule",
   this);
    m_Context=(com.kivasoft.IContext) obj;

    IModuleState2 stateModule = (IModuleState2)
  ((IDictionary)m_Context).get("com.kivasoft.IState2Module");
```

```
if (stateModule != null)

// get a root node
{
  IState2 extensionState =
     stateModule.getStateTree(m_extensionNodeName,
     GXSTATE.GXSTATE_LOCAL);

  if (extensionState != null)

  // create an application-specific node
  {
    IState2 auctionState =
      extensionState.createStateChild(m_auctionNodeName,
      0, GXSTATE.GXSTATE_LOCAL);
    if (auctionState != null)

    // populate the state data with merchandise names
    // and minimum bid prices
    {
      IValList merchandiseList =
         auctionState.getStateContents();
      merchandiseList.setValInt(new
         String("15 Inch Monitor"), 250);
      merchandiseList.setValInt(new
         String("PCS Phone"), 190);
      merchandiseList.setValInt(new
         String("GPS System"), 1200);
      merchandiseList.setValInt(new
         String("Web Browser"), 0);
      merchandiseList.setValInt(new
         String("Pebble Beach Vacation"), 1500);
      merchandiseList.setValInt(new
         String("DVD Player"), 150);
      merchandiseList.setValInt(new
         String("Digital Camera"), 300);
      auctionState.setStateContents(merchandiseList);
      auctionState.saveState(GXSTATE.GXSTATE_LOCAL);
      m_auctionState = auctionState;
    }
  }
}
```

```
  return res;
}

// return the state interface
public IState2 getState()
{
    return m_auctionState;
}

// Helper functions

// Method bodies for interface: IAuctionModule
public Auction.IAuctionPool createAuctionPool()
{
  // one pool only
  if (m_auctionPool != null)
    return m_auctionPool;
  else
  {
    CAuctionPool auctionPool = new CAuctionPool(this);
    return auctionPool;
  }
}

public Auction.IParticipant getParticipant(
  java.lang.String UserId)
{
  CParticipant user = new CParticipant(this, UserId);
  return user;
}

// if a bid is made, update the state data for the
// corresponding merchandise
public void bid(Auction.IBid bidorder)
{
  IValList merchandiseList = getState().getStateContents();
  merchandiseList.setValInt(bidorder.getName(),
     bidorder.getPrice());
  getState().setStateContents(merchandiseList);
  getState().saveState(GXSTATE.GXSTATE_LOCAL);
}
```

```
// remainder of this file is automatically generated

public static final String
    appNameLocal="CAuctionModuleL";
public static final String
    appNameDistributed="CAuctionModuleD";
public static final String
    appNameCluster="CAuctionModuleC";

public ISession2 getThreadSession(int sessionType)
{
  String appName;
  ISession2 session=null;
  int dwflags;

  if ((sessionType & GXSESSION.GXSESSION_LOCAL) > 0)
  {
    appName = appNameLocal;
    dwflags = GXSESSION.GXSESSION_LOCAL;
  }
  else if ((sessionType & GXSESSION.GXSESSION_CLUSTER) > 0)
  {
    appName = appNameCluster;
    dwflags = GXSESSION.GXSESSION_CLUSTER;
  }
  else
  {
    appName = appNameDistributed;
    dwflags = GXSESSION.GXSESSION_DISTRIB;
  }

  IModuleExtensionData extData = (IModuleExtensionData)
     ((IDictionary)m_Context).get
     ("com.kivasoft.IModuleExtensionData");

  if(extData!=null) {
    String sessionID = extData.getSessionID(null);
    IModuleSession modSession = (IModuleSession)
       ((IDictionary)m_Context).get
       ("com.kivasoft.IModuleSession");

    ISessionGroup sessionGroup =
```

```
            modSession.getSessionGroup(appName);
        session = sessionGroup.getSession(sessionID, null, 1);
        if (session == null) {

          // sync with application's session
          int tmpflags = extData.getSessionFlags(null);
          if ((tmpflags & GXSESSION.GXSESSION_TIMEOUT_CREATE)
               > 0)
            dwflags |= GXSESSION.GXSESSION_TIMEOUT_CREATE;
          else
            dwflags |= GXSESSION.GXSESSION_TIMEOUT_ABSOLUTE;

          if ((tmpflags & GXSESSION.GXSESSION_PERSISTENT) > 0)
            dwflags |= GXSESSION.GXSESSION_PERSISTENT;

          int timeout = extData.getSessionTimeout(null);

          session = sessionGroup.createSession(dwflags,
             timeout, null, sessionID);
        }
        return session;
      }
      return null;
    }
}
```

Implement the ISession2 Interface

In this example, the Participant class corresponds to an individual user of the auction application. The Participant class is represented by this file:

`sample\auction\java\Auction\myext\CParticipant.java`

The complete contents of this file, including additional comments, are as follows:

```
//
//  This file is initially generated by KIDL - Edit as
//  necessary to complete
//

package Auction.myext;
```

```
import com.kivasoft.*;
import com.kivasoft.types.*;

public class CParticipant
  implements Auction.IParticipant
{
  public Auction.myext.CAuctionModule m_Module;
  private String m_userId;

  public CParticipant(Auction.myext.CAuctionModule module,
      String userId)
  {
    m_Module=module;
    m_userId = userId;
  }
  public void finalize()
  {

    //***
    //*** Add your own code to the destructor here.
    //***
  }

  // Method bodies for interface: IParticipant
  // Get a list of your bids
  public com.kivasoft.IValList getBiddingList()
  {
    ISession2 session = m_Module.getThreadSession
       (GXSESSION.GXSESSION_DISTRIB);
    IValList list = session.getSessionData();
    return list;
  }
  // Put an IBid object into a session
  public int submitBid(
    Auction.IBid bid)
  {
    ISession2 session = m_Module.getThreadSession
       (GXSESSION.GXSESSION_DISTRIB);
    IValList list = session.getSessionData();

    // Put data into an IValList.
```

```
    // For this example, assume the name of
    // each merchandise item is unique
    list.setValInt(bid.getName(), bid.getPrice());

    // Assign the IValList to the session
    session.setSessionData(list);

    // Save the session data
    session.saveSession(GXSESSION.GXSESSION_DISTRIB);

    // Submit the bid to the Manager class
    m_Module.bid(bid);
    return 0;
  }
  public Auction.IBid newBid(
    java.lang.String name,
    int price)
  {
    CBid bid = new CBid(m_userId, name, price);
    return bid;
  }
}
```

Chapter

9

Using Object Pools

This chapter describes object pooling, a solution to potential limited-resource issues, including bottlenecks that occur when there are not enough resources to meet clients' demands.

The following topics are included in this chapter:

- Introduction to Object Pooling
- Process Overview for Object Pooling
- Example of Object Pooling
- Inside a Pool-Enabled Extension
- Inside the Object Pool Manager
- Adding Decorations for Object Pooling
- Completing Stubs for Pooled-Object Creation Methods
- Completing Stubs for Object Evaluation Methods
- Reference List of Object Pooling Decorations

Introduction to Object Pooling

Object pooling solves potential limited-resource issues. Limited resources can cause performance bottlenecks when there are not enough resources to meet clients' demands. For example, connections to networked resources, such as databases, require non-trivial amounts of time to create and destroy. Often, in high-throughput applications, client objects must wait for a connection object to become available, creating a bottleneck in the flow of the application.

With object pooling, many clients can share a limited resource such as a connection, using it only when they need it. In this way, the performance cost of creating and destroying the resource is reduced. This benefit applies to any client of the pool-enabled extension, whether the calling client is an application or another extension.

Summary of Object Pooling Tasks

To enable object pooling, extension writers and server administrators work together as follows:

- In Netscape Extension Builder Designer, an extension writer adds object pooling decorations. These tasks are described in "Adding Decorations for Object Pooling" on page 140.
- In the generated source code, an extension writer completes method stubs related to object pooling. These tasks are described in the following sections:
  - "Completing Stubs for Pooled-Object Creation Methods" on page 150.
  - "Completing Stubs for Object Evaluation Methods" on page 152.

  Sample implementation of these methods is also presented in Appendix E, "The ConnManager.cpp File."
- In the Netscape Application Server registry file, the server administrator configures any object pools referenced in extension code. This task is described in Chapter 11, "Deploying and Managing a Netscape Extension."

Key Concepts

Object pooling involves the following key concepts:

- Object Pool
- Virtual and Physical Objects
- Client
- Object Pool Manager

Object Pool

An object pool is a set of limited resources, such as connections, that can be reserved for use by clients and then returned to the pool (for probable reuse) when the object is no longer needed. Reserving and returning pooled objects avoids the overhead of separately creating and destroying an object each time a client requests it. Multiple object pools can be used. For example, one object pool might contain database connection objects, and another pool might contain CICS connection objects.

Virtual and Physical Objects

Without object pooling, whenever a client of an extension (typically an application) requests an object, a physical object is created and destroyed when no longer needed.

By contrast, when an extension uses object pooling, the application's request for a poolable object generates a virtual object instead. The virtual object supports all the methods of the requested object, but the application sees only the virtual object.

When an application calls an interface method from the virtual object, the virtual object's implementation requests a physical object from the pool and delegates the request to the physical object. When the request is complete, the extension returns the physical object to the Object Pool Manager for use by other virtual objects.

Client

In the context of object pooling, a client is the code that calls into the extension. The client is typically an application but can also be another extension. The extension requests objects, and the Object Pool Manager makes callbacks to the extension to determine how to fulfill the request.

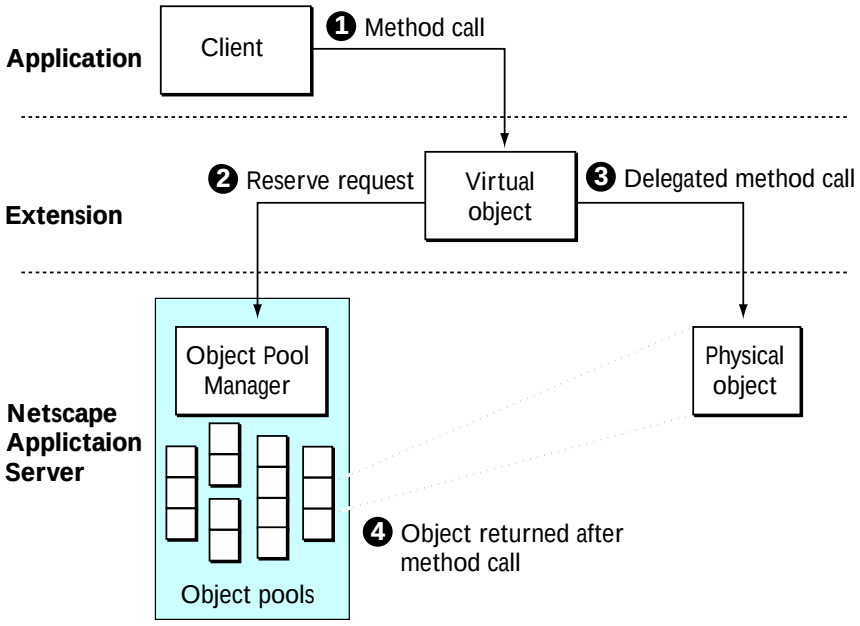
Object Pool Manager

The Object Pool Manager is a service of Netscape Application Server. In response to clients' requests for objects, the Object Pool Manager controls one or more pools by reserving and releasing the objects in the pool. The Object Pool Manager performs the following tasks:

- Queues virtual objects' requests for physical objects.
- Marks physical objects as either free or reserved.
- Attempts to create physical objects when necessary.
- Destroys physical objects in a pool, based on idle time or usage limits.

Process Overview for Object Pooling

The following figure shows the interrelationship between a pool-enabled extension and the key components of object pooling:



1. The client of an extension calls an interface method on the virtual object.
2. The virtual object's implementation reserves a matching physical object from a named pool.
3. The method call is delegated to the physical object.
4. When the method call is completed, the physical object is returned to the appropriate pool for use by other virtual objects. The Object Pool Manager uses a timer thread that periodically releases unused physical objects after a timeout.

For a more detailed description of object pooling processes, see the following sections:

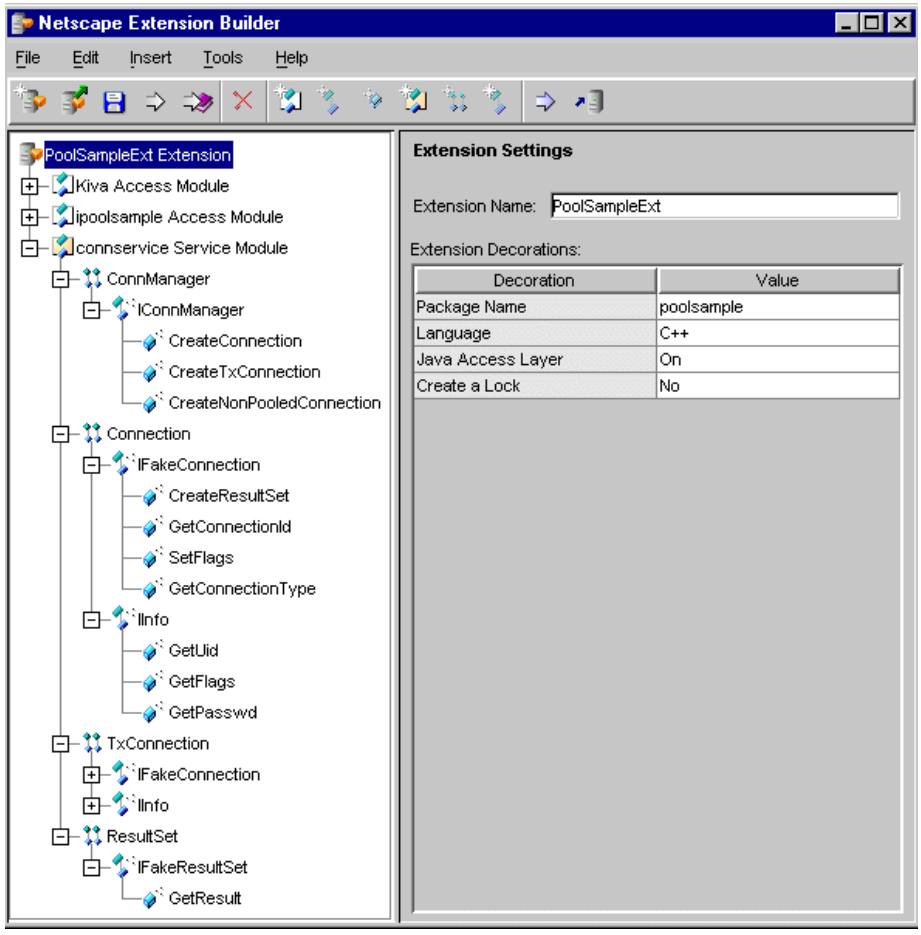
- Inside a Pool-Enabled Extension
- Inside the Object Pool Manager

Example of Object Pooling

In the Netscape Extension Builder sample\poolsample directory is an extension named PoolSampleExt, built in C++ with a Java Access Layer. Many topics in this chapter refer to this sample extension. To better understand the concepts and tasks presented, you might want to open the sample extension while reading about object pooling.

To open PoolSampleExt

1. Start Netscape Extension Builder Designer.
2. From the File menu, open the poolsample.gxp project file.
3. After you expand some of the nodes in the tree view, a window similar to the following appears:


Netscape Extension Builder
File
Edit
Insert
Tools
Help
PoolSampleExt Extension
Kiva Access Module
ipoolsample Access Module
connservice Service Module
ConnManager
IConnManager
CreateConnection
CreateTxConnection
CreateNonPooledConnection
Connection
IFakeConnection
CreateResultSet
GetConnectionId
SetFlags
GetConnectionType
IInfo
GetUid
GetFlags
GetPasswd
TxConnection
IFakeConnection
IInfo
ResultSet
IFakeResultSet
GetResult
Extension Settings
Extension Name: PoolSampleExt
Extension Decorations:
Decoration
Value
Package Name
poolsample
Language
C++
Java Access Layer
On
Create a Lock
No

About PoolSampleExt

The PoolSampleExt extension demonstrates the use of object pooling. The extension implements a “dummy” SQL execution service. In other words, no backend SQL engine exists, and the results returned are just an echo of the SQL provided to the connection.

Based on user input, the IConnManager interface can be asked to create either of two types of connections to the back-end data source: non-transactional or transactional. All connections are returned as an IFakeConnection object. Given some SQL, a connection can be asked to create a result set, which keeps a reference to the parent connection. The result set can be asked to return the SQL execution result in the form of a string.

About the Application

Using an HTML file, poolsample.html, users provide input to the application whose functionality is extended by PoolSampleExt. The application accepts the following parameters:

- The number of connections to create.

  Values exceeding 8 will cause empty result sets to be displayed, because the maximum pool size is set to 8. Maximum pool size is part of the pool configuration, found in the Netscape Application Server registry under CCS0/POOLS/POOLSAMPLE.

- The type of connections to create, either pooled, pooled transactional, or non-pooled.

- A numeric user ID and a password string, both used in creating or matching a physical connection.

- Numeric flag settings, which are used as additional matching criteria for pooled transactional connections.

- Input text, which will be given to a result set and echoed back by it.

The application creates the specified number and type of connections. It then queries the connections for their associated physical connection IDs, and returns the IDs. These IDs appear in the first section of the application’s response page.

For each connection the application creates, it also creates a result set with the given SQL, executes the GetResult( ) method from each result set, and displays the results. Each result is a string containing the connection ID of the associated physical connection. The results appear in the second section of the application's response page.

Inside a Pool-Enabled Extension

This section describes the interactions within an extension, using PoolSampleExt as an example. Two main process flows are described:

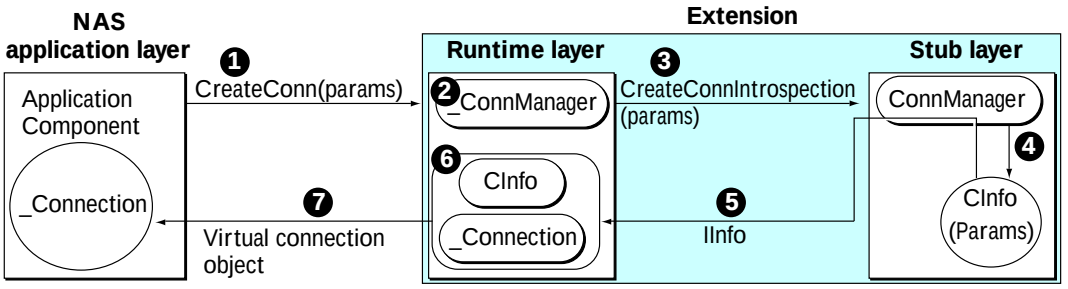
- The creation of a pooled object—a connection in this example.
- A method call on a pooled object—a virtual connection in this example.

Creating a Connection

As shown in the following figure, creating a connection involves this process:

1. An application calls CreateConnection( ), a method of the ConnManager class. ConnManager is defined by the extension programmer.
2. CreateConnection( ) acts on _ConnManager, a runtime layer object that corresponds to the ConnManager class.
3. Using the same parameters as CreateConnection( ), _ConnManager calls CreateConnectionIntrospection( ) on ConnManager.
4. ConnManager creates an introspection object, CInfo, which encapsulates the method parameters from CreateConnection( ). For more information, see "The Introspection Interface" on page 135.
5. ConnManager passes CInfo back to the Netscape Extension Builder Runtime Layer as an interface, IInfo. IInfo is the introspection interface that is implemented by CInfo and Connection.
6. The Netscape Extension Builder Runtime Layer creates a virtual connection object, _Connection, which contains the introspection object, CInfo.

7. The Netscape Extension Builder Runtime Layer passes _Connection back to the application.

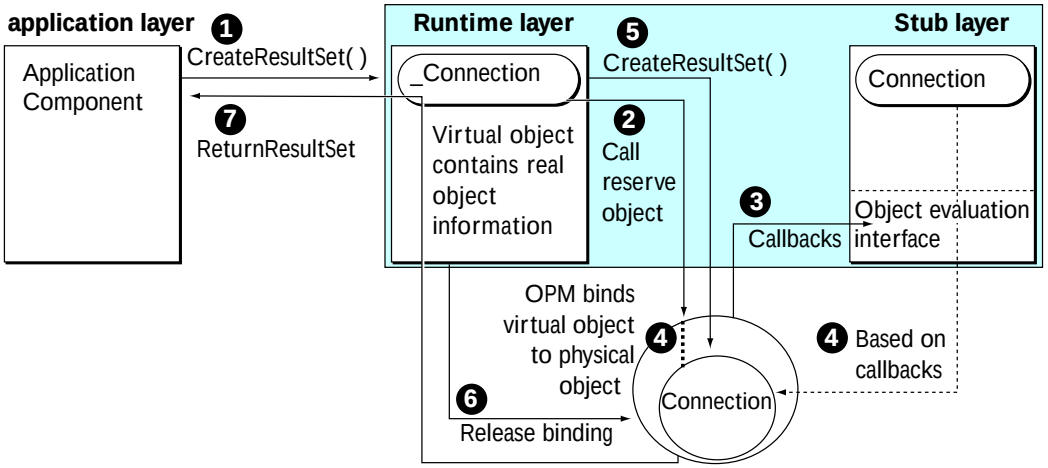


Calling a Method on a Virtual Connection

As shown in the following figure, calling a method on a virtual connection involves this process:

1. The application calls CreateResultSet( ) on the virtual object, _Connection.
2. The virtual object makes a reserve request, asking the Object Pool Manager to reserve a physical connection object that matches the virtual connection's introspection object, CInfo.
3. The Object Pool Manager makes callbacks to the object evaluation interface in the stub-file component of the extension. For more information, see "The Object Evaluation Interface" on page 136.
4. Based on the callbacks, the Object Pool Manager passes back a matching physical object. In other words, the virtual object binds to a physical object.
5. The virtual connection delegates the CreateResultSet( ) method call to the physical connection, and a result set is created.
6. After the method call is finished, the virtual connection returns the physical connection to the Object Pool Manager. In other words, the binding between virtual and physical object is broken.

7. The virtual connection object returns results to the application.



The Introspection Interface

In Netscape Extension Builder Designer, suppose you decorate a method to be a pooled-object creation method. In PoolSampleExt for example, CreateConnection( ) has its Pooled Object Creation decoration set to Yes.

When you set this decoration, subsequent code generation produces two related method stubs: one for the normal creation method, and a “twin” stub known as the create-introspection method. The twin method has a naming scheme of

```
<method>Introspection()
```

For example:

```
CreateConnectionIntrospection()
```

In addition to filling out the normal creation method, you fill out the *create-introspection method*, whose purpose is to return an introspection object.

An *introspection object* encapsulates the information that is used as the basis for matching by the Object Pool Manager, and the introspection interface provides access to this information.

An *introspection interface* is one whose methods are used to query attributes of virtual or physical objects. This interface is implemented by the introspection object.

You define an introspection interface in Netscape Extension Builder Designer. Then, when you set a coclass's Poolable Object decoration, you are also required to specify this coclass's introspection interface. For more information on specifying the introspection interface, see "Designating Objects That Can Be Pooled" on page 144.

The Object Evaluation Interface

The object evaluation interface provides methods to help the Object Pool Manager reserve objects and return them to a pool. This Netscape Application Server-defined interface is named IGXObjectEvaluation in C++ extensions, or IJavaObjectEvaluation in Java extensions.

Each class that contains pooled-object creation methods will implement the object evaluation interface. Therefore, each class's corresponding stub file contains the object evaluation interface and its method stubs. For example, PoolSampleExt has a ConnManager class for creating pooled objects, so the ConnManager.cpp file is where you would complete stubs for the IGXObjectEvaluation interface.

You never call these methods directly. Instead, the Object Pool Manager makes callbacks to your extension classes to obtain specific data. You decide how the Object Pool Manager evaluates this data when you fill out the method stubs, supplying appropriate return values for given input conditions.

The object evaluation interface defines the methods as shown in the following table. The C++ versions are listed. The Java versions are equivalent, but their names begin with a lowercase letter.

Method	Description
MatchObject( )	Determines whether a virtual object matches a physical object from a specified pool.
CreateObject( )	For a given virtual object, creates a matching physical object in a specified pool.
StealObject( )	Determines whether a physical object is eligible to be replaced.

Method	Description
ReleaseObject( )	Destroys a physical object in a specified pool.
InitObject( )	Marks a physical object as reserved.
UninitObject( )	Marks a physical object as returned to the pool (unreserved).
GetHint( )	Provides a hint string that may improve performance of the MatchObject( ) method.

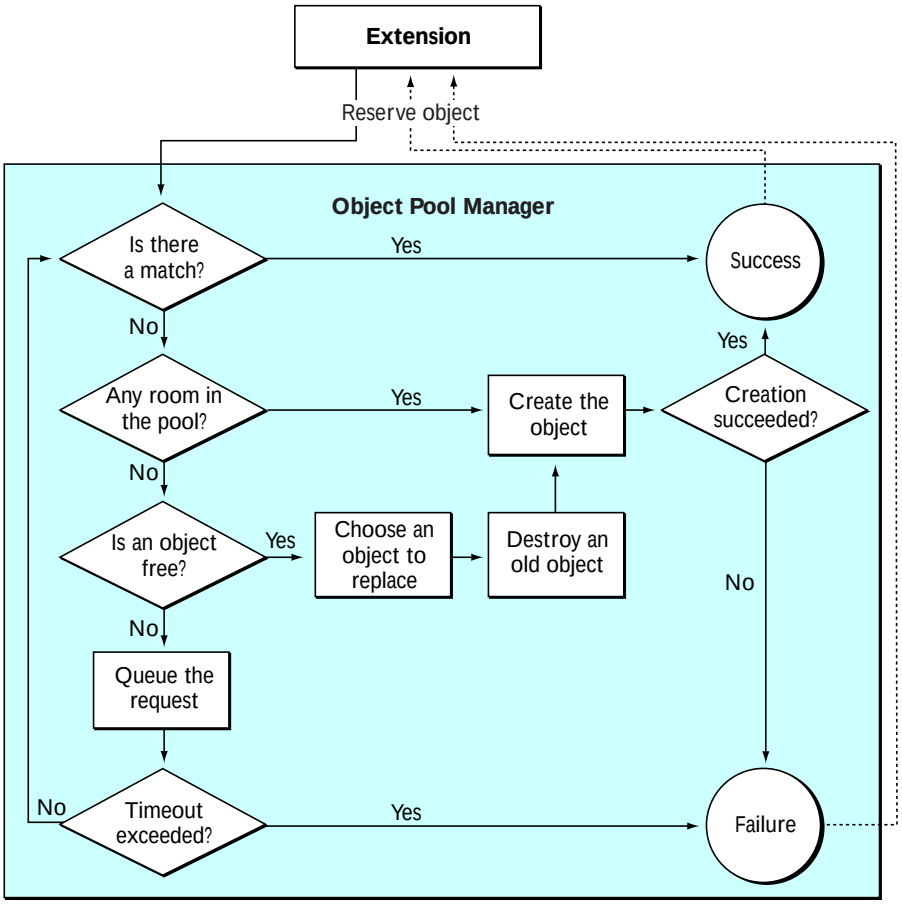
For more information on syntax and usage of the object evaluation methods, see “Completing Stubs for Object Evaluation Methods” on page 152.

Inside the Object Pool Manager

When an extension makes a request to reserve an object, the Object Pool Manager processes the request through the following possible phases:

- Matching a pooled object
- Creating a pooled object
- Replacing a pooled object
- Queuing a request
- Returning a FAILURE state

The following flowchart and the next several sections describe the lifetime of a request received by the Object Pool Manager:



Matching a Pooled Object

A client makes a request of a virtual object. The extension communicates with the Object Pool Manager to bind the virtual object to a matching physical object. The virtual object then repeats the client's request, making this same request of the matched physical object.

If a match is found (if MatchObject( ) returns TRUE), then the Object Pool Manager reserves the matching physical object and sends it back to the requesting extension. If no matching object is found (if MatchObject( ) returns FALSE), then the Object Pool Manager proceeds to the next phase of processing logic—attempting to create a physical object.

Creating a Pooled Object

If no matching object is found, the Object Pool Manager checks its MaxPoolSize variable before trying to create a physical object. MaxPoolSize determines the maximum number of objects a pool can contain. This variable is set in the registry by the server administrator.

If the number of objects in the pool is less than MaxPoolSize, then the pool is allowed to grow. As a result, the Object Pool Manager calls the CreateObject( ) method. A successful call returns a physical object to the requesting client. An unsuccessful call indicates that a critical resource is unavailable. For example, if an application requests a database connection but the database server is down, then the call to CreateObject( ) would fail, and a FAILURE state would be returned immediately.

Replacing a Pooled Object

If an object pool is at the limit set by MaxPoolSize, then the pool is considered full. As a result, instead of trying to create an object, the Object Pool Manager attempts to replace a non-matching object with a physical object that matches the one requested.

Only unreserved objects can be replaced, so the Object Pool Manager first checks whether any objects are free. Assuming there are unreserved objects, the Object Pool Manager next calls the StealObject( ) method to determine which object to replace.

The Object Pool Manager calls the ReleaseObject( ) method to destroy the object to be replaced. As a result, the pool now contains one slot to fill, and the CreateObject( ) method is immediately called.

Queuing a Request

The fourth phase of processing occurs under the following conditions: if no matches are found, if the pool is full, and if none of the objects is free to be replaced. Under these conditions, the Object Pool Manager queues the request. The request waits until an object is returned to the pool. Pooled objects are returned to their originating pool when a client is done using them.

If the waiting period is within a maximum allowable idle time, then the request processing starts over from the beginning, with an attempt to match the newly returned object. The maximum allowable idle time is configurable through the MaxWait variable in the registry.

Returning a FAILURE State

If the request is queued for longer than the MaxWait time, then the request finally fails. Frequent request failures may indicate a need for the server administrator to review an object pool's configuration settings.

Adding Decorations for Object Pooling

Just as you do for other decorations, use Netscape Extension Builder Designer to add decorations for object pooling. The order in which you add decorations does not matter. However, there are different sets of decorations you must specify, depending on what you want to do.

Object pooling decorations can be categorized according to the following tasks:

- Creating Pooled Objects
- Designating Objects That Can Be Pooled
- Changing a Pooled Object's Lifetime
- Passing a Pooled Object

Details of the preceding tasks are based on PoolSampleExt, whose project file is poolsample.gxp.

Creating Pooled Objects

To create pooled objects, specify the following decorations:

- Pooled Object Creation
- Pooled Object Name
- Object Pool Name
- Object Pool Config

Pooled Object Creation

You set the Pooled Object Creation decoration at the method level. A value of Yes indicates that this method creates a pooled object through its Out parameter. For example, in the connService module, the IConnManager interface provides three creation methods. Note the values for each method's Pooled Object Creation decoration:

Method	Pooled Object Creation Value
CreateConnection( )	Yes
CreateTxConnection( )	Yes
CreateNonPooledConnection( )	No

The CreateConnection( ) method is used to create a non-transactional connection, and the CreateTxConnection( ) method is used to create a transactional connection. In both cases, the object created is allocated from a pool. However, the CreateNonPooledConnection( ) method, creates a non-pooled, non-transactional connection.

When Pooled Object Creation is set to its default value of No, a corresponding default set of decorations appears for the method's Out parameter. For example, the decorations on the Out parameter of CreateNonPooledConnection( ) appear as follows:

Parameter Name	Decoration	Value
ppConn	C++ Class	Connection
ppConn	Java Class	Connection
ppConn	Uses Pooled Object	(None)
ppConn	Pooled In Parameter	(None)

By contrast, when Pooled Object Creation is set to Yes, four additional decorations become available to the method's Out parameter. For example, the decorations on the Out parameter of CreateConnection( ) appear as follows:

Parameter Name	Decoration	Value
ppConn	C++ Class	Connection
ppConn	Java Class	Connection
ppConn	Uses Pooled Object	(None)
ppConn	Pooled In Parameter	(None)
ppConn	Pooled Object Name	CONNECTION
ppConn	Object Pool Name	CONN_POOL
ppConn	Object Pool Config	POOLSAMPLE
ppConn	Keep Pooled Object	No

Compare the Out parameter decorations for CreateConnection( ) and CreateTxConnection( ):

Decoration	Value in CreateConnection( )	Value in CreateTxConnection( )
C++ Class	Connection	TxConnection
Java Class	Connection	TxConnection
Pooled Object Name	CONNECTION	CONNECTION
Object Pool Name	CONN_POOL	TXCONN_POOL
Object Pool Config	POOLSAMPLE	POOLSAMPLE
Keep Pooled Object	No	Yes

CreateConnection( ) creates a Connection object allocated from a pool named CONN_POOL, whereas CreateTxConnection( ) creates a TxConnection object allocated from a pool named TXCONN_POOL.

Another difference appears in the Keep Pooled Object decoration. This decoration affects the lifetime of a pooled object and is described in “Changing a Pooled Object’s Lifetime” on page 145.

The next three sections describe Pooled Object Name, Object Pool Name, and Object Pool Config.

Pooled Object Name

The Pooled Object Name decoration is an arbitrary name you assign to the pooled object you want to create. The Object Pool Manager uses this name to track objects as they are passed from place to place in the object pooling request flow.

The example uses a value of CONNECTION as a convenient indication that the object is a connection. However, the value is not indicating the Connection class itself. Note that CONNECTION is also used as the name for TxConnection objects. Even though CONNECTION is created by both methods, the created objects are allocated from different pools. As a result, the Object Pool Manager can distinguish Connection objects and TxConnection objects.

The Pooled Object Name decoration is required if an Object Pool Name is specified.

Object Pool Name

The Object Pool Name decoration determines which pool will contain the objects. Based on the example, the Object Pool Manager will be managing two pools: CONN_POOL will contain connection objects; TXCONN_POOL will contain transactional connection objects.

In general, the number of pools that the Object Pool Manager controls at any one time is determined by the number of unique values for the Object Pool Name decoration, in all extensions running on the same server. If the specified pool does not exist, it is created the first time an object is requested from this pool.

One reason for using different pools is to allow the Object Pool Manager to evaluate matches differently. For example, you might want to have more restrictive matching criteria for TxConnection objects. If so, you would implement the MatchObject( ) method to check for a pool name of

TXCONN_POOL, and use different programming logic depending on the pool name. Similarly, you can implement CreateObject( ) to create a TxConnection object based on the pool name.

Another reason for using different pools is to manage them differently. For example, the Keep Pooled Object decoration is No for objects from the CONN_POOL, but is Yes for objects from TXCONN_POOL.

Object Pool Config

The Object Pool Config decoration is the configuration key in the Netscape Application Server registry. For example, both CONN_POOL and TXCONN_POOL use the same configuration key, POOLSAMPLE. If the value is null, the extension uses Object Pool Name as the default value for the configuration key.

Typically, the server administrator assigns a key name to configure the server for optimal performance under object pooling conditions. Different configurations can be created depending on server resources. Check with the server administrator for the value to enter for the Object Pool Config decoration.

Designating Objects That Can Be Pooled

To designate objects that can be pooled, specify the following decorations:

- Poolable Object
- Introspection Interface

You set both of these decorations on a coclass. For example, the decorations on the Connection coclass appear as shown in the following figure:

Decoration	Value
Manager Class	No
Template Streaming	No
Java Access Layer	On
Create a Lock	No
Poolable Object	Yes
Introspection Interface	IInfo

Poolable Object

If you want clients to be able to share an object, set the Poolable Object decoration to Yes. Just because an object can be pooled does not necessarily mean that the object always will be pooled. For example, CreateNonPooledConnection( ) is not decorated as an pooled-object creation method; therefore, the Connection objects it creates are not pooled.

Introspection Interface

Setting Poolable Object to Yes causes another decoration to appear just below it: Introspection Interface. The value you specify must be an interface you have defined in your extension. Also, the poolable coclass, Connection, must implement this interface.

In PoolSampleExt, the introspection interface is set to IInfo. This means that IInfo will be used to encapsulate data into a virtual object. The Object Pool Manager will use this data from the virtual object and will either find a matching physical object or create one if needed. In PoolSampleExt, the IInfo interface defines methods that will return a user ID, password, and some flags.

Changing a Pooled Object's Lifetime

To change the lifetime of a pooled object, specify either of the following decorations:

- Keep Pooled Object
- Pooled No Reuse

Keep Pooled Object

The Keep Pooled Object decoration determines how long to keep a virtual object bound to a physical object. The default value, No, causes the binding to last only as long as a method call on the pooled object. In other words, successive method calls on the same virtual object are not guaranteed to bind to the same physical object. A value of No will also produce better interleaved use of the pooled object, because it is held for shorter periods.

By contrast, setting Keep Pooled Object to Yes extends the binding to the lifetime of the virtual object, thereby ensuring that successive method calls use the same physical object. This is useful in the case of the CreateTxConnection( ) method, because the resource being requested is a transactional connection, so preserving the session context is important.

You can set Keep Pooled Object in either of two places:

- On the Out parameter of a pooled-object creation method. For example, the Out parameter of CreateTxConnection( ) has its Keep Pooled Object decoration set to Yes.

  A setting of Yes means that when a method call creates a virtual object through its Out parameter, the virtual object is instantly bound to the physical object. The virtual object does not relinquish this physical object to the Object Pool Manager for the lifetime of the virtual object.

- On the method of a pooled object, such as the CreateResultSet( ) method of the Connection coclass.

  A setting of Yes means that when a particular method is called, the binding between the virtual and physical object persists for subsequent calls on the virtual object. The virtual object is ResultSet in this case.

Note that Keep Pooled Object is set to Yes for CreateTxConnection( ) but is set to No for CreateConnection( ). This difference can be seen in the first section of the application's response page:

- For non-pooled connections, each execution will display a fresh set of physical connection IDs.
- For pooled connections from CONN_POOL, the same physical connection ID will be displayed.
- For pooled transactional connections, repeated executions will keep displaying the same set of physical connection IDs.

Pooled No Reuse

Use the Pooled No Reuse decoration on the method of a pooled object. By default, Pooled No Reuse is set to No. This setting means that after a physical object is returned to the Object Pool Manager, the physical object is available for reuse by other virtual objects.

In some cases, a method call on a pooled object changes the state of this object so it becomes unusable. When this happens, the physical object should not be reused by other virtual objects.

To ensure that such physical objects are not reused, set the Pooled No Reuse decoration to Yes on any method that might make a physical object unusable for future method calls. A setting of Yes causes a physical object to be destroyed when it is returned to the Object Pool Manager.

Passing a Pooled Object

When passing a pooled object, specify the following decorations:

- Uses Pooled Object
- Pooled In Parameter

Although these decorations affect the lifetime of a pooled object, they are used particularly when passing pooled objects. For example, suppose a pooled object, Connection, passes itself to a ResultSet object. By default, the ResultSet will try to use the pooled Connection, but the Object Pool Manager may have already given away this physical object to another request.

Instead, to preserve the pooled object for use by ResultSet, it's necessary to bind the reservation lifetime of the pooled object to the lifetime of the ResultSet object. In other words, as long as the ResultSet object exists, the pooled Connection that it uses must not be returned to its pool. You specify the necessary binding by setting Uses Pooled Object and Pooled In Parameter.

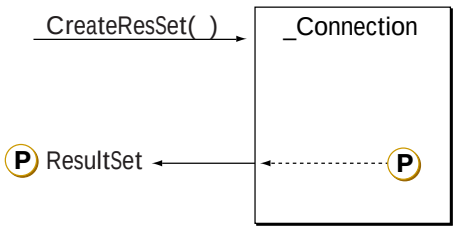
A pooled object may be passed around in any of the following ways:

How Pooled Object Is Passed	Decorations to Set
From the called object to an Out parameter.	Uses Pooled Object (on Out parameter)
From an In parameter to the called object.	Uses Pooled Object, Pooled In Parameter (on a method)
From an In parameter to an Out parameter.	Uses Pooled Object, Pooled In Parameter (on Out parameter)

The next three sections describe these situations in detail. The most common case is the first one, passing from the called object to an Out parameter. But you may find use for any of the three situations, depending on how you design the extension. Note that you specify Pooled In Parameter only when Uses Pooled Object is also specified.

From Called Object to Out Parameter

The following figure represents the case of passing a pooled object from the called object to an Out parameter:



In this case, specify the Uses Pooled Object decoration on the Out parameter.

For example, in IFakeConnection, the CreateResultSet( ) method creates a ResultSet object, which keeps a reference to the IFakeConnection. Since the ResultSet object intends to use the IFakeConnection, the associated physical connection must not be returned to its pool. The Uses Pooled Object decoration prevents the physical connection from being returned to its pool, for the lifetime of the ResultSet object.

The Out parameter settings for CreateResultSet( ) are shown in the following figure:

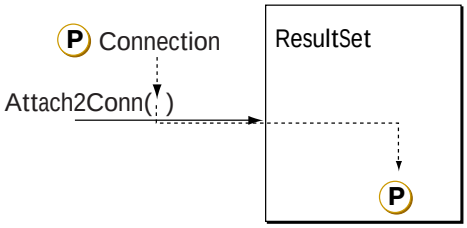
Parameter Name	Decoration	Value
ppSet	C++ Class	ResultSet
ppSet	Java Class	ResultSet
ppSet	Uses Pooled Object	CONNECTION
ppSet	Pooled In Parameter	(None)

Note that you specify CONNECTION as the value for Uses Pooled Object. This corresponds to the arbitrary name you assigned to the IFakeConnection object when you decorated the CreateConnection( ) method. In CreateConnection( ), you previously set the Out parameter's Pooled Object Name to CONNECTION:

Parameter Name	Decoration	Value
ppConn	C++ Class	Connection
ppConn	Java Class	Connection
ppConn	Uses Pooled Object	(None)
ppConn	Pooled In Parameter	(None)
ppConn	Pooled Object Name	CONNECTION
ppConn	Object Pool Name	CONN_POOL
ppConn	Object Pool Config	POOLSAMPLE
ppConn	Keep Pooled Object	No

From In Parameter to Called Object

The following figure represents the case of passing a pooled object from an In parameter to the called object:

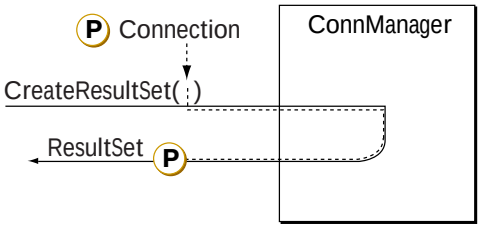


In this case, specify both Pooled In Parameter and Uses Pooled Object on the method.

For example, suppose Attach2Conn( ) is a method that uses a pooled Connection object. If an Attach2Conn( ) method call is made on a ResultSet object, then you want to prevent the pooled Connection object from being returned to its pool, for the lifetime of the ResultSet object. To do so, set the decorations on Attach2Conn( ). In the example, you might set Pooled In Parameter to ppConn, and Uses Pooled Object to CONNECTION.

From In Parameter to Out Parameter

The following figure represents the case of passing a pooled object from an In parameter to an Out parameter:



In this case, specify both Pooled In Parameter and Uses Pooled Object on the method's Out parameter.

For example, suppose a CreateResultSet( ) method call is made on a ConnManager object. In this case, a pooled object is being passed in, some processing occurs, and the pooled object is passed out again. To prevent the pooled object from being returned to its pool, for the lifetime of the Out parameter, set the decorations on the Out parameter of CreateResultSet( ). In the example, you might set Pooled In Parameter to ppConn, and Uses Pooled Object to CONNECTION.

Completing Stubs for Pooled-Object Creation Methods

In PoolSampleExt, the connService service module contains the ConnManager coclass. Therefore, after successful code generation, the ConnManager coclass is represented in source code by the following file:

```
poolsample\cpp\connService\ConnManager.cpp
```

ConnManager implements the IConnManager interface, which in turn has three creation methods. Each method has a corresponding stub to be completed, which is found in ConnManager.cpp. The following code fragment shows the completed stub for the CreateConnection( ) method:

```
// Stub method bodies for interface: IConnManager
HRESULT
ConnManager::CreateConnection(
      /* [in] */ unsigned long uid,
      /* [in] */ LPSTR passwd,
      /* [in] */ unsigned long flags,
      /* [out] */ IFakeConnection **ppConn)
```

```
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   *ppConn = new Connection(this, uid, passwd, flags);
   return hr;
}
```

However, in addition to a method stub for CreateConnection( ), there is also a stub for a “twin method” called CreateConnectionIntrospection( ). The completed stub is shown in the following code fragment:

```
HRESULT
ConnManager::CreateConnectionIntrospection(
      /* [in] */ unsigned long uid,
      /* [in] */ LPSTR passwd,
      /* [in] */ unsigned long flags,
      /* [out] */ IInfo  **ppConn)
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
   *ppConn = new Info(uid, passwd, flags);
   return hr;
}
```

This twin method, known as a create-introspection method, appears because the Pooled Object Creation decoration is set to Yes for the CreateConnection( ) method. The purpose of a create-introspection method is to create an introspection object, which encapsulates the data that the Object Pool Manager uses to match a physical object.

Note how CreateConnectionIntrospection( ) returns an introspection object:

```
/* [out] */ IInfo  **ppConn)
```

This object must implement the introspection interface, which was set to IInfo in Netscape Extension Builder Designer.

The other pooled object creation method, CreateTxConnection( ), has its own introspection method, CreateTxConnectionIntrospection( ). For any method whose Pooled Object Creation decoration is set to Yes, a twin method stub is generated using the following naming scheme:

```
<method>Introspection()
```

Completing Stubs for Object Evaluation Methods

For any class containing pooled-object creation methods, a stub file is generated that contains the object evaluation interface and its methods. The Object Pool Manager makes callbacks to these methods to determine criteria for matching and creating objects.

You must complete the method stubs in this interface, which is named IGXObjectEvaluation in C++ extensions or IJavaObjectEvaluation in Java extensions. For example, in PoolSampleExt, the object evaluation method stubs are found in ConnManager.cpp.

Most or all of the object evaluation methods use the following parameters:

poolName	The name of an object pool. poolName is needed because a class may create pooled objects from more than one pool, and each pool may contain different types of objects. However, the methods in the object evaluation interface can be called for any pool. You may want to use different decision logic depending on which pool a request comes from.
pVirtual	A pointer to a virtual object.
pPhysical	A pointer to a physical object.

Implementing MatchObject( )

The Object Pool Manager makes a callback to MatchObject( ) to ask if a given physical object matches a given virtual object. Given these two objects, you determine whether they are considered a match, and you set the return value accordingly to TRUE or FALSE.

C++ Example

In PoolSampleExt, MatchObject( ) is implemented in ConnManager.cpp as shown in the following code. The matching criteria differ for TXCONN_POOL, which contains transactional connection objects created by CreateTxConnection( ).

```
// Stub method bodies for interface: IGXObjectEvaluation
HRESULT
ConnManager::MatchObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [in] */ IGXObject *pPhysical,
      /* [out] */ BOOL *pMatches)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Depending on poolName, determine if the objects match:
   // Navigate to the introspection interface of the
   // virtual & physical objects, get their attributes,
   // and see if they match.

   // CONN_POOL: match on userid & passwd
   // TXCONN_POOL: match on userid, passwd AND flags
   //       (match criteria is stricter than CONN_POOL)

   if (!strcmp(poolName, "CONN_POOL"))
   {
      *pMatches = FALSE;

      IInfo* pV = NULL;
      IInfo* pP = NULL;

      // Navigate to IInfo interfaces

      if((((hr=pVirtual->QueryInterface(IID_IInfo,
          (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))
```

```
            &&(((hr=pPhysical->QueryInterface(IID_IInfo,
         (LPVOID*)&pP))==GXE_SUCCESS)&&(pP)))
      {
         ULONG PUid = 0;
         ULONG VUid = 0;
         char PPasswd[256];
         char VPasswd[256];

         // Get uid, passwd for both virtual
         // & physical objects

         if(hr==GXE_SUCCESS)
            hr=pV->GetUid(&VUid);
         if(hr==GXE_SUCCESS)
            hr=pV->GetPasswd(VPasswd, 256);
         if(hr==GXE_SUCCESS)
            hr=pP->GetUid(&PUid);
         if(hr==GXE_SUCCESS)
            hr=pP->GetPasswd(PPasswd, 256);

         // Check if userid & password match

         if ((hr==GXE_SUCCESS)
         && (PUid==VUid)
         && (!strcmp(PPasswd, VPasswd)))
            *pMatches = TRUE;
      }

      // Release interfaces
      if (pP)
         pP->Release();
      if (pV)
         pV->Release();
   }

   else // TXCONN_POOL
...
```

Implementing CreateObject( )

The Object Pool Manager makes a callback to CreateObject( ) to ask you to create a physical object that matches the given virtual object.

This callback may occur for two reasons:

- as part of an initial creation attempt, because no match was found.
- as part of an attempt to replace an object, because the pool is full. The maximum pool size is handled by the Object Pool Manager. You cannot use CreateObject( ) to override the maximum pool size.

If CreateObject( ) fails to create a physical object, it means a critical resource is unavailable. As a result, the Object Pool Manager will not queue the associated request but will return a FAILURE status immediately.

Note the Out parameter for CreateObject( ):

```
/* [out] */ IGXPoolObject **ppPhysical)
```

The previous line of code creates an instance of a coclass that is marked as poolable. This coclass implements the IGXPoolObject interface, which is automatically code-generated for you.

C++ Example

In PoolSampleExt, CreateObject( ) is implemented in ConnManager.cpp as shown in the following code. The creation logic differs depending on the pool name.

```
HRESULT
ConnManager::CreateObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [out] */ IGXPoolObject **ppPhysical)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
```

```
// Depending on the poolName, navigate to the
// introspection interface of the virtual object,
// get the encapsulated attributes,
// and create the appropriate physical object with the
// same attributes.

if (!strcmp(poolName, "CONN_POOL"))
{
   IInfo* pV = NULL;
   IFakeConnection* pConn = NULL;
   ULONG uid = 0;
   ULONG flags = 0;
   char passwd[256];

   // Navigate to IInfo interface

   if(((hr=pVirtual->QueryInterface(IID_IInfo,
      (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))
   {
      // Get the uid, passwd, flags

      if(hr==GXE_SUCCESS)
         hr=pV->GetUid(&uid);
      if(hr==GXE_SUCCESS)
         hr=pV->GetPasswd(passwd, 256);
      if(hr==GXE_SUCCESS)
         hr=pV->GetFlags(&flags);
   }

   // Create a matching Connection object.

   if (hr==GXE_SUCCESS)
   {
      hr = CreateConnection(uid, passwd, flags, &pConn);

      // QueryInterface to IGXPoolObject
      if((hr==GXE_SUCCESS)&& pConn)
         hr=pConn->QueryInterface(IID_IGXPoolObject,
            (LPVOID*)ppPhysical);
   }

   // Release interfaces
```

```
      if (pV)
         pV->Release();
      if (pConn)
         pConn->Release();
   }

   else // TXCONN_POOL
...
```

Implementing StealObject( )

The Object Pool Manager makes a callback to this method to ask if a given physical object can be destroyed. Destroying the object makes room in the pool to create another object that matches a given virtual object. Set the return value to TRUE or FALSE depending on the physical object to be replaced and the virtual object to be reserved.

In most cases, you may replace any physical object, so you should return TRUE. In rare cases, the object may be an important resource that you want to prevent from being replaced. Return FALSE in this case. Returning FALSE does not guarantee that the object will never be destroyed. Instead, a value of FALSE effectively raises that object's priority. The Object Pool Manager will pass over this object when searching for objects to destroy.

If you return FALSE, the request will probably be queued until another object is returned to the pool.

C++ Example

In PoolSampleExt, StealObject( ) is implemented in ConnManager.cpp as shown in the following code. The implementation means that any object is equally eligible to be replaced.

```
HRESULT
ConnManager::StealObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [in] */ IGXObject *pPhysical,
      /* [out] */ BOOL *pCanSteal)
{
   HRESULT hr=GXE_SUCCESS;
```

```
    //***
    //*** Provide your implementation here
    //*** and remove the GXASSERT below
    //***

    // Allow any connection to be replaced.

    *pCanSteal = TRUE;

    return hr;
}
```

Implementing ReleaseObject( )

The Object Pool Manager makes a callback to ReleaseObject( ) to ask you to dispose of any resource a physical object might be holding. The physical object will never be used again. This callback may occur if an object needs to be created but the pool is at its maximum size.

However, do not use ReleaseObject( ) to actually destroy the object. The object is automatically destroyed like any other COM object, when its reference count drops to zero.

For pooled objects, use ReleaseObject( ) for time-consuming actions such as closing network connections. If the destructor is used for time-consuming tasks instead of ReleaseObject( ), other requesting threads will be blocked.

It is recommended that you isolate time-consuming actions in a separate method of the physical object. You can use a state flag to ensure that the method executes its logic. A non-pooled instance of the object can then call this logic from the destructor, and a pooled instance can call the logic from ReleaseObject( ).

C++ Example

In PoolSampleExt, ReleaseObject( ) is implemented in ConnManager.cpp as follows. The `reason` parameter does nothing but is reserved for future use.

```
HRESULT
ConnManager::ReleaseObject(
```

```
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pPhysical,
      /* [in] */ unsigned long reason)

{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Would have closed any backend connection here.
   // Nothing to do in this dummy implementation.

   return hr;
}
```

Implementing InitObject( )

The Object Pool Manager makes a callback to InitObject( ) whenever the given physical object is reserved against the given virtual object; that is, after a successful match, creation, or replacement. Use InitObject( ) to make any initializations on the physical object that are specific to the given virtual object.

For example, suppose you have a connection object. For this object to match exactly with a given virtual object, both the userid/password and any connection settings must match. If you assume that allocating a network connection is time consuming, it is more efficient to do so in the object constructor or in the CreateObject( ) method. In that case, you would return TRUE from MatchObject( ) as long as the userid/password match.

By contrast, use InitObject( ) to initialize the settings that take little processing time. This breakup of actions will ensure maximum pooling efficiency.

C++ Example

In PoolSampleExt, InitObject( ) is implemented in ConnManager.cpp as follows.

```
HRESULT
```

```
ConnManager::InitObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [in] */ IGXObject *pPhysical)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

// Remember: the match criteria for objects in
// CONN_POOL did not include flag settings.
// We therefore need to restore the flag settings
// of the physical object, to match those of the
// virtual object.

   // For CONN_POOL objects, do flag settings here.
   if (!strcmp(poolName, "CONN_POOL"))
   {
      // Get to IInfo interface
      IInfo* pV = NULL;

      if(((hr=pVirtual->QueryInterface(IID_IInfo,
          (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))
      {
         ULONG flags = 0;
         hr=pV->GetFlags(&flags);

         // Get Connection implementation
         IFakeConnection* pIConn=NULL;

         if(((hr=pPhysical->QueryInterface
             (IID_IFakeConnection,(LPVOID*)&pIConn))
             ==GXE_SUCCESS)&&(pIConn))
         {
            // Set flags
            pIConn->SetFlags(flags);
         }
      }
   }
```

```
   return hr;
}
```

Implementing UninitObject( )

The Object Pool Manager makes a callback to UninitObject( ) whenever the given physical object is returned to the given pool. Use this method as a complement to InitObject( ). Do not use UninitObject( ) nor the object destructor for time-consuming actions such as closing network connections; use ReleaseObject( ) instead.

C++ Example

In PoolSampleExt, UninitObject( ) is implemented in ConnManager.cpp as follows.

```
HRESULT
ConnManager::UninitObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pPhysical)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Nothing to do.

   return hr;
}
```

Implementing GetHint( )

The Object Pool Manager makes a callback to GetHint( ) to ask for a hint string for the given virtual object. This hint string is used to group objects in the pool, to limit MatchObject( ) searches to this group. Using GetHint( ) may improve performance if the MatchObject( ) method is time-consuming. In most cases, however, return an empty string.

C++ Example

In PoolSampleExt, GetHint( ) is implemented in ConnManager.cpp as shown in the following code:

```
HRESULT
ConnManager::GetHint(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [out] */ LPSTR hint,
      /* [in] */ unsigned long Size)
{
   HRESULT hr=GXE_SUCCESS;
   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Don't want to provide any hint.

   strcpy(hint, "");
   return hr;
}
```

Reference List of Object Pooling Decorations

You add decorations for object pooling in coclasses, methods, and Out parameters.

Coclass Decorations

Add decorations on a coclass if it represents objects that are pooled.

Decoration	Description
Poolable Object	Specifies whether this object can be pooled. Pooled objects are shared between clients so as to minimize the time clients wait to be served. This option is useful for objects that are linked to limited and costly resources, such as data connections.
Introspection Interface	Only available when Poolable Object is set to Yes. This decoration specifies the interface that provides the methods used to query the virtual object. This is an interface you have defined in the extension, and the poolable coclass must implement this interface.

Method Decorations

The following decorations are available in all methods:

Decoration	Description
Uses Pooled Object	Indicates that when this method is called, a pooled object is retrieved from an In parameter. Specify an arbitrary name for the pooled object. Use this decoration to extend the reservation lifetime of pooled objects that are passed to other objects.
Pooled In Parameter	Enter the name of the In parameter that holds the pooled object. The In parameter may be the pooled coclass or any other downstream coclass that has previously grabbed the pooled object. This decoration is required if the Uses Pooled Object decoration is set.
Pooled Object Creation	Defines this method as one that creates a pooled object through its Out parameter. Setting this decoration makes additional decorations available on its Out parameter.

The following method decorations appear only within coclasses that are decorated as poolable objects:

Decoration	Description
Keep Pooled Object	Once this method is called, this decoration specifies whether the virtual object will keep the real object reserved (not allowing other matches) for the life of the virtual object. Establishes a persistent binding between the virtual object and the real object.
Pooled No Reuse	Once this method is called, this decoration specifies the reserved object will be destroyed when the virtual object returns it to the object pool. This is useful if a method is known to have side-effects on a pooled resource.

Out Parameter Decorations

The following Out parameter decorations can be set in any method:

Decoration	Description
Uses Pooled Object	Indicates that when this method is called, this Out parameter retrieves a pooled object. If this Out parameter is also decorated with a Pooled In Parameter, then this Out parameter retrieves the pooled object from the specified In parameter. If this Out parameter is not decorated with a Pooled In Parameter, then this Out parameter retrieves the pooled object from the called object. Use this decoration to extend the reservation lifetime of pooled objects that are passed to other objects.
Pooled In Parameter	Enter the name of the In parameter that holds the pooled object. The In parameter may be the pooled coclass or any other downstream coclass that has previously grabbed the pooled object. This decoration is optional and supports the Uses Pooled Object decoration.

The following Out parameter decorations are available only in methods that are decorated as pooled object creation methods:

Decoration	Description
Pooled Object Name	Name of the pooled object that is being created. This decoration is required if the Object Pool Name is specified.
Object Pool Name	Name of the pool that will contain objects being pooled. This decoration is required if the Pooled Object Name is specified.
Object Pool Config	Configuration key for this pool in the Netscape Application Server registry. If null, the extension uses Object Pool Name as the default value for the configuration key.
Keep Pooled Object	If set to No, the virtual object binds to a physical object only for the duration of a method call on the virtual object. If set to Yes, the virtual object created binds itself immediately to a physical object and reserves the physical object for the life of the virtual object. A setting of Yes is useful when you want to preserve session context.

Chapter

10

Compiling the Extension Source Code

This chapter describes compiling, the repeated process that you perform after generating output files with Netscape Extension Builder. Extension code is compiled by running the make command.

The following topics are included in this chapter:

- About Compiling the Source Code
- Introduction to Editing Makefiles
- Editing Makefiles on Solaris
- Editing Makefiles On Windows NT
- The make Harness

About Compiling the Source Code

When you run make the first time, the initial compilation produces a "boilerplate" extension—a valid extension containing no useful functionality. After you add functionality—by filling out method stubs and by completing other code—you must run make a second time.

As long as your extension uses only the source files automatically generated by Netscape Extension Builder, you can recompile simply by running make. However, in many cases your extension needs to encapsulate legacy code, and this requires you to link libraries or create source code by hand. If your extension uses components that were not automatically generated, then these components will not appear in the makefiles. You must add this information to the makefiles before you can successfully compile the extension code.

Introduction to Editing Makefiles

You may need to edit one or more files in the make harness under any of the following conditions:

- If your extension includes additional source code files that were not automatically generated. For example, to support a method stub that you are completing, you might create extra code but place it in a separate file.
- If your extension must link with additional libraries that were not automatically generated.
- If you want to compile or link your extension using additional command-line flags. For example, in the testing phase, you might use the -DVERBOSE flag to generate diagnostic messages while your extension is running.

When you modify the make harness, edit only the makefile in each affected directory. For example, suppose you created new .cpp files by hand in a service module directory named EmployeeServices. If so, you would edit that directory's makefile and re-run the make command from that directory.

If instead, you added source files to many directories, you would still edit each directory's makefile, but you might find it easier to re-run the make command just once, from the top-level directory.

The specific changes you add to a makefile depend on your computing platform (Solaris or Windows NT) and on the extension language (C++ or Java).

Editing Makefiles on Solaris

This section describes the following tasks:

- Editing Makefiles for C++ Extensions
- Editing Makefiles for Java Extensions

Editing Makefiles for C++ Extensions

If you are building a C++ extension, you may need to edit the GNUmakefile.sun file in any of the following ways:

- Adding New Source File Names
- Linking Additional Libraries
- Adding Flags
- Adding Support for Non-Pic Objects

Adding New Source File Names

If you created new extension source files:

1. Locate the section where SERVICE_SRCS is defined, and append the new file names. For example:

```
SERVICE_SRCS = \
   CExtModule.cpp \
   CExtFirstClass.cpp \
   CExtSecondClass.cpp \
   CMyNewClass.cpp
```

2. Locate the section where SERVICE_OBJS is defined, and append the name of the corresponding object files. For example:

```
SERVICE_OBJS = \
   $(INTERMEDIATE_OUTDIR_NAME)/CExtModule.o \
   $(INTERMEDIATE_OUTDIR_NAME)/CExtFirstClass.o \
   $(INTERMEDIATE_OUTDIR_NAME)/CExtSecondClass.o \
   $(INTERMEDIATE_OUTDIR_NAME)/CMyNewClass.o
```

Linking Additional Libraries

1. Locate the section where LIBS_KIVA is defined.
2. Append your library names to the LIBS_KIVA definition. For example:

```
LIBS_KIVA += -lMyExtension
```

If you extend the definition of LIBS_KIVA, check that the listed libraries reside in <NAS_ROOTDIR>/APPS/bin; otherwise, the Deployment Manager will not be able to deploy them.

Adding Flags

1. For C++ compilation flags, edit the CFLAGS definition. For example, to add the DEBUG flag:

```
CFLAGS += -DDEBUG
```

2. For link flags, edit the LFLAGS_SERVICE definition. For example:

```
LFLAGS_SERVICE += -L anotherLibDirectory
```

Adding Support for Non-Pic Objects

By default, the linker supports position-independent code (PIC) objects. On Solaris, you must edit makefiles if a C++ extension uses non-PIC objects. The edits you make will differ, depending on whether you can recompile the non-PIC objects.

If you can recompile the objects, then use the -KPIC option in the compile command. Using this option produces objects with position-independent code.

If you cannot recompile the objects, then override the LFLAGS_ZTEXT variable in the makefile. For example:

```
include $(GX_KETROOTDIR)/makefiles/GNUdefaults_sun.mak

LIB_TARGETS += $(LIB_INSTALL_DIR)/lib$(SERVICE_TARGET).so
LIBS_KIVA += FileWriterNOPIC.a -lketgxidl -lketutil \
   -lSampleExt_ext
INCLUDE_PATH += -I$(KIVA_EXT_INCLUDE) -I$(INCLUDE_INSTALL_DIR)

# override the default LFLAGS_ZTEXT value
```

```
LFLAGS_ZTEXT =
```

The default value of LFLAGS_ZTEXT is “-z text”. This value causes a fatal error whenever text relocations need to occur, and text relocations always occur when linking with a non-PIC object. By overriding the default value of LFLAGS_ZTEXT to nothing, the linker does not use the “-z text” flag. As a result, non-PIC objects can be successfully linked.

Editing Makefiles for Java Extensions

If you are building a Java extension, you edit the makefile only if you add new source files. No edits are needed to account for libraries, and no flags are used when compiling or linking Java. Any Java class files you want to import in your extension must reside in <NAS_ROOTDIR>/APPS/bin; otherwise, the Deployment Manager will not be able to deploy them.

If you created new Java source files, edit GNUmakefile.sun as follows.

1. Locate the section where JAVA_SRCS is defined, and append the new file names. For example:

```
JAVA_SRCS = \
   CExtModule.java \
   CExtFirstClass.java \
   CExtSecondClass.java \
   CMyNewClass.java
```

2. Locate the section where JAVA_CLASSES is defined, and append the name of the corresponding Java class files. For example:

```
JAVA_CLASSES = \
   $(JAVA_INSTALL_DIR)/$(JAVA_PACKAGEDIR)/CExtModule.class \
   $(JAVA_INSTALL_DIR)/$(JAVA_PACKAGEDIR)/ \
       CExtFirstClass.class \
   $(JAVA_INSTALL_DIR)/$(JAVA_PACKAGEDIR)/ \
       CExtSecondClass.class \
   $(JAVA_INSTALL_DIR)/$(JAVA_PACKAGEDIR)/CMyNewClass.class
```

Editing Makefiles On Windows NT

This section describes the following tasks:

- Editing Makefiles for C++ Extensions
- Editing Makefiles for Java Extensions
- Configuring the make Harness

Editing Makefiles for C++ Extensions

If you are building a C++ extension, you may need to edit the GNUmakefile.nt file in any of the following ways:

- Adding New Source File Names
- Linking Additional Libraries
- Adding Flags

Adding New Source File Names

If you created new extension source files:

1. Locate the section where SERVICE_SRCS is defined, and append the new file names. For example:

```
SERVICE_SRCS = \
   CExtModule.cpp \
   CExtFirstClass.cpp \
   CExtSecondClass.cpp \
   CMyNewClass.cpp
```

2. Locate the section where SERVICE_OBJS is defined, and append the name of the corresponding object files. For example:

```
SERVICE_OBJS = \
   $(INTERMEDIATE_OUTDIR_NAME)\CExtModule.obj \
   $(INTERMEDIATE_OUTDIR_NAME)\CExtFirstClass.obj \
   $(INTERMEDIATE_OUTDIR_NAME)\CExtSecondClass.obj \
   $(INTERMEDIATE_OUTDIR_NAME)\CMyNewClass.obj
```

3. Locate the section where INTERMEDIATE_OUTDIR_NAME is defined, and add similar entries for each of the new files. For example:

```
$(INTERMEDIATE_OUTDIR_NAME)\CMyNewClass.obj: CMyNewClass.cpp
   $(CPP) $(CFLAGS) -Fo"$@" CMyNewClass.cpp
```

Linking Additional Libraries

1. Locate the section where LIBS_KIVA is defined.
2. Append your library names to the LIBS_KIVA definition. For example:

```
LIBS_KIVA = $(LIBS_KIVA) ketgxidl.lib ketutil.lib MyExt.lib
```

3. If the new libraries reside in directories other than those defined by LIB_PATH, append the directory paths to LIB_PATH. For example:

```
LIB_PATH = $(LIB_PATH); C:\my_legacy_code\lib
```

If any of the files you appended to LIBS_KIVA are .dll files, check that they reside in <NAS_ROOTDIR>\APPS\bin; otherwise, the Deployment Manager will not be able to deploy them. It is recommended, though not required, that .lib files reside in <NAS_ROOTDIR>\APPS\bin as well.

Adding Flags

1. For C++ compilation flags, edit the CFLAGS definition. For example, to add the DEBUG flag:

```
CFLAGS = $(CFLAGS) -DDEBUG
```

2. For link flags, edit the LFLAGS_SERVICE definition. For example:

```
LFLAGS_SERVICE = $(LFLAGS_SERVICE) -nologo
```

Editing Makefiles for Java Extensions

If you are building a Java extension, you edit the makefile only if you add new source files. No edits are needed to account for libraries, and no flags are used when compiling or linking Java. Any Java class files you want to import in your extension must reside in <NAS_ROOTDIR>\APPS\bin; otherwise, the Deployment Manager will not be able to deploy them.

If you created new Java source files, edit GNUmakefile.nt as follows.

1. Locate the section where JAVA_SRCS is defined, and append the new file names. For example:

```
JAVA_SRCS = \
   CExtModule.java \
   CExtFirstClass.java \
   CExtSecondClass.java \
   CMyNewClass.java
```

2. Locate the section where JAVA_CLASSES is defined, and append the name of the corresponding Java class files. For example:

```
JAVA_CLASSES = \
   $(JAVA_INSTALL_DIR)\$(JAVA_PACKAGEDIR)\CExtModule.class \
   $(JAVA_INSTALL_DIR)\$(JAVA_PACKAGEDIR)\ \
         CExtFirstClass.class \
   $(JAVA_INSTALL_DIR)\$(JAVA_PACKAGEDIR)\ \
         CExtSecondClass.class \
   $(JAVA_INSTALL_DIR)\$(JAVA_PACKAGEDIR)\CMyNewClass.class
```

3. Locate the section where JAVA_INSTALL_DIR is defined, and add similar entries for each of the new files. For example:

```
$(JAVA_INSTALL_DIR)\$(JAVA_PACKAGEDIR)\CMyNewClass.class :
          CMyNewClass.java
   $(JAVAC) $(JAVAC_FLAGS) CMyNewClass.java
```

Configuring the make Harness

You may need to configure Netscape Extension Builder so that the make harness can locate your C++ development environment. You do this by setting several variables in the env.mak file. This file resides in the <NEB_ROOTDIR>\makefiles directory.

For example, to set your C++ development environment, you might define variables as shown in this sample env.mak file:

```
# GNUenv_nt.mak
# Contains machine-specific directories.
```

```
# Where the MSDEV environment is.
# Make sure you keep the double quotes if the
# paths you specify here include spaces.
# For example, "c:\Program Files\DevStudio"
#
WIN_SDK = "c:\msdev"
CPP = "c:\msdev\bin\cl.exe"
LINK = "c:\msdev\bin\link.exe"
WIN_SDK_INCLUDE = "c:\msdev\include"
WIN_SDK_LIB = "c:\msdev\lib"
```

The installer attempts to set these variables to their correct values. Check the env.mak file and edit these values as needed.

The make Harness

After Netscape Extension Builder Designer generates IDL files and a make harness, you run the gmake command from the top-level directory. Running this command performs the following main tasks:

- Invokes the KIDL Compiler to translate IDL files into Java or C++ source files.
- Copies source files into appropriate locations in the code tree.
- Compiles source files.

In particular, the top-level makefile recursively calls the makefiles in the idl subdirectory and in the source code subdirectories (cpp, java, or both).

Inside the IDL Directory

The gmake command in the idl directory performs the following tasks:

- Runs the KIDL Compiler on <extension>.idl to produce public interfaces.
- Copies the public interfaces to the appropriate location, depending on whether they are Java or C++.

- For a C++ extension, compiles this source code and puts <extension>_ext.lib in

Windows `<NAS_ROOTDIR>\APPS\extensions\lib`

Solaris `<NAS_ROOTDIR>/APPS/bin`

- Traverses into each service module subdirectory and runs make, which in turn performs the following tasks:
  - Runs the KIDL Compiler on the service IDL file, producing stub files, runtime feature files, Java Access Layer files (if specified in IDL), and accessor files.
  - Copies these source files into appropriate locations in the code tree.

Inside the cpp Directory

The cpp directory is created if you are building a C++ extension. The gmake command in the cpp directory performs the following tasks:

- Checks for the existence of a makefile in each <service_module> subdirectory. If no makefile exists, it is copied from gen.<service_module>.
- Traverses into the accessor subdirectory and runs make, which in turn compiles the accessor files.
- Traverses into each _<service_module> subdirectory and runs make, which in turn compiles the code for Netscape Extension Builder runtime features.
- Traverses into each <service_module> subdirectory and runs make, which in turn performs the following tasks:
  - Checks for the existence of the stub files. If none are present, they are copied from the gen.<service_module> directory.
  - Compiles the stub files. For the initial compile, the stub files are empty. You must complete the method stubs in these files and then recompile.

Inside the Java Directory

The java directory is created if you are building a Java extension or a C++ extension with a Java Access Layer. The gmake command in the java directory traverses down multiple directories, recursively running make until it reaches the <package> directory. For example, if the Java package is named com.mycompany.Hello, then the make command is run from the <extension>\com\mycompany\Hello directory.

The gmake command in the <package> directory performs the following tasks:

- Compiles public Java interfaces and Java accessors.
- For Java extensions, checks for the existence of a makefile in each <service_module> subdirectory. If no makefile exists, it is copied from gen.<service_module>.
- For Java extensions, traverses into each _<service_module> subdirectory and runs make, which in turn compiles the code for Netscape Extension Builder runtime features.
- Traverses into each <service_module> subdirectory and runs make. If the extension is written in Java, running make in this directory performs the following tasks:
    - Checks for the existence of the stub files. If none are present, they are copied from the gen.<service_module> directory.
    - Compiles the stub files. For the initial compile, the stub files are empty. You must complete the method stubs in these files and then recompile.

    If the extension is written in C++ and has a Java Access Layer, then running make from the <service_module> directory performs the following tasks:

    - Compiles the Java classes with native signatures.
    - Runs `javah` and `javah -stubs` to produce the Java native interface stubs.
    - Compiles the Java native interface stubs and the KIDL-generated native wrapper code.

Chapter

11

Deploying and Managing a Netscape Extension

This chapter describes the process of deploying a Netscape extension, including deploying the Netscape Extension Builder Runtime Layer, the extension classes and libraries, and the Netscape Application Server application that will use the extension.

The following topics are included in this chapter:

- Deploying a Netscape Extension
- Verifying Extension Configuration
- Disabling an Extension
- Adjusting Object Pooling Configurations

Deploying a Netscape Extension

Before you can deploy an extension, you must first register the server(s) to which you want to deploy. During this process you must provide a username and password that is authorized by that server to perform both Administration and Deployment. In preparation for deployment, you must first setup this username using the NAS Administrator.

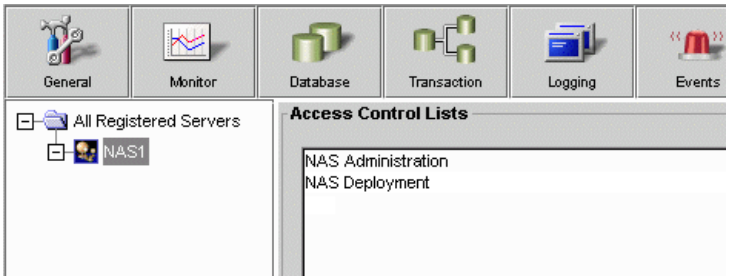
To setup a username in NAS Administration and NAS Deployment to use when deploying an extension, perform the following steps:

1. From the Start menu choose Programs and then Netscape Application Server 4.0.

   The NAS Administrator opens.

2. In the left navigation pane, highlight the server to which you want to deploy and click the Security button.

   The following window opens:



3. Highlight NAS Administration and click Modify near the bottom of the window.

   The Modify Access Control List dialog box appears.

4. Click Add User or Group.

5. In the text box highlight the username “admin21” and click OK.

6. In the Modify Access Control List dialog, click the checkbox for user admin21.

   This grants this user ADMIN privileges for NAS Administration.

7. Click OK.

8. Repeat steps 3-6 to grant user admin21 ADMIN privileges for NAS Deployment.

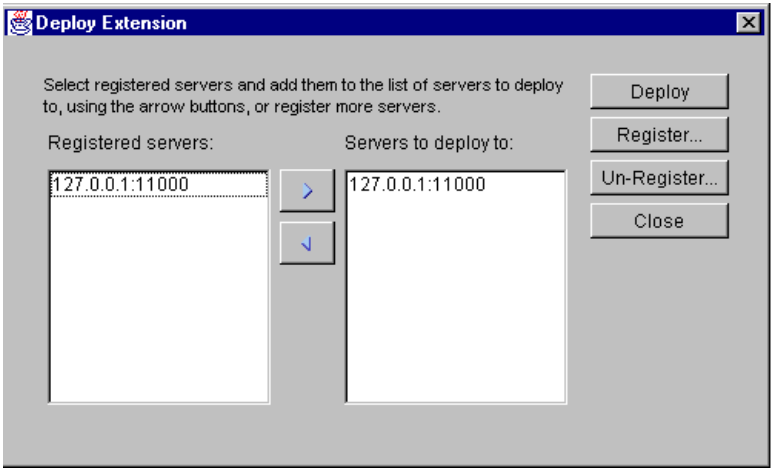
   You can now deploy an extension to this server.

You deploy an extension from the project file (.gxp) that corresponds to the extension project. To generate the .gxp file, you must first compile the extension source code. For more information about compiling the extension, see Chapter 10, “Compiling the Extension Source Code.”

To deploy a Netscape extension, perform the following steps:

1. Open the extension .gxp file in Netscape Extension Builder Designer.
2. Choose Tools - Deploy or click the Deploy Extension tool-bar icon.

The Deploy Extension dialog box appears:



3. In the Deploy Extension dialog box, select the server or servers that you will deploy the extension to, then click the right arrow button.

   The selected servers appear in the “Servers to deploy to:” box.

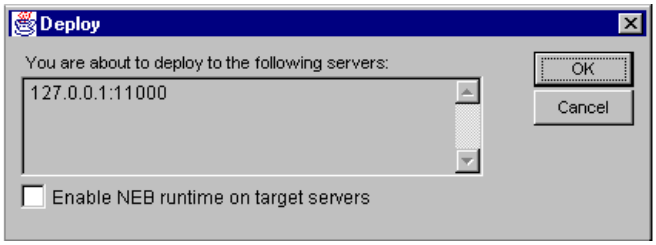
4. If you need to make more Netscape Application Servers available, click Register and specify the information necessary to connect and add those servers.

   The Register Server dialog appears, prompting you for the name of the server, a username, and a password.

5. Enter the name of the server, the username “admin21” (without the quotes) and the admin password used to logon to the server.

6. In the Deploy Extension dialog box, click Deploy.

   The Deploy dialog box appears:



7. If you are deploying this extension to a Netscape Application Server that does not already host an extension, you must first enable the Netscape Extension Builder Runtime Layer by selecting the “Enable NEB runtime on target servers” checkbox.

   The Runtime Layer consists of the files associated with the Netscape Extension Builder Runtime Features. These files need be enabled only once for each Netscape Application Server.

8. Click OK.

   A log dialog box appears, displaying the deployment process of the extension. Read the log to verify proper deployment of the extension.

9. When finished reading the log, click Close.

   Deployment of the extension is complete.

Verifying Extension Configuration

Once you have completed the deployment process, you can verify that the extension is properly configured using Netscape Application Server Administrator. This is particularly useful for servers that have multiple extensions installed on them and for extensions that use many services.

Netscape Application Server Administrator displays each extension hosted on the server as well as the service or services that compose each extension, and the engines in which those services are run. This allows you to verify that a C++ service is properly hosted in the C++ Server, a Java service is hosted in the Java Server, and a C++ service with a Java access layer is hosted in the C++ Server and the Java Server.

As the extension developer and deployer, you must coordinate with the server administrator to verify extension configuration using Netscape Application Server Administrator.

To verify extension configuration

1. Open Netscape Application Server Administrator.
2. Select the server to which you deployed.
3. Click View Extensions.

   The View Extensions dialog box appears.
4. Under Extension Name, find the extension you deployed.
5. Under Extension In, verify that each service of the extension is hosted in the appropriate Netscape Application Server process, as described by the following table:

Extension In	Description
C++ Server	Extension services written in C++ must be hosted by the C++ Server.
Java Server	Extension services written in Java must be hosted by the Java Server.
C++ Server/Java Server	Extension services written in C++ with a Java access layer are hosted in the C++ Server and Java Server.

6. When you are finished, click OK.

Disabling an Extension

You might have to disable an extension for a variety of reasons. Using the Netscape Application Server registry, you can disable an extension whenever necessary.

To disable an extension

1. Open the Netscape Application Server registry editor by typing `kregedit` at the command or shell prompt.

   The Netscape Application Server registry editor opens and displays the keys and values that apploy to the Netscape Application Server.

   For Windows NT, use regedit and start at `HKEY_LOCAL_MACHINE\\SOFTWARE\\KIVA`.

2. Open the following key:

```
KIVA\\Enterprise\\4.0\\CCSO\\EXTENSIONS\\<ExtensionName>\\ServiceName
```

3. Double-click the enabled DWORD value.

   The DWORD editor dialog box appears.

4. Change the value data to 0 and click OK.

Adjusting Object Pooling Configurations

If your extension is pooling objects, there are performance-optimizing configurations the server administrator can make. Currently, these settings must be made in the Netscape Application Server registry for each machine on which Netscape Application Server is running an extension.

To adjust object pooling configurations

1. Open the Netscape Application Server registry editor by typing `kregedit` at the command or shell prompt.

   The Netscape Application Server registry editor opens and displays the keys and values that apply to the Netscape Application Server.

For Windows NT, use `regedit` and start at HKEY_LOCAL_MACHINE\\SOFTWARE\\KIVA.

2. Open the following key:

   KIVA\\Enterprise\\2.1\\CCSO\\POOLS

3. Adjust the pooling configurations as described in the following table:

Object Pooling Values	Description
MaxPoolSize	The maximum number of objects allowed in the pool. For example, if the pooled objects are CICS connections, set this number to the peak number of connections available to the server. If the number of objects is less than MaxPoolSize but exceeds SteadyPoolSize, objects are destroyed immediately after they are returned to the pool.
SteadyPoolSize	The number of unused objects that are kept in the pool until they time out. For example, if the pooled objects are CICS connections, set SteadyPoolSize to the steady state (RMS) number of connections available from the CICS server.
MaxWait	The maximum time, in seconds, a request for a physical object is held in the queue before the request times out and is destroyed.
UnusedMaxLife	The maximum time, in seconds, that a physical object remains unused in the pool. After this time, the physical object is destroyed.
MonitorInterval	(Optional) The time, in seconds, that a thread is spawned to monitor the current status of the pool. Default is 30 seconds. Typically, setting this number too low degrades performance, but it should be set to a number less than UnusedMaxLife.
DebugLevel	(Optional) Determines type of message logging, as described by the following choices: 0: Logging turned off. 1: Logs only callback messages. 2: Logs all messages.

These values can also be set at the engine level (`CCS0\\ENG\\<eng #>`). Settings made for an engine override the settings made in the POOLS key only for that engine. This allows you to fine tune, on an engine by engine bases, extensions running on multiple engines.

4. When finished, close the Netscape Application Server registry editor.

Chapter

12

Example Extension: HelloWorld

This section of *Developer's Guide* explains the steps that were performed to create the sample extension HelloWorld and application.

The following topics are included in this chapter:

- About the Example
- Design the Extension
- Complete the Extension
- Build the Application
- Test the Application

About the Example

Netscape Extension Builder includes a sample directory that contains a C++ extension named HelloWorld and a Java servlet that uses the extension. The steps to follow can be grouped into the following project phases:

- Design the Extension
- Complete the Extension

- Build the Application
- Test the Application

Design the Extension

Designing the sample extension consists of four main tasks:

1. Open Netscape Extension Builder Designer.
2. Create an access module.
3. Create a service module.
4. Generate IDL files.

Open Netscape Extension Builder Designer

1. On Windows NT, double-click the Netscape Extension Builder icon or run the neb command. On Solaris, run the neb command. The neb command executable is in the bin directory of your Netscape Extension Builder installation directory.

   The Netscape Extension Builder Designer start-up window appears.
2. Name the extension HelloWorld.

Create an Access Module

1. In the tree view, highlight the iSample access module.
2. Name the module ihello.
3. In the access module, define an interface named IHelloWorld.
4. In the interface, define a method named helloThere.

5. In the method, define a parameter named pHelloMessage. Define it as an Out parameter of type String.

Create a Service Module

1. In the tree view, highlight the service module.
2. Name the module chello. Leave the values for service module decorations as they are.
3. In the service module, define a coclass named CHelloWorld.
4. Change the value of the Manager Class decoration to Yes. This marks CHelloWorld as the manager class for the chello service.
5. Leave the other values for CHelloWorld decorations as they are.
6. In the CHelloWorld coclass, implement the IHelloWorld interface.
7. In the IHelloWorld interface, decorate the parameter of the helloThere method. Set the value of the pHelloMessage parameter to 256. This sets the maximum buffer size for the out string parameter.

Generate IDL Files

Select the Generate IDL command to produce IDL files and makefiles.

Complete the Extension

Note Stop the Netscape Application Server before compiling the extension and then re-start it before testing the application. Extensions are loaded as you restart Netscape Application Server.

1. Generate source code by running the gmake command from the top-level directory of the extension.

Windows

```
gmake -f GNUmakefile.nt
```

Solaris `gmake -f GNUmakefile.sun`

The gmake command places stub files in the cpp\gen.chello\ directory. The gmake command also copies stub files to the cpp\chello\ directory if it is empty.

1. Go to the cpp\chello\ directory and locate the stub file named CHelloWorld.cpp. This file corresponds to the class you created.
2. Edit CHelloWorld.cpp by completing the stub for the helloThere method.

   Note: in the sample extension provided, the cpp\chello\CHelloWorld.cpp file already contains the completed stub. To view the stub file before it is completed, compare the sample code in cpp\gen.chello\CHelloWorld.cpp.
3. Rerun the gmake utility from the cpp\chello directory. In general, you only need to rerun gmake from directories in which you've edited files.
4. Register the extension:

   ```
   kreg < Hello.gxr
   ```

   Registering an extension makes it available to C++ and Java applications.

Build the Application

1. Locate the callHello.java application in the sample\hello\servlet directory.
2. Edit the application to use the accessors for the extension. When you are done, the application should look like the following:

```
package HelloTest;

import java.lang.*;
import java.util.*;
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;

import com.netscape.server.servlet.extension.*;
import com.kivasoft.applogic.*;
```

```
import helloExtension.*;    // extension classes

public class callHello extends HttpServlet {


/**
 * Call the hello extension
 */


    public void service(HttpServletRequest req, HttpServletResponse
res)
      throws ServletException, IOException
    {
       // access servlet's underlying AppLogic instance
    HttpServletRequest2 req2;
    req2 = (HttpServletRequest2) req;  // legal cast within NAS
    AppLogic al = req2.getAppLogic();
    com.kivasoft.IContext alContext;
    alContext = al.context;

       // Attempt to get to the helloExtension through the accessor
IHelloWorld hello=access_chello.getchello(alContext, null, al);

      // Let's build the result string
         String result;

      // Go to the extension to get the greeting
      if (hello != null) {
          result = hello.helloThere();
      }
      else {
              result = "ERROR! Can't get access to the Hello
extension.";
}
   req.setAttribute("strHelloMsg", result);

   res.setContentType("text/html");

   RequestDispatcher dispatcher;

   dispatcher = getServletContext().getRequestDispatcher("HelloTest/
   hello.jsp");
```

```
    dispatcher.include(req, res);
    return;
   }
  }
```

3. After you finish editing the application code, register the application:

```
kreg < appinfo.gxr
```

Note You can also register the application by running `servletreg` on `appInfo.ntv`.

4. Copy `hello.jsp` into the `<NAS_ROOTDIR>\APPS\helloTest` directory.

5. Copy `appInfo.ntv` and `servInfo.ntv` into the `<NAS_ROOTDIR>\APPS\helloTest\ntv` directory.

Test the Application

Before testing the application, verify that Netscape Enterprise Server (Web server) and Netscape Application Server are running. To test the application:

1. From your web browser, load a test page that uses the application. Point your browser to http://<my_host_name>/extensions, and select the Hello example.

2. Test the application from the web page.

   You have completed all tasks for the code walkthrough.

Appendix

A

C++ Helper Functions

This appendix contains information about C++ helper functions.

The following helper functions are described in this appendix:

GXContextCreateDataConn( )
GXContextCreateDataConnSet( )
GXContextCreateHierQuery( )
GXContextCreateMailbox( )
GXContextCreateQuery( )
GXContextCreateTrans( )
GXContextDestroySession( )
GXContextGetAppEvent( )
GXContextGetObject( )
GXGetStateTreeRoot( )
GXContextIsAuthorized( )
GXContextLoadHierQuery( )
GXContextLoadQuery( )
GXContextLog( )
GXContextNewRequest( )
GXContextNewRequestAsync( )

GXContextCreateDataConn( )

Creates a new data connection object and opens a connection to a database or data source.

Syntax

```
HRESULT GXContextCreateDataConn(
  IGXContext *pContext,
  DWORD flags,
  DWORD driver,
  IGXValList *props,
```

```
IGXDataConn **ppConn);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

flags. One or more optional flags used for connecting to the specified data source.

- To try to use a cached connection, if one is available, specify GX_DA_CACHED. If no cached connections are currently available, a new one is created.
- To always create a new connection (instead of using a cached connection), specify GX_DA_NEW.
- To retry if a connection is not available, specify GX_DA_CONN_BLOCK.
- To return a failure immediately after the first attempt if a connection is not available, specify GX_DA_CONN_NOBLOCK.

The caller can pass one parameter from both mutually exclusive pairs, as shown in the following example:

(GX_DA_CACHED | GX_DA_CONN_BLOCK)

Specify 0 (zero) to use the system's default settings: GX_DA_CACHED and GX_DA_CONN_BLOCK

driver. Specify one of the following:

GX_DA_DRIVER_ODBC	GX_DA_DRIVER_SYBASE_CTLIB
GX_DA_DRIVER_MICROSOFT_JET	GX_DA_DRIVER_MICROSOFT_SQL
GX_DA_DRIVER_INFORMIX_SQLNET	GX_DA_DRIVER_INFORMIX_CLI
GX_DA_DRIVER_INFORMIX_CORBA	GX_DA_DRIVER_DB2_CLI
GX_DA_DRIVER_ORACLE_OCI	GX_DA_DRIVER_DEFAULT

If GX_DA_DRIVER_DEFAULT is specified, the Netscape Application Server evaluates the drivers and their associated priorities set in the registry to determine the driver to use. Specify GX_DA_DRIVER_DEFAULT if your system uses ODBC and native drivers, and if you want the Netscape Application Server to choose between an ODBC driver and a native driver at connection time.

props. IGXValList of connection-specific information required to log in to the data source. Use the following keys for the connection parameters:

- `"DSN"` for the data source name.
- `"DB"` for the database name.
- `"USER"` for the user name.
- `"PSWD"` for the password.

ppConn. A pointer to the created IGXDataConn object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage A data connection is a communication link or session with a database or other data source. Before interacting with a data source, an extension must first establish a connection with it. Each connection is represented by a data connection object, which contains all the information needed to communicate with a database or data source, such as the name of the database, database driver, user name, password, and so on. A data connection object is an instance of the IGXDataConn interface.

Use GXContextCreateDataConn( ) to set up a separate connection for each database or data source you want to access. Extensions refer to the data connection object in their methods that perform subsequent operations on the database.

Rules

- Call GXContextCreateDataConn( ) before running any other database operations requiring a data connection object.
- Your network and the database server must be correctly configured and running so that the extension on your application server can log into the database management system with which it will communicate.
- The data source name, database name, user name, and password must be valid for the database management system to which you want to connect.
- The extension must log in with sufficient access rights to perform all operations it attempts on the data source.

Tips

- Before logging in to the database, the extension should check the user's security level to verify sufficient access rights to perform intended operations on the database.

- The Data Access Engine (DAE) manages database connections and related housekeeping tasks, such as shutdown and cleanup. While the DAE performs these tasks automatically and intermittently, an extension can also explicitly close data connections using CloseConn( ) in the IGXDataConn interface.
- Before using an ODBC connection, you must use the ODBC administration utility supplied with your database software to define and name a data source. For more information about how to do this, refer to your ODBC documentation.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example

```
// Method to open a connection to a database
STDMETHODIMP
CEmployeeMgr::GetOBDataConn(IGXDataConn **ppConn)
{
   HRESULT hr=GXE_SUCCESS;

   // Create a vallist for the connection parameters
   IGXValList *pList=GXCreateValList();
   if(pList) {
      // Set up the connection parameters
      GXSetValListString(pList, "DSN", OB_DSN);
      GXSetValListString(pList, "DB", "");
      GXSetValListString(pList, "USER", OB_USER);
      GXSetValListString(pList, "PSWD", OB_PASSWORD);

      // Attempt to create the connection
      hr = GXContextCreateDataConn(m_pContext, 0, GX_DA_DRIVER_DEFAULT,
       pList, ppConn);

      // Release pList when it's no longer needed
      pList->Release();
   }
   return hr;
}
```

GXContextCreateDataConnSet( )

Creates a collection used to dynamically assign query name/data connection pairs before loading a query file.

Syntax

```
HRESULT GXContextCreateDataConnSet(
   IGXContext *pContext,
   DWORD flags,
   IGXDataConnSet **ppDataConnSet);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

flags. Specify 0. Internal use only.

ppDataConnSet. Pointer to the created IGXDataConnSet object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage

Use GXContextCreateDataConnSet( ) only if you are loading a query file using GXContextLoadHierQuery( ). To use a query file, an extension first establishes a data connection with each database on which any queries will be run.

Next, the extension calls GXContextCreateDataConnSet( ) to create an IGXDataConnSet object, then populates this collection with query name / data connection pairs. Each query name in the collection matches a named query in the query file. IDataConnSet provides a method for adding query name / data connection pairs to the collection. In this way, extensions can use standardized queries and select and assign data connections dynamically at runtime.

Finally, the extension calls GXContextLoadHierQuery( ) to create the hierarchical query object.

Return Value

HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Related Topics

GXContextLoadHierQuery( )

GXContextCreateHierQuery( )

Creates a new query object used for building and running a hierarchical query.

Syntax

```
HRESULT GXContextCreateHierQuery(
    IGXContext *pContext,
    IGXHierQuery **pHQ);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

pHQ. A pointer to the created IGXHierQuery object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage A hierarchical query can be more complex than a flat query. A hierarchical query combines one or more flat queries which, when run on the database server, returns a result set with multiple nested levels of data. The number of nested levels is limited only by system resources.

The hierarchical query is not necessarily a single query. In fact, a hierarchical query is a collection of one or more flat queries arranged in a series of cascading parent-child, one-to-many relationships. The parent query obtains the outer level of information, or summary, and the child query obtains the inner level of information, or detail. The parent level of information determines the grouping of information in its child levels. The child query is run multiple times, once for each row in the parent query's result set.

Tips

- Use GXContextCreateQuery( ) instead for simple, flat queries requiring tabular, non-nested output that is merged with HTML templates.
- To use a hierarchical query, an extension first creates each individual flat query and defines its selection criteria. Next, it creates the IGXHierQuery object with GXContextCreateHierQuery( ), then calls AddQuery( ) repeatedly to add a child query to a parent query for each level of detail in the hierarchical query.
- Alternatively, an extension can create a hierarchical query by loading a query file using GXContextLoadHierQuery( ). With this technique, the Netscape Application Server can cache query objects to service requests for identical queries more quickly.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example

```
// Create the hierarchical query
IGXHierQuery *pHq=NULL;

if(((hr=GXContextCreateHierQuery(m_pContext, &pHq))
   ==GXE_SUCCESS)&&pHq) {
   // Add a query
```

```
pHq->AddQuery(pQuery, pConn, "SelCusts", "", "");
```

Related Topics GXContextCreateDataConn( )

GXContextCreateQuery( )

GXContextCreateMailbox( )

Creates an electronic mailbox object used for communicating with a user's mailbox.

Syntax

```
HRESULT GXContextCreateMailbox(
   IGXContext *pContext,
   LPSTR pHost,
   LPSTR pUser,
   LPSTR pPassword,
   LPSTR pUserAddr,
   IGXMailbox **ppMailbox);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

pHost. Address of POP and SMTP server, such as mail.myOrg.com. If the POP and SMTP servers are running on different hosts, you must use two separate GXContextCreateMailbox( ) calls.

pUser. Name of user's POP account, such as jdoe.

pPassword. Password for POP server.

pUserAddr. Return address for outgoing mail, such as john@myOrg.com. Usually the electronic mail address of the user sending the message.

ppMailbox. Pointer to the created IGXMailbox object. When the extension has finished using the object, call its Release( ) method to release the interface instance.

Usage Use GXContextCreateMailbox( ) to set up a mail session for sending and receiving electronic mail messages.

In the Internet electronic mail architecture, different servers are used for incoming and outgoing messages.

- POP (post-office protocol) servers process incoming mail and forward messages to the recipient's mailbox.
- SMTP (simple mail transport protocol) servers forward outgoing mail to the addressee's mail server.

Rules

- The specified user account and password must be valid for the specified POP host name.
- The user address must be valid for the specified SMTP server.

Tip Once instantiated, use the methods in the IGXMailBox interface to open and close a mailbox, as well as send and receive mail messages.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

GXContextCreateQuery( )

Creates a new query object used for building and running a flat query.

Syntax

```
HRESULT GXContextCreateQuery(
  IGXContext *pContext,
  IGXQuery **ppQuery);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

ppQuery. A pointer to the created IGXQuery object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage A flat query is the simplest type of query. It retrieves data in a tabular, non-hierarchical result set. Unlike a hierarchical query, a flat query returns a result set that is *not* divided into levels or groups.

An extension can also use GXContextCreateQuery( ) to create a query object to perform SELECT, INSERT, DELETE, or UPDATE operations on a database.

Tips

- To query a database, the extension first uses GXContextCreateQuery( ) to create the query object, then constructs the query selection criteria using methods in the IGXQuery interface, and finally runs the query on a database server. The extension can process results using methods in the IGXResultSet interface.
- Alternatively, extensions can pass a SQL SELECT statement directly to the database server using SetSQL( ) in the IGXQuery interface.
- To retrieve data with nested levels of information, use GXContextCreateHierQuery( ) instead.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example

```
// Create a query to insert data into a table
IGXQuery *pUserQuery=NULL;

if(((hr=GXContextCreateQuery(m_pContext,
   &pUserQuery))==GXE_SUCCESS)&&pUserQuery) {
   pUserQuery->SetSQL("INSERT INTO OBUser(userName, password, userType,
   eMail) VALUES (:userName, :password, :userType, :eMail)");
```

Related Topics GXContextCreateDataConn( )

GXContextCreateQuery( )

GXContextCreateTrans( )

Creates a new transaction object used for transaction processing operations on a database.

Syntax

```
HRESULT GXContextCreateTrans(
  IGXContext *pContext,
  IGXTrans **ppTrans);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

ppTrans. A pointer to the created IGXTrans object. When the extension is finished using the object, after a call to either Commit( ) or Rollback( ), call the Release( ) method to release the interface instance.

Usage Transaction processing allows the extension to define a series of database operations that succeed or fail as a group. If all operations in the group succeed, then the system commits, or saves, all of the modifications from the operations. If any operation in the group fails for any reason, then the extension can roll back, or abandon, any proposed changes to the target table(s).

If your application requires transaction processing, use GXContextCreateTrans( ) to create a transaction object. Pass this transaction object to subsequent methods, such as AddRow( ) or ExecuteQuery( ), that make up a transaction.

Tips

- Use this function in conjunction with AddRow( ), UpdateRow( ), and DeleteRow( ) methods in the IGXTable interface and ExecuteQuery( ) in the IGXDataConn interface.
- To manage transaction processing operations, use GXContextCreateTrans( ) to create an instance of the IGXTrans interface, then use Begin( ), Commit( ), and Rollback( ) in the IGXTrans interface to begin, commit, and rollback the transaction, respectively.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example

```
// Create a transaction for several insert operations
IGXTrans *pTx=NULL;

if(((hr=GXContextCreateTrans(m_pContext, &pTx))==GXE_SUCCESS)&&pTx) {
   // Begin the transaction
   pTx->Begin();
   IGXResultSet *pRset=NULL;

   // Update User
   if(((hr=pUserPQuery->Execute(0, pUserValList, pTx, NULL,
     &pRset))==GXE_SUCCESS)&&pRset) {

      // The result set is not needed; release it
      pRset->Release();

      // Update Customer
      if(((hr=pCustPQuery->Execute(0, pCustValList, pTx, NULL,
          &pRset))==GXE_SUCCESS)&&pRset) {

         // All updates succeeded. Commit the transaction
```

```
            pTx->Commit(0, NULL);
            GXSetValListString(pValIn, "ssn", m_pSsn);
            GXSetValListString(pValIn, "OUTPUTMESSAGE", "Successfully
             updated customer record");

            if(GXContextNewRequest("AppLogic CShowCustPage", pValIn,
                pValOut, 0)!=GXE_SUCCESS)
               HandleOBSystemError("Could not chain to CShowCustPage
                AppLogic");
            }
         else {
            pTx->Rollback();
            HandleOBSystemError("Could not insert checking account record
                for new customer");
         }
      }
      else {
         pTx->Rollback();
         HandleOBSystemError("Could not insert checking account record for
                new customer");
      }
      pTx->Release();
   }
   else
      HandleOBSystemError("Could not start transaction");
```

GXContextDestroySession( )

Deletes a user session.

Syntax

```
HRESULT GXContextDestroySession(
   IGXContext *pContext,
   LPSTR pAppName,
   LPSTR pSessionID,
   IGXSessionIDGen *pIDGen);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

pAppName. Name of the application associated with the session. The application name enables the Netscape Application Server to determine which extensions have access to the session data.

pSessionID. The session ID to use.

pIDGen. The session ID generation object used to generate session IDs. Specify NULL to use the default IGXSessionIDGen object, or specify a custom session ID generation object.

Usage To increase security and conserve system resources, use GXContextDestroySession( ) to delete a session between a user and the application when the session is no longer required. An extension typically calls GXContextDestroySession( ) when the user logs out of an application.

Tip If the extension set a timeout value for the session when it was created, you need not delete the session explicitly with GXContextDestroySession( ). The session is deleted automatically when the timeout expires.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

GXContextGetAppEvent( )

Retrieves the application event object.

Syntax

```
HRESULT GXContextGetAppEvent(
   IGXContext *pContext,
   IGXAppEvent **ppAppEvent);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

ppAppEvent. A pointer to the retrieved IGXAppEvent object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage Use GXContextGetAppEvent( ) to retrieve an IGXAppEvent object. Through the IGXAppEvent interface, you can create and manage application events. An extension uses application event objects to define events that are triggered at a specified time or times or when triggered explicitly.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

GXContextGetObject( )

Retrieves an object from the context.

Syntax

```
HRESULT GXContextGetObject(
   IGXContext *pContext,
   LPSTR service
   IID iid
   LPVOID *ppObj);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

service. Name of the service in the context to retrieve.

iid. Identifier of the interface sought on the service.

ppObj. Returned service object. If the returned object is an IGXObject, then call Release( ) when done with the service object.

Usage Call GXContextGetObject to retrieve a service (from another extension, for example), in order to access the services of another loaded extension.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example

```
// Get the data access engine object from the context
IGXObject *pObj=NULL;
HRESULT hr;
hr=GXContextGetObject(m_pContext,
                      "IID_IGXModuleData",
                      IID_IGXModuleData,
                      (LPVOID *)&pObj);
```

GXGetStateTreeRoot( )

Returns an existing root node of a state tree or creates a new one.

Syntax

```
HRESULT GXGetStateTreeRoot(
   IGXContext *pContext,
   DWORD dwFlags,
   LPSTR pName,
   IGXState2 **ppStateTree)
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

dwFlags. Specify one of the following flags or zero to use the default settings:

- GXSTATE_LOCAL to make the node visible to the local process only.
- GXSTATE_CLUSTER to make the node visible within the cluster.
- GXSTATE_DISTRIB, the default, to make the node visible on all servers.
- GXSTATE_PERSISTENT to write the data to a persistent store that survives server crashes.

pName. The name of the root node. If a node with this name doesn't already exist, a new node is created.

ppStateTree. A pointer to the created IGXState2 object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage Use GXGetStateTreeRoot( ) to create a state tree. A state tree is a hierarchical data storage mechanism. It is used primarily for storing application data that needs to be distributed across server processes and clusters.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example The following code shows how to create a state tree and a child node:

```
HRESULT hr;

hr = GXGetStateTreeRoot(m_pContext, GXSTATE_DISTRIB|GXSTATE_PERSISTENT,
  "Grammy", &m_pStateRoot);

if (hr == NOERROR && m_pStateRoot)
{
   IGXState2 *pState = NOERROR;
   hr = m_pStateRoot->GetStateChild("Best Female Vocal",
      &pState);
   if (hr != NOERROR || !pState)
   {
     hr = m_pStateRoot->CreateStateChild("Best Female Vocal",
         0, GXSTATE_DISTRIB|GXSTATE_PERSISTENT, &pState);
```

GXContextIsAuthorized( )

Checks a user's permission level to a specified action.

Syntax 1 Use in most cases.

```
HRESULT GXContextIsAuthorized(
   IGXContext *pContext,
   LPSTR pTarget,
   LPSTR pPermission,
   DWORD *pResult);
```

Syntax 2 Contains several parameters that are place holders for future functionality.

```
HRESULT GXContextIsAuthorized(
   IGXContext *pContext,
   LPSTR pDomain,
   LPSTR pTarget,
   LPSTR pPermission,
   DWORD method,
   DWORD flags,
   IGXCred *pCred,
   IGXObject *pEnv,
   DWORD *pResult);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

pDomain. The type of Access Control Lists (ACL). An ACL (created by the server administrator) defines the type of operations, such as Read or Write, that a user or group can perform. There are two types of ACLs: AppLogic and general. For this parameter, specify one of the following strings, which specifies the type of ACL to check for this user:

"kiva:acl,logic"

"kiva:acl,general"

pTarget. The name of the ACL, if the ACL is a general type. If the ACL is an AppLogic ACL, specify the AppLogic name or GUID string.

pPermission. The type of permission, for example, "EXECUTE."

method. Specify 0.

flags. Specify 0.

pCred. Specify NULL.

pEnv. Specify NULL.

pResult. Pointer to the client-allocated variable that contains the returned permission status. The variable is set to one of the following enum constants:

Constant	Description
GXACL_ALLOWED	The specified permission is granted to the user.
GXACL_NOTALLOWED	The specified permission is not granted to the user.
GXACL_DONTKNOW	The specified permission is unlisted or there is conflicting information.

Usage Use GXContextIsAuthorized( ) in portions of the code where application security is enforced through Access Control Lists (ACL). This function lets an application check if a user has permission to execute an AppLogic or perform a particular action. The application can use the result of GXContextIsAuthorized( ) as a condition in an If statement. It can, for example, return a message to users who are denied access to an AppLogic.

Application developers should obtain the list of registered ACLs, users and groups from the server administrator who created these items. ACLs are created through the Enterprise Administrator tool or through the kreg tool.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Value	Description
GXACLPERMSTATUS.GXACL_ALLOWED	The specified permission is granted to the user.
GXACLPERMSTATUS.GXACL_NOTALLOWED	The specified permission is not granted to the user.
GXACLPERMSTATUS.GXACL_DONTKNOW	The specified permission is unlisted or there is conflicting information.

Example

```
DWORD auth_result = 0;

if (GXContextIsAuthorized("Shop_Inventory", "WRITE", &auth_result) !=
    NOERROR || auth_result != (DWORD) GXACL_ALLOWED)
{
   GXContextLog("Unauthorized access: Shop_Inventory");

   return GXE_FAIL;
}
else
// Update inventory
```

GXContextLoadHierQuery( )

Creates a hierarchical query by loading a query file containing one or more query names and associated data connections.

Syntax

```
HRESULT GXContextLoadHierQuery
   IGXContext *pContext,
   LPSTR pFileName,
   IGXDataConnSet *pDataConnSet,
   DWORD flags,
   IGXValList *pParams,
   IGXHierQuery **ppHierQuery);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

pFileName. Name of the query (.GXQ) file, including the path. Use a relative path when possible.

A query file is an ASCII text file containing one or more SQL statements. You can create the file using any ASCII text editor. Use the following syntactical guidelines:

- The file for a hierarchical query contains several SQL SELECT statements (compliant with ANSI SQL89) with the following additions:
  - Each query is preceded by the following line:

    query *queryName* using (*driverCode*, *DSN*, *UserName*) is

- For a child query, append the following line after the SQL SELECT statement:

  ```
  join currentQueryName to parent parentName where
  currentQueryName.table.column = parentName.colOrAlias
  ```

- In the query file, do not use any semicolons (;) or other vendor-specific SQL statement terminators.

pDataConnSet. Collection of query name/data connection pairs. The query names in the collection must match the named queries in the query file. The associated IDataConn object identifies the data connection for the query.

flags. Specify 0 (zero). Internal use only.

pParams. IGXValList of query file parameters, or NULL. A collection of place holders for the WHERE clause. A place holder may be a name or a number. It is prefixed by a colon (:) character. The place holders can be replaced by specifying replacement values in the ValList parameter.

ppHierQuery. Pointer to the created IGXHierQuery object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage Use GXContextLoadHierQuery( ) to create a hierarchical query object. An extension can retrieve standardized queries stored in a data file and, at runtime, can dynamically select and assign the data sources on which the query is run. You create the query file separately using the Netscape Application Builder query editor window or an ASCII text editor, ANSI 89 standard SQL SELECT statements, and specialized Netscape Application Server syntax. A query file can define both flat and hierarchical queries.

To use a query file, the extension first establishes a data connection with each database on which any queries will be run. Next, the extension calls GXContextCreateDataConnSet( ) to create an IGXDataConnSet collection, then populates this collection with query name / data connection pairs. Each query name in the collection matches a named query in the query file.

IGXDataConnSet provides a method for adding query name / data connection pairs to the collection. In this way, extension can use standardized queries and assign data connections dynamically at runtime. Finally, the extension calls GXContextLoadHierQuery( ) to create the hierarchical query object.

Rules

- The extension must first call GXContextCreateDataConnSet( ) to create an IGXDataConnSet, then add query name / data connection pairs using AddConn( ) in the IGXDataConnSet interface.
- The query names in the collection must match the query names in the query file.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example The following example shows a query (GXQ) file and a section of an extension that loads the hierarchical query file and creates an HTML report:

Query file:

```
/* STATES */
query STATES using (ODBC, kstates, kuser) is
select STATES.STATE as STATES_STATE
from STATES
where (STATES.REGION = ':REGION')
order by STATES.STATE asc

/* DETAILS */
query DETAILS using (ODBC, kdetails, kuser) is
select COUNTIES.COUNTYNAM as COUNTIES_COUNTYNAM,
   COUNTIES.POP as COUNTIES_POP,
   COUNTIES.STATE as COUNTIES_STATE
from COUNTIES
order by COUNTIES.COUNTYNAM asc

join DETAILS to parent STATES
where DETAILS.COUNTIES.STATE = 'STATES.STATES_STATE'
```

Extension code snippet:

```
IGXDataConnSet *connSet = NULL;
HRESULT hr;
hr = GXContextCreateDataConnSet(m_pContext, 0, &connSet);
if (hr == GXE_SUCCESS)
{
   // Create database connections
   IGXDataConn *conn_detailDB = NULL;
   IGXDataConn *conn_statesDB = NULL;

   IGXValList *pList=GXCreateValList();
```

```
pList->SetValString("DSN", "kdetails");
pList->SetValString("DB", "");
pList->SetValString("USER", "kuser");
pList->SetValString("PSWD", "kpassword");

// Create first connection
hr = GXContextCreateDataConn(m_pContext, 0, GX_DA_DRIVER_DEFAULT,
pList,
       NULL, &conn_detailDB);
if (hr == GXE_SUCCESS)
{
   pList->SetValString("DSN", "dstates");
   pList->SetValString("DB", "");
   pList->SetValString("USER", "kuser");
   pList->SetValString("PSWD", "kpassword");

   // Create second connection
   hr = GXContextCreateDataConn(m_pContext, 0, GX_DA_DRIVER_DEFAULT,
    pList,
                        NULL, &conn_statesDB);
   pList->Release();

   if (hr == GXE_SUCCESS)
   {
      // Specify query / db connection pairs
      connSet->AddConn("DETAILS", conn_detailDB);
      connSet->AddConn("STATES", conn_statesDB);

      // Create IGXValList that contains the
      // REGION parameter value to pass to the
      // hierarchical query
      IGXValList param = GXCreateValList();
      param->SetValString("REGION", "WEST");

      IGXHierQuery *hqry;
      // Load the GXQ file with the db connection set
      // and parameter value

      hr = GXContextLoadHierQuery(m_pContext, "state.gxq", connSet,
       0,
                           param, &hqry);
```

```
if (hr == GXE_SUCCESS)
{
   // Run the report ....
}
```

Related Topics GXContextCreateDataConnSet( )

GXContextLoadQuery( )

Creates a flat query by loading a query file.

Syntax

```
HRESULT GXContextLoadQuery(
   IGXContext *pContext,
   LPSTR pFileName,
   LPSTR pQueryName,
   DWORD flags,
   IGXValList *pParams,
   IGXQuery **ppQuery);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

pFileName. Name of the query (.GXQ) file, including the path. Use a relative path when possible.

A query file is an ASCII text file containing one or more SQL statements. You can create the file using any ASCII text editor. Use the following syntactical guidelines:

- The query file for a flat query contains a SQL SELECT statement (compliant with ANSI SQL89) preceded by the following line:

  ```
  /* optional comments */
  query queryName using (driverCode, DSN, UserName) is
  ```

  where *queryName* is the name of the flat query. Do not use any semicolons (;) in the query file.

- In the query file, do not use any semicolons (;) or other vendor-specific SQL statement terminators. The SQL statement may contain place holders in the WHERE clause.

pQueryName. Name of the query in the query file.

flags. Specify 0 (zero). Internal use only.

pParams. IGXValList of query file parameters, or null. A collection of place holders for the WHERE clause. A placeholder may be a name or a number. It is prefixed by a colon (:) character. The place holders can be replaced by specifying replacement values in the IGXValList parameter.

ppQuery. Pointer to the created IGXQuery object. When the extension is finished using the object, call the Release( ) method to release the interface instance.

Usage Use GXContextLoadQuery( ) to create a flat query object by loading a query (.GXQ) file. An extension can retrieve standardized queries stored in a data file and, at runtime, can dynamically select and assign the data source on which the query is run.

You create the query file separately using the Netscape Application Builder query editor window or an ASCII text editor, ANSI 89 standard SQL SELECT statements, and special Netscape Application Server syntax.

To run the flat query, call ExecuteQuery( ) in the IGXDataConn interface.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example The following example shows a query (GXQ) file and a section of an extension that loads and executes the query:

Query file:

```
/* STATES */
query STATES using (ODBC, kstates, kuser) is
select STATES.STATE as STATES_STATE
from STATES
where (STATES.REGION = ':REGION')
order by STATES.STATE asc
```

Extension code snippet:

```
HRESULT hr;

// Create database connection
IGXDataConn *conn = NULL;

IGXValList *pList=GXCreateValList();
pList->SetValString("DSN", "kstates");
```

```
pList->SetValString("DB", "");
pList->SetValString("USER", "kuser");
pList->SetValString("PSWD", "kpassword");

hr = GXContextCreateDataConn(m_pContext, 0, GX_DA_DRIVER_DEFAULT,
pList,
                        NULL, &conn);
if (hr == GXE_SUCCESS)
{
   // Create IGXValList that contains the REGION
   // parameter value to pass to the
   // hierarchical query
   IGXValList param = GXCreateValList();
   param->SetValString("REGION", "WEST");

   IGXQuery *qry;

   // Load the GXQ file with the parameter value
   hr = GXContextLoadQuery(m_pContext, "state.gxq", "STATES", 0,
                    param, &qry);

   // Execute the query
   IGXResultSet *rs = NULL;
   hr = conn->ExecuteQuery(GX_DA_RS_BUFERRING, qry, NULL,
               NULL, &rs);
```

GXContextLog( )

Writes a log message to an output device.

Syntax 1 Logs a message (type = GXEVENTTYPE_INFORMATION and category = 0).

```
HRESULT GXContextLog(
   IGXContext *pContext,
   LPSTR msg);
```

Syntax 2 Logs an event with a message, specifying the type and category of event.

```
HRESULT GXContextLog(
   IGXContext *pContext,
   DWORD type,
   DWORD category,
   LPSTR msg);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

msg. Message text to log.

type. Message type. Use one of the following variables:

- GXEVENTTYPE_INFORMATION
- GXEVENTTYPE_ERROR
- GXEVENTTYPE_SYSTEM
- GXEVENTTYPE_WARNING

category. User-defined message category. Do not use the range of values reserved for the Netscape systems, which is 0 to 65535, inclusive.

Usage Use GXContextLog( ) for displaying or storing simple messages or for debugging. The output can be directed to the console, to a text file, or to a database table. To direct output, use the Netscape Application Server Administrator. For more information, see the *Administration Guide*.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example

```
hr = GXContextGetSession(m_pContext, 0, "Catalog", NULL, &m_pSession);
if (hr != GXE_SUCCESS)
{
   GXContextLog(m_pContext, "Could not get session; creating a new
     one");
   hr = GXContextCreateSession(m_pContext, GXSESSION_DISTRIB, 0, NULL,
          NULL, NULL, &m_pSession);
```

GXContextNewRequest( )

Calls an AppLogic.

Syntax 1 Passes in the specified IGXValList of input parameters and result values. The location of the AppLogic execution depends on partitioning and load balancing criteria.

```
HRESULT GXContextNewRequest(
   IGXContext *pContext,
   LPSTR guid,
   IGXObject *vIn,
```

```
   IGXObject *vOut,
   DWORD flag);
```

Syntax 2 Same as Syntax 1, but explicitly specifies the location of AppLogic execution.

```
HRESULT GXContextNewRequest(
   IGXContext *pContext,
   LPSTR guid,
   IGXObject *vIn,
   IGXObject *vOut,
   DWORD host,
   DWORD port,
   DWORD flag);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

guid. String GUID or name of the AppLogic to execute.

vIn. IGXValList object containing input parameters to pass to the called AppLogic.

vOut. IGXValList object containing result values of the called AppLogic.

host. IP address of the Internet host of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

port. Internet port of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

flag. Specify zero.

Usage Use GXContextNewRequest( ) to call an AppLogic. When you call GXContextNewRequest( ), the extension passes to the Netscape Application Server the GUID or name of the AppLogic to execute and, optionally, any input and output parameters.

Netscape Application Server constructs a request using the parameters specified and processes it like any other request, by instantiating the AppLogic and passing in its parameters. The results from the called AppLogic module are returned to the calling extension.

The AppLogic that GXContextNewRequest( ) invokes can do one of the following tasks:

- Process application logic and return result values in the vOut parameter.
- Process application logic and return the resulting data form (such as a report) by streaming the output or by calling Result( ).
- Process application logic and return result values in the vOut parameter as well as return the resulting data form (such as a report) by streaming the output or by calling Result( ).

If the called AppLogic uses EvalOutput( ) to stream results, EvalOutput( ) returns HTML results by default. The caller can, however, specify that EvalOutput( ) return a non-HTML data stream by setting the gx_client_type key to "ocl" in the input IGXValList of GXContextNewRequest( ). For example:

```
vallist.SetValString("gx_client_type", "ocl");
```

Rule The specified GUID string, input parameters, and output parameters must be valid for the specified AppLogic.

Tips
- The calling code can create new input and output IGXValLists so as to avoid changing its own input and output IGXValLists.
- Use GXContextNewRequestAsync( ) instead of GXContextNewRequest( ) to execute asynchronous request.
- Called AppLogics might reside on different servers, depending on partitioning and load balancing configurations, might be written in a different language, or might have cached results. The calling code can be unaware or independent of these conditions.
- Using GXContextNewRequest( ), you can modularize parts of the application, build dynamic header/footer information and smart reporting templates, and hide complex or confidential business logic in secure submodules or even separate servers.
- Use GXContextNewRequest( ) judiciously. Each invoked AppLogic uses a certain amount of communications and server resources.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

GXContextNewRequestAsync( )

Calls another AppLogic and runs it asynchronously.

Syntax 1 Passes in the specified IGXValList of input parameters and result values. The location of the AppLogic execution depends on partitioning and load balancing criteria.

```
HRESULT GXContextNewRequestAsync(
   IGXContext *pContext,
   LPSTR guid,
   IGXObject *vIn,
   IGXObject *vOut,
   DWORD flag,
   IGXOrder **ppOrder);
```

Syntax 2 Same as Syntax 1, but explicitly specifies the location of AppLogic execution.

```
HRESULT GXContextNewRequestAsync(
   IGXContext *pContext,
   LPSTR guid,
   IGXObject *vIn,
   IGXObject *vOut,
   DWORD host,
   DWORD port,
   DWORD flag
   IGXOrder **ppOrder);
```

pContext. A pointer to an IGXContext object, which provides access to Netscape Application Server services. Specify m_pContext if accessing this helper function from a manager class; otherwise, use m_pModule->m_pContext.

guid. String GUID or name of the AppLogic to execute.

vIn. IGXValList object containing input parameters to pass to the called AppLogic.

vOut. IGXValList object containing result values of the called AppLogic.

host. IP address of the Internet host of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

port. Internet port of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

flag. Specify 0.

ppOrder. Pointer to the returned IGXOrder object, which the caller can use to obtain the status of the request. When the caller is finished using the order object, call the Release( ) method to release the interface instance.

Usage Use GXContextNewRequestAsync( ) to call another AppLogic and run it asynchronously. Executing an AppLogic asynchronously is useful if the AppLogic performs a lengthy operation, or if the AppLogic acts as a monitor or remains persistent. For example, an asynchronous AppLogic may perform a lengthy database query to produce a complex result set that it e-mails to a destination address. Another AppLogic module may run continuously and re-index HTML pages every 24 hours.

When an extension calls GXContextNewRequestAsync( ), it passes to the Netscape Application Server the GUID of the AppLogic module to execute and, optionally, any input and output parameters.

The Netscape Application Server constructs a request using the parameters specified and processes it like any other request, by instantiating the AppLogic and passing in its parameters. The results from the called AppLogic module are returned to the calling code.

The AppLogic that GXContextNewRequestAsync( ) invokes can do one of the following tasks:

- Process application logic and return result values in the vOut parameter.
- Process application logic and return the resulting data form (such as a report) by streaming the output or by calling Result( ).
- Process application logic and return result values in the vOut parameter as well as return the resulting data form (such as a report) by streaming the output or by calling Result( ).

Rules

- The specified AppLogic must be accessible to the Netscape Application Server.
- The specified GUID string, input parameters, and output parameters must be valid for the specified AppLogic module.

Tips

- To get the current status of the request, use the GetState( ) method in the returned IGXOrder object.
- The calling code can use GXWaitForOrder( ) to wait for multiple asynchronous requests to return.

- Using GXContextNewRequestAsync( ), you can modularize parts of the application, build dynamic header/footer information and smart reporting templates, and hide complex or confidential business logic in secure submodules or even separate servers.
- Use GXContextNewRequestAsync( ) judiciously. Each invoked AppLogic uses a certain amount of communications and server resources.

Return Value HRESULT, which is set to GXE_SUCCESS if the function succeeds.

Example

```
IGXOrder *pOrder;
ULONG     nOrder;
HRESULT hr, ReqResult;

if (GXContextNewRequestAsync(m_pContext, asyncGUIDStr, m_pValIn,
                        m_pValOut, 0, &pOrder) == GXE_SUCCESS)
{
   GXContextLog(m_pContext, "Successfully invoked async AppLogic\n");

   // wait for async AppLogic to finish (max 100 seconds)
   hr = GXWaitForOrder(&pOrder, 1, &nOrder, m_pContext, 100);
   if (hr != NOERROR)
   {
      return hr;
   }
   else
   {
      pOrder->GetState(NULL, &ReqResult, NULL);
      if (ReqResult != NOERROR)
         return ReqResult;
   }
}
else
{
GXContextLog(m_pContext,
"Failed to invoke async AppLogic\n");
}
```

Appendix

B

Java Helper Static Methods

The GXContext class is a utility class that enables Java extension writers to access core Netscape Application Server functionality. The GXContext class provides a suite of helper static methods that can be used anywhere in an extension's method stubs.

You call the methods in the GXContext class by using the following convention: GXContext.<method>

The following Java helper static methods are described in this appendix:

GXContext.CreateDataConn( )
GXContext.CreateDataConnSet( )
GXContext.CreateHierQuery( )
GXContext.CreateMailbox( )
GXContext.CreateQuery( )
GXContext.CreateTrans( )
GXContext.DestroySession( )
GXContext.GetAppEvent( )
GXContext.GetObject( )
GXContext.GetStateTreeRoot( )
GXContext.IsAuthorized( )
GXContext.LoadHierQuery( )
GXContext.LoadQuery( )
GXContext.Log( )
GXContext.NewRequest( )
GXContext.NewRequestAsync( )

CreateDataConn( )

Creates a new data connection object and opens a connection to a database or data source.

Syntax 1 Use this version for most database drivers.

```
public static IDataConn CreateDataConn(
  IContext context,
  int flags,
  int driver,
  String datasource,
  String database,
  String username,
  String password)
```

Syntax 2 Use this version for database drivers requiring parameters not found in Syntax 1. Provides parameters through an IValList object instead. To use this syntax, you must first create an instance of the IValList interface and use setValString( ) to specify the connection parameter names and values.

```
public static IDataConn CreateDataConn(
  IContext context,
  int flags,
  int driver,
  IValList prop)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

flags. One or more optional flags used for connecting to the specified data source.

- To try to use a cached connection, if one is available, specify GX_DA_CONN_FLAGS.GX_DA_CACHED. If no cached connections are currently available, a new one is created.
- To always create a new connection (instead of using a cached connection), specify GX_DA_CONN_FLAGS.GX_DA_NEW.
- To retry if a connection is not available, specify GX_DA_CONN_FLAGS.GX_DA_CONN_BLOCK.
- To return a failure immediately after the first attempt if a connection is not available, specify GX_DA_CONN_FLAGS.GX_DA_CONN_NOBLOCK.

The caller can pass one parameter from both mutually exclusive pairs, as shown in the following example:

(GX_DA_CONN_FLAGS.GX_DA_CACHED |
GX_DA_CONN_FLAGS.GX_DA_CONN_BLOCK)

Specify 0 (zero) to use the system's default settings:
GX_DA_CONN_FLAGS.GX_DA_CACHED and
GX_DA_CONN_FLAGS.GX_DA_CONN_BLOCK

driver. Specify one of the following static variables in the GX_DA_DAD_DRIVERS class:

GX_DA_DRIVER_ODBC	GX_DA_DRIVER_SYBASE_CTLIB
GX_DA_DRIVER_MICROSOFT_JET	GX_DA_DRIVER_MICROSOFT_SQL
GX_DA_DRIVER_INFORMIX_SQLNET	GX_DA_DRIVER_INFORMIX_CLI
GX_DA_DRIVER_INFORMIX_CORBA	GX_DA_DRIVER_DB2_CLI
GX_DA_DRIVER_ORACLE_OCI	GX_DA_DRIVER_DEFAULT

If GX_DA_DRIVER_DEFAULT is specified, the Netscape Application Server evaluates the drivers and their associated priorities set in the registry to determine the driver to use. Specify GX_DA_DRIVER_DEFAULT if your system uses ODBC and native drivers, and if you want the Netscape Application Server to choose between an ODBC driver and a native driver at connection time.

datasource. Name of the data source to connect to. Used for databases that organize data into data source, database, and table objects and sub-objects. For example, a data source for Accounting might contain separate databases for GL, AP, and AR. Each database, such as Accounts Payable, might be a collection of individual tables, such as a vendor table, purchase order table, materials table, and so on. See your database server documentation for more information.

database. Name of the database to connect to. Used for database servers that organize data into databases and tables. See your database server documentation for more information.

username. Login user name that is valid for the specified database.

password. Login password that is valid for the specified user name and database.

props. IValList of connection-specific information required to log in to the data source. Use the following keys for the connection parameters:

- `"DSN"` for the data source name.

- `"DB"` for the database name.
- `"USER"` for the user name.
- `"PSWD"` for the password.

Usage A data connection is a communication link or session with a database or other data source. Before interacting with a data source, an extension must first establish a connection with it. Each connection is represented by a data connection object, which contains all the information needed to communicate with a database or data source, such as the name of the database, database driver, user name, password, and so on. A data connection object is an instance of the IDataConn interface.

Use CreateDataConn( ) to set up a separate connection for each database or data source you want to access. Extension objects refer to the data connection object in their methods that perform subsequent operations on the database.

Rules
- Call CreateDataConn( ) before running any other database operations requiring a data connection object.
- Your network and the database server must be correctly configured and running so that the extension on your application server can log into the database management system with which it will communicate.
- The data source name, database name, user name, and password must be valid for the database management system to which you want to connect.
- The extension must log in with sufficient access rights to perform all operations it attempts on the data source.

Tips
- Before logging in to the database, the extension should check the user's security level to verify sufficient access rights to perform intended operations on the database.
- The Data Access Engine (DAE) manages database connections and related housekeeping tasks, such as shutdown and cleanup. While the DAE performs these tasks automatically and intermittently, an extension can also explicitly close data connections using closeConn( ) in the IDataConn interface.
- Before using an ODBC connection, you must use the ODBC administration utility supplied with your database software to define and name a data source. For more information about how to do this, refer to your ODBC documentation.

Return Value IDataConn object representing the specified data connection, or null if a failure occurred. An extension uses the data connection object to uniquely identify this data connection in subsequent operations, such as queries and record insert, update, and delete operations.

Example

```
// Method to open a connection to a database
protected com.kivasoft.IDataConn getOBDataConn()
   {
      String username = "kdemo";
      String password = "kdemo";
      IDataConn newDataConn = CreateDataConn(m_Context, 0,
              GX_DA_DAD_DRIVERS.GX_DA_DRIVER_ODBC,
              /* Datasource name. */ "ksample",
              /* Database name.   */ "",
              /* userName.        */ username,
              /* password.        */ password);

      if (newDataConn == null)
      {
         Log(m_Context, "ERROR: Could not create database connection");
      }
         return newDataConn;
   }
```

CreateDataConnSet( )

Creates a collection used to dynamically assign query name/data connection pairs before loading a query file.

Syntax

```
public static IDataConnSet CreateDataConnSet(
  IContext context,
   int flags)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

flags. Specify 0. Internal use only.

Usage Use CreateDataConnSet( ) only if you are loading a query file using LoadHierQuery( ). To use a query file, an extension first establishes a data connection with each database on which any queries will be run.

Next, the extension calls CreateDataConnSet( ) to create an IDataConnSet object, then populates this collection with query name / data connection pairs. Each query name in the collection matches a named query in the query file. IDataConnSet provides a method for adding query name / data connection pairs to the collection. In this way, the extension can use standardized queries and select and assign data connections dynamically at runtime.

Finally, the extension calls LoadHierQuery( ) to create the hierarchical query object.

Return Value IDataConnSet object that can hold a collection of data connections, or null for failure (such as insufficient memory).

Example

```
IDataConnSet connSet;
connSet = CreateDataConnSet(m_Context, 0);
// Specify query / db connection pairs
connSet.addConn("employee", conn_empDB);
connSet.addConn("sales", conn_salesDB);
IHierQuery hqry;
// Load the GXQ file with the db connection set
hqry = LoadHierQuery(m_Context, "employeeReport.gxq",
                     connSet, 0, null);

// Run the report
evalTemplate("employeeReport.html", hqry);
```

Related Topics LoadHierQuery( )

CreateHierQuery( )

Creates a new query object used for building and running a hierarchical query.

Syntax

```
public static IHierQuery CreateHierQuery(
  IContext context
)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

Usage A hierarchical query can be more complex than a flat query. A hierarchical query combines one or more flat queries which, when run on the database server, returns a result set with multiple nested levels of data. The number of nested levels is limited only by system resources.

The hierarchical query is not necessarily a single query. In fact, a hierarchical query is a collection of one or more flat queries arranged in a series of cascading parent-child, one-to-many relationships. The parent query obtains the outer level of information, or summary, and the child query obtains the inner level of information, or detail. The parent level of information determines the grouping of information in its child levels. The child query is run multiple times, once for each row in the parent query's result set.

Tips

- Use CreateQuery( ) instead for simple, flat queries requiring tabular, non-nested output that is merged with HTML templates.
- To use a hierarchical query, an extension first creates each individual flat query and defines its selection criteria. Next, it creates the IHierQuery object with CreateHierQuery( ), then calls addQuery( ) repeatedly to add a child query to a parent query for each level of detail in the hierarchical query.
- Alternatively, an extension can create a hierarchical query by loading a query file using LoadHierQuery( ). With this technique, the Netscape Application Server can cache query objects to service requests for identical queries more quickly.

Return Value IHierQuery object representing a hierarchical query, or null for failure. An extension uses this object to uniquely identify this hierarchical query in subsequent operations, such as defining the query criteria, executing the query, retrieving query results, and processing templates.

Example 1

```
// Create the flat query
IQuery qry = CreateQuery(m_Context);
qry.setTables("CTLusers");
qry.setFields("loginName, Password, AccessLevel")
qry.setOrderBy("LoginName);

// Create the hierarchical query used for template processing
IHierQuery hqry = CreateHierQuery(m_Context);

// Add the flat query object and data connection to hqry
hqry.addQuery(qry, conn, "USERS", "", "");
```

```
// Pass hierarchical query to evalTemplate() for reporting
if(evalTemplate("apps/template/userinfo.html", hqry)== GXE.SUCCESS)
   return result("");
else
   return result("Failed to Generate HTML");
}
```

Example 2

```
// Create the flat query
IQuery qry = CreateQuery(m_Context);
qry.setTables("CTLusers");
qry.setFields("loginName, Password, AccessLevel")
qry.setWhere("UserId > 100");

// Create the hierarchical query
IHierQuery hqry = CreateHierQuery(m_Context);
hqry.addQuery(qry, conn, "USERS", "", "");

// Execute the hierarchical query
IHierResultSet hrs = hqry.execute(0, 0, null);

// Process rows in result set
if(hrs.getRowNumber("USERS")!=0)
. . .
```

Related Topics CreateDataConn( )

CreateMailbox( )

Creates an electronic mailbox object used for communicating with a user's mailbox.

Syntax

```
public static IMailbox CreateMailbox(
   IContext context,
   String pHost,
   String pUser,
   String pPassword,
   String pUserAddr)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

pHost. Address of POP and SMTP server, such as mail.myOrg.com. If the POP and SMTP servers are running on different hosts, you must use two separate CreateMailbox( ) calls.

pUser. Name of user's POP account, such as jdoe.

pPassword. Password for POP server.

pUserAddr. Return address for outgoing mail, such as john@myOrg.com. Usually the electronic mail address of the user sending the message.

Usage Use CreateMailbox( ) to set up a mail session for sending and receiving electronic mail messages.

In the Internet electronic mail architecture, different servers are used for incoming and outgoing messages.

- POP (post-office protocol) servers process incoming mail and forward messages to the recipient's mailbox.
- SMTP (simple mail transport protocol) servers forward outgoing mail to the addressee's mail server.

Rules
- The specified user account and password must be valid for the specified POP host name.
- The user address must be valid for the specified SMTP server.

Tip Once instantiated, use the methods in the IMailbox interface to open and close a mailbox, as well as send and receive mail messages.

Return Value IMailbox object representing a mailbox, or null for failure (such as an invalid parameter).

CreateQuery( )

Creates a new query object used for building and running a flat query.

Syntax
```
public static IQuery CreateQuery(
  IContext context
)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

Usage A flat query is the simplest type of query. It retrieves data in a tabular, non-hierarchical result set. Unlike a hierarchical query, a flat query returns a result set that is *not* divided into levels or groups.

An extension can also use CreateQuery( ) to create a query object to perform SELECT, INSERT, DELETE, or UPDATE operations on a database.

Tips

- To query a database, the extension first uses CreateQuery( ) to create the query object, then constructs the query selection criteria using methods in the IQuery interface, and finally runs the query on a database server. The extension can process results using methods in the IResultSet interface.
- Alternatively, the extension can pass a SQL SELECT statement directly to the database server using setSQL( ) in the IQuery interface.
- To retrieve data with nested levels of information, use CreateHierQuery( ) instead.

Return Value IQuery object representing a query, or null for failure. An extension uses this object to uniquely identify this query in subsequent operations, such as defining the query criteria, executing the query, retrieving query results, and processing templates.

Example

```
// Create the flat query object
IQuery qry = CreateQuery(m_Context);
// Set up the query
qry.setTables("CTLcust");
qry.setFields("CustomerID, Customer");
qry.setWhere("Customer"+"='"+String.valueOf(custId)+"'");

// Execute the query
IResultSet rs = conn.executeQuery(0, qry, null, null);

// Check for a result set with rows
if((rs!=null)&&(rs.getRowNumber()>0))
   return result("Sorry, this user ("+
   firstName+" "+lastName+") already exists");
// Otherwise, process the result set . . .
```

Related Topics CreateDataConn( ),

CreateQuery( )

CreateTrans( )

Creates a new transaction object used for transaction processing operations on a database.

Syntax

```
public static ITrans CreateTrans(
   IContext context
)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

Usage

Transaction processing allows the extension to define a series of operations that succeed or fail as a group. If all operations in the group succeed, then the system commits, or saves, all of the modifications from the operations. If any operation in the group fails for any reason, then the extension can roll back, or abandon, any proposed changes to the target table(s).

If your application requires transaction processing, use CreateTrans( ) to create a transaction object. Pass this transaction object to subsequent methods, such as addRow( ) or executeQuery( ), that make up a transaction.

Tips

- Use this method in conjunction with addRow( ),updateRow( ), and deleteRow( ) methods in the ITable interface and executeQuery( )in the IDataConn interface.
- To manage transaction processing operations, use CreateTrans( ) to create an instance of the ITrans interface, then use begin( ), commit( ), and rollback( ) in the ITrans interface to begin, commit, and rollback the transaction, respectively.

Return Value

ITrans object representing a transaction, or null for failure. The extension uses this object to uniquely identify this transaction in subsequent transaction processing operations, such as beginning, committing, or rolling back a transaction.

Example

```
// Create and begin a transaction
ITrans trx = CreateTrans(m_Context);
trx.begin();

// 1) Process the credit card
if(!processCreditCard(cusId, card, number, expirationDate, trx)) {
   trx.rollback();
```

```
   return result("Could not process the credit card information"); }

// 2) Process the invoice record
InfoHolder info=new InfoHolder();
if(!makeInvoiceRecord(cusId, number, trx, info)) {
   trx.rollback();
   return result("Could not create the invoice record"); };

// 3) Process products on the invoice
if(!makeInvoiceEntries(info.invoiceId, trx)) {
   trx.rollback();
   return result("Can't create product records"); };

// 4) Process optional shipping information
if(shippingInfo && !makeShippingRecord(info.invoiceId, trx, addr1,
   addr2, city, state, zip)) {
   trx.rollback();
   return result("Could not create the shipping information record"); };

// 5) Process the inventory for each purchased product
if(!reduceProductInventory(trx)) {
   // Problem occurred - abandon everything
   trx.rollback();
return result("Could not reduce inventory");
}

// No problem occurred - save everything
trx.commit(0);
// Return success message / report
```

DestroySession( )

Deletes a user session.

Syntax

```
public static int DestroySession(
   IContext context,
   String pAppName
   String pSessionID
   ISessionIDGen pIDGen)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

pAppName. Name of the application associated with the session. The application name enables the Netscape Application Server to determine which extensions have access to the session data.

pSessionID. The session ID to use.

pIDGen. The session ID generation object used to generate session IDs. Specify null.

Usage To increase security and conserve system resources, use DestroySession( ) to delete a session between a user and the application when the session is no longer required. An extension typically calls DestroySession( ) when the user logs out of an application.

Tip If the extension set a timeout value for the session when it was created, you need not delete the session explicitly with DestroySession( ). The session is deleted automatically when the timeout expires.

Return Value GXE.SUCCESS if the method succeeds.

Related Topics CreateSession( )

GetAppEvent( )

Retrieves the application event object.

Syntax

```
public static IAppEvent GetAppEvent(
   IContext context
)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

Usage Use GetAppEvent( )to retrieve an IAppEvent object. Through the IAppEvent interface, you can create and manage application events. An extension uses application event objects to define events that are triggered at a specified time or times or when triggered explicitly.

Return Value IAppEvent object, or null for failure.

GetObject( )

Retrieves an object from the context.

Syntax

```
public static Object GetObject(
   IContext context,
   String contextName)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

contextName. The name of the object in the context. If an object with this name does not exist, null is returned.

Usage Call GetObject( ) to retrieve a service (from another extension, for example), in order to access the services of another loaded extension.

Return Value The returned object, or null for failure.

Example

```
IModuleData modData=GXContext.GetObject(context,
     "com.kivasoft.IModuleData");
```

GetStateTreeRoot( )

Returns an existing root node of a state tree or creates a new one.

Syntax

```
public static IState2 GetStateTreeRoot(
   IContext context,
   int dwFlags,
   String pName)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

dwFlags. Specify one of the following flags or zero to use the default settings.

- GXSTATE.GXSTATE_LOCAL to make the node visible to the local process only.
- GXSTATE.GXSTATE_CLUSTER to make the node visible within the cluster.
- GXSTATE.GXSTATE_DISTRIB, the default, to make the node visible on all servers.

- GXSTATE.GXSTATE_PERSISTENT to write the data to a persistent store that survives server crashes.

pName. The name of the root node. If a node with this name doesn't already exist, a new node is created.

Usage Use GetStateTreeRoot( ) to create a state tree. A state tree is a hierarchical data storage mechanism. It is used primarily for storing application data that needs to be distributed across server processes and clusters.

Return Value IState2 object representing the root node, or null for failure.

Example The following code shows how to create a state tree and a child node.

```
IState2 tree = GetStateTreeRoot(m_Context,
   GXSTATE.GXSTATE_DISTRIB|GXSTATE.GXSTATE_PERSISTENT,
   "Grammy");

if (tree!=null)
{
   IState2 child = tree.getStateChild("Best Female Vocal");
   if (child == null)
   {
      child = tree.createStateChild("Best Female Vocal", 0,
        GXSTATE.GXSTATE_DISTRIB|GXSTATE.GXSTATE_PERSISTENT);
```

IsAuthorized( )

Checks a user's permission level to a specified action.

Syntax 1 Use in most cases.

```
public static int IsAuthorized(
   IContext context,
   String pTarget,
   String pPermission)
```

Syntax 2 Contains several parameters that are place holders for future functionality.

```
public static int IsAuthorized(
   IContext context,
   String pDomain,
   String pTarget,
   String pPermission,
   int method,
```

```
int flags,
ICred pCred,
IObject pEnv)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

pDomain. The type of Access Control Lists (ACL). An ACL (created by the server administrator) defines the type of operations, such as Read or Write, that a user or group can perform. There are two types of ACLs: AppLogic and general. For this parameter, specify one of the following strings, which specifies the type of ACL to check for this user:

"kiva:acl,logic"

"kiva:acl,general"

pTarget. The name of the ACL, if the ACL is a general type. If the ACL is an AppLogic ACL, specify the AppLogic name or GUID string.

pPermission. The type of permission, for example, “EXECUTE.”

method. Specify 0.

flags. Specify 0.

pCred. Specify null.

pEnv. Specify null.

Usage Use IsAuthorized( ) in portions of the code where application security is enforced through Access Control Lists (ACL). This method lets an application check if a user has permission to execute an AppLogic or perform a particular action. The application can use the result of IsAuthorized( ) as a condition in an If statement. It can, for example, return a message to users who are denied access to an AppLogic.

Application developers should obtain the list of registered ACLs, users and groups from the server administrator who created these items. ACLs are created through the Enterprise Administrator tool or through the kreg tool.

Return Value One of the following:

Value	Description
GXACLPERMSTATUS.GXACL_ALLOWED	The specified permission is granted to the user.
GXACLPERMSTATUS.GXACL_NOTALLOWED	The specified permission is not granted to the user.
GXACLPERMSTATUS.GXACL_DONTKNOW	The specified permission is unlisted or there is conflicting information.

Example

```
if (IsAuthorized(m_Context, "MonthlyForecast", "READ") !=
GXACLPERMSTATUS.GXACL_ALLOWED)
   return result("<html><body>sorry no access</body><html>);
else
// run monthly forecast report
```

LoadHierQuery( )

Creates a hierarchical query by loading a query file containing one or more query names and associated data connections.

Syntax

```
public static IHierQuery LoadHierQuery(
   IContext context,
   String pFileName,
   IDataConnSet pDataConnSet,
   int flags,
   IValList pParams)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

pFileName. Name of the query (.GXQ) file, including the path. Use a relative path when possible.

A query file is an ASCII text file containing one or more SQL statements. You can create the file using any ASCII text editor. Use the following syntactical guidelines:

- The file for a hierarchical query contains several SQL SELECT statements (compliant with ANSI SQL89) with the following additions.

- Each query is preceded by the following line.

  ```
  query queryName using (driverCode, DSN, UserName) is
  ```

- For a child query, append the following line after the SQL SELECT statement.

  ```
  join currentQueryName to parent parentName where
  currentQueryName.table.column = parentName.colOrAlias
  ```

- In the query file, do not use any semicolons (;) or other vendor-specific SQL statement terminators.

pDataConnSet. Collection of query name/data connection pairs. The query names in the collection must match the named queries in the query file. The associated IDataConn object identifies the data connection for the query.

flags. Specify 0 (zero). Internal use only.

pParams. IValList of query file parameters, or null. A collection of place holders for the WHERE clause. A place holder may be a name or a number. It is prefixed by a colon (:) character. The place holders can be replaced by specifying replacement values in the ValList parameter.

Usage Use LoadHierQuery( ) to create a hierarchical query object. An extension can retrieve standardized queries stored in a data file and, at runtime, can dynamically select and assign the data sources on which the query is run. You create the query file separately using the Netscape Application Builder query editor window or an ASCII text editor, ANSI 89 standard SQL SELECT statements, and specialized Netscape Application Server syntax. A query file can define both flat and hierarchical queries.

To use a query file, the extension first establishes a data connection with each database on which any queries will be run. Next, the extension calls CreateDataConnSet( ) to create an IDataConnSet collection, then populates this collection with query name / data connection pairs. Each query name in the collection matches a named query in the query file.

IDataConnSet provides a method for adding query name / data connection pairs to the collection. In this way, the extension can use standardized queries and assign data connections dynamically at runtime. Finally, the extension calls LoadHierQuery( ) to create the hierarchical query object.

Rules

- The extension must first call CreateDataConnSet( ) to create an IDataConnSet, then add query name / data connection pairs using addConn( ) in the IDataConnSet interface.
- The query names in the collection must match the query names in the query file.

Return Value IHierQuery object representing a hierarchical query, or null for failure (such as query file not found). The extension uses this object to uniquely identify this hierarchical query in subsequent operations, such as defining the query criteria, executing the query, retrieving query results, and processing templates.

Example The following example shows a query (GXQ) file and a section of an extension that loads the hierarchical query file and creates an HTML report:

Query file:

```
/* STATES */
query STATES using (ODBC, kstates, kuser) is
select STATES.STATE as STATES_STATE
from STATES
where (STATES.REGION = ':REGION')
order by STATES.STATE asc

/* DETAILS */
query DETAILS using (ODBC, kdetails, kuser) is
select COUNTIES.COUNTYNAM as COUNTIES_COUNTYNAM,
   COUNTIES.POP as COUNTIES_POP,
   COUNTIES.STATE as COUNTIES_STATE
from COUNTIES
order by COUNTIES.COUNTYNAM asc

join DETAILS to parent STATES
where DETAILS.COUNTIES.STATE = 'STATES.STATES_STATE'
```

Extension code snippet:

```
IDataConnSet connSet;
connSet = CreateDataConnSet(m_Context, 0);

// Create database connections
IDataConn conn_detailDB = CreateDataConn(m_Context, 0,
GX_DA_DAD_DRIVERS.GX_DA_DRIVER_DEFAULT, "kdetails", "", "kuser",
"kpassword");
```

```
IDataConn conn_statesDB = CreateDataConn(m_Context, 0,
GX_DA_DAD_DRIVERS.GX_DA_DRIVER_ODBC, "kstates", "", "kuser",
"kpassword");

// Specify query / db connection pairs
connSet.addConn("DETAILS", conn_detailDB);
connSet.addConn("STATES", conn_statesDB);

// Create IValList that contains the REGION parameter
// value to pass to the hierarchical query
IValList param = GX.CreateValList();
param.setValString("REGION", "WEST");

IHierQuery hqry;
// Load the GXQ file with the db connection set and
// parameter value
hqry = LoadHierQuery(m_Context, "state.gxq", connSet, 0,
     param);

// Run the report
evalTemplate("state.html", hqry);
```

Related Topics CreateDataConnSet( )

LoadQuery( )

Creates a flat query by loading a query file.

Syntax

```
public static IQuery LoadQuery(
   IContext context,
   String pFileName,
   String pQueryName,
   int flags,
   IValList pParams)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

pFileName. Name of the query (.GXQ) file, including the path. Use a relative path when possible.

A query file is an ASCII text file containing one or more SQL statements. You can create the file using any ASCII text editor. Use the following syntactical guidelines:

- The query file for a flat query contains a SQL SELECT statement (compliant with ANSI SQL89) preceded by the following line:

  ```
  /* optional comments */
  query queryName using (driverCode, DSN, UserName) is
  ```

  where *queryName* is the name of the flat query. Do not use any semicolons (;) in the query file.

- In the query file, do not use any semicolons (;) or other vendor-specific SQL statement terminators. The SQL statement may contain place holders in the WHERE clause.

pQueryName. Name of the query in the query file.

flags. Specify 0 (zero). Internal use only.

pParams. IValList of query file parameters, or null. A collection of place holders for the WHERE clause. A place holder may be a name or a number. It is prefixed by a colon (:) character. The place holders can be replaced by specifying replacement values in the IValList parameter.

Usage Use LoadQuery( ) to create a flat query object by loading a query (.GXQ) file. An extension can retrieve standardized queries stored in a data file and, at runtime, can dynamically select and assign the data source on which the query is run.

You create the query file separately using the Netscape Application Builder or an ASCII text editor, ANSI 89 standard SQL SELECT statements, and special Netscape Application Server syntax.

To run the flat query, call executeQuery( ) in the IDataConn interface.

Return Value IQuery object, or null for failure (such as query file not found).

Example The following example shows a query (GXQ) file and a section of an extension that loads and executes the query:

Query file:

```
/* STATES */
query STATES using (ODBC, kstates, kuser) is
```

```
select STATES.STATE as STATES_STATE
from STATES
where (STATES.REGION = ':REGION')
order by STATES.STATE asc
```

Extension code snippet:

```
// Create database connection
IDataConn conn = CreateDataConn(m_Context, 0,
GX_DA_DAD_DRIVERS.GX_DA_DRIVER_DEFAULT, "kstates", "", "kuser",
"kpassword");

// Create IValList that contains the REGION parameter
// value to pass to the query
IValList param = GX.CreateValList();
param.setValString("REGION", "WEST");

IQuery qry;
// Load the query file with the parameter value
qry = LoadQuery(m_Context, "state.gxq", "STATES", 0, param);

// Execute the query
IResultSet rs = conn.executeQuery(
GX_DA_EXECUTEQUERY_FLAGS.GX_DA_RS_BUFFERING, qry, null, null);
```

Log( )

Writes a log message to an output device.

Syntax Logs an event with a message, specifying the type and category of event.

```
public static int Log(
   IContext context,
   int type,
   int category,
   String msg)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

msg. Message text to log.

type. Message type. Use one of the following variables:

- GXLOG.GXEVENTTYPE_INFORMATION
- GXLOG.GXEVENTTYPE_ERROR
- GXLOG.GXEVENTTYPE_SYSTEM
- GXLOG.GXEVENTTYPE_WARNING

category. User-defined message category. Do not use the range of values reserved for the Netscape systems, which is 0 to 65535, inclusive.

Usage Use Log( ) for displaying or storing simple messages or for debugging. The output can be directed to the console, to a text file, or to a database table. To direct output, use the Netscape Application Server Administrator. For more information, see the *Administration Guide*.

Return Value GXE.SUCCESS if the method succeeds.

Example

```
// Log messages that include String variables
Log(m_Context, firstName+lastName+password+"
    "+String.valueOf(cusId));
Log(m_Context, GXLOG.GXEVENTTYPE_ERROR", -1,
    Cannot find table: "+"Shipping");
```

NewRequest( )

Calls an AppLogic.

Syntax 1 Passes in the specified valIn and valOut.

```
public static int NewRequest(
   IContext context,
   String guidSTR,
   IValList vIn,
   IValList vOut,
   int flags)
```

Syntax 2 Use to explicitly specify the location of AppLogic execution.

```
public static int NewRequest(
   IContext context,
   String guidSTR,
   IObject vIn,
   IObject vOut,
   int host,
```

```
   int port)
```

Syntax 3 Use to explicitly specify the location of AppLogic execution.

```
public static int NewRequest(
   IContext context,
   String guidSTR,
   IObject vIn,
   IObject vOut,
   int host,
   int port,
   int flags)
```

context. An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

guidSTR . String GUID or name of the AppLogic to execute.

vIn. IValList object containing input parameters to pass to the called AppLogic.

vOut. IValList object containing result values of the called AppLogic.

host. IP address of the Internet host of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

port. Internet port of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

flags. Specify zero.

Usage Use NewRequest( ) to call an AppLogic. When you call NewRequest( ), the extension passes to the Netscape Application Server the GUID or name of the AppLogic to execute and, optionally, any input and output parameters.

Netscape Application Server constructs a request using the parameters specified and processes it like any other request, by instantiating the AppLogic and passing in its parameters. The results from the called AppLogic module are returned to the calling extension.

The AppLogic that NewRequest( ) invokes can do one of the following tasks:

- Process application logic and return result values in the vOut parameter.
- Process application logic and return the resulting data form (such as a report) by streaming the output or by calling result( ).

- Process application logic and return result values in the vOut parameter as well as return the resulting data form (such as a report) by streaming the output or by calling result( ).

If the called AppLogic uses evalOutput( ) to stream results, evalOutput( ) returns HTML results by default. The caller can, however, specify that evalOutput( ) return a non-HTML data stream by setting the gx_client_type key to "ocl" in the input IValList of NewRequest( ). For example:

```
vallist.setValString("gx_client_type", "ocl");
```

Rule The specified GUID string, input parameters, and output parameters must be valid for the specified AppLogic.

Tips

- The calling code can create new input and output IValLists so as to avoid changing its own input and output IValLists.
- Use NewRequestAsync( ) instead of NewRequest( ) to execute asynchronous request.
- Called AppLogics might reside on different servers, depending on partitioning and load balancing configurations, might be written in a different language, or might have cached results. The calling code can be unaware or independent of these conditions.
- Using NewRequest( ), you can modularize parts of the application, build dynamic header/footer information and smart reporting templates, and hide complex or confidential business logic in secure submodules or even separate servers.
- Use NewRequest( ) judiciously. Each invoked AppLogic uses a certain amount of communications and server resources.

Return Value GXE.SUCCESS if the method succeeds.

Example 1

```
// Call specified AppLogic and pass parameters
NewRequest(m_Context,
   "{E5CA1000-6EEE-11cf-96FD-0020AFED9A65}",
   paramsToModule, paramsReturned);
```

Example 2

```
// Use DisplayBasket AppLogic to display the contents
if(NewRequest(m_Context, "DisplayBasket",
       valIn, valOut)==0){
   return 0;
   }
```

```
else
   return result("Cannot execute DisplayBasket AppLogic");
```

NewRequestAsync( )

Calls another AppLogic and runs it asynchronously.

Syntax 1 Passes in the specified valIn and valOut.

```
public static IOrder NewRequestAsync(
   IContext context,
   String guidSTR,
   IValList vIn,
   IValList vOut,
   int flags)
```

Syntax 2 Use to explicitly specify the location of AppLogic execution.

```
public static IOrder NewRequestAsync(
   IContext context,
   String guidSTR,
   IObject vIn,
   IObject vOut,
   int host,
   int port)
```

Syntax 3 Use to explicitly specify the location of AppLogic execution.

```
public static IOrder NewRequestAsync(
   IContext context,
   String guidSTR,
   IObject vIn,
   IObject vOut,
   int host,
   int port,
   int flags)
   context
```

An IContext object, which provides access to Netscape Application Server services. Specify m_Context if accessing this helper static method from a manager class; otherwise, use m_Module.m_Context.

guidSTR . String GUID or name of the AppLogic to execute.

vIn. IValList object containing input parameters to pass to the called AppLogic.

vOut. IValList object containing result values of the called AppLogic.

host. IP address of the Internet host of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

port. Internet port of the Netscape Application Server where the AppLogic is to be executed. Specify 0 to execute the AppLogic locally.

flags. Specify 0.

Usage

Use NewRequestAsync( ) to call another AppLogic and run it asynchronously. Executing an AppLogic asynchronously is useful if the AppLogic performs a lengthy operation, or if the AppLogic acts as a monitor or remains persistent. For example, an asynchronous AppLogic may perform a lengthy database query to produce a complex result set that it e-mails to a destination address. Another AppLogic module may run continuously and re-index HTML pages every 24 hours.

When an extension calls NewRequestAsync( ), it passes to the Netscape Application Server the GUID of the AppLogic module to execute and, optionally, any input and output parameters.

The Netscape Application Server constructs a request using the parameters specified and processes it like any other request, by instantiating the AppLogic and passing in its parameters. The results from the called AppLogic module are returned to the calling code.

The AppLogic that NewRequestAsync( ) invokes can do one of the following tasks:

- Process application logic and return result values in the vOut parameter.
- Process application logic and return the resulting data form (such as a report) by streaming the output or by calling result( ).
- Process application logic and return result values in the vOut parameter as well as return the resulting data form (such as a report) by streaming the output or by calling result( ).

Rules

- The specified AppLogic must be accessible to the Netscape Application Server.
- The specified GUID string, input parameters, and output parameters must be valid for the specified AppLogic module.

Tips

- To get the current status of the request, use the getState( ) method in the returned IOrder object.
- The calling code can use GX.WaitForOrder( ) to wait for multiple asynchronous requests to return.
- Using NewRequestAsync( ), you can modularize parts of the application, build dynamic header/footer information and smart reporting templates, and hide complex or confidential business logic in secure submodules or even separate servers.
- Use NewRequestAsync( ) judiciously. Each invoked AppLogic uses a certain amount of communications and server resources.

Return Value IOrder object, or null for failure.

Example

```
IOrder  Orders[1];
ULONG     nOrder;
HRESULT hr, ReqResult;

Orders[0] = NewRequestAsync(m_Context, asyncGUIDStr, valIn,
                            valOut);
if (Orders[0] != null)
{
   Log(m_Context, "Successfully invoked async AppLogic\n");

   // wait for async applogic to finish (max 100 seconds)
   hr = waitForOrder(Orders, context, 100);
   if (hr != GXE.SUCCESS)
   {
       return result("Error in executing async request:
          order wait returned an error");
   }
   else
   {
      getStateIOrder state = Orders[0].getState();
      if (state == null || state.pdwState != GXE.SUCCESS)
         return result("Error in executing async

           request");
   }
}
else
```

```
{
   Log(m_Context, "Failed to invoke async AppLogic\n");
}
```

Appendix

C

Java Class Decorations

Information in this section applies only to C++ extensions with a Java Access Layer.

Netscape Application Server provides a set of predefined interfaces. As you define your own interface methods in an access module, you can specify any one of these Netscape Application Server-defined interfaces as the value for the Out parameter type. If you define an Out parameter as a Netscape Application Server-defined type, you must set the Java Class decoration for this parameter in any coclass that implements your interface.

In the following table, the lefthand column shows the parameter types you can specify for a method's Out parameter. You specify this value in an access module. The righthand column shows the corresponding decoration you must set in any coclass where you implement the method. You specify decorations in a service module.

Predefined Netscape Application Server Interface	Value to Use for Java Class Decoration
IGXBuffer	com.kivasoft.util.Buffer
IGXCallableStmt	com.kivasoft.data.CallableStmt
IGXColumn	com.kivasoft.data.Column
IGXDataConn	com.kivasoft.data.DataConn
IGXDataConnSet	com.kivasoft.loadq.DataConnSet

Predefined Netscape Application Server Interface	Value to Use for Java Class Decoration
IGXEnumObject	com.kivasoft.util.EnumObject
IGXHierQuery	com.kivasoft.datap.HierQuery
IGXHierResultSet	com.kivasoft.datap.HierResultSet
IGXMailbox	com.kivasoft.mailbox.Mailbox
IGXObject	com.kivasoft.types.COM
IGXOrder	com.kivasoft.util.Order
IGXPreparedQuery	com.kivasoft.data.PreparedQuery
IGXQuery	com.kivasoft.data.Query
IGXResultSet	com.kivasoft.data.ResultSet
IGXSequence	com.kivasoft.data.Sequence
IGXSession2	com.kivasoft.session.Session2wrap
IGXState2	com.kivasoft.state.State2
IGXStream	com.kivasoft.util.Stream
IGXStreamBuffer	com.kivasoft.util.StreamBuffer
IGXTable	com.kivasoft.data.Table
IGXTemplateData	com.kivasoft.tmpl.TemplateData
IGXTemplateMap	com.kivasoft.tmpl.TemplateMap
IGXTrans	com.kivasoft.trans.Trans
IGXValList	com.kivasoft.util.ValList

Appendix

D

Reserved Words

This appendix specifies keywords used internally by Netscape Extension Builder.

The following keywords are reserved. Do not use any of these keywords as object identifiers in Netscape Extension Builder.

__stdcall
abstract
appobject
attribute
bool
boolean
byte
call_as
char
class
coclass
const
cpp_class
cpp_code
cpp_quote
cpp_wrapper_class
default_value
double
dword
endpoint
enum
exception
extends
extension_language
extension_name
final
fixed_size
float
garbage_collect
gen_dir
global
helpcontext
helpstring
hide
hresult
iid_is
implements
import
import_file
in
in/out
inout

int
interface
java_class
java_code
java_gen_only
java_type
keep_pooled_object
length_is
local
long
lpbyte
lpcstr
lpstr
lpvoid
module
native
no_corba_gen
no_corba_glue_gen
no_corba_idl_gen
no_corba_proxy_gen
no_corba_proxy_java_gen
no_cpp_gen
no_gen
no_interface_header_gen
no_java_gen
no_java_interface_gen
no_java_wrapper_gen
no_native_c_gen
object
object_pool_config
object_pool_name
octet
out
package
pointer_default
pooled_inparam
pooled_introspection
pooled_no_reuse
pooled_object
pooled_object_name
private
project_file
protected
ptr
public
raises
readonly
readwrite
ref
sequence
short
size_is
smart_cpp_proxy
smart_java_proxy
smart_proxy
smart_proxy_data
static
stdmethod
stdmethod_
struct
synch_level
synchronized
throws_exceptions
transient
typedef
ulong
union
unique
unsigned
use_coclass_lock
use_extension_lock
use_module_lock
use_no_lock
uses_pooled_object
uuid
version
void
volatile
word
wrapper_uuid
writeonly

Appendix

E

The ConnManager.cpp File

This appendix contains sample source code.

The following code sample is found in the file:

`sample\poolsample\cpp\connService\ConnManager.cpp`

This file is used by PoolSampleExt, the sample extension described in Chapter 9, “Using Object Pools.” ConnManager.cpp is the manager class of the extension. The file shows one way of implementing creation methods as well as the methods in IGXObjectEvaluation interface.

```
//
//    This file is initially generated by KIDL - Edit as
//    necessary to complete
//

#include <stdio.h>
#include <gxplat.h>
#include <gxutil.h>
#include <gxdlm.h>

#include "connservice.h"

#include "gxiextdata.h"
#include "gxextutil.h"

LPSTR ConnManager::m_appNameLocal = "ConnManagerLocal";
```

```
LPSTR ConnManager::m_appNameDistributed = "ConnManagerDistributed";
LPSTR ConnManager::m_appNameCluster = "ConnManagerCluster";

/*
[uuid(55863a3d-bacd-1505-f0f3-0800208055c0),
wrapper_uuid(55873c6d-bacd-1505-f0f3-0800208055c0)]
*/
GUID ConnManagerGUID =
{ 0x55863a3d,0xbacd,0x1505,{0xf0,0xf3,0x08,0x00,0x20,0x80,0x55,0xc0} };

// The following must belong in one, and only one, C++ file
GXDLM_IMPLEMENT_BEGIN()
GXDLM_IMPLEMENT(ConnManager, ConnManagerGUID);
GXDLM_IMPLEMENT_END();

ConnManager::ConnManager()
{
    GXDllLockInc();
    m_nInits=0;
    m_pContext=NULL;
    //***
    //*** Add your own code to the constructor here.
    //*** Note: You can change this constructor or add new
    //*** constructors
    //***
    GXSYNC_INIT(&(Connection::G_Sync));
}

ConnManager::~ConnManager()
{
    if(m_pContext)
        m_pContext->Release();
    //***
    //*** Add your own code to the destructor here.
    //***
    GXSYNC_DESTROY(&(Connection::G_Sync));
    GXDllLockDec();
}

HRESULT
ConnManager::Init(IGXObject *pObj)
{
```

```
    // If we haven't been initialized ...
    if(!m_nInits) {
        // If we have a context already, release it
        if(m_pContext)
            m_pContext->Release();
        // Pull the context from the argument
        m_pContext = GXGetContext(pObj);
        if (m_pContext) {
            // And set myself in the context, so other folks
            // can see me
            m_pContext->SetObject("IID_IConnManager", 0, 0, 0,
              (IGXObject*)(IConnManager*)this, 0, 0, 0);
        }
        //***
        //*** Add your own service initialization code here.
        //*** This method (Init) gets called by the runtime
        //*** when the service is initially loaded by the
        //*** Extension Manager
        //***
    }
    m_nInits++;
    return GXE_SUCCESS;
}

HRESULT
ConnManager::Uninit(IGXObject *pObj)
{
    if(m_nInits > 0)
        m_nInits--;
    return GXE_SUCCESS;
}

// Stub method bodies for interface: IConnManager
HRESULT
ConnManager::CreateConnection(
        /* [in] */ unsigned long uid,
        /* [in] */ LPSTR passwd,
        /* [in] */ unsigned long flags,
        /* [out] */ IFakeConnection **ppConn)
{
    HRESULT hr=GXE_SUCCESS;
    //***
```

```
    //*** Provide your implementation here
    //*** and remove the GXASSERT below
    //***
    *ppConn = new Connection(this, uid, passwd, flags);
    return hr;
}

HRESULT
ConnManager::CreateConnectionIntrospection(
        /* [in] */ unsigned long uid,
        /* [in] */ LPSTR passwd,
        /* [in] */ unsigned long flags,
        /* [out] */ IInfo  **ppConn)
{
    HRESULT hr=GXE_SUCCESS;
    //***
    //*** Provide your implementation here
    //*** and remove the GXASSERT below
    //***
    *ppConn = new Info(uid, passwd, flags);
    return hr;
}

HRESULT
ConnManager::CreateTxConnection(
        /* [in] */ unsigned long uid,
        /* [in] */ LPSTR passwd,
        /* [in] */ unsigned long flags,
        /* [out] */ IFakeConnection **ppConn)
{
    HRESULT hr=GXE_SUCCESS;
    //***
    //*** Provide your implementation here
    //*** and remove the GXASSERT below
    //***
    *ppConn = new TxConnection(this, uid, passwd, flags);
    return hr;
}

HRESULT
ConnManager::CreateTxConnectionIntrospection(
        /* [in] */ unsigned long uid,
```

```
        /* [in] */ LPSTR passwd,
        /* [in] */ unsigned long flags,
        /* [out] */ IInfo  **ppConn)
{
    HRESULT hr=GXE_SUCCESS;
    //***
    //*** Provide your implementation here
    //*** and remove the GXASSERT below
    //***
    *ppConn = new Info(uid, passwd, flags);
    return hr;
}

HRESULT
ConnManager::CreateNonPooledConnection(
        /* [in] */ unsigned long uid,
        /* [in] */ LPSTR passwd,
        /* [in] */ unsigned long flags,
        /* [out] */ IFakeConnection **ppConn)
{
    HRESULT hr=GXE_SUCCESS;
    //***
    //*** Provide your implementation here
    //*** and remove the GXASSERT below
    //***
    *ppConn = new Connection(this, uid, passwd, flags);
    return hr;
}

// Stub method bodies for interface: IGXObjectEvaluation
HRESULT
ConnManager::MatchObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [in] */ IGXObject *pPhysical,
      /* [out] */ BOOL *pMatches)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
```

```
//***

// Depending on poolName, determine if the objects match:
// Navigate to the introspection interface of the
// virtual & physical objects, get their attributes,
// and see if they match.

// CONN_POOL: match on userid & passwd
// TXCONN_POOL: match on userid, passwd AND flags
//       (match criteria is stricter than CONN_POOL)

if (!strcmp(poolName, "CONN_POOL"))
{
   *pMatches = FALSE;

   IInfo* pV = NULL;
   IInfo* pP = NULL;

   // Navigate to IInfo interfaces

   if((((hr=pVirtual->QueryInterface(IID_IInfo,
       (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))
          &&(((hr=pPhysical->QueryInterface(IID_IInfo,
       (LPVOID*)&pP))==GXE_SUCCESS)&&(pP)))
   {
      ULONG PUid = 0;
      ULONG VUid = 0;
      char PPasswd[256];
      char VPasswd[256];

      // Get uid, passwd for both virtual
      // & physical objects

      if(hr==GXE_SUCCESS)
         hr=pV->GetUid(&VUid);
      if(hr==GXE_SUCCESS)
         hr=pV->GetPasswd(VPasswd, 256);
      if(hr==GXE_SUCCESS)
         hr=pP->GetUid(&PUid);
      if(hr==GXE_SUCCESS)
         hr=pP->GetPasswd(PPasswd, 256);
```

```
      // Check if userid & password match

      if ((hr==GXE_SUCCESS)
         && (PUid==VUid)
         && (!strcmp(PPasswd, VPasswd)))
         *pMatches = TRUE;
   }

   // Release interfaces
   if (pP)
      pP->Release();
   if (pV)
      pV->Release();
}

else // TXCONN_POOL
{
   *pMatches = FALSE;

   IInfo* pV = NULL;
   IInfo* pP = NULL;

   // Navigate to IInfo interfaces

   if((((hr=pVirtual->QueryInterface(IID_IInfo,
      (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))
       &&(((hr=pPhysical->QueryInterface(IID_IInfo,
      (LPVOID*)&pP))==GXE_SUCCESS)&&(pP)))
   {
      ULONG PUid = 0;
      ULONG VUid = 0;
      ULONG PFlags = 0;
      ULONG VFlags = 0;
      char PPasswd[256];
      char VPasswd[256];
      if(hr==GXE_SUCCESS)
         hr=pV->GetUid(&VUid);
      if(hr==GXE_SUCCESS)
         hr=pV->GetPasswd(VPasswd, 256);
      if(hr==GXE_SUCCESS)
         hr=pP->GetUid(&PUid);
      if(hr==GXE_SUCCESS)
```

```
            hr=pP->GetPasswd(PPasswd, 256);
         if(hr==GXE_SUCCESS)
            hr=pP->GetFlags(&PFlags);
         if(hr==GXE_SUCCESS)
            hr=pV->GetFlags(&VFlags);

         // Check if userid, password & flags match

         if ((hr==GXE_SUCCESS)
            && (PUid==VUid)
            && (PFlags==VFlags)
            && (!strcmp(PPasswd, VPasswd)))
            *pMatches = TRUE;
      }

      // Release interfaces
      if (pP)
         pP->Release();
      if (pV)
         pV->Release();
   }

   return hr;
}

HRESULT
ConnManager::StealObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [in] */ IGXObject *pPhysical,
      /* [out] */ BOOL *pCanSteal)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Allow any connection to be replaced.

   *pCanSteal = TRUE;
```

```
   return hr;
}

HRESULT
ConnManager::GetHint(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [out] */ LPSTR hint,
      /* [in] */ unsigned long Size)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Don't want to provide any hint.

   strcpy(hint, "");
   return hr;
}

HRESULT
ConnManager::CreateObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [out] */ IGXPoolObject **ppPhysical)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Depending on the poolName, navigate to the
   // introspection interface of the virtual object,
   // get the encapsulated attributes,
   // and create the appropriate physical object with the
   // same attributes.
```

```
if (!strcmp(poolName, "CONN_POOL"))
{
   IInfo* pV = NULL;
   IFakeConnection* pConn = NULL;
   ULONG uid = 0;
   ULONG flags = 0;
   char passwd[256];

   // Navigate to IInfo interface

   if(((hr=pVirtual->QueryInterface(IID_IInfo,
      (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))
   {
      // Get the uid, passwd, flags

      if(hr==GXE_SUCCESS)
         hr=pV->GetUid(&uid);
      if(hr==GXE_SUCCESS)
         hr=pV->GetPasswd(passwd, 256);
      if(hr==GXE_SUCCESS)
         hr=pV->GetFlags(&flags);
   }

   // Create a matching Connection object.

   if (hr==GXE_SUCCESS)
   {
      hr = CreateConnection(uid, passwd, flags, &pConn);

      // QueryInterface to IGXPoolObject
      if((hr==GXE_SUCCESS)&& pConn)
         hr=pConn->QueryInterface(IID_IGXPoolObject,
            (LPVOID*)ppPhysical);
   }

   // Release interfaces
   if (pV)
      pV->Release();
   if (pConn)
      pConn->Release();
}
```

```
else // TXCONN_POOL
{
   IInfo* pV = NULL;
   IFakeConnection* pConn = NULL;
   ULONG uid = 0;
   ULONG flags = 0;
   char passwd[256];

   // Get to IInfo interface

   if(((hr=pVirtual->QueryInterface(IID_IInfo,
       (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))
   {
      // Get uid, passwd, flags

      if(hr==GXE_SUCCESS)
         hr=pV->GetUid(&uid);
      if(hr==GXE_SUCCESS)
         hr=pV->GetPasswd(passwd, 256);
      if(hr==GXE_SUCCESS)
         hr=pV->GetFlags(&flags);
   }

   // Create a matching TxConnection object.

   if (hr==GXE_SUCCESS)
   {
      hr = CreateTxConnection(uid, passwd, flags, &pConn);

      // QueryInterface to IGXPoolObject
      if((hr==GXE_SUCCESS)&& pConn)
         hr=pConn->QueryInterface(IID_IGXPoolObject,
            (LPVOID*)ppPhysical);
   }

   // Release interfaces
   if (pV)
      pV->Release();
   if (pConn)
      pConn->Release();
}
```

```
   return hr;

}

HRESULT
ConnManager::InitObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pVirtual,
      /* [in] */ IGXObject *pPhysical)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

// Remember: the match criteria for objects in
// CONN_POOL did not include flag settings.
// We therefore need to restore the flag settings
// of the physical object, to match those of the
// virtual object.

   // For CONN_POOL objects, do flag settings here.

   if (!strcmp(poolName, "CONN_POOL"))
   {
      // Get to IInfo interface

      IInfo* pV = NULL;

      if(((hr=pVirtual->QueryInterface(IID_IInfo,
          (LPVOID*)&pV))==GXE_SUCCESS)&&(pV))

      {
         ULONG flags = 0;
         hr=pV->GetFlags(&flags);

         // Get Connection implementation

         IFakeConnection* pIConn=NULL;
```

```
         if(((hr=pPhysical->QueryInterface
             (IID_IFakeConnection,(LPVOID*)&pIConn))
             ==GXE_SUCCESS)&&(pIConn))
         {
            // Set flags
            pIConn->SetFlags(flags);
         }
      }
   }

   return hr;
}

HRESULT
ConnManager::UninitObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pPhysical)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***

   // Nothing to do.

   return hr;
}

HRESULT
ConnManager::ReleaseObject(
      /* [in] */ LPSTR poolName,
      /* [in] */ IGXObject *pPhysical,
      /* [in] */ unsigned long reason)
{
   HRESULT hr=GXE_SUCCESS;

   //***
   //*** Provide your implementation here
   //*** and remove the GXASSERT below
   //***
```

```
   // Would have closed any backend connection here.
   // Nothing to do in this dummy implementation.

   return hr;
}

// Stub method bodies for interface: IGXCreateIntrospection
HRESULT
ConnManager::CreateIntrospectObject(
        /* [in] */ LPSTR methodname,
        /* [in] */ LPSTR signature,
        /* [in] */ LPVOID ParamStack)
{
    if (!strcmp(methodname,"CreateConnection"))
    {
        if (strcmp(signature, "unsigned long.LPSTR.unsigned
          long.IInfo*."))
            return GXE_FAIL;
        return CreateConnectionIntrospection(
                *((unsigned long*)((LPVOID*)ParamStack)[0])
                ,*((LPSTR*)((LPVOID*)ParamStack)[1])
                ,*((unsigned long*)((LPVOID*)ParamStack)[2])
                ,((IInfo**)((LPVOID*)ParamStack)[3])
        );
    }
    if (!strcmp(methodname,"CreateTxConnection"))
    {
        if (strcmp(signature, "unsigned long.LPSTR.unsigned
          long.IInfo*."))
            return GXE_FAIL;
        return CreateTxConnectionIntrospection(
                *((unsigned long*)((LPVOID*)ParamStack)[0])
                ,*((LPSTR*)((LPVOID*)ParamStack)[1])
                ,*((unsigned long*)((LPVOID*)ParamStack)[2])
                ,((IInfo**)((LPVOID*)ParamStack)[3])
        );
    }
    return GXE_FAIL;
}

STDMETHODIMP
```

```
ConnManager::QueryInterface(REFIID riid, LPVOID *ppvObject)
{
    if(!ppvObject)
        return GXE_INVALID_ARG;
    *ppvObject=NULL;
    if(GXGUID_EQUAL(riid, IID_IUnknown))
        *ppvObject=(LPVOID)(IUnknown*)(IConnManager*)this;
    if(GXGUID_EQUAL(riid, IID_IGXObject))
        *ppvObject=(LPVOID)(IGXObject*)(IConnManager*)this;
    if(GXGUID_EQUAL(riid, IID_IGXModule))
        *ppvObject=(LPVOID)(IGXModule*)this;
    if(GXGUID_EQUAL(riid, IID_IConnManager))
        *ppvObject=(LPVOID)(IConnManager*)this;
    if(GXGUID_EQUAL(riid, IID_IGXObjectEvaluation))
        *ppvObject=(LPVOID)(IGXObjectEvaluation*)this;
    if(GXGUID_EQUAL(riid, IID_IGXCreateIntrospection))
        *ppvObject=(LPVOID)(IGXCreateIntrospection*)this;
    if(*ppvObject) {
        ((IUnknown*)*ppvObject)->AddRef();
        return NOERROR;
    }
    return E_NOINTERFACE;
}
STDMETHODIMP_(ULONG)
ConnManager::AddRef()
{
    GXUTIL_ADDREF();
}
STDMETHODIMP_(ULONG)
ConnManager::Release()
{
    GXUTIL_RELEASE();
}

STDMETHODIMP
ConnManager::GetThreadSession(DWORD sessionType, IGXSession2
**ppSession)
{
    LPSTR appName;
    DWORD dwFlags;
    if (sessionType & GXSESSION_LOCAL)
    {
```

```
        appName = m_appNameLocal;
        dwFlags = GXSESSION_LOCAL;
    }
    else if (sessionType & GXSESSION_CLUSTER)
    {
        appName = m_appNameCluster;
        dwFlags = GXSESSION_CLUSTER;
    }
    else
    {
        appName = m_appNameDistributed;
        dwFlags = GXSESSION_DISTRIB;
    }
    return GXContextGetThreadSession(m_pContext, dwFlags, appName,
ppSession);
}
```

Glossary

access module A grouping of interfaces that servlet and Enterprise JavaBean (EJB) application objects use to access extension objects. The interfaces within the access module have no implementation details.

accessor A reference to the manager class of an extension's service. A servlet, EJB or AppLogic calls an accessor in order to make use of a particular service in an extension. Netscape Extension Builder creates accessors automatically during code generation. A C++ accessor is a global function, whereas a Java accessor is a static method in an accessor class.

application A computer program that performs a task or service for a user. Also see **web application**.

AppLogic object A set of programming instructions that accomplish a well-defined, modular task within the application. AppLogic objects run on the Netscape Application Server and are managed and hosted by it. Typically, an application includes several to many AppLogics, which can be deployed across many servers. These AppLogics provide some or all of the procedural, or logic, portion of the application. Each AppLogic object is derived, directly or indirectly, from AppLogic class in the Netscape Application Server Foundation Class Library. AppLogic components are used if your application is written in C++ or if it will run in the NAS 2.x environment.

attribute Attributes are name-value pairs in a request object that can be set by servlets. Contrast with *parameter*. More generally, an attribute is a unit of metadata.

business logic The implementation rules determined by an application's requirements.

cache See **result cache**.

class A named set of methods and member variables that defines the characteristics of a particular type of object. The class defines what types of data and behavior are possible for this type of object.

client A computer or application that contacts and obtains data from a server on another computer. A client program is designed to work with one specific type of server. For example, a Web browser is a client application that contacts a Web server and requests Web pages. The server might be in the next room, across town, or on the other side of the world.

coclass The design-time representation of a Netscape Application Server extension class. A coclass becomes a class after the IDL files are compiled using the KIDL Compiler.

code generation The process of translating IDL files into a source code tree. In Netscape Extension Builder, the KIDL Compiler uses IDL files to generate Java or C++ source code. The KIDL Compiler is invoked by running the gmake utility from an extension's top-level build directory.

COM class An internal base class of Netscape Application Server's Foundation Class Library. The COM class implements a simple Java wrapper of native Component Object Model (COM) objects.

component A servlet, Enterprise JavaBean (EJB), or JavaServer Page (JSP).

constructor A method that instantiates a class.

container A process that executes and provides services for an EJB.

database A generic term for Relational Database Management System (RDBMS). A software package that enables the creation and manipulation of large amounts of related, organized data.

decoration A KIDL-specific value that is added to generic IDL to help the KIDL Compiler generate code.

Enterprise JavaBean (EJB) A business logic component for applications in a multitiered, distributed architecture. EJBs conform to the Java EJB standard specifications, which defines beans in terms of their expected roles. An EJB encapsulates one or more application tasks or application objects, including data structures and the methods that operate on them. Typically they also take parameters and send back return values. EJBs always work within the context of a container, which serves as a link between the EJBs and the server that hosts them. See also **container**, **session**, and **entity EJB**.

entity EJB An entity Enterprise JavaBean (EJB) relates to physical data, such as a row in a database. Entity beans are long-lived, because they are tied to persistent data. Entity beans are always transactional and multiuser aware. Also see **session**.

extension A long-lived set of libraries that reside on Netscape Application Server and provides one or more services to application logic also residing on the server. An extension provides a way to integrate legacy code or encapsulate functionality that needs to be called repeatedly from code or shared between different calling code modules.

.gxp file See **project file**.

.gxr file See **registration**.

HTTP A protocol for communicating hypertext documents across the Internet.

IDL (Interface Definition Language) A high-level, programming-language-neutral description of the services an object or interface will provide.

implementation code Code that specifies the behavior defined in an interface.

inheritance A technique in which a subclass automatically includes the method and variable definitions of its superclass. A programmer can change or add to the inherited characteristics of a subclass without affecting the superclass.

instance An object that is based on a particular class. Each instance of the class is a distinct object, with its own variable values and state. However, all instances of a class share the variable and method definitions specified in that class.

instantiation The process of allocating an object to memory at runtime.

interface A description of the services provided by an object. An interface defines a set of functions, called methods. The interface includes no implementation code. An interface, like a class, defines the characteristics of a particular type of object. However, unlike a class, an interface is always abstract. A class can be instantiated to form an object, but an interface cannot be instantiated.

introspection interface An interface whose methods are used to query attributes of virtual or physical objects. An introspection interface is used as part of object pooling.

Java Access Layer A set of Java classes and Java native methods that allows a Java interface to communicate with a C++ interface. For example, if you want a Java AppLogic to communicate with a C++ extension, create a Java Access Layer for the extension.

JavaServer Page (JSP) A text page written using a combination of HTML or XML tags, JSP tags, and Java code. JSPs combine the layout capabilities of a standard browser page with the power of a programming language.

KIDL Compiler The Netscape Extension Builder component that uses IDL as input and translates it into source code. The KIDL Compiler is invoked by running the gmake command from the top-level directory of the extension code. The KIDL Compiler generates a nearly complete source code tree. However, some output files will contain method stubs you must fill out before you finish building the extension.

load balancing A technique for distributing the user load evenly among multiple servers in a cluster.

make harness The set of makefiles within a specific source code tree. The make harness generated by Netscape Extension Builder Designer consists of a makefile in every directory and subdirectory of the tree.

metadata Information about a component, such as its name, and specifications for its behavior.

member A variable or method declared in a class is a member of that class.

member variable A variable with the following characteristics:

- The variable is declared inside a class declaration.
- A member variable specifies a piece of data that can be stored by an object instantiated from that class.

method A function with the following characteristics:

- The method is declared inside a class or interface.
- A method specifies an action that can be performed by an object instantiated from that class.

method locking A runtime feature of Netscape Extension Builder that facilitates legacy integration by ensuring thread safety.

method stub A method containing sample code that does nothing other than act as a placeholder. Developers replace method stubs with actual code specific to their needs.

module See **access module** or **service module**.

Netscape Application Server Foundation Class Library A set of interfaces and classes provided by Netscape Communications Corporation that can be used to develop object-oriented Netscape Application Server applications. The classes in the Netscape Application Server Foundation

Class Library define many types of objects you can include in Netscape Application Server applications, such as servlet and EJB objects, data connections, queries, and result sets.

Netscape Extension Builder A product that extends the functionality of the Netscape Application Server by integrating third-party solutions into the Netscape Application Server environment.

Netscape Extension Builder Designer A GUI tool provided by Netscape Extension Builder for designing an extension. In Netscape Extension Builder Designer, you define interfaces within interface modules, and you specify the implementation hints within service modules. These modules can then be stored in IDL files, which are generated into C++ or Java source code.

object A programmed entity with the following characteristics:

- An object embodies both data and behavior.
- Objects come into existence at runtime through the process of instantiation.
- Each object is based on a definition, which is called a class.

Many parts of a Netscape Application Server application, such as servlet and EJB objects, queries, and result sets, are objects.

object pooling A runtime feature of Netscape Extension Builder that improves performance by enabling extensions to share limited resources, such as connections.

object-oriented programming A method for writing programs using classes, not algorithms, as the fundamental building blocks. At runtime, the classes give rise to objects which perform the tasks of the application.

override To write new code that replaces the default code of an inherited method.

parameter Parameters are name-value pairs sent from the client, including form field data, HTTP header information, etc., and encapsulated in a request object. Contrast with attribute. More generally, an argument to a Java method.

persistent Refers to the creation and maintenance of a bean throughout the lifetime of the application. In NAS 4.0, beans are responsible for their own persistence, called *bean-managed persistence*. Opposite of *transient*.

pooling See **object pooling**.

presentation layout Creating and formatting page content.

presentation logic Activities that create a page in an application, including processing a request, generating content in response, and formatting the page for the client.

registration The process of informing Netscape Application Server of the existence of an servlet object, code module, extension, or security information. This information is stored in a registration file, which has a .gxr suffix.

request object An object that contains page and session data produced by a client, passed as an input parameter to a servlet or JavaServer Page (JSP).

result cache Storage in Netscape Application Server that holds the output from an servlet object so that the output can be accessed repeatedly without the necessity of running the servlet object again.

server A computer or software package that provides a specific kind of service to client software running on other computers. A server is designed to communicate with a specific type of client software.

service module A grouping of coclasses. In a service module, coclasses typically implement one or more of the interfaces defined in an access module. Client applications access the service module through the interfaces that the classes implement. The applications never access the service class types directly.

servlet An instance of the `Servlet` class. A servlet is a reusable application that runs on a server. In NAS, a servlet acts as the central dispatcher for each interaction in your application by performing presentation logic, invoking business logic, and invoking or performing presentation layout.

servlet engine An internal object that handles all servlet metafunctions. Collectively, a set of processes that provide services for a servlet, including instantiation and execution.

servlet runner Part of the servlet engine that invokes a servlet with a request object and a response object. See **servlet engine**.

session A continuous series of interactions between a user and a Netscape Application Server application. The term *session* is widely used to refer to a Web browser session, but in this manual, the term *session* refers more specifically to a series of user interactions that are tracked by a Netscape Application Server application. The user's session with a Web browser or other client software might start before the Netscape Application Server application begins tracking the user, and could continue after the application stops tracking the user.

session EJB A session Enterprise JavaBean (EJB) relates to a unit of work, such as a request for data. Session beans are short lived—the lifespan of the client request is the same as the lifespan of the session bean. Session beans can be stateless or stateful, and they can be transaction aware. See **stateful session EJB** and **stateless session EJB**. Also see **entity EJB**.

session load balancing See **sticky load balancing**.

stateful session EJB An Enterprise JavaBean (EJB) that represents a session with a particular client and which automatically maintains state across multiple client-invoked methods.

stateless session EJB An Enterprise JavaBean (EJB) that represents a stateless service. A stateless session bean is completely transient and encapsulates a temporary piece of business logic needed by a specific client for a limited time span.

state and session management A runtime feature of Netscape Extension Builder that enables an extension to cache an application's session or state.

sticky load balancing A technique that forces all client requests in a session to go to the same Netscape Application Server. In this way, the load is balanced by session, not by request. Sticky load balancing allows a session to store and retrieve complex objects within one process, instead of making distributed requests.

streaming

1. A runtime feature of Netscape Extension Builder that improves performance by allowing an extension to stream results. In other words, smaller portions of a result set are continuously returned, instead of waiting for the entire result set to be returned.

2. A technique for managing how data is communicated via HTTP. When results are streamed, the first portion of the data is available for use immediately. When results are not streamed, the whole result must be received before any part of it can be used. Streaming provides a way to allow large amounts of data to be returned in a more useful way, increasing the perceived performance of the application.

stub See **method stub**.

transaction A set of database commands that succeed or fail as a group. All the commands involved must succeed for the entire transaction to succeed.

transient A resource that is released when it is not being used. Opposite of *persistent*.

web application A computer program that uses the World Wide Web for connectivity and user interface (UI). A user connects to and runs a web application by using a web browser on any platform. The user interface of the application is the HTML pages displayed by the browser. The application itself runs on a web server and/or application server.

Index

A

accessing NAS services 89

access modules 33
  creating by importing 52
  creating in Netscape Extension Builder Designer 48
  IDL output location 69

accessors 90

application components 14

application model 14

applications, calling extensions from 90

C

C++ Class decoration 65

coclasses 57

compiling source code 77

component view 42

configuring extensions 182

ConnManager.cpp sample file 257

constants
  GXACL_ALLOWED 208
  GXACL_DONTKNOW 208
  GXACL_NOTALLOWED 208

Create a Lock decoration 58

CreateDataConn( ) 223

CreateDataConnSet( ) 227

CreateHierQuery( ) 228

CreateMailbox( ) 230

CreateObject( ) 155

CreateQuery( ) 231

CreateTrans( ) 233

D

decorations
  C++ Class 65
  Create a Lock 48, 51
  for coclasses 58
  for methods 63
  for object pooling 59, 63, 162
  Java Access Layer 48, 51, 58, 61
  Java Class 65
  Language 48
  Package Name 48
  Template Streaming 95

deploying extensions 179

DestroySession( ) 234

directory structure 70
  specifying in Netscape Extension Builder Designer 76

distributed state and session data 113

E

extension modules 32, 47

extensions
  benefits of building 16
  configuring 182
  defined 13
  deploying 179
  design-time components 30
  diagram of source code components 80
  method locking 26
  object pooling 27
  state and session management 27
  template streaming 26
  top-level directory 68

G

gen. directories 78, 79
Generate IDL command 76
generating source code 77
GetAppEvent( ) 235
GetHint( ) 162
GetStateTreeRoot( ) 111, 236
GetThreadSession( ) 112
global constants 49
GNUenv_nt.mak file 174
GXACL_ALLOWED 208
GXACL_DONTKNOW 208
GXACL_NOTALLOWED 208
GXContextCreateDataConnSet( ) 196
GXContextCreateHierQuery( ) 197
GXContextCreateMailbox( ) 199
GXContextCreateQuery( ) 200
GXContextCreateTrans( ) 201
GXContextDestroySession( ) 203
GXContextGetAppEvent( ) 204
GXContextIsAuthorized( ) 207
GXContextLoadHierQuery( ) 209
GXContextLoadQuery( ) 213
GXContextLog( ) 215
GXContextNewRequest( ) 216
GXContextNewRequestAsync( ) 218
GXGetStateTreeRoot( ) 111, 205

I

IDL files 69
  importing into Netscape Extension Builder Designer 52
IGXObjectEvaluation interface 136
IGXSession2 interface 112
IGXState2 interface 110, 112
IGXTemplateData interface 96, 97
IJavaObjectEvaluation interface 136
importing files 52
InitObject( ) 159
interfaces
  designing in Netscape Extension Builder Designer 53
  implementing in Netscape Extension Builder Designer 59
  locating in source code tree 85
introspection 135
Introspection Interface decoration 59, 145, 163
IsAuthorized( ) 237
ISession2 interface 112
IState2 interface 110, 112
ITemplateData interface 96, 99

J

Java Access Layer 73
  decoration 48, 51, 58, 61
Java Class decoration 65
Java native methods 74

K

Keep Pooled Object decoration 64, 145, 164
keywords, reserved 255
KIDL 29
KIDL Compiler 25, 77

L

LoadHierQuery( ) 239
LoadQuery( ) 242
local state and session data 113
Log( ) 244

M

make harness 27, 70
  configuring on Windows NT 174

details of program execution 175
editing makefiles 168
Manager Class decoration 58
MatchObject( ) 152
menu bar 42
method locking 26
Create a Lock decoration 58
methods
decorations for 63
defining in Netscape Extension Builder Designer 54
implementing in Netscape Extension Builder Designer 62
method stubs 71
completing implementation code 87
extending the class definition 88
implementing template streaming 96
locating in source code tree 84

N

NAS_ROOTDIR 12, 71
NAS services 89
native methods 74
NEB
See Netscape Extension Builder
NEB_ROOTDIR 12, 71
NEB Designer 24
Netscape's Interface Definition Language
see KIDL
Netscape Application Server registration file 69
Netscape Extension Builder 23
Netscape Extension Builder Designer 40
creating modules in 47
designing coclasses 57
designing interfaces 53
generating files 76
GUI overview 40
output from 68
procedural overview 43
starting the program 46
Netscape Extension Builder Runtime Layer 72
Netscape extensions
See extensions
Netscape project file 69
NewRequest( ) 245
NewRequestAsync( ) 248

O

Object Pool Config decoration 144
object pooling 27
adding decorations 140
completing method stubs 150
configuration 184
decorations 63
process flowchart 128
sample implementation file 257
Object Pool Manager 128
internal processes 137
Object Pool Name decoration 143

P

parameters
defining in Netscape Extension Builder Designer 55
physical objects 127
Poolable Object decoration 59, 145, 163
Pooled In Parameter decoration 63, 147, 163, 164
Pooled No Reuse decoration 64, 146, 164
Pooled Object Creation decoration 63, 141, 163
Pooled Object Name decoration 143
PoolSampleExt extension example 130

R

ReleaseObject( ) 158
reserved words in Netscape Extension Builder 255
runtime features 25

S

service modules 35
  creating by importing 52
  creating in Netscape Extension Builder Designer 50
  IDL output location 69
servlet 91
source code
  generating with gmake 77
  output from gmake 71
state and session management 115
state tree 111
StealObject( ) 157

T

templates, creating 102
template streaming 26
  decoration 58
  implementing interfaces for 96
tool bar 43
top-level directory 68, 77
tree view 41

U

UninitObject( ) 161
Uses Pooled Object decoration 63, 147, 163, 164
Use Synchronization Lock decoration 63

V

value-added features
  See runtime features
virtual objects 127

W

working directories 78, 79