Programmer's Guide (C++)

iPlanet™ Application Server

Version 6.5

806-4793-01
February 2002

Contents

Preface

This manual is intended for application developers who will be programming server-side or client-side code, or both. It provides conceptual sections that will prove useful to anyone working with iPlanet Application Server. It contains an overview of useful iPlanet Application Server concepts, application design principles, and detailed information about developing iPlanet Application Server applications in C++.

This preface describes the iAS documentation set and illustrates what you can expect to find in this Programmer's Guide.

This preface contains the following sections:

- Using the Documentation
- About This Guide
- What You Should Already Know
- How This Guide Is Organized
- Naming Conventions
- Documentation Conventions
- Related Information

Using the Documentation

The following table lists the tasks and concepts that are described in the iPlanet Application Server manuals and *Release Notes*. If you are trying to accomplish a specific task or learn more about a specific concept, refer to the appropriate manual.

Note that the printed manuals are also available online in PDF and HTML format, at: http://docs.iplanet.com/docs/manuals/ias.html

For information about	See the following	Shipped with
Late-breaking information about the software and the documentation	*Release Notes*	Available on the Web, at http://docs.iplanet.com
Installing iPlanet Application Server and its various components (Web Connector plug-in, iPlanet Application Server Administrator), and configuring the sample applications	*Installation Guide*	iPlanet Application Server 6.5
Creating iPlanet Application Server 6.5 applications that follow the open Java standards model (Servlets, EJBs, JSPs, and JDBC), by performing the following tasks: • Creating the presentation and execution layers of an application • Placing discrete pieces of business logic and entities into Enterprise Java Bean (EJB) components • Using JDBC to communicate with databases • Using iterative testing, debugging, and application fine-tuning procedures to generate applications that execute correctly and quickly	*Developer's Guide*	iPlanet Application Server 6.5

For information about	See the following	Shipped with
Administering one or more application servers using iPlanet Application Server Administrator Tool to perform the following tasks: • Monitoring and logging server activity • Implementing security for iPlanet Application Server • Enabling high availability of server resources • Configuring web-connector plugin • Administering database connectivity • Administering transactions • Configuring multiple servers • Administering multiple-server applications • Load balancing servers • Managing distributed data synchronization • Setting up iPlanet Application Server for development	*Administrator's Guide*	iPlanet Application Server 6.5
Migrating your applications to the new iPlanet Application Server 6.5 programming model from the Netscape Application Server version 2.1, including a sample migration of an Online Bank application provided with iPlanet Application Server	*Migration Guide*	iPlanet Application Server 6.5
Using the public classes and interfaces, and their methods in the iPlanet Application Server class library to write Java applications	*Server Foundation Class Reference (Java)*	iPlanet Application Server 6.5
Using the public classes and interfaces, and their methods in the iPlanet Application Server class library to write C++ applications	*Server Foundation Class Reference (C++)*	Order separately

About This Guide

This guide describes how to create applications intended to run on iPlanet Application Server.

This guide is intended for information technology developers in the corporate enterprise who want to extend client-server applications to a broader audience through the World Wide Web. In addition to describing programming concepts and tasks, this guide offers sample code, implementation tips, and reference material that includes a glossary.

What You Should Already Know

This guide assumes you are familiar with the following topics:

- the Java 2 Platform, Enterprise Edition (J2EE) specification
- the Internet and World Wide Web
- Hypertext Markup Language (HTML)
- C++ programming
- C++ APIs as defined in specifications for Enterprise JavaBeans, JavaServer Pages, and JDBC
- structured database query languages such as SQL
- relational database concepts
- software development processes, including debugging and source code control

How This Guide Is Organized

This guide is organized into twelve chapters and an appendix.

- Chapter 1, “Introduction to Applications”
- Chapter 2, “Designing Applications”
- Chapter 3, “Application Development Techniques”
- Chapter 4, “Writing Server-Side Appication Code”
- Chapter 5, “Working with Databases”

- Chapter 6, “Querying a Database”
- Chapter 7, “Working with Templates”
- Chapter 8, “Managing Session and State Information”
- Chapter 9, “Writing Secure Applications”
- Chapter 10, “Integrating Applications with Email”
- Chapter 11, “Running and Debugging Applications”
- Chapter 12, “Sample Code Walkthrough”
- Chapter 13, “Writing Secure Applications”
- Appendix A, “Implementation Tips”

Finally, a Glossary and Index are provided.

Naming Conventions

Item	Convention
Class name	“GX” prefix, followed by mixed case with initial uppercase. For example, GXTemplateMapBasic class.
Interface name	“IGX” prefix, followed by mixed case with initial uppercase. For example, IGXPreparedQuery.
Method name	Mixed case with initial uppercase. For example, GetLogin( ).
Parameters	Mixed case with initial lowercase. For example, myQuery.
Variables	Mixed case with initial lowercase. For example, myVar.

Documentation Conventions

File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except that slashes are used instead of backslashes to separate directories.

This guide uses URLs of the form:

```
http://server.domain/path/file.html
```

In these URLs, *server* is the name of server on which you run your application; domain is your Internet domain name; path is the directory structure on the server; and file is an individual filename. Italic items in URLs are placeholders.

This guide uses the following font conventions:

- The monospace font is used for sample code and code listings, API and language elements (such as function names and class names), file names, pathnames, directory names, and HTML tags.
- *Italic* type is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

Related Information

Specifications related to the iAS programming model are provided in the directory installdir/ias/docs/, where installdir refers to the directory in which you installed iAS. You can find a directory of all iAS-related documentation at installdir/ias/docs/index.htm.

Chapter 1

Introduction to Applications

This chapter provides an overview of iPlanet Application Server applications.

The following topics are included in this chapter:

- About iPlanet Application Server Applications
- Introduction to the iPlanet Application Server Foundation Class Library
- Introduction to Interfaces and COM

About iPlanet Application Server Applications

The cross-platform and universal access nature of the Internet and Intranet is enabling corporations to deliver online services to large numbers of users with ease and speed. Some examples of business services include travel, banking, online shopping, and stock trading services. iPlanet Application Server makes it possible for corporations to build, deploy, and manage applications that drive these on-demand, transaction-based services.

A key distinguishing feature between traditional client/server applications and iPlanet Application Server applications is the two-tier versus multi-tier model. In the two-tier client/server model, complex business and presentation code reside on the client. The client code needs to be configured and maintained on individual clients that often run different operating systems.

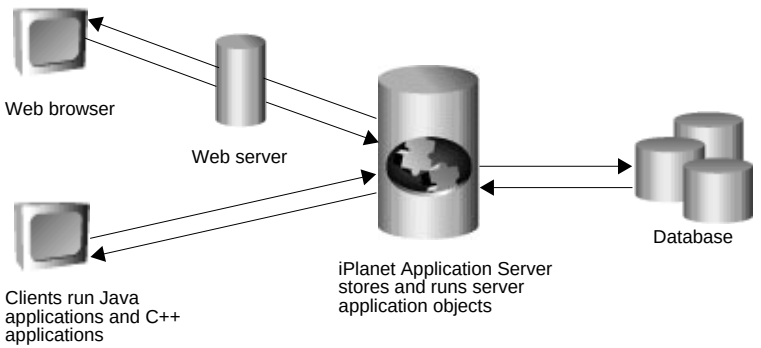
In the multi-tier model, iPlanet Application Server is the middle tier. Business code is stored and processed on iPlanet Application Server rather than on clients. An application is deployed and managed in a single location, and is accessible to large numbers of heterogeneous clients.

The iPlanet Application Server software is shipped with several sample applications which you can look at to get a more detailed overview. For example, the Online Bank sample application provides online banking and customer management. Customers using Web browsers can log in to Online Bank over the Internet, view account information, and move funds between accounts. Bank employees can use Online Bank to display information about customers, add new customers, and update or delete existing customer data.

Applications as Part of a Three-Tiered Environment

iPlanet Application Server applications run in a distributed, three-tiered environment. This means that an iPlanet Application Server system might consist of several computers running multiple copies of iPlanet Application Server software, along with multiple database servers and Web servers. Your application code can be distributed among the iPlanet Application Servers and on client machines. Overall, the machines and software involved are divided into three layers, or *tiers*.

The following illustration shows iPlanet Application Server application code in the three-tiered environment:



Client Tier Code

The first tier is the client, or user interface, tier. End users interact with client software in order to use the application. The software providing the user interface can be either a Web browser displaying HTML pages or a client application installed on desktop PCs.

Middle Tier Code

The middle tier is iPlanet Application Server itself. This tier consists of server machines running both iPlanet Application Server software and your server-side application code. You write this portion of the application code using the iPlanet Application Server Foundation Class Library.

The iPlanet Application Server handles requests from clients by running the appropriate application code, then returns the results to the clients. The means of communication between the client tier and iPlanet Application Server is the Internet or an Intranet. In the case of HTML clients, a Web server also stands between the client and iPlanet Application Server. The Web server passes requests and responses back and forth between HTML clients and iPlanet Application Server.

Database Tier Code

The third tier is the database tier. This tier consists of one or more database servers, which can be from different vendors. The database tier stores the information that forms the basis of the application. For example, in the Online Bank sample application, a database is used to keep track of customer information, account information, and transactions.

The code in the database tier can be created in a variety of database applications, and is therefore outside the scope of this documentation. You will need to write code in the server tier, using the iPlanet Application Server Foundation Class Library, to interact with the database tier. For more information, see “Working with Databases” on page 97.

Example Three-Tiered Application

As mentioned earlier, the Online Bank sample application, which is shipped with iPlanet Application Server, provides online banking and customer management. Online Bank is a good example of a three-tiered iPlanet Application Server application. The following discussion describes a typical user session with this application.

The User Logs In

If the client tier strategy used in Online Bank consists of HTML pages displayed in Web browsers, a bank customer can log in to the application from home, or wherever they have access to the Web. To begin working with the application, the user opens a Web browser and requests URL of the first HTML page in the application, the login page.

The Server Responds By Accessing The Database

After typing a user ID and password, the user clicks the Login button. This causes an action request to be sent to iPlanet Application Server (if the clients are Web browsers, the request is first sent to the Web server, which forwards it to iPlanet Application Server). The iPlanet Application Server handles requests by running the appropriate application code.

In this case, the code needed is that which logs in the user. In order to validate the user's password, the application accesses the database tier, where information about users is stored.

The Server Returns Results

After looking up the user's ID and password in the database, the application can validate the user and send a response back to the client tier. In a Web browser application, the server sends an HTML page containing the main menu of the application back to the user's Web browser (assuming the user was validated successfully).

The User Continues Using the Application

The user might next ask for the current balance in their checking account, which the iPlanet Application Server handles by querying the database, formatting the data into a report, and sending the report back to the user. This cycle of request, query, and response is repeated many times as the customer makes other selections and navigates through the application. Requests come in from the client tier and are processed in the server tier, often by accessing the database tier. Responses are then sent back from the server to the client.

Introduction to the iPlanet Application Server Foundation Class Library

A library is a set of predefined interfaces and class declarations that can be used in object-oriented programs. The iPlanet Application Server Foundation Class Library is designed for building server-side code as part of iPlanet Application Server applications. The class library is stored in the iPlanet Application Server installation directory, and is copied to your disk when you install iPlanet Application Server.

The classes and interfaces in the iPlanet Application Server Foundation Class Library define many types of objects you can include in iPlanet Application Server applications. Each object provides a specific type of functionality that is commonly needed. The following list shows some of the major types of objects and functionality you can include in your application by using the iPlanet Application Server Foundation Class Library:

- AppLogic objects
- Data connections
- Queries and other database commands
- Dynamic reports
- Electronic mailboxes
- User sessions and session-related data
- Security

This is just a partial list. For complete information about the iPlanet Application Server Foundation Class Library, see the iPlanet Application Server Foundation Class Reference.

Introduction to Interfaces and COM

The iPlanet Application Server programming API is based on interfaces and the Component Object Model (COM). This section provides an overview of the concepts involved in such a programming model. You do not need to read this section if you are already familiar with these concepts.

What Is COM?

The Component Object Model (COM) is a specification that provides a standard way for objects and their clients to interact. COM specifies only how objects interact, not how applications are structured internally or how they are implemented.

The fundamental mechanism which COM defines for this purpose is called an *interface*. An interface is a description of the services provided by an object. Clients wishing to use a COM object need only know what interface the object supports.

For more information about interfaces, see “What Is an Interface?” on page 23.

Benefits of COM

The purpose of COM is to promote software interoperability. Applications are made up of objects that can easily cooperate with one another, even when the objects are:

- Written in different programming languages
- Running on different machines under different operating systems
- Used by different client applications.

Unlike other object-oriented programming techniques, the COM standard does not depend on what type of client is attempting to use a particular software object, nor does it depend on which programming language is used to implement the object. COM provides a standard framework through which all software objects can interact.

COM provides a more productive way to design, implement, distribute, and reuse software. By reusing components, developers are free to spend more time on the functionality that is specific to their business situation.

How to Use COM

You use the Component Object Model to create reusable software components. These components must adhere to the COM standard and use interfaces to define their expected behavior to clients, but you can implement the internal behavior of components in any way you wish.

At the minimum, every COM object provides two basic operations:

- Find out which interfaces an object supports by calling the QueryInterface( ) method. For more information, see “Getting Information About Interfaces” on page 43.
- Control the object's lifetime by calling the reference counting methods AddRef( ) and Release( ). For more information, see “Reference Counting” on page 45.

All three of these methods are defined in the IUnknown interface, which is the base interface from which all COM interfaces inherit.

What Is an Interface?

The objects in iPlanet Application Server applications interact through *interfaces*. An interface is a description of the services provided by an object. An interface is like a contract between an object and its user (the code that wishes to interact with it). The contract describes a set of expected behavior. Code that wishes to use an object need only know what interface the object supports, and does not need to know anything about the internal implementation of the object.

An interface defines a set of functions, called methods. The interface includes no implementation code. It only describes the parameters and return types of its methods. The code for the methods is written separately, in a class, which is said to *implement* the interface. Typically, the interfaces and their implementations are written by different groups of people.

How is an interface different from a class? A class is a set of data and functions (member variables and methods) that define the characteristics of one type of object. An object is an instantiation of a class. 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 can not be instantiated, because it has no implementation code to determine what to do when each method is called.

Every interface has a name that serves as an identifier you can refer to in code. By convention, the name of each interface begins with a capital I, such as IGXQuery.

Benefits of Using Interfaces

Interfaces provide a level of abstraction that enables objects to interoperate more easily. The code which wants to use the object needs to know how to connect to the object and call its methods. The object needs to expose its services to any code that wishes to connect. The interface provides the connection that allows the code to access the object, and allows the object to expose its services.

When using an interface-based programming model, you can modify the internal implementation of an object at will. As long as it continues to implement the same interface, the code that uses that object does not require any rewriting or recompilation.

Chapter 2

Designing Applications

This chapter describes the various parts of the iPlanet Application Server application, as well as important elements of the development process, including choosing a programming language.

The following topics are included in this chapter:

- Parts of an iPlanet Application Server Application
- Questions to Ask Before You Start
- Designing the Components of the Application

Parts of an iPlanet Application Server Application

An iPlanet Application Server application is made up of several components, which fall into three categories:

- Database tier
- Client tier
- Server tier

Database Tier

The foundation of any iPlanet Application Server application is the database. It is vital that the database portion of the application be designed so that it lends itself to working with the application code.

Client Tier

Assuming the application uses HTML as its client strategy, then the user interface is made up of HTML Pages, which are stored in the server tier.Server Tier

The following application components reside in the server tier. Not all of these components are used in every application.

- AppLogic Objects
- HTML Pages
- Templates
- Query Files
- Other Code

The rest of this section gives a brief description of the client- and server-tier application components. The database tier is outside the scope of this documentation.

AppLogic Objects

An AppLogic object is a set of programming instructions that accomplish a well-defined, modular task within the application. AppLogic objects run on the iPlanet 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 the `GXAppLogic` class in the iPlanet Application Server Foundation Class Library. AppLogics can be written in Java or C++. In this manual, it is assumed that you are using C++.

For more information about AppLogics, see Chapter 4, “Writing Server-Side Application Code

HTML Pages

HTML pages are used to provide the user interface for Web-browser based applications. An HTML page might include the following items:

- Input fields for the user to provide information.
- A Submit button for the user to request action.
- An AppLogic call (not visible to the user) that tells iPlanet Application Server which AppLogic object to run to perform the requested action when the user clicks a Submit button.

- If the page was generated dynamically using an HTML template, it will also contain live data.

Templates

A template is a text file that can be merged with dynamic data to produce formatted output. Templates include special GX markup tags, which specify how to merge the data with the page.

For more information, see , “Working with Templates.”

Query Files

A query file is a file with a .gxq extension that contains the specifications for one or more database commands, such as queries or any other SQL command. Query files are generated automatically when you use the Query Designer to build queries. You can also write query files yourself using any text editor.

For more information, see “Working with Query Files” on page 178 of , “Working with Query Files.”

Other Code

In addition to AppLogic objects, your application can include other code to perform various tasks. For example, you might have a file that contains helper functions such as data conversions that are specific to your application. You can also subclass from classes in the iPlanet Application Server Foundation Class Library if you want to modify or add to the default behavior provided.

Questions to Ask Before You Start

During the planning phase of the project, it is advisable to ask the following questions and make the following decisions. Make sure everyone concerned with the project is in agreement about the decisions that have been made.

- What is the purpose of the application? What result is desired when people start using it?
- How many users are expected?.

- Will the users be anonymous, or closely tracked and secured? A website that gives travel tips probably has thousands of anonymous users. In contrast, an online banking application probably requires users to log in to a secured session before accessing certain parts of the application, such as the screen that is used to transfer funds between accounts.
- Will you use iPlanet Application Server sessions to track the users? Which types of sessions, secure or non-secured? The answers to these questions affect how you use sessions and manage users. Even if security is not an issue, you might use sessions for other purposes, such as to record a user's preferences as they move through the application. If you want to require the user to log in, sessions provide the best security mechanism.
- If using secured sessions, you must decide which AppLogic objects in the application are accessible to all users, and which are accessible only to users who have logged in with a password. A single application is likely to contain both secured and unsecured AppLogics. For example, the login screen is, of necessity, available to any user, as are the AppLogics that display it and handle the user's login request.
- Is there an existing application that will be replaced or augmented by the iPlanet Application Server application? Which features of that application are to be kept, and which discarded? Will the legacy code be incorporated into the application? These answers lead into decisions about whether to code the application in C++ or use Java native calls.
- Are there any particular features that are desired in the application in order to showcase a company's products or promote a certain development approach? These features are in addition to those that are dictated by the practical purpose of the application.
- What strategy will you use to keep track of the parts of the application? A pictorial flowchart is useful for visualizing the HTML templates, AppLogics, and other components of an application. Once development is underway, a good source control system is advisable if the application is of any size or if several developers are involved.
- Lastly, and most importantly: What components, such as AppLogics and templates, are needed in the application? Considerations such as reusability and time available for development should be weighed when answering this question. These considerations are addressed in detail in "Designing the Components of the Application" on page 29.

Designing the Components of the Application

This section describes some of the considerations and techniques you can use when deciding how to design the various components in an application.

User Interface Design

It is advisable to design the user interface of the application first. If your development team includes interface (UI) specialists, be sure to include them early in the design process. Changes they make to the screens can have major effects on how the application must be designed. For example, the use of frames in a Web browser can have a profound effect on the business logic layout of an application.

HTML Interface Design

In an HTML-based application, you need to inform the UI designer of which GX tags and database fields will be available for them to use in the HTML pages and templates. GX tags are placeholders for the dynamic data that is merged with the templates at runtime to create live HTML pages. These are the basics of communication between the code and the templates. As long as the database fields and GX tags match what is expected in the code, the UI expert can design the appearance and other features of the UI with a free hand.

Impact of Caching on the User Interface

The capability of iPlanet Application Server to cache AppLogic results can dramatically improve performance. A cache is a fast-access area in the computer's memory. The first time an AppLogic runs, it can store its results in the cache. When the iPlanet Application Server receives additional requests for the same AppLogic, instead of running the AppLogic again, the iPlanet Application Server returns the results directly from the cache.

Each AppLogic has one result cache. The cache can contain multiple results, which are produced by running the AppLogic with different input parameter values. When you set up caching in code, you specify which input parameters are significant in deciding when to cache a result. For example, you might want to cache sales reports for certain products and not others, so you would use the input parameter containing the product code to control caching.

Plan on caching early in the design process, because its use affects how you design the client side of the application. All the criteria needed for caching must be present in the input parameters of the AppLogic request. For example, when the clients are Web browsers, this means the caching criteria must be present as fields on an HTML form, or as arguments in the URL that calls the AppLogic.

For more information, see “Caching AppLogic Results to Improve Performance,” on page 86 of Chapter 4, “Writing Server-Side Application Code

Combining or Splitting Application Components

When designing your application, you will find that there are certain approaches that can help you find the optimum way to divide up the functionality. You will need to make some decisions about the size and number of AppLogics, templates, and other components in the application.

The application will fall somewhere on a continuum from a very large number of components, each of which performs an extremely limited task, to a very small number of components, each of which can perform many tasks. Where exactly the application falls in this spectrum depends on your development needs and goals.

The following considerations affect your decision to combine or split application components:

- Redundancy
- Reusability
- Caching
- Performance

Redundancy

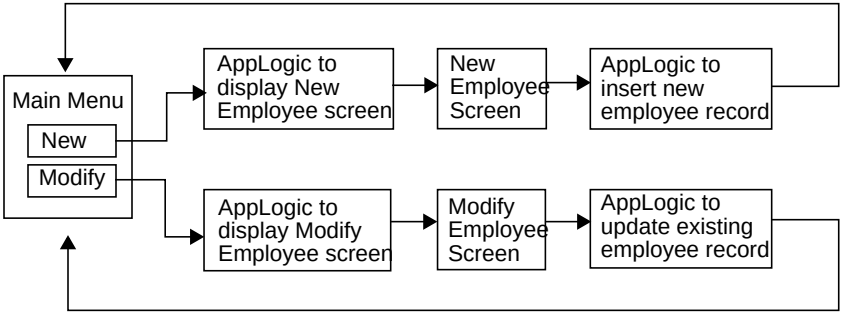
If you use a flowchart to represent a user’s navigation through an application, you will find it helpful in making decisions about combining functionality. Flowcharts are especially useful for spotting redundant code or screens.

For example, if the flowchart shows two screens that are almost identical in an HTML-based application, it may be advisable to combine the functionality. You could write one AppLogic object that uses different templates depending on runtime conditions. You could also write one HTML template which can vary its output. The template could be populated with different data, or could contain conditional portions of HTML.

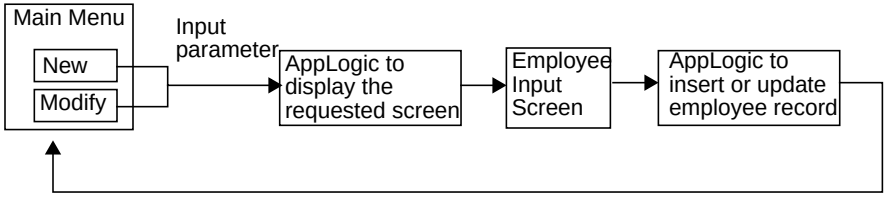
Similarly, a flowchart can help you spot redundant AppLogics. Deciding when to group functionality into one AppLogic, and when to split one AppLogic into several, is an important design decision. By grouping similar functionality into one AppLogic that takes slightly different actions, perhaps depending on an input parameter, you reduce the amount of coding and development time. By using fewer AppLogics, you produce a simpler design and a more sleek deployment profile.

For example, consider an application that allows the user to work with employee records. The user can either add a new employee or modify the data that is already stored for an existing employee. The forms displayed to the user for both operations are similar, with fields for the employee name, address, and so on. The only difference between the two screens is that in the case of modifying data, the fields are already filled in with the existing data, and the screen prompts and action button are labeled Modify instead of Add.

If you drew a flowchart of this application, as shown in the following illustration, it would be apparent that the HTML templates and AppLogics required by these two operations are redundant:



More efficiently, you could write a single AppLogic to be executed in response to the main menu, and a single HTML template using conditional GX markup tags to produce slightly different screens. The AppLogic would take an input parameter to indicate whether the user wanted to display the Add Employee screen or the Modify Employee screen. This new design would look like the following illustration:



Alternatively, if updating a record and inserting a new record involve radically different business logic, you might keep separate AppLogics for inserting and updating records. A new employee is likely to require more business rules or processes. However, you could still combine the screen display AppLogic and the HTML templates.

Reusability

If a particular task is performed repeatedly and is called from several parts of the application, it is preferable to divide this code into a separate, reusable component.

For example, suppose your application contains a Main Menu screen which is displayed after each user-selected operation is complete. Rather than write code to display the Main Menu at the end of every AppLogic in the application, it is preferable to create a Main Menu AppLogic that performs only the single task of displaying this screen. The other AppLogics can then call the Main Menu AppLogic whenever it is necessary to display the Main Menu.

You can also reuse a component in several applications. If you consider this when designing the first application in a suite, it will make the task of developing the subsequent applications easier. For example, you might want to create separate HTML templates that contain standard look-and-feel elements of the user interface.

The HTML template SuccessMessage.html in the Online Bank sample application is an example of a reusable component. This template is called throughout the application code whenever it is necessary to display an informational message to the user.

Caching

For the purposes of caching AppLogic results, it is best to keep functionality in separate AppLogics. If you group too much functionality into too few AppLogics, you might encounter more difficulty in caching results. Because caching results can greatly increase performance, it is important to consider caching during the application design phase.

When an AppLogic performs one task and returns one type of result, it is relatively simple to set up the caching criteria. If the AppLogic is capable of returning multiple types of results, the caching criteria can become complex. It is likely your cache criteria will require more input parameters to the AppLogic, and it may be difficult to predict exactly how caching will occur.

For reasons of resource usage, it is advisable not to keep too much data in a single AppLogic cache. Therefore, caching five or six types of results in one cache for a multi-purpose AppLogic might have a negative impact. In addition, the iPlanet Application Server keeps a least-recently-used list of the cached items, and if too many items are cached, some might be removed from the end of the list.

For more information, see Chapter 4, “Writing Server-Side Application Code of Chapter 4, “Writing Server-Side Application Code

Deployment and Partitioning

Deployment and partitioning considerations affect the desired granularity of code. For example, if the task of displaying the Main Menu is performed from many parts of your code, you might want to put that functionality into a single AppLogic, not only for reusability, but also so that you can distribute the code to every machine in the installation. In this way, the code will have maximum availability and the performance of the application will be improved.

In the Online Bank sample application, the AppLogic objects OBLogin, ShowMenuPage, and OBLogout are good candidates for widespread distribution of this kind.

Performance

Increasing the number of AppLogics does not have a significant negative effect on application performance. The iPlanet Application Server can quickly find the correct AppLogic for any given request, even if there are many AppLogics in the application. As noted earlier, increasing the number of AppLogics can actually improve performance by making the code more readily available and improving the ease of caching.

At a Glance

The following table summarizes the reasons for combining functionality into fewer components or splitting it into more components:

Goal	Action
Reduce redundant code or templates	Combine
Make best use of limited development time	Combine
Reuse components within application or throughout suite of applications	Split
Maximize granularity for caching AppLogic results	Split
Increase availability of components when application is deployed	Split
Optimize performance	Split

Designing an AppLogic Base Class

Instead of collapsing functionality into fewer AppLogics, another strategy to reduce redundant coding is to create a custom AppLogic base class, derived from the GXAppLogic class in the iPlanet Application Server Foundation Class Library. The custom base class contains helper methods to perform the routine tasks that are specific to your application. Derive all the other AppLogics in the application from this base class, rather than directly from the GXAppLogic class. This will give you a head start and reduce repetitive coding.

The following list shows a few of the tasks you might perform in a custom AppLogic base class:

- Check input parameters.

  For more information, see “Validating Input to AppLogic Objects,” on page 265 of Chapter 9, “Writing Secure Applications

- Provide standard, application-specific data conversion helper functions.
- Verify the session ID.

  For more information, see Chapter 8, “Managing Session and State Information

- Set up part of the template map.

  For more information, see “Using a Template Map” on page 207 of , “Working with Templates.”

- Produce common presentation elements, such as corporate banners.

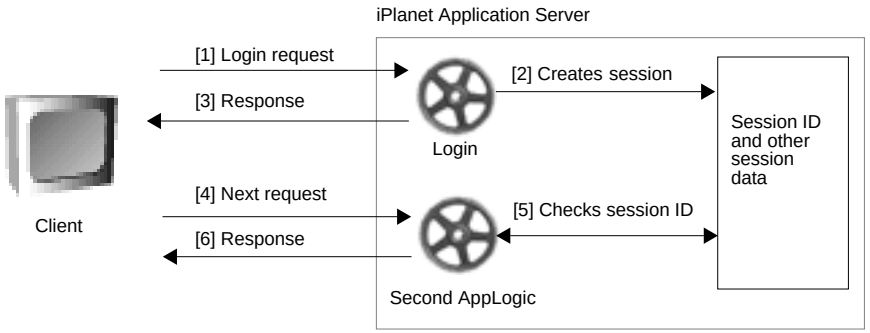
- Provide debugging helper functions, such as to log the current date and time and the last executed AppLogic.

The Online Bank sample application uses a customized AppLogic base class called OBBaseAppLogic. This class contains methods to create a user session, initiate contact with the database, handle certain types of errors, and show a success message to the user. For more information, see “The Online Bank Base AppLogic,” on page 297 of Chapter 12, “Sample Code Walkthrough

Designing a Login AppLogic Object

A typical way to design a secured application is to write one AppLogic object that is the main entry point to the application. This AppLogic responds to the user’s first request. The AppLogic gathers and authenticates the user’s login ID and password. If the user is authorized to run the application, the login AppLogic creates a session and assigns a session ID to the user’s session with the application.

The user then continues to make requests and run other AppLogics in the application. Each subsequent AppLogic checks for the session ID and user’s authorization level before proceeding with its main task, as shown in the following illustration:



The Online Bank sample application includes a login AppLogic, OBLogin, which checks the user’s password, looks the user up in a database, creates a user session and stores information about the user in it, then displays the appropriate menu depending on the type of user.

Designing Local, Distributed, and Global AppLogic Objects

When your application is ready for production use, the system administrator who deploys your application can specify the extent to which each AppLogic object is distributed by assigning one of the following types to the AppLogic:

- A local AppLogic is an AppLogic instance that runs on a particular iPlanet Application Server machine and does not participate in load balancing.
- A distributed AppLogic can run on any iPlanet Application Server machines that you and the system administrator specify, such as the machines in a cluster at a single company office. Each iPlanet Application Server might store a copy of the same AppLogic code, and the choice of which server runs the AppLogic at any particular time is made by the load balancing module of iPlanet Application Server.
- A global AppLogic can run on any iPlanet Application Server machine in the system, as determined by the load balancing module.

When designing an application, consider which AppLogics should be assigned to each category. These categories do not affect how you write the AppLogic code, but you need to communicate with the system administrator about which categories to use when the application is deployed.

Application Development Techniques

This chapter describes your application development tools (code editor, compiler, debugger, and so on), as well as the iPlanet Application Builder and sample applications.

The following topics are included in this chapter:

- Your Development Environment
- Accessing Libraries
- Using Interfaces
- Instantiating Objects
- Declaring and Defining Methods
- Reference Counting
- Working with Data
- Exporting Classes
- Using Events
- Using Cookies

Your Development Environment

Your development environment includes the following components:

- Your application development tools (code editor, compiler, debugger, and so on).

- The iPlanet Application Builder and sample applications that you installed using the iPlanet Application Server installation procedure, described on the product CD.

If you are using iPlanet Application Builder, the application files are automatically placed in appropriate directories. If you are not using the iPlanet Application Builder, you should create a separate directory (<Code>$APP_ROOTDIR) to contain the files that belong to each application or project. The following table shows the suggested locations for these directories:

Component	Unix	Windows NT
Source files (.cpp)	<Code>$APP_ROOTDIR	<Code>$APP_ROOTDIR
header files (.h)	<Code>$APP_ROOTDIR	<Code>$APP_ROOTDIR
makefile	<Code>$APP_ROOTDIR	<Code>$APP_ROOTDIR
html templates (.html)	<Code>$APP_ROOTDIR/templates	<Code>$APP_ROOTDIR\templates
html graphics (.gif, .jpeg)	<Code>$APP_ROOTDIR/images	<Code>$APP_ROOTDIR\images
registration file (.gxr)	<Code>$APP_ROOTDIR	<Code>$APP_ROOTDIR

Accessing Libraries

To use classes from a particular file in one of the libraries, include that file at the start of your AppLogic code. You include a file by using the `#include` statement. Any code module can contain one or more `#include` statements to gain access to code in other files.

When writing AppLogic objects, you will be working with the files in the iPlanet Application Server Foundation Class Library. These files are located in `$IAS/include`, where $IAS is the directory in which iPlanet Application Builder is installed. Typically, you will need at least the following files:

```
#include <gxutil.h>

#include <gxapplogic.h>

#include <gxidl.h>
```

Example

When you write an AppLogic object, you make it a subclass of the GXAppLogic class. This class is in the file gxapplogic.h. Before you can reference the GXAppLogic class to derive the subclass, you must include the GXAppLogic class's file. The following example code shows how to include the file gxapplogic.h:

```
#include <gxapplogic.h>

// . . .

class MyAppLogic : public GXAppLogic

   // . . .

}
```

Using Interfaces

You can use the interfaces in the iPlanet Application Server Foundation Class Library in either of the following ways:

- Create a reference to the interface, in order to interact with an object that supports that interface. For example, to interact with a query object, you need to reference the IGXQuery interface.
- Create a class that implements an interface, in order to write your own custom behavior for a particular type of object. For example, to implement a session ID object that creates session IDs in a way that is unique to your application, you implement the IGXSessionIDGen interface.

How to Reference Objects Through Interfaces

Once an interface is implemented in a class, and the class is instantiated into an object, calling code access the object through a pointer to the object's interface. This is the only way calling code can access an object: through its clearly-defined contract, the interface. When calling code accesses an object, it uses the interface pointer to access the methods in the interface. This use of interface pointers hides the internal implementation from calling code.

For example, the following code shows how to access an object through an interface pointer:

```
// Declare pointer variable
```

```
IGXQuery *pqry;

// Get pointer to an IGXQuery interface by calling
// CreateQuery(), which returns such a pointer
HRESULT hr = CreateQuery(&pqry);

// Use the pointer to call SetTables(), which is
// a method of the IGXQuery interface
pqry->SetTables("customers, orders");
```

How to Implement Interfaces

When you implement an interface, you write a class that contains code to perform the behavior that is defined in the interface. The class declaration statement is similar to subclassing except that in place of the superclass name, you put the interface name. For example:

```
class MySession : public IGXSession2 {
   // code to implement the interface
}
```

The class inherits from the interface it wishes to implement, declares whatever variables are necessary for maintaining the object state, and overrides all the member functions of the interface.

For example, the IGXSession2 interface in the iPlanet Application Server Foundation Class Library contains the SetSessionData( ) and SaveSession( ) methods. The interface defines the parameters and return types for these methods. You can create a class to implement IGXSession2 and write code to make these methods work in any way you choose. You can even create several different classes that implement the interface in very different ways. Calling code accesses the objects instantiated from these classes only through their interfaces, and know nothing else about the objects. Therefore, different implementations can be used interchangeably.

When implementing an interface, you must provide an implementation for every method in the interface. However, the implementation can be simply a `return` statement. When working with interfaces provided by iPlanet, you can implement some methods so that they simply call the iPlanet version of the same method. To do this, make an instance of the original interface and use it to call the original method versions.

For example, suppose you implement the IGXSession2 interface in order to perform some special processing during a single method. For the other methods in the interface, you would simply call the original method as shown in the sample code below.

The following code appears in the header file:

```
class AcmeSession : public IGXSession
{
// Pointer to IGXSession instance to delegate to
private:
   IGXSession2 *original;
public:
   AcmeSession(IGXSession2 *orig)
   {
      original = orig;
      if (original)
         original->AddRef();
   }
   ~AcmeSession()
   {
      if (original)
         original->Release();
   }
   // ...
   STDMETHOD(GetSessionData)    (IGXValList **ppSessionData);
   long GetShoppingCartItemCount();
   // ...
   // Additional code ...
```

```
   // ...
```

The following code appears in the source file:

```
// You are making no changes to this method
// so call the original version
STDMETHODIMP
AcmeSession::GetSessionData(IGXValList **ppSessionData)
{
    if (!original)
        return GXE_FAIL;
    return original->GetSessionData(ppSessionData);
}

// You are customizing this method
long AcmeSession::GetShoppingCartItemCount()
{
   long count = 0;
   IGXValList *data = NULL;

   if (GetSessionData(&data) == NOERROR &&
      data)
   {
      data->GetValInt("cart_item_count", &count);
      data->Release();
   }
   return count;
}
```

Getting Information About Interfaces

You can find out whether an object is capable of providing particular services by calling the QueryInterface( ) method, which is provided by all interfaces in the libraries. QueryInterface( ) tells whether a given object implements a given interface (and, therefore, provides the services you need). QueryInterface( ) takes as a parameter the unique identifier of the interface in which you are interested. If the object implements that interface, it returns a pointer to the interface. Your code then uses this pointer to interact with the object.

If QueryInterface( ) successfully obtains the interface pointer, it automatically calls AddRef( ) to increment the reference count on the object. Therefore, for every successful QueryInterface( ) call, your calling code must make a corresponding single Release( ) call, through the returned pointer, in order to match the implicit AddRef( ) call made by QueryInterface( ). Otherwise, a memory leak can occur.

Example

The following code shows how to query an interface:

```
HRESULT hr;

// Make sure the result set supports the IGXTemplateData

// interface (it always should)

IGXTemplateData *pTD=NULL;

if(((hr=pHRset->QueryInterface(IID_IGXTemplateData,
   (LPVOID *)&pTD))==GXE_SUCCESS)&&pTD) {

   // Everything is fine, so continue; release when done

   // ...

pTD->Release();

}
```

Instantiating Objects

Instantiation is the process of allocating an object to memory at runtime. An object is an instance of a class. The class defines the characteristics of a type of object. When an application runs, one or more objects can be instantiated, or created, from each class in the application. In an iPlanet Application Server application, most access to objects is accomplished through interfaces.

To instantiate an object

1. Declare a variable to refer to the object. For example:

   ```
   IGXSession2 *pSess=NULL;
   ```

   In this example, the variable `pSess` is declared using the IGXSession2 interface to specify that the variable will reference an object implemented from that interface. In step 2, you create the instance.

2. Instantiate the object by calling the appropriate method. For example, in the following code, the GetSession( ) method is used to retrieve an instance of a session object.

   ```
   HRESULT hr;

   hr=GetSession(0, OB_APPNAME, NULL, &pSess);
   ```

   Calling the special instantiation and object retrieval methods and functions in the iPlanet Application Server Foundation Class Library takes the place of the new keyword. These methods and functions, such as GetSession( ), perform extra tasks above and beyond what is accomplished with the new keyword. The new functionality is performed in the iPlanet Application Server method code and takes place automatically.

Declaring and Defining Methods

Methods often return HRESULT, a 32-bit result code that equals zero (NOERROR) for success or non-zero for error conditions. In your AppLogic, if you create a virtual method that returns HRESULT, you must use the following macros for cross-platform portability:

- STDMETHOD macro for the method declaration in the header file (.h).
- STDMETHODIMP macro for the method definition in the source code file (.cpp).

For example, in the OBLogin.h file in the Online Bank sample application, the following STDMETHOD command declares the OBLogin AppLogic's Execute( ) method:

```
class OBLogin : public OBBaseAppLogic
{
   // ...
   STDMETHOD(Execute) ();
}
```

In addition, in the source file OBLogin.cpp, the following STDMETHODIMP command defines the Execute( ) method:

```
STDMETHODIMP
OBLogin::Execute()
{
   // ...
}
```

Reference Counting

It is necessary to free objects when they are no longer in use, just as you must free memory. The code that is using an object is responsible for freeing the object when it is no longer needed. The mechanism that calling code uses to accomplish this is called *reference counting*.

You perform reference counting by using AddRef( ) and Release( ), which are provided by all interfaces in the libraries. The AddRef( ) and Release( ) methods are defined in the IGXObject interface, from which all the other interfaces inherit.

You can use these two methods to increment and decrement the count of references to any iPlanet Application Server object that your code uses. When you call AddRef( ), you are informing an object that you are using it. When you call Release( ), you are informing the object that you are finished using it. The object keeps track of how many other code modules are using it, and when the count drops to zero, the object deletes itself from memory.

Reference counting provides lifecycle control over objects to ensure proper housekeeping and avoid memory leaks. Each object contains an internal reference counter that tracks the number of other objects relying on it at runtime. When the reference count is decremented to zero (0), the object is deleted.

Unless otherwise noted, methods in the iPlanet Application Server Foundation Class Library that return objects automatically increment the reference count on these objects on behalf of your calling code. However, you must explicitly decrement the reference count in your AppLogic, using the object's Release( ) method, when a pointer to the instance is no longer needed.

For example, the following code shows how to release a query object. The CreateQuery( ) method returns a query object and performs one implicit AddRef( ). The caller is responsible for the matching Release( ) call for that implicit AddRef( ) call.

```
HRESULT hr;
IGXQuery *pQ=NULL;
if(((hr=CreateQuery(&pQ))==GXE_SUCCESS)&&pQ) {
   // Set up the query
   pQ->SetTables("OBAccount, OBTransaction,
      OBTransactionType");
   // Use the query ...
   pQ->Release();
}
```

In the following example, housekeeping for reference counters is performed in the destructor method.

```
MyAppLogic::~MyAppLogic() {
   if (m_pProps)
      m_pProps->Release();
   if (m_pConn)
      m_pConn->Release();
   if (m_pQuery1)
      m_pQuery1->Release();
   if (m_pHierQuery)
      m_pHierQuery->Release();
   GXDllLockDec(); // Update count of references to the
                   // AppLogic library
};
```

In more advanced applications, you may need to explicitly increment the reference count in your AppLogic using the object's AddRef( ) method. You might want to do this in order to ensure that the object is valid when using the object for a long time.

Even if you are extremely careful about reference counting, it is likely that some memory will not be released properly. Over time, such memory leaks will consume the available resources on your machine. Therefore, when using C++, it is advisable to use a tool capable of memory use analysis, such as Purify by Pure Atria.

Working with Data

When developing applications, you need to work with data of various types. This section describes the techniques provided to allow you to access and modify data. Manipulation of data will arise routinely throughout an application, such as when preparing data to pass into a function call, dealing with the return value from a function call, or dealing with the data returned by a database query.

iPlanet Application Server applications can include the usual data types such as integers and strings. The libraries also provide some special types of data, such as globally unique identifiers (GUIDs), data objects such as IGXValList objects, and mechanisms such as memory buffer management and spin locks.

Managing Memory Buffers

The iPlanet Application Server Foundation Class Library provides the IGXBuffer interface to manipulate memory blocks. The IGXBuffer interface represents a block of memory that multiple objects can share, allowing the multiple users of an IGXBuffer object to control the lifetime of the memory block by using reference counting.

Several methods return IGXBuffer objects, such as GetFields( ) in the IGXQuery interface. These methods automatically create the IGXBuffer object and return it. You can then use the methods in the IGXBuffer interface to access the data in the buffer after it is returned from the method call.

Other methods take IGXBuffer objects as parameters. Before calling such a method, you must create the buffer and place the appropriate data in it.

To create an IGXBuffer object

1. Call the function.
2. You must first specify the size of the memory block by calling Alloc( ) before calling any of the other methods in the IGXBuffer interface.

Examples

The first example calls a method that returns a buffer, and then uses the buffered data.

```
IGXBuffer *buff = NULL;

HRESULT hr = pQuery->GetFields(&buff);

if(hr==NOERROR && buff) {
```

```
   // ... work with buff, such as using GetAddress()
   // and GetSize()
   buff->Release();
}
```

The following code allocates a buffer object and uses it to temporarily store a string. It then passes the buffer as a parameter to the Put( ) method in the GXTemplateMapBasic class.

```
LPSTR pOutput=NULL;
pOutput=GXGetValListString(m_pValIn, "OUTPUTMESSAGE");


IGXBuffer *pBuffOutput=NULL;
if(pOutput) {
   pBuffOutput=GXCreateBufferFromString(pOutput);
   pTM->Put("OUTPUTMESSAGE", pBuffOutput);
   }
```

Using Spin Locks

Use spin locks to ensure synchronous access to shared resources such as a counter variable used by several threads. Use spin locks for only short processes consisting of just one or several brief operations. Extensive or careless use of spin locks (such as for longer processes like memory allocation or ODBC calls) can reduce AppLogic performance. For longer processes, use critical sections instead. For more information, see “Using Critical Sections” on page 50.

Using a Spin Lock for General Operations

To use a spin lock, call the following functions in the following order:

1. Call the GXSYNC_INIT( ) function to initialize a synchronization variable (of type GXSYNCVAR) to be used to synchronize access to shared resources, via a spin lock, in subsequent operations.
2. Call the GXSYNC_LOCK( ) function to acquire exclusive access to the shared resource(s) that the specified spin lock protects. While your code owns the spin lock, other code cannot acquire it.
3. Perform the brief process or operations.

4. Call the GXSYNC_UNLOCK( ) function to release a spin lock that was acquired in a preceding GXSYNC_LOCK( ) call. Releasing the spin lock allows other code to acquire it.

5. Call the GXSYNC_DESTROY( ) function to remove a spin lock that is no longer needed. Calling GXSYNC_DESTROY( ) releases the system resources allocated for the spin lock. Subsequent calls to the spin lock are invalid. To use the spin lock again, you must subsequently initialize the spin lock using GXSYNC_INIT( ).

Example

The following code shows how to use a spin lock to perform the simple operation of incrementing a counter. In the class MyClass, the following member variables are declared:

```
int counter;

GXSYNCVAR sync;
```

The following code appears in the constructor method:

```
GXSYNC_INIT(&sync);

counter = 0;
```

The following code appears in the destructor method:

```
GXSYNC_DESTROY(&sync);
```

The following code appears in a method in the class:

```
void MyClass::method() {
   // ...
   GXSYNC_LOCK(&sync);
   counter++;
   GXSYNC_UNLOCK(&sync);
   // ...
}
```

Incrementing and Decrementing Variables

Alternatively, if you want to increment or decrement a variable using a spin lock, call the following functions in the following order:

1. Call the GXSYNC_INIT( ) function to initialize a synchronization variable (of type GXSYNCVAR) to be used to synchronize access to shared resources, via a spin lock, in subsequent operations.
2. Call one of the following functions:
   - GXSYNC_INC( ) function to increment a variable by one (1), using a spin lock to ensure synchronized access to it.
   - GXSYNC_DEC( ) function to decrement a variable by one (1), using a spin lock to ensure synchronized access to it.
3. Call the GXSYNC_DESTROY( ) function to destroy a spin lock that is no longer needed. Calling GXSYNC_DESTROY( ) releases the system resources allocated for the spin lock. Subsequent calls to the spin lock are invalid. To use the spin lock again, you must subsequently initialize the spin lock using GXSYNC_INIT( ).

GXSYNC_INC( ) and GXSYNC_DEC( ) call GXSYNC_LOCK( ) automatically before changing the variable, and call GXSYNC_UNLOCK( ) automatically after changing the variable.

Using Critical Sections

In multithreaded programming, use critical sections in your code to ensure synchronization when multiple threads can manipulate the same object.

To use a critical section

1. Call the GXInitCriticalSection( ) function to initialize a critical section object (of type GXCRIT_SECTION) to be used in subsequent operations to synchronize thread access to a particular process.
2. Call the GXEnterCriticalSection( ) function to obtain exclusive thread access to a shared resource before performing any operations on the protected resource. GXEnterCriticalSection( ) blocks until the thread is granted ownership.
3. Run the protected operations on the thread.
4. Call the GXLeaveCriticalSection( ) function to release exclusive thread access to shared resources after completing operations on the protected resource. Releasing ownership allows other threads to acquire the critical section.

5. Call the GXDeleteCriticalSection( ) function to destroy a critical section object that is no longer needed, which releases the system resources allocated for the critical section object. Subsequent calls to the critical section are invalid. To use the critical section again, you must subsequently initialize the critical section using GXInitCriticalSection( ).

Example

The following code implements a class that uses a critical section.

```
class MyClass {
   GXCRIT_SECTION myCS;
   public:MyClass() {
      GXInitCriticalSection(&myCS);
   }
   ~MyClass() {
      GXDeleteCriticalSection(&myCS);
   }
   void method() {
      // ...
      GXEnterCriticalSection(&myCS);
      // ...
      // Perform long, protected operation here.
      // ...
      GXLeaveCriticalSection(&myCS);
   }
```

Working with Strings

When calling methods that return strings through out parameters, the caller is usually responsible for allocating the memory buffer for the string and for passing in the size of the buffer. The methods and functions called will fill the buffer with the string value. For example:

```
char buff[200];
buff[0]='\0';
HRESULT hr;
```

```
hr = m_pValIn->GetValString("PHONE", buff, sizeof(buff));
if (hr==NOERROR)
   printf("Phone number is %s\n", buff);
```

Numerous methods and functions in the libraries provide ways to specify and retrieve string values. For more information, see the relevant class, interface, or function description in the <Italic>iPlanet Application Server Foundation Class Reference.

Working with IGXValList Objects

An IGXValList object is an unordered list of named values. IGXValList objects are supported by the IGXValList interface. You use the GXCreateValList( ) function to create an IGXValList object, and you use the following commands to manipulate IGXValList objects:

- the SetVal( ) methods in the IGXValList interface to specify values in an IGXValList object
- the GetVal( ) methods in the IGXValList interface to retrieve values from an IGXValList object

The following example shows how to create an IGXValList object, populate it with database connection properties, and then pass it as a parameter to CreateDataConn( ):

```
m_pProps = GXCreateValList();
GXSetValListString(m_pProps, "DSN", "ksample");
GXSetValListString(m_pProps, "DB", "ksample");
GXSetValListString(m_pProps, "USER", "kdemo");
GXSetValListString(m_pProps, "PSWD", "kdemo");

// Create a database connection using properties
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, m_pProps,
   m_pContext, &m_pConn);

IGXValList *pList=GXCreateValList();
if(pList) {
```

```
// Specify database connection properties
   GXSetValListString(pList, "DSN", OB_DSN);
   GXSetValListString(pList, "DB", "");
   GXSetValListString(pList, "USER", OB_USER);
   GXSetValListString(pList, "PSWD", OB_PASSWORD);

   // Create a database connection using properties
   hr=CreateDataConn(0, GX_DA_DRIVER_DEFAULT, pList,
      m_pContext, ppConn);
   // Release the list
   pList->Release();
}
```

Working with GUIDs

Each registered AppLogic has a Globally Unique Identifier (GUID) associated with it. A GUID is a 128-bit hexadecimal number and has an associated GUID struct.

You can use the following commands to manipulate GUID structs:

- GXGetValListGUID( ) and GXSetValListGUID( ) functions
- GXGUIDToString( ) and GXStringToGUID( ) functions
- GXGUID_EQUAL macro

When passing a GUID to a macro, such as GXDLM_DECLARE or GXDLM_IMPLEMENT, you pass in a GUID struct. This format is shown in the following code example, taken from the source file OBLogin.cpp:

```
//
// Set the GUID for OBLogin to
// {C1B5E720-6153-11D1-A1AE-006008293C54}
//
GUID OBLoginGUID =
{ 0xC1B5E720, 0x6153, 0x11D1, { 0xA1, 0xAE, 0x00, 0x60, 0x08, 0x29,
0x3C, 0x54 } };
```

The following code is from a header file and shows how the GUID is passed to the GXDLM_DECLARE macro. OBLogin is the name of the AppLogic class associated with the GUID struct stored in OBLoginGUID.

```
extern GUID OBLoginGUID;

GXDLM_DECLARE( OBLogin,  OBLoginGUID);
```

Note that the string version of the GUID is embedded in the comments (for readability purposes only) and that the GUID struct is what gets parsed. However, in some cases, the string version of the GUID is passed, such as with the NewRequest( ) method.

For more information about GUIDs, see “Requests, AppLogic Names, and GUIDs” on page 70 of , “Writing Server-Side Application Code.”

Working with Binary Large Objects (BLOBs)

A binary large object (BLOB) is a large block of bits that can be stored in a database. A BLOB is useful for storing any large piece of data, such as pictures or sounds, that do not need to be interpreted by the database. Use the following methods for manipulating BLOB data in your AppLogic:

- SetValueBinary( ) or SetValueBinaryPiece( ) in the IGXTable interface for inserting or updating BLOB values in a table
- GetValueBinary( ) or GetValueBinaryPiece( ) in the IGXResultSet interface for retrieving BLOB values returned in a result set
- GetValBLOB( ) or SetValBLOB( ) in the IGXValList interface for retrieving or assigning BLOB values in an IGXValList object

Example

The following code retrieves a BLOB value from a database.

```
HRESULT hr;

IGXQuery       *pQuery = NULL;

IGXResultSet   *pRS    = NULL;


CreateQuery(&pQuery);

pQuery->SetTables("blobtable");

pQuery->SetFields("blobcol");

hr = pConn->ExecuteQuery(0, pQuery, NULL, NULL, &pRS);
```

```
ULONG nRows;
hr = pRS->GetRowNumber(&nRows);

LPBYTE pBlobChunk = NULL;
ULONG expectSize, gotSize;
expectSize = 65535;
pBlobChunk = new LPBYTE[65536];

hr = pRS->GetValueBinaryPiece(1, expectSize, &pBlobChunk,
   65536);
pRS->GetValueSize(1, &gotSize);
if (gotSize == expectSize)
   fprintf(stderr, "got a full chunk, size = %d\n", gotSize);
else
   fprintf(stderr, "got a partial chunk, size = %d\n", gotSize);

pRS->Release();
```

Working with Dates and Times

Date and time values are implemented as a GXDATETIME struct. You can use the GXGetCurrentDateTime( ) function to obtain the system time.

The following example shows how to retrieve and print the current system time:

```
GXDATETIME curtime;
GXGetCurrentDateTime(&curtime);
printf("Time is [%02d/%02d/%02d %02d:%02d:%02d:%01d]\n",
   curtime.month,
   curtime.day,
   curtime.year-1900,
   curtime.hour,
```

```
    curtime.minute,
    curtime.second,
    curtime.fraction/100);
```

Exporting Classes

In order for your classes to be loaded properly at runtime, you must export some specific functions. iPlanet Application Server expects to find these exported functions when it loads your shared library at runtime. The exported functions are required to fully initialize the shared library and to create instances from the classes in it.

To export a class

1. In a header file (.h), call the GXDLM_DECLARE macro to associate a class in a dynamically loadable, shared library module (DLM) with an already declared GUID struct.
2. In a source file (.cpp) that is associated with the header file, call the GXDLM_IMPLEMENT_BEGIN macro to begin a block of one or more GXDLM_IMPLEMENT calls.
3. Call the GXDLM_IMPLEMENT macro to establish to the iPlanet Application Server the entry point in a dynamically loadable, shared library module (DLM) for one exported class.
4. Call the GXDLM_IMPLEMENT_END macro to end a block of one or more of GXDLM_IMPLEMENT calls.

The following example shows how these macros are used. The following code fragment appears in the header file:

```
extern GUID OBLoginGUID;
class OBLogin : public OBBaseAppLogic
{
   // ...
}
GXDLM_DECLARE( OBLogin,  OBLoginGUID);
```

The following code fragment appears in the source file:

```
// {C1B5E720-6153-11D1-A1AE-006008293C54}
```

```
//
GUID OBLoginGUID =
{ 0xC1B5E720, 0x6153, 0x11D1, { 0xA1, 0xAE, 0x00, 0x60,
  0x08, 0x29, 0x3C, 0x54 } };
// ...
GXDLM_IMPLEMENT_BEGIN();
GXDLM_IMPLEMENT( OBLogin, OBLoginGUID);
GXDLM_IMPLEMENT_END();
```

Using Events

In an iPlanet Application Server environment, you can create and use named events. The term *event* is widely used to refer to user actions, such as mouse clicks, that trigger code. However, the events described in this section are not caused by users. Rather, an event is a named action that you register with the iPlanet Application Server. The event occurs either when a timer expires or when the event is activated from application code at runtime.

Events are stored persistently in the iPlanet Application Server, and are removed only when your application explicitly deletes them. Typical uses for events include periodic backups, reconciling accounts at the end of the business day, or sending alert messages. For example, you can set up an event that sends an email to alert your company's buyer when inventory levels drop below a certain level.

Each event has a name, a timer (optional), and one or more actions to take when the event is triggered. Application events have the following characteristics:

- Each event can cause the execution of one or more actions, which can include sending email or running an Applogic.
- Actions can be synchronous or asynchronous with the calling environment.
- Multiple actions can be configured to execute concurrently with one another, or serially, one after the other.
- Multiple actions are executed in a specific order (the order in which they are registered).
- Request data can be passed to an application event in an IValList object.

You can set up events to occur at specific times or at intervals, such as every hour or once a week. You can also trigger an event by calling the event by name from code. When an event's timer goes off or it is called from code, the associated action occurs.

The Application Events API

iAS uses two interfaces to support events:

- The `IGXAppEventMgr` interface manages application events. This interface defines methods for creating, registering, triggering, enabling, disabling, enumerating, and deleting events.
- The `IGXAppEventObj` interface represents the defined events an application supports. This interface defines methods not only for getting or setting attributes of an event, but also for adding, deleting, or enumerating actions of the event.

For more details, see the entries for these interfaces in the *iPlanet Application Server Foundation Class Reference*.

Creating a New Application Event

To access an IGXAppEventMgr object, use the C++ helper function GXContextGetAppEventMgr( ):

```
HRESULT GXContextGetAppEventMgr(
    IGXContext *pContext
    IGXAppEventMgr **ppAppEventMgr);
```

The pContext parameter is a pointer to an IContext object, which provides access to iPlanet Application Server services. Specify a value of m_pContext.

The ppAppEventMgr is a pointer to the returned manager object.

After creating the `IGXAppEventMgr` object, you can create an application event (an instance of `IGXAppEventObj`) by calling `CreateEvent()` on the `IGXAppEventMgr` object.

You must then register the event, or make iAS aware of it, by calling `RegisterEvent()`. Further, you must also instruct iAS to enable the event for access by calling `EnableEvent()`. Once the event is registered and enabled, you can trigger it by hand using `TriggerEvent()`.

Using an Application Event

You can perform any of the following tasks with an event by using the associated methods in the `IAppEventMgr` object:

Method	Task
Using Events	Creates an empty application event object.
Using Events	Removes a registered event from iAS.
Using Events	Temporarily isables a registered event.
Using Events	Enables a registered event.
Using Events	Enumerates through the list of registered events.
Using Events	Retrieves the IGXAppEventObj for a registered event.
Using Events	Registers a named event for use in applications.
Using Events	Triggers a registered event.

Within the event object itself, you can set and examine the event's attributes as well as define actions for the event. Use the methods in the `IAppEventObj` interface:

Method	Description
Using Events	Appends an action to an ordered list of actions.
Using Events	Deletes all actions added to this IGXAppEventObj.
Using Events	Enumerates the actions added to this IGXAppEventObj.
Using Events	Retrieves the list of attributes of an IGXAppEventObj.
Using Events	Retrieves the name of the IGXAppEventObj.
Using Events	Sets a list of attribute values for the IGXAppEventObj.

For more details about these methods, see the *iPlanet Application Server Foundation Class Reference*.

Using Cookies

Cookies are a mechanism that Web applications can use to store information on the client (Web browser) side of the application. Cookies are variables that your application sends to the browser to be stored there for a specified length of time. Each time a Web browser requests an HTML page in your application, the cookies from that browser are sent to the application.

Cookies are domain-specific and can take advantage of the same Web server security features as other data interchange between your application and the server. Thus, cookies are useful for privately exchanging data between your application and the Web browser.

Some browsers do not support cookies, but they are supported by all versions of Netscape Navigator and by Microsoft Internet Explorer version 2.0 and later.

Sending a Cookie

To send a cookie, call SetVariable( ). For example:

```
SetVariable("preference", "green");
```

Referencing a Cookie

Whenever an AppLogic is executed, all cookies stored in the Web browser are sent to the AppLogic as part of the AppLogic's input parameters. The iPlanet Application Server receives the cookies as part of the AppLogic request, along with other input parameters for the AppLogic. When iPlanet Application Server runs the AppLogic, it passes all the input parameters to the AppLogic, including the cookies. The input parameters are passed in the input IGXValList object.

Therefore, to reference a cookie from AppLogic code, use the AppLogic's m_pValIn variable, which refers to the input IGXValList. The name of the parameter is the same as the name you specified when you called SetVariable( ) to send the cookie to the Web browser. For more information about m_pValIn, see "Passing Parameters to AppLogic Objects" on page 71 of , "Writing Server-Side Application Code."

Chapter 4

Writing Server-Side Application Code

This chapter describes AppLogic objects, which are a set of programming instructions that accomplish a well-defined, modular task within an application.

The following topics are included in this chapter:

- What Is An AppLogic Object?
- Introduction to Writing AppLogic Objects
- Steps for Writing AppLogic Objects
- Performing the Main Task in an AppLogic Object
- Calling an AppLogic From Code
- Requests, AppLogic Names, and GUIDs
- Passing Parameters to AppLogic Objects
- Returning Results From an AppLogic Object
- Caching AppLogic Results to Improve Performance

What Is An AppLogic Object?

AppLogic objects run on the iPlanet 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 the GXAppLogic class in the iPlanet Application Server Foundation Class Library. AppLogic source files are stored in files with the `.cpp` extension.

AppLogics perform the tasks in the server side of the application. For example, some of the AppLogic objects in the Online Bank sample application perform the following tasks:

- Display a menu page.
- Retrieve and display the current balance in a customer's account.
- Retrieve and display transactions.
- Transfer funds from one account to another.

When writing AppLogic objects, you can use the classes and interfaces provided in the iPlanet Application Server Foundation Class Library. This class library provides the specialized functionality you need to write AppLogic objects for iPlanet Application Server applications.

Introduction to Writing AppLogic Objects

To write an AppLogic object,use one of the following techniques:

- Use the iPlanet Application Builder to create an AppLogic visually. This tool provides a framework that gives you a head start on the most commonly-used types of AppLogic.

  For more information, see iPlanet Application Builder User's Guide.

- If you are not using the AppLogic Designer, you can use your favorite code editor to write the AppLogic, using and call the iPlanet Application Server Foundation Class Library when necessary. You can also use a code editor to modify an AppLogic that was produced by a development tool or imported into your project. The rest of this chapter describes how to write AppLogic objects without using the AppLogic Designer.

Parts of a Typical AppLogic Object

The following skeleton code shows the syntax of a typical AppLogic object.

First, the AppLogic can use one or more `#include` statements to gain access to code in other files.

```
#include <file.h>
```

After the `#include` statements, the body of the AppLogic begins with a subclass declaration. All AppLogics are derived, directly or indirectly, from the GXAppLogic class.

```
class NewAppLogicName : public GXAppLogic {
```

The main task of the AppLogic is specified in code that overrides the Execute( ) method, which is inherited from the GXAppLogic superclass.

```
   STDMETHODIMP
   NewAppLogicName::Execute() {
```

The code within the Execute( ) method can perform any desired task. In this skeleton example, it is not possible to show all the typical tasks.

```
   // AppLogic code
   // . . .
```

The final task of a typical AppLogic is to send a response back to the calling entity. The result can be any type of data, and it can be sent to any calling entity, including another AppLogic. There are several alternative techniques available, such as using the EvalTemplate( ) or EvalOutput( ) method to stream data back to the client. In the following code, `methodCall` is a placeholder for a method call, where result is a success code that indicates to the iPlanet Application Server system if the request was processed correctly. You use ValOut or streaming or EvalTemplate( ) / EvalOutput( ) to return application specific results.

```
      result = MethodCall(params);
      return result;
   }
}
```

For more information, see “Returning Results From an AppLogic Object,” on page 79.

Example

The following example code shows a simplified version of the OBShowMenuPage AppLogic in the Online Bank sample application. This AppLogic displays a main menu in an HTML page.

The following code is in the header file:

```
// Declare an indirect subclass of AppLogic: OBShowMenuPage
// uses OBBaseAppLogic, which is derived from AppLogic
```

```
class OBShowMenuPage : public OBBaseAppLogic
{
   // ...
};
```

The following code is in the source file:

```
// Include header files
#include <stdio.h>
#include <gxplat.h>
#include <gxutil.h>
#include <gxagent.h>
#include <gxdlm.h>
#include "ShowMenuPage.h"
#include "gxval.h"
#include "common.h"
// ...
// ...

// Override Execute() method
STDMETHODIMP
OBShowMenuPage::Execute()
{
   HRESULT hr=GXE_SUCCESS;
   if(!IsSessionValid())
      return HandleOBSessionError();
   OBSession *pSession=NULL;
   if(((hr=GetOBSession(&pSession))==GXE_SUCCESS)&&pSession)    {
      ULONG userType=pSession->GetUserType();

      if(userType==OB_USERTYPE_CUSTOMER)

// Return results for user type customer
```

```
            EvalTemplate("GXApp/COnlineBank/templates/
               CustomerMenu.html", (IGXHierQuery*)NULL, NULL,
               NULL, NULL);
         else if(userType==OB_USERTYPE_REP)

// Return results for user type representative of bank
            EvalTemplate("GXApp/COnlineBank/templates/
               RepMenu.html", (IGXHierQuery*)NULL, NULL, NULL,
               NULL);
         pSession->Release();
      }
      else
         Result("<HTML>Call to getOBSession() failed
            in Login</HTML>");
      return GXE_SUCCESS;
}
```

Steps for Writing AppLogic Objects

This section provides an overview of the process of writing an AppLogic object. It assumes that you are familiar with using the code editor on your development platform.

To write AppLogic using the iPlanet Application Builder, follow the general steps defined in the rest of this section. It uses examples from the OBLogin AppLogic in the Online Bank sample application, which is described in greater detail in What Is An AppLogic Object?“What Is An AppLogic Object?,” on page 61 of Chapter 4, “Writing Server-Side Application Code.”

Header File

These instructions use examples from the header file OBLogin.h.

1. In the header file, include the other necessary header files:

```
#include <stdio.h>          // standard I/O routines
#include <gxapplogic.h>     // KIVA AppLogic base class
#include "BaseAppLogic.h"   // AppLogic base class for this
                            // application
```

2. Declare the GUID variable for the AppLogic (a unique GUID is associated with each AppLogic):

```
extern GUID OBLoginGUID;
```

3. Subclass from the application's AppLogic base class, which is derived from GXAppLogic:

```
class OBLogin : public OBBaseAppLogic
{
```

4. Define the constructor and destructor methods, then declare the Execute( ) method using the STDMETHOD macro:

```
public:
   OBLogin();
   virtual ~OBLogin();
   STDMETHOD(Execute) ();
};
```

5. Associate the AppLogic with its GUID variable using the GXDLM_DECLARE macro:

```
GXDLM_DECLARE( OBLogin,  OBLoginGUID);
```

Source File

These instructions use examples from the source file OBLogin.cpp.

1. In the source file, include the necessary header file(s):

```
#include "OBLogin.h"
#include "gxval.h"
#include "common.h"
```

2. Create a GUID for the AppLogic using the `kguidgen` utility, as described in "Getting Ready to Run an Application"Getting Ready to Run an Application," on page 275 of Chapter 11, "Running and Debugging Applications

3. Paste the generated GUID for the AppLogic into the source file (.cpp). Note that the text version is embedded in the comments (for readability purposes only) and that the GUID struct is what gets parsed.

```
//
// {C1B5E720-6153-11D1-A1AE-006008293C54}
//
GUID OBLoginGUID =
{ 0xC1B5E720, 0x6153, 0x11D1, { 0xA1, 0xAE, 0x00, 0x60,
   0x08, 0x29, 0x3C, 0x54 } };
```

4. In the source file, establish to the iPlanet Application Server the entry point in a dynamically loadable, shared library module using the GXDLM_IMPLEMENT_BEGIN, GXDLM_IMPLEMENT, and GXDLM_IMPLEMENT_END macros. These macros define certain exported functions which the iPlanet Application Server expects to find when it loads the AppLogic at runtime. The macros are needed in order to fully initialize and create instances of the AppLogic.

```
GXDLM_IMPLEMENT_BEGIN();
GXDLM_IMPLEMENT(COBLogin, OBLoginGUID);
GXDLM_IMPLEMENT_END();
OBLogin::OBLogin() {
   GXDllLockInc(); // Update count of refs to applogic lib
};
OBLogin::~OBLogin() {
   GXDllLockDec(); // Update count of refs to applogic lib
};
```

It is advisable to use the GXDllLockInc( ) function at the *beginning* of the constructor method and the GXDllLockDec( ) function at the *end* of the destructor method.

5. Override the Execute( ) method with your own code. The STDMETHODIMP macro specifies that this execute method has a return type value of `virtual HRESULT`.

```
STDMETHODIMP
OBLogin::Execute()
{
```

```
        // Overriding code
    }
```

Performing the Main Task in an AppLogic Object

To write code that performs the main task of an AppLogic object, override the AppLogic's Execute( ) method. The Execute( ) method is inherited from the GXAppLogic class, from which you derived the AppLogic. iPlanet Application Server automatically calls Execute( ) when a request comes in for the AppLogic.

You can write code in Execute( ) to perform any desired task. A typical AppLogic's Execute( ) method might contain code to perform the following tasks:

- Check input parameters.

  For more information, see "Validating Input to AppLogic Objects," on page 265 of "Writing Secure Applications," Chapter 9.

- Query a database to retrieve data requested by the user.

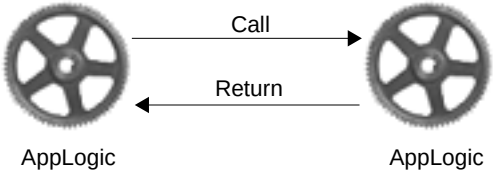
  For more information, see Chapter 5, "Working with Databases

- Return results.

  For more information, see "Returning Results From an AppLogic Object" on page 79.

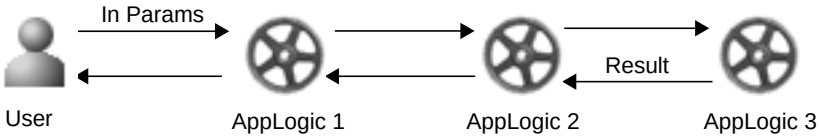
Calling an AppLogic From Code

In addition to being executed by iPlanet Application Server in response to user requests, AppLogic objects can be called by other AppLogic objects or by other code. The called AppLogic returns results to the calling code, as shown in the following illustration.



AppLogic objects can call each other whether they are running on the same iPlanet Application Server or on different iPlanet Application Servers. In your AppLogic code, you do not specify the location of the called AppLogic. This allows you to change the partitioning and location of AppLogic objects and redeploy an application without having to modify AppLogic code.

In some cases, the user might submit a request that runs an AppLogic, which calls another AppLogic, which calls another one, and so on. The input parameters are passed down the chain automatically by iPlanet Application Server, and the results from the called AppLogics are passed back up the chain until they reach the end user. You can modify the input parameters or intercept the results at any point if desired, but typically, the parameters and results are passed along as shown in the following illustration.



To call an AppLogic from another AppLogic, use the NewRequest( ) method. For server-side code, this method is in the GXAppLogic class. For client-side code, it is in the IGXConnection interface. In the NewRequest( ) call, specify the name or globally unique identifier (GUID) of the AppLogic you want to call. The name or GUID is assigned when you register the AppLogic with iPlanet Application Server. For more information about registration, see “Registering Code And Security Information” on page 282 of , “Running and Debugging Applications.”

iPlanet Application Server uses the arguments of the NewRequest( ) method to construct an AppLogic request, which it then processes by executing the AppLogic. For more information, see “Requests, AppLogic Names, and GUIDs” on page 70.

You can pass parameters to and from the called AppLogic by using IGXValList objects. For more information, see “Passing Parameters to AppLogic From Code” on page 76.

Example

The following code calls an AppLogic by GUID:

```
hr = NewRequest("{E5CA1000-6EEE-11cf-96FD-0020AFED9A65}",
                    m_pValIn, m_pValOut, 0);
```

The following code shows how to call the same AppLogic by name. In this code, it is assumed that you have registered the AppLogic with the name CShowMenuPage.

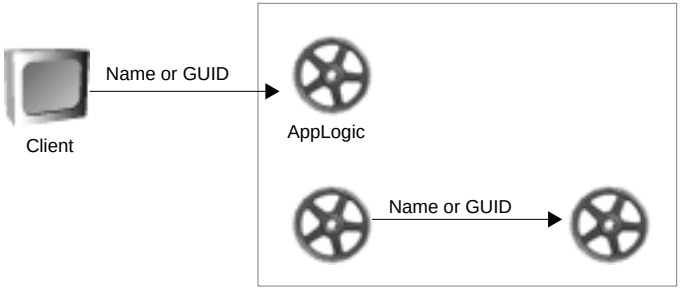
```
hr = NewRequest("AppLogic CShowMenuPage",
                m_pValIn, m_pValOut, 0);
```

Requests, AppLogic Names, and GUIDs

When an AppLogic object is called, whether from a user or from code, a message called a *request* is sent to iPlanet Application Server. In response to the request, iPlanet Application Server runs the AppLogic. Requests from users and from within program code can use either of the following techniques to identify the proper AppLogic to handle the request:

- unique AppLogic name
- globally unique identifier (GUID)

The following illustration shows how AppLogics are called.



Every AppLogic has a unique name and a unique GUID. The GUID is a 128-bit hexadecimal number in the following format:

```
{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}
```

For example:

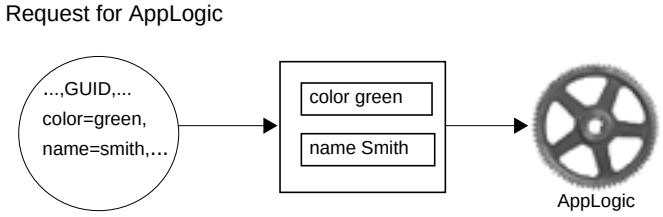
```
{C1B5E720-6153-11D1-A1AE-006008293C54}
```

When an AppLogic is registered with iPlanet Application Server, a unique GUID and a name are assigned to the AppLogic.You can register AppLogics and assign names to them using the technique described in “Registering Code And Security Information” on page 282 of , “Running and Debugging Applications.”

Passing Parameters to AppLogic Objects

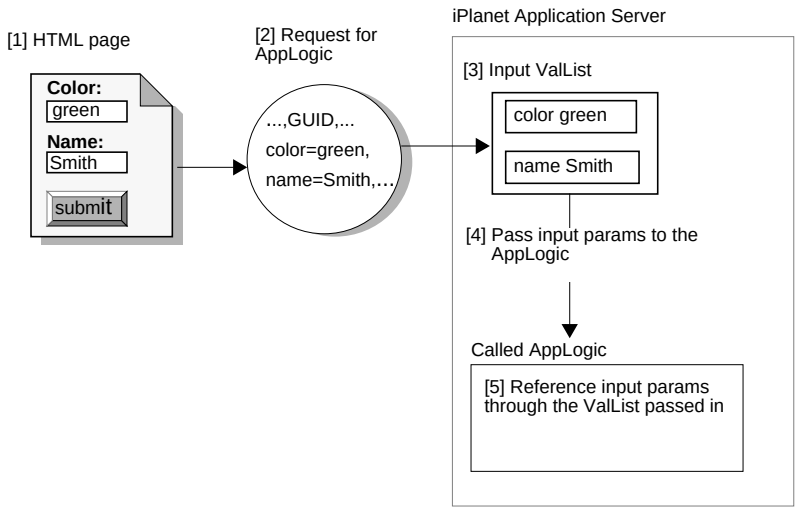
When iPlanet Application Server processes a request to run an AppLogic object, it checks the request to see whether it contains any parameters, and it passes the parameters to the AppLogic. For example, the parameters may be values the user has supplied from input fields on a form. AppLogic objects can also call an AppLogic and pass parameters to it, setting up the parameters in code.

iPlanet Application Server passes parameters in the form of IGXValList objects. An IGXValList object is an unordered collection of named parameters. Each parameter has a data type and value. To pass parameters to an AppLogic, iPlanet Application Server constructs an IGXValList object based on the names and values it finds in the request, as shown in the following illustration.



Passing Parameters To AppLogic From An HTML Page

The following illustration and list summarize the steps you follow and the sequence of events that occurs when you pass parameters to an AppLogic object from an HTML page.



To pass parameters to AppLogic from an HTML page

1. Create an HTML form and write an AppLogic object to handle the input from the form. The form typically includes the following items:

   - One or more named controls that accept input, either as typed text or selected values, such as radio buttons and single- or multiple-selection list boxes. If a user selects several items from a multiple-selection list box, the value of the field is returned as a semicolon-delimited list.
   - A Submit button that the user clicks after filling out the form.
   - The URL that specifies which AppLogic to run when the user clicks the Submit button.

   For more information about how to code this URL, see “Calling an AppLogic Object From an HTML Page” on page 192 of , “Writing Server-Side Application Code.”

2. The user runs the application and fills out the form. When the user clicks Submit, a request is issued that includes the input field names, data values, and AppLogic name or GUID from the form. The Web server passes this request to iPlanet Application Server.

3. iPlanet Application Server instantiates an IGXValList object and populates it with the data values from the form. Each value in the list is named after one of the input fields on the form. The input parameters might also include cookies, if any are currently stored in the Web browser for this application.
4. iPlanet Application Server then instantiates the AppLogic, sets the AppLogic's m_pValIn member variable to the IGXValList object that contains the AppLogic's input parameters, and then calls the Execute( ) method of the AppLogic.
5. The AppLogic can get the parameter values by referencing m_pValIn and using the GetVal**( ) methods of the IGXValList interface.

Example

In the Online Bank sample application, an employee of the bank can use the Find Customer form to get information about a given customer. This form is presented as an HTML page with input fields into which the employee types the search criteria, such as the customer's last name:

```
<!-- Preliminary HTML tags ... -->
<FORM method="POST" action="/cgi-bin/gx.cgi/
   AppLogic+FindCust">
<H2>Search for Customer</H2>
<H5>Please enter search criteria:</H5>
<TABLE BORDER=0 COLS=2 WIDTH=100% BGCOLOR=#CCCC80>
<TR>
<TD>Last Name</TD><TD><INPUT TYPE="TEXT" NAME="lastName"
   SIZE=30 VALUE=""></TD>
</TR>
<TR>
<TD>First Name</TD><TD><INPUT TYPE="TEXT"
   NAME="firstName" SIZE=30 VALUE=""></TD>
</TR>

<!-- Other criteria fields ... -->
</TABLE>
```

```
<input type="submit" name="go" value="Search">
</FORM>
<!-- Closing HTML tags ... -->
```

When the employee clicks the form's Search button, a request is sent to the Web server. The request includes the name of the FindCust AppLogic, which is designed to process this form. The names of fields from the form and the data the bank employee typed in each field are also included in the request.

The Web server forwards the request to the iPlanet Application Server, which places the input data in an IGXValList object. iPlanet Application Server then runs the FindCust AppLogic, setting the AppLogic's m_pValIn parameter to the IGXValList object that contains the AppLogic's parameters.

The AppLogic's Execute( ) method contains code to get the parameter values out of the IGXValList object. For example, the following code places the data value from the LastName field on the form into a variable named pLastName.

```
LPSTR pLastName=NULL;
pLastName=GXGetValListString(m_pValIn, "lastName");
```

After using the parameter values to look up the requested customer data from the bank's database, the AppLogic returns an HTML page to display the results.

Uploading Files From a Web Browser

By using AppLogic input parameters, you can send a text or binary file from a Web browser to an application running on the iPlanet Application Server. You can also upload files in Microsoft Internet Information Server (MS IIS). This feature is useful for applications which could benefit from the submission of files of data. The file is passed in the input IGXValList object, just like any other parameters coming from the HTML page.

For example, in a Human Resource management application, a form could prompt the user to attach a resume file along with other information such as the job for which they are applying. The AppLogic receiving the submitted resume could store the resume and other information in a database.

For applications that use the file upload feature to work, file uploading must be supported by the Web server to which the iPlanet Application Server is connected.

Example

The following example HTML code uploads the user's file when the user clicks the Send File button. The file name of the user's file becomes the value of the variable `userFile`.

```
<FORM ENCTYPE="multipart/form-data"
ACTION="/cgi-bin/gx.cgi/GUIDGX-{0F6D8100-6E1F-11cf-96FD-0020AFED9A6
5}" METHOD=POST>

Send this file: <INPUT NAME="userFile" TYPE="file">

<INPUT TYPE="submit" VALUE="Send File">

</FORM>
```

The following example code handles the submission of the file and processes its contents. This code would appear as part of the AppLogic corresponding to the GUID {0F6D8100-6E1F-11cf-96FD-0020AFED9A65}.

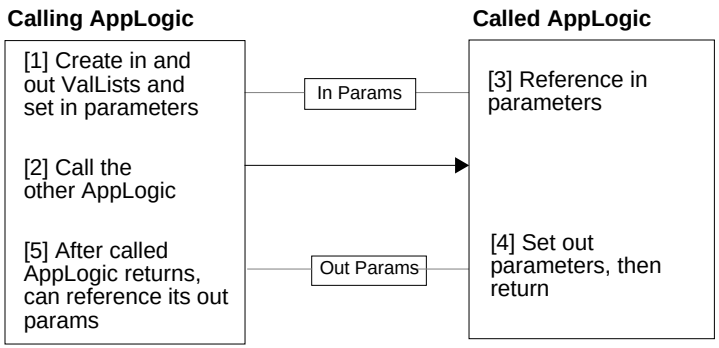
```
StreamResult("<HTML><HEAD><TITLE>File Submit</TITLE>
   </HEAD>");
StreamResult("<BODY BGCOLOR=#FFFFFF>");
StreamResult("<H2>File Info</H2>");
char tmp[1024];
tmp[0] = '\0';
m_pValIn->GetValString("userFile_file", tmp, sizeof(tmp));
StreamResult("<br>File name: ");
StreamResult(tmp);
tmp[0] = '\0';
m_pValIn->GetValString("userFile_type", tmp, sizeof(tmp));
StreamResult("<br>File type: ");
StreamResult(tmp);
tmp[0] = '\0';
m_pValIn->GetValString("userFile_size", tmp, sizeof(tmp));
StreamResult("<br>File size: ");
StreamResult(tmp);
int   bufferSize = atoi(tmp) + 16;
char *buffer     = new char[bufferSize];
buffer[0] = '\0';
m_pValIn->GetValBLOB("userFile", (LPBYTE) buffer,
   bufferSize);
StreamResult("<H2>File Content:</H2><PRE>");
```

```
StreamResult(buffer);
delete [] buffer;
StreamResult("</PRE></BODY></HTML>");
return 0;
```

Passing Parameters to AppLogic From Code

You can pass parameters to an AppLogic object when you call it from another AppLogic or from other code. To call an AppLogic from code, construct an IGXValList object in code, populate it with the desired parameter values, and pass the IGXValList object when you call the AppLogic. You can also return output parameters from the called AppLogic in another IGXValList object.

The following illustration and list summarize the steps you follow and the sequence of events that occurs when you pass parameters to an AppLogic from code. In the following discussion, the use of output parameters is also described.



To pass parameters to an AppLogic from code

1. Instantiate two IGXValList objects. Populate one IGXValList object with the parameters you want to pass to the AppLogic. The second IGXValList object is for the parameters that will be returned from the called AppLogic. To set up these objects, use the following methods:

   - To instantiate the IGXValList objects, use the GXCreateValList( ) function. For example:

     ```
     m_pParamsTo = GXCreateValList();
     ```

```
m_pParamsReturned = GXCreateValList();
```

- To populate the first IGXValList object with parameters you want to pass to the AppLogic, use the SetVal**( ) methods of the IGXValList interface. With each call to any of these methods, you specify a parameter name and its value. For example, the following line specifies a string parameter named stateVal and sets it to the value Oklahoma:

```
GXSetValListString(m_pParamsTo, "stateVal","Oklahoma");
```

2. Call the AppLogic using the NewRequest( ) method from the GXAppLogic class. This method issues a request that includes the AppLogic name or GUID and the input and output IGXValList objects. For example:

```
NewRequest(calledGUID, paramsTo, paramsReturned, 0);
```

   In response to the request, iPlanet Application Server instantiates the called AppLogic and sets its m_pValIn and m_pValOut member variables to the IGXValList objects specified by the calling code. In the previous example, m_pValIn is paramsTo and m_pValOut is paramsReturned.

3. The AppLogic can obtain its input parameter values by referencing m_pValIn with theGetVal**( ) methods from the IGXValList interface, as appropriate. For example, the called AppLogic could use the following code to get the value of the `stateVal` input parameter:

```
LPSTR state=GXGetValListString(m_pValIn, "stateVal");
```

4. If desired, you can return output parameters from the called AppLogic by filling the output IGXValList object with values. To do so, use the m_pValOut member variable to reference the output IGXValList object, and call the SetVal**( ) methods of the IGXValList interface. The called AppLogic uses the same techniques to set up its output IGXValList object as the calling code used to set up the input IGXValList object. For example, the called AppLogic could use the following code to set the value of the `Pop` output parameter:

```
m_pValOut->SetValString("Pop", result);
```

5. After the AppLogic is finished running, the calling code can use its second IGXValList object to reference the parameters that were passed back from the called AppLogic. For example:

```
paramsReturned->GetValString("Pop", population,
   sizeof(population));
```

Example

The following code shows part of an AppLogic that receives the name of a state as a parameter. The state comes from a field named StateField in a form that is filled out by the user. The AppLogic passes the state name to another AppLogic, which returns the population of that state. The called AppLogic passes the population out in a parameter named Pop.

```
// Get input parameter.
char state[128];
m_pValIn->GetValString("StateField", state, sizeof(state));
// Set up parameter to pass to other AppLogic.
IGXValList *paramsTo;
paramsTo = GXCreateValList();
paramsTo->SetValString("st", state);
// Set up IGXValList to receive result from other AppLogic.
IGXValList *paramsReturned;
paramsReturned = GXCreateValList();
// Call other AppLogic.
HRESULT hr;
hr = NewRequest("{E5CA1000-6EEE-11cf-96fd-0020AFED9A65}",
   paramsTo, paramsReturned, 0);
if (hr == NOERROR)
{
   // Get value from IGXValList returned by
   // other AppLogic.
   char population[128];
   paramsReturned->GetValString("Pop", population,
      sizeof(population));
   // ...
}
```

The following code shows how the called AppLogic references the state name that is passed in to it, and how it returns the Pop parameter to the calling AppLogic:

```
m_pValIn->GetValString("st", state, sizeof(state));
```

```
// ...
m_pValOut->SetValString("Pop", result);
```

Returning Results From an AppLogic Object

The final task an AppLogic object typically performs is returning its results. iPlanet Application Server directs the AppLogic results to the entity that called the AppLogic, whether it is a Web browser or another AppLogic.

For example, in response to a user request from a Web browser, an AppLogic might be called to look up data and merge it with an HTML template to create an HTML report. The report is the result of the AppLogic, and iPlanet Application Server sends the report back to the Web browser so the user can view it.

Types of Results

The techniques for returning AppLogic results fall into the following categories:

- The return value of the Execute( ) method. It is advisable for every AppLogic to return this type of result to indicate success or failure. For more information, see “Using the Return Value of Execute( )” on page 79.
- HTML pages. You should return HTML only when all clients are Web browsers and you want to create special HTML-specific output effects. For more information, see “Returning HTML Results” on page 81.
- Output parameters in an IGXValList object. You can return this type of result when the AppLogic is called from another AppLogic. For more information, see “Returning Output Parameters in an IGXValList Object” on page 85.

Using the Return Value of Execute( )

The `return` statement stops execution of a method or function and returns a value indicating either a successful completion of the task or an error. For example, the following statement indicates success:

```
return GXE_SUCCESS;
```

A nonzero value indicates failure.

In an iPlanet Application Server application, you use the `return` statement in each AppLogic when you override its Execute( ) method (unless the Execute( ) method has been modified to return a data type other than `int`). Inside Execute( ), the `return` statement is commonly combined with a method call, such as EvalTemplate( ) or EvalOutput( ), that returns results to the entity that called the AppLogic.

The iPlanet Application Server Foundation Class Library provides several such methods. By combining a method call with the `return` statement, you can accomplish two objectives: you can send output such as an HTML report from the AppLogic, and you can return the numeric code that indicates success or failure.

For example, the following code uses EvalTemplate( ) to send a dynamically-generated HTML page back to the entity that called the AppLogic. When EvalTemplate( ) is finished, it returns a code indicating success or failure. This code is, in turn, used as the argument to the `return` statement.

```
return EvalTemplate("GXApp/COnlineBank/templates/
   CustomerMenu.html",(IGXHierQuery*)NULL, NULL, NULL, NULL);
```

If a single method call is not enough to return all the output from the AppLogic, you can use several method calls before the `return` statement. For example, the following code uses streaming to return an HTML result in three parts:

```
StreamResultHeader(header);
StreamResult(body1);
StreamResult(body2);
return 0;
```

The streaming methods return data as it becomes available, instead of waiting for all the data to be ready before sending output to the user. For more information about streaming, see “Streaming Results” on page 83.

The method calls need not occur immediately before the `return` statement as they do in the previous example. They can be separated by additional lines of code. For example, you might want to return the results in several portions from several points in the AppLogic code.

Application-specific return values, such as error values or messages, should be returned using the AppLogic’s m_pValOut variable, output streaming, or EvalTemplate( ) or EvalOutput( ).

Returning HTML Results

You can explicitly return HTML when all clients are Web browsers. The advantage of forcing HTML results, rather than using the client-independent programming model, varies from application to application. If you wish to use particular features of HTML and are certain that all clients will be Web browsers for the life of the AppLogic, you can return HTML results using one of the following techniques:

- Merge the result set from a hierarchical query with an HTML template to produce a dynamically generated HTML page. For example, use this technique to return a database report or a dynamically populated form.

  For more information, see “Returning Results Using an HTML Template” on page 81.

- If the HTML result is not complicated or you prefer not to use an HTML template, you can construct and return HTML results programmatically. In most cases, it is preferable to use HTML templates because they are easier to maintain.

  For more information, see “Returning HTML Results Without a Template” on page 83.

You can also combine these two techniques to stream several portions of HTML in succession. When results are streamed, the first portion of the data is available for use immediately. This increases the perceived performance of the application. For example, although the AppLogic may process and return a lengthy query result, the AppLogic can use streaming to send back a report header, which is displayed quickly to the user. For more information, see “Streaming Results” on page 83.

Returning Results Using an HTML Template

To return HTML that is merged with dynamic data, use an HTML template. An HTML template is an HTML page that contains placeholders where data is to be merged with the template. For more information, see , “Working with Templates.”

The dynamic data is obtained from one of the following sources:

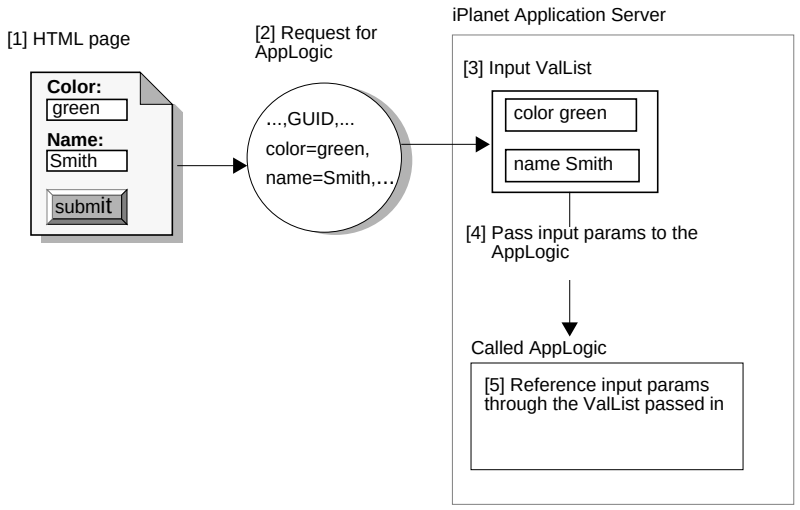
- A hierarchical query, which retrieves data from a database. For more information, see “Using Hierarchical Queries” on page 149 of , “Types of Queries.”

- A template data object, in which you have placed the data programmatically. For more information, see “Constructing a Hierarchical Result Set with GXTemplateDataBasic” on page 212 of , “Working with Templates.”

- To return an HTML page from an AppLogic, use EvalTemplate( ).

- This technique merges an HTML template with a hierarchical result set, then uses HTTP streaming to return the results to the calling entity, which is usually a Web browser.

You can return additional results after EvalTemplate( ) has finished streaming its results. To do so, call StreamResult( ) after calling EvalTemplate( ). For more information, see “Streaming Results” on page 83.



CAUTION The SaveSession( ) method in the GXAppLogic class performs some processing of HTTP headers, which must be sent before the HTTP body. The EvalTemplate( ) method streams an HTTP body. Therefore, if your application uses sessions and also streams HTML results to a Web browser, be sure to call SaveSession( ) before calling EvalTemplate( ) and StreamResult( ).

Example

The following code, from the ShowMenuPage AppLogic in the Online Bank sample application, uses EvalTemplate( ) to return an output HTML page using the HTML template CustomerMenu.html.

```
EvalTemplate("GXApp/COnlineBank/templates/CustomerMenu.html",
   (IGXHierQuery*)NULL, NULL, NULL, NULL);
```

Returning HTML Results Without a Template

An AppLogic can return HTML that is constructed programmatically, without using an HTML template. In most cases, it is preferable to use HTML templates, because they are easier to maintain.

To prepare HTML output programmatically, set the value of a string variable to the HTML string, and return that variable using Result( ) or StreamResult( ). For example, the following code is from the Logout AppLogic in the Online Bank sample application:

```
return Result("<HTML><BODY>Thanks for using the Online
   Bank.</BODY></HTML>");
```

You can also construct the HTML result in two parts, an HTTP header and HTTP body, and return the two parts sequentially. Use StreamResultHeader( ) to return the header and StreamResult( ) to return the body. For example:

```
StreamResultHeader(headerStr)
StreamResult(bodyStr);
```

Streaming Results

Streaming is a technique for managing how data is returned to the user. 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 prepared before any part of it can be sent to the client. Streaming provides a way to return large amounts of data in a more timely manner.

In an iPlanet Application Server application, you can use streaming to return either HTML or client-independent results.

Methods That Affect Streaming

The following list summarizes the methods in the libraries that perform streaming. For details about these methods, see the *iPlanet Application Server Foundation Class Reference.*

The following methods affect the streaming of header information:

- The SaveSession( ) method in the GXAppLogic class performs some processing of HTTP headers, which must be sent before the HTTP body. Therefore, if your application uses sessions and also streams HTML results to a Web browser, be sure to call SaveSession( ) before calling any streaming methods, including EvalTemplate( ) or EvalOutput( ).

For more information about SaveSession( ), see “Starting a Session” on page 228 of , “Managing Session and State Information.”

- The SetVariable( ) method sets the value of a variable and streams the variable out in an HTTP header. This method supports HTTP cookies. Call SetVariable( ) before calling any HTTP body streaming methods, such as EvalTemplate( ) or EvalOutput( ).

  For more information about SetVariable( ), see “Using Cookies” on page 60 of , “Application Development Techniques.”

- The StreamResultHeader( ) method is used to explicitly stream an HTTP header to a Web browser. To use this method, construct the data you want to pass as a stream header and pass it to StreamResultHeader( ) as a parameter.

  For more information, see the iPlanet Application Server Foundation Class Reference.

The following methods stream body information:

- The EvalTemplate( ) method merges data with a template. As soon as a segment of the output page is finished, EvalTemplate( ) streams it to the waiting client.

  For more information about EvalTemplate( ), see “Returning Results Using an HTML Template” on page 81.

- The StreamResult( ) method is used to explicitly stream an HTTP body to a Web browser. To use this method, construct the data you want to pass as a stream body and pass it to StreamResult( ) as a parameter.

  For more information, see the iPlanet Application Server Foundation Class Reference.

HTTP Header and Body Components

Streamed HTML results are communicated using the Hypertext Transfer Protocol (HTTP), which is the protocol used for communicating hypertext documents across the Internet and the World Wide Web. The HTTP protocol specifies the order in which data must be passed. If your AppLogic calls HTTP methods out of order, the AppLogic violates the protocol and causes a runtime error at the Web browser.

The HTTP protocol divides data into two categories: header and body. A discussion of the requirements for HTTP headers and bodies is outside the scope of iPlanet Application Server documentation. For more information, refer to the literature on the HTTP protocol.

When streaming HTML, be sure to stream header data before body data. You can stream the header or body in several parts, using several method calls, as long as all the header calls occur before the body calls. For example:

```
StreamResultHeader(startHeader);
StreamResultHeader(finishHeader);
StreamResult(bodyStart);
StreamResult(bodyMiddle);
StreamResult(bodyEnd);
```

Returning Output Parameters in an IGXValList Object

AppLogic objects that are called by other AppLogics or by other code can return one or more output parameters in an IGXValList object. To return output parameters, populate the AppLogic's output IGXValList object with the values you want to return. Use the AppLogic's m_pValOut member variable to refer to the output IGXValList object, and use the SetVal**( ) methods of the IGXValList interface to populate the list. With each call to one of these methods, you name an item in the list and set its value.

For more information, see "Passing Parameters to AppLogic From Code" on page 76.

Example

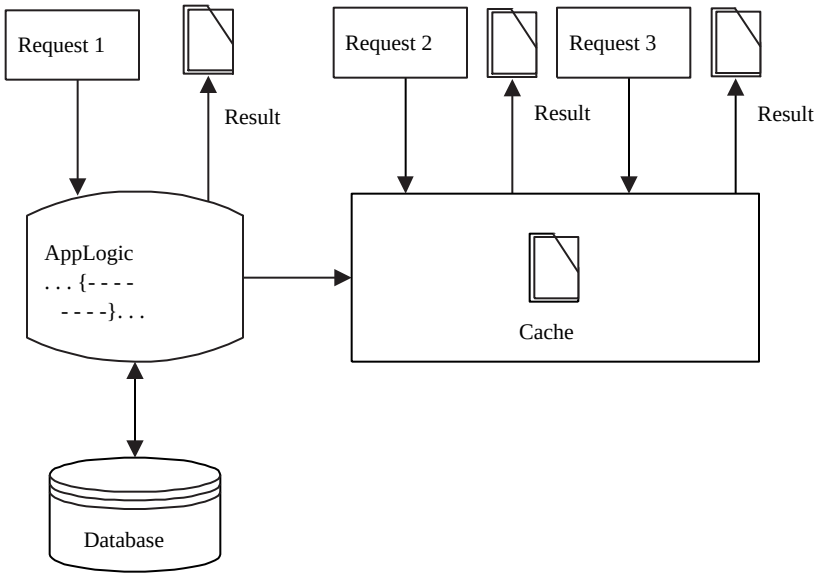
The following code returns the product of two numbers in an output IGXValList object.

```
int product = a * b;

m_pValOut->SetValInt("answer", product);

return 0;
```

Caching AppLogic Results to Improve Performance

Results from AppLogic objects can be cached. A cache is a fast-access area in the computer's memory. Caching improves performance when AppLogics perform time-consuming operations, such as lengthy database queries and report generation. Only streamed results, such as reports, can be cached, not other results such as output parameters. For example, output from EvalOutput( ) or EvalTemplate( ) can be cached.

The first time an AppLogic runs, it can store its results in the cache. When the iPlanet Application Server receives additional requests for the same AppLogic, instead of running the time-consuming operations again, the iPlanet Application Server returns the results directly from the cache. The following illustration shows how caching works.



When the most up-to-date results are needed, it is necessary to run AppLogics every time they are requested, rather than using cached results. However, it is often appropriate to cache and reuse results. For example, sales reports that are generated daily can be cached for 24 hours. Stock market price quotes can be provided on a 15-minute delay basis, with results cached between each update.

Each AppLogic has one result cache. The cache can contain multiple results, which are produced by running the AppLogic with different input parameter values. For example, an AppLogic might produce a report that shows the order history for a product. The AppLogic would accept the name of the product as an input parameter from the user. The same AppLogic might, therefore, produce several different reports for different products, and some or all of these reports can be cached.

How to Cache Results

To specify that you want the results from an AppLogic to be cached, call the SetCacheCriteria( ) method. The parameters to SetCacheCriteria( ) define the conditions of the caching, such as how long cache results are kept, size of cache, and which input parameter values are significant in controlling when results are cached.

The SetCacheCriteria( ) method also clears the cache of any existing results. Therefore, before calling SetCacheCriteria( ), call IsCached( ) to avoid accidentally discarding the current contents of the cache. For example:

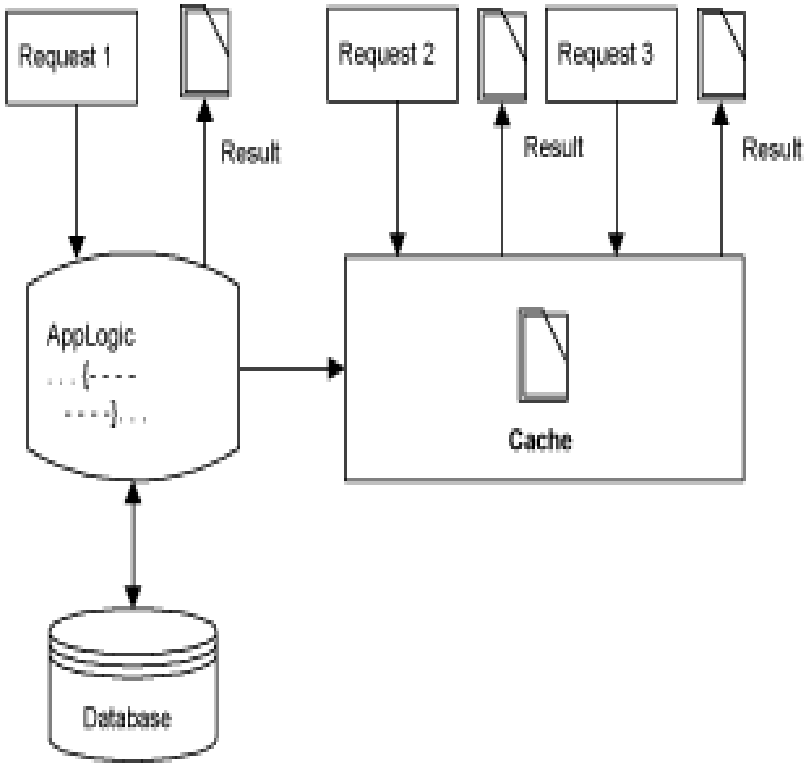
```
if (!IsCached())
{
   SetCacheCriteria(3600, 1, "");
}
IGXHierQuery *hqry;
// ...create and define hierarchical query here...
LPSTR templateName;
// ...assign template name here...
// Run template report.
//
EvalTemplate(templateName, hqry, NULL, NULL, NULL);
hqry->Release();
delete [] templateName;
return 0;
```

Using Cache Criteria

When you call SetCacheCriteria( ) to start caching, you specify the names and, optionally, the relevant values of certain AppLogic input parameters. You can specify criteria for some or all of the AppLogic's parameters. The criteria set limits on which parameters and which values are significant to iPlanet Application Server when the server determines whether to cache a particular AppLogic result.

Each time an AppLogic runs with caching enabled, iPlanet Application Server checks to see whether the AppLogic's input parameters match the criteria in the SetCacheCriteria( ) call. If so, iPlanet Application Server places the AppLogic's results in a cache. The cache also stores the values of the relevant parameters.

Each cached result is the output from running the AppLogic with a certain set of parameter values that fall within the criteria specified for that cache. For example, an EmployeeReport AppLogic might cache different reports for employees in different company departments, as shown in the following illustration.



How To Specify Caching Criteria

iPlanet Application Server provides several types of criteria that you can use to specify the conditions under which you want caching to occur. Use any of the following formats in the caching criteria parameter of SetCacheCriteria( ):

- Parameter Name
- Matching Value
- List of Values
- Range of Values
- List of Several Criteria

Parameter Name

Use this to cache multiple results for every value of the specified parameter. For example:

```
SetCacheCriteria(3600,1,"Department");
```

Alternatively, you can also use the following syntax:

```
SetCacheCriteria(3600,1,"Department=*");
```

iPlanet Application Server caches a new result every time the AppLogic runs with a different value for the Department parameter.

Matching Value

Use this to cache a single result for a given value of a parameter. For example:

```
SetCacheCriteria(3600,1,"Department=Operations");
```

iPlanet Application Server caches only one result from this AppLogic, when the Department parameter is Operations.

List of Values

Use this to cache multiple results for several distinct values of a single parameter. For example:

```
SetCacheCriteria(3600,1,
   "Department=Research | Engineering");
```

iPlanet Application Server caches a maximum of two results from this AppLogic, one for Research and one for Engineering.

Range of Values

Use this to cache multiple results for a continuum of values of a single parameter. For example:

```
SetCacheCriteria(3600,1,"Salary=40000-60000");
```

iPlanet Application Server caches results from this AppLogic whenever the Salary parameter contains a new value between 40000 and 60000, inclusive.

List of Several Criteria

Use this to cache multiple results using the values of several parameters, up to the total number of parameters accepted by the AppLogic. The list can contain any combination of the previously described types of criteria, separated by commas. iPlanet Application Server caches results when all the criteria in the list are met. For example:

```
SetCacheCriteria(3600,1,
   "Department=Sales,Salary=40000-60000");
```

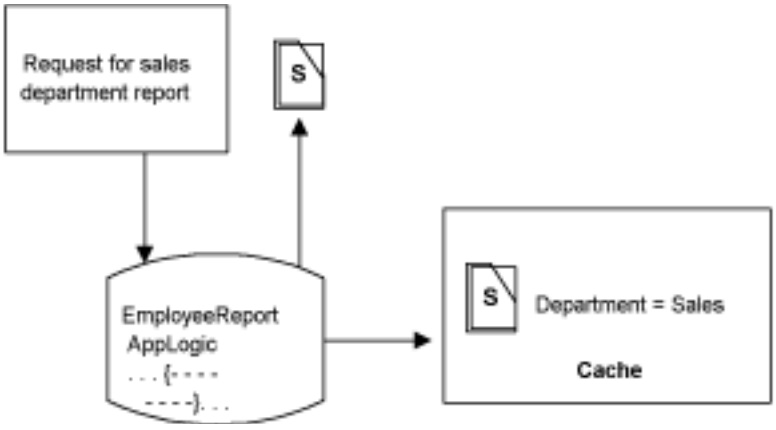
iPlanet Application Server caches results from this AppLogic whenever the Department parameter is Sales *and* the Salary parameter contains a new value between 40000 and 60000, inclusive.

Example

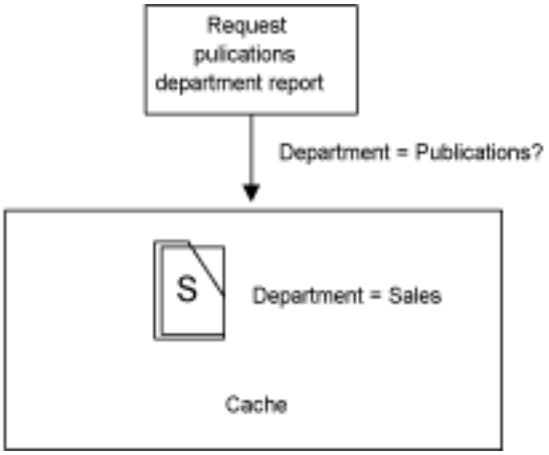
The following code is part of an AppLogic called EmployeeReport. The caching criteria supplied in the SetCacheCriteria( ) call specify that, if the AppLogic's Department parameter is either Sales or Publications, the AppLogic's results are to be cached. The value of the Department parameter is stored along with the cached results.

```
SetCacheCriteria(3600,1,
   "Department=Sales | Publications");
```

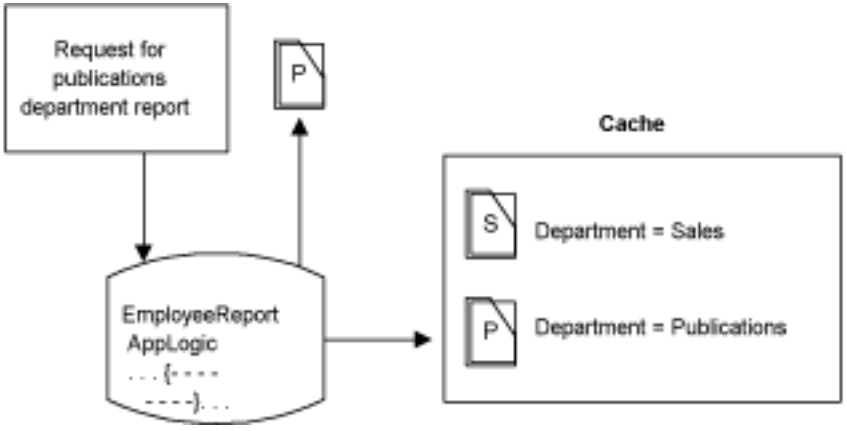
If this AppLogic runs once with a Department parameter value of "Sales", the results are cached, along with the parameter name and value, as shown in the following illustration.



Now suppose the iPlanet Application Server receives another request to run this same AppLogic, but this time the value of Department is "Publications." iPlanet Application Server checks to see whether the value of Department in the request matches the value of Department in the cached result, as shown in the following illustration:



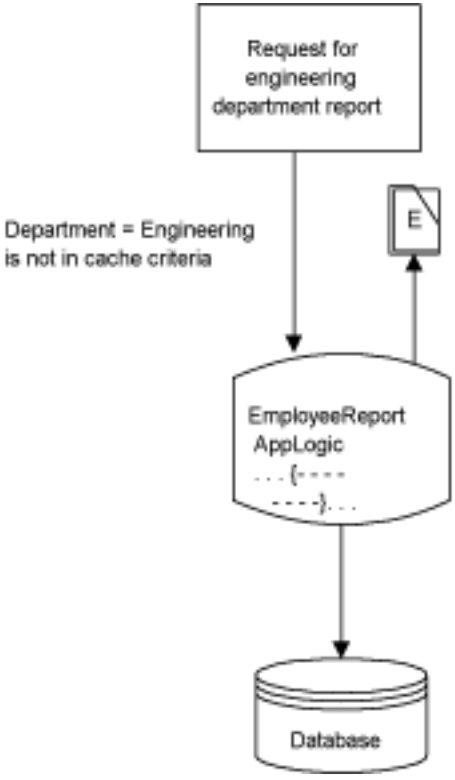
Because the values do not match, the iPlanet Application Server runs the AppLogic again. Because the parameter name and value match the criteria in the SetCacheCriteria( ) call, iPlanet Application Server creates a new entry in the AppLogic's cache and stores the AppLogic's results. iPlanet Application Server also stores the current value (Publications) of the Department parameter. As a result, now two cached results exist for this AppLogic, as shown in the following illustration.



The next time iPlanet Application Server receives a request to run EmployeeReport with a Department parameter of Sales or Publications, it uses the cached results for the appropriate department.

Keep in mind that the Department parameter can have other values besides Sales and Publications, or might be omitted entirely if it is an optional parameter. The AppLogic might also have other parameters. None of these other values or parameters have any bearing on caching, because the cache criteria make no reference to them. Only those parameters and values that are specified in the SetCacheCriteria( ) call, and no others, are used to identify the various cached results for an AppLogic.

For example, suppose an AppLogic request specifies “Engineering” for the Department parameter. The AppLogic runs without caching its results, because this value of the Department parameter does not fall within the caching criteria, as shown in the following illustration.



How to Change Caching Criteria

To change caching criteria, call SetCacheCriteria( ) again. Each subsequent call to SetCacheCriteria( ) supersedes the criteria set in the previous call. If the criteria are changed in the call, the contents of the AppLogic's result cache are discarded, and the new criteria replace the old criteria for subsequent executions of that AppLogic.

For example, suppose your mail-order business puts a certain sweater on sale for a week. In anticipation of increased user interest in the sale item, you could change the cache criteria so that requests for information about that sweater are cached.

How to Remove Cached Results

You can remove some or all of the current contents of the cache without changing the cache criteria. To remove some specific cached results, call RemoveCachedResult( ). For example, the following code removes a cached result for a given AppLogic. The first parameter specifies which AppLogic's results are to be removed, and the second gives the criteria that tell which particular results to remove:

```
hr = RemoveCachedResult(guid, m_pValIn);
```

To clear the cache of all results, call RemoveAllCachedResults( ). For example:

```
hr = RemoveAllCachedResults(guid);
```

Example

The following code removes cached results for a given AppLogic.

```
// Get the input parameter that tells which AppLogic's
// cache is to be flushed
guid = GXGetValListString(m_pValIn, "applogic");


// Get the input parameter that gives the criteria used
// in selecting which result(s) to flush from the cache
LPSTR specific = GXGetValListString(m_pValIn, "specific");


if (specific)
// If result criteria were passed in, flush the
// corresponding results from the cache
```

```
   hr = RemoveCachedResult(guid, m_pValIn);
else
{
   // If no specific cache criteria were passed in,
   // either delete the cache or remove all entries,
   // depending on value of the "delete" parameter
   LPSTR del = GXGetValListString(m_pValIn, "delete");
   if (del)
      hr = DeleteCache(guid);
   else
      hr = RemoveAllCachedResults(guid);
}
```

How to Stop Caching

To stop caching results, call DeleteCache( ). For example:

```
hr = DeleteCache(guid);
```

When this method is called, the current contents of the AppLogic's result cache, if any, are deleted. Unless the AppLogic makes a subsequent call to SetCacheCriteria( ), no further results are cached for this AppLogic.

You can also stop caching temporarily by calling SkipCache( ). The AppLogic results are not cached this time the AppLogic runs, but the cache is not cleared, and when the AppLogic runs again in the future, it will continue to cache results.

Chapter 5

Working with Databases

This chapter describes iPlanet Application Server's interaction with databases, which are used for a wide variety of purposes, such as storing information about users or customers, tracking inventory levels, and recording sales or banking transactions.

The following topics are included in this chapter:

- Introduction to Working with Databases
- About Database Connections
- Getting Information About Columns or Fields
- Inserting Records in a Database
- Updating Records in a Database
- Deleting Records From a Database
- Using Pass-Through Database Commands
- Using Prepared Database Commands
- Using Parameters in Database Commands
- Using Stored Procedures
- Using Triggers
- Using Sequences
- Managing Database Transactions

Introduction to Working with Databases

iPlanet Application Server applications typically interact with databases. For example, an application might perform a database query to retrieve data for a report.

In addition to querying for data, you can perform several other types of database commands. The following list summarizes the operations you can include in an iPlanet Application Server application:

- Queries
- INSERT
- UPDATE
- DELETE
- Any other database command supported by the database server (through the `SetSQL( )` method)

Supported Databases

iPlanet Application Server supports the following databases:

- Oracle
- Sybase
- Informix CLI
- Microsoft SQL Server
- DB2
- ODBC 1.0 and ODBC 2.0 compliant databases

Summary of Database Interaction

The following list summarizes the steps involved in adding database interaction to an iPlanet Application Server application:

To add database interaction to an iPlanet Application Server application

1. Check the user's security level to see whether they have access to the tables and other database objects that the application will reference.

For more information, see , “Writing Secure Applications.”

2. Open a database connection.

   For more information, see “Opening a Database Connection” on page 99.

3. Write the database command (an insert, update, delete, or query operation).

   Several sections in this chapter describe in detail how to write database commands.

4. Run the command.

5. Retrieve and process the results of the command (if any).

About Database Connections

A database connection is a communication link with a database or other data source. Your code can create and manipulate several database connections simultaneously to access data.

Each database connection is represented by a database connection object, which is an instance of the IGXDataConn interface. Each IGXDataConn object contains information such as the user name, password, and any other parameters that are necessary to establish the connection to a particular database.

Opening a Database Connection

Before running a query or another database command, you must open a connection to a data source. To do this, call the CreateDataConn( ) method in the GXAppLogic class. This method instantiates a data connection object. The connection parameters are placed in an IGXValList which is then passed as a parameter to CreateDataConn( ). For example:

```
IGXValList *loginParams;
loginParams = GXCreateValList();

loginParams->SetValString("DSN", "salesDB");
loginParams->SetValString("DB", "salesDB");
loginParams->SetValString("USER", "steve");
loginParams->SetValString("PSWD", "pass7878");
```

```
IGXDataConn *conn;
CreateDataConn(0,GX_DA_DRIVER_ODBC,
               loginParams, NULL, &conn);
```

In this example, the connection uses the following connection parameters:

- Type of driver is ODBC.
- Data source name is `salesDB`.
- Database name is also `salesDB`.
- User name is `steve`.
- Password is `pass7878`.

Specifying Connection Parameters

In the `CreateDataConn( )` method call, the AppLogic specifies which data access driver should be used and lists the connection parameters required by that type of driver. In the example given earlier, the ODBC data access driver is specified. This type of driver requires connection parameters, such as a data source name, user name, and password.

One of the connection parameters is the name of a data source. 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.

Closing a Database Connection

It is not necessary to close a connection object. The Data Access Engine service of iPlanet Application Server manages these connections for you. It will perform any shutdown and cleanup that is necessary.

If you are opening many connections simultaneously, you might want to close one or more of them explicitly in order to free up the connection for others who might be waiting to use it. You can explicitly close a database connection by using `Release( )`. For example:

```
IGXDataConn *conn1;
HRESULT hr = CreateDataConn(0,GX_DA_DRIVER_ODBC,
                            loginParams1, NULL, &conn1);
```

```
IGXDataConn *conn2;
HRESULT hr = CreateDataConn(0,GX_DA_DRIVER_ODBC,
                            loginParams2, NULL, &conn2);
IGXDataConn *conn3;
HRESULT hr = CreateDataConn(0,GX_DA_DRIVER_ODBC,
                            loginParams3, NULL, &conn3);
// ...
// Code using the three connections...
// Finished using connection 1
conn1->Release();
// Continue using connections 2 and 3...
// Release them also when finished.
```

Getting Information About Columns or Fields

When you are working with databases, you often need to refer to columns in a database table or fields in a result set. If you do not know the names of the columns or fields, use one of the following methods:

- For a table, use the `GetColumnByOrd( )` method in the `IGXTable` interface.
- For a flat result set, use the `GetColumnByOrd( )` method in the `IGXResultSet` interface.
- For a hierarchical result set, use the `GetColumnByOrd( )` method in the `IGXHierResultSet` interface.

Each of these methods returns one column object at a time. The column object describes the name, data type, size, and other properties of a column. You can use a loop and an incrementing counter to retrieve all the available columns in a table or result set.

Alternatively, to walk through all the available columns, call `EnumColumns( )` inside a loop to iterate through all the columns in a table or all the fields in a result set's row structure. To find out how many columns or fields there are, call `GetNumColumns( )`. Before the start of the loop, call `EnumColumnReset( )` so that you start at the first column.

Once you have retrieved a column object, you can get more information about it by using the following methods in the `IGXColumn` interface: `GetName( )`, `GetNullsAllowed( )`, `GetPrecision( )`, `GetScale( )`, `GetSize( )`, and `GetType( )`. For more information about these methods, see the iPlanet Application Server Foundation Class Reference.

Example

The following code enumerates columns and constructs a segment of HTML that displays the name and type code of all columns in a database table:

```
hr = table->EnumColumnReset();
while (TRUE)
{
   IGXColumn *column = NULL;
   hr = table->EnumColumns(&column);
   if (hr == NOERROR &&
      column)
   {
      char buffer[256];
      buffer[0] = '\0';
      column->GetName(buffer, sizeof(buffer));
      StreamResult("Column Name = ");
      StreamResult(buffer);
      StreamResult(", ");
      DWORD type;
      type = 0;
      column->GetType(&type);
      sprintf(buffer, "Column Type = %d", type);
      StreamResult(buffer);
      StreamResult("<br>");
      column->Release();
   }
   else
   {
```

```
    // No more columns, exit loop.
    //
    break;
    }
}
```

Inserting Records in a Database

You can write code to insert records, or rows, into tables. To do so, write an INSERT command. An INSERT command is a database command that adds a new row to an existing table.

You can use two techniques to write an INSERT command programmatically:

- Use a series of method calls to write the command. The rest of this section describes how to use this technique.
- Write a SQL INSERT statement and pass it to the `SetSQL( )` method. Use this technique only if you are very familiar with SQL syntax. With this technique, you can allow part of the INSERT command to be set through runtime parameters. For more information, see “Using Pass-Through Database Commands” on page 108.

To insert a row by calling a series of methods

1. Open a connection to a data source.
2. Get a table in the database by calling `GetTable( )`.
3. Create a temporary buffer for data by calling `AllocRow( )`.
4. Set the values of columns by calling the SetValue**( ) methods of the `IGXTable` interface. You must refer to columns by their ordinal position in the table, using `GetColumnOrdinal( )` if necessary.
5. Add the row to the database by calling `AddRow( )`.

Example

The following code, from the Transfer AppLogic in the Online Bank sample application, adds a row to the OBTransaction table for a new transaction:

```
// Get table
IGXTable *pTable=NULL;
```

```
hr=pConn->GetTable("OBTransaction",
   &pTable))==GXE_SUCCESS)&&pTable);
// Look up the column ordinals for the table
ULONG transTypeCol=0; pTable->GetColumnOrdinal("transType",
   &transTypeCol);
ULONG postDateCol=0; pTable->GetColumnOrdinal("postDate",
   &postDateCol);
ULONG acctNumCol=0; pTable->GetColumnOrdinal("acctNum",
   &acctNumCol);
ULONG amountCol=0; pTable->GetColumnOrdinal("amount",
   &amountCol);

// Allocate a new row for the withdrawal
pTable->AllocRow();

// Set values in the row
pTable->SetValueString(acctNumCol, pFromAcct);
pTable->SetValueInt(transTypeCol, OB_TRANSTYPE_WITHDRAWAL);
pTable->SetValueDateString(postDateCol, dateStr);
pTable->SetValueDouble(amountCol, amount*-1.0);

// Perform the insert
pTable->AddRow(0, pTx);
```

Updating Records in a Database

You can write code to modify records, or rows, in tables. To do so, write an UPDATE command. An UPDATE command is a database command that changes the values of one or more columns in one or more existing rows in a table.

You can use two techniques to write an UPDATE command programmatically:

- Use a series of method calls to write the command. The rest of this section describes how to use this technique.
- Write a SQL UPDATE statement and pass it to the `SetSQL( )` method. Use this technique only if you are very familiar with SQL syntax. With this technique, you can allow part of the UPDATE command to be set through runtime parameters. For more information, see "Using Pass-Through Database Commands" on page 108.

To update table rows by calling a series of methods

1. Open a connection to a data source.
2. Get a table in the database by calling `GetTable( )`.
3. Create a temporary buffer for data by calling `AllocRow( )`.
4. Set the values of the columns you want to modify by calling `SetValue**( )` methods of the `IGXTable` interface. You can refer to columns by name or by column number.
5. Specify which row(s) to update and actually update them by calling **`UpdateRow( )`**. Use the same syntax as a SQL WHERE clause to specify which row(s) to update.

   For more information about SQL syntax, see your SQL documentation.

Example

The following code changes the value of the region column from West to East, where the current region is West:

```
// Create a connection
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);
```

```
// Get a table
IGXTable *table = NULL;
hr = conn->GetTable("employees", &table);
// Allocate the row
hr = table->AllocRow();

// Set the new values in the row
ULONG col;
table->GetColumnOrdinal("region", &col);
table->SetValueString(col, "East");

// Perform the update
table->UpdateRow(0, "region='West'", NULL);

// Release resources
table->Release();
conn->Release();
conn_params->Release();
```

Deleting Records From a Database

You can write code to delete records, or rows, from tables. To do so, write a DELETE command. A DELETE command is a database command that removes one or more existing rows from a table.

You can use two techniques to write a DELETE command programmatically:

- Use a series of method calls to write the command. The rest of this section describes how to use this technique.
- Write a SQL DELETE statement and pass it to the `SetSQL( )` method. Use this technique only if you are very familiar with SQL syntax. For more information, see “Using Pass-Through Database Commands” on page 108.

To delete rows by calling a series of methods

1. Open a connection to a data source.
2. Get a table in the database by calling `GetTable( )`.
3. Specify which row(s) to delete and actually remove them by calling `DeleteRow( )`. Use the same syntax as a SQL WHERE clause to specify which row(s) to delete.

   For more information about SQL syntax, see your SQL documentation.

Example

The following code deletes all rows that contain data about sales employees whose last name is Smith:

```
// Create a connection
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

// Get the table
IGXTable *table = NULL;
hr = conn->GetTable("employees", &table);
// Perform the delete
table->DeleteRow(0, "lastname='Smith'", NULL);
// Release resources used
table->Release();
conn->Release();
conn_params->Release();
```

Using Pass-Through Database Commands

A pass-through database command is a statement you write using SQL syntax and pass directly to the database. Use pass-through commands anytime you want to send literal SQL or database server-specific commands to the database. Use this technique only if you are familiar with SQL syntax.

To write a pass-through command

1. Instantiate an `IGXQuery interface` object.

   Query objects are used for all types of pass-through database commands, not just queries.

2. Write a SQL statement and pass it as a parameter to a `SetSQL( )` method call. The statement must not contain a statement termination character at the end. For example, commands passed to Oracle databases must not contain semicolons (;).

   The statement must comply with the requirements of the database driver. For more information about this syntax, refer to your driver or SQL documentation.

To run a pass-through command

1. Open a connection to a data source.
2. If necessary, declare a variable to refer to the output from the command. For example, if the command is a query, you will need a way to refer to the result set:

   ```
   IGXResultSet *rs;
   ```

3. Call the `ExecuteQuery( )` method in the `IGXDataConn interface`. This method runs all types of database commands that are specified using `SetSQL( )`, not just queries. For example:

   ```
   HRESULT hr = conn->ExecuteQuery(0, qry, NULL, NULL, &rs);
   ```

Examples

The following code passes a flat query (SELECT statement) to `SetSQL( )`:

```
// Set the sql string

char sqlStr[400];

sprintf(sqlStr, "SELECT * FROM OBUser, OBCustomer WHERE
OBUser.userName = OBCustomer.userName AND OBCustomer.ssn = '%s'",
pSsn);
```

```
// Pass to SetSQL()
pQuery->SetSQL(sqlStr);
```

The following code passes an INSERT command to SetSQL( ):

```
pUserQuery->SetSQL("INSERT INTO OBUser(userName, password,
   userType, eMail) VALUES (:userName, :password, :userType,
   :eMail)");
```

The following code passes an UPDATE command to SetSQL( ):

```
pUserQuery->SetSQL("UPDATE OBUser SET password = :password,
   eMail = :eMail WHERE userName = :userName");
```

The following code passes a DELETE command to SetSQL( ):

```
pQuery->SetSQL("DELETE FROM OBUser WHERE userName =
   :userName");
```

Using Prepared Database Commands

When you run a database command, the database engine performs certain routine tasks such as determining efficient paths to the data referenced in the command. These tasks become repetitive when running the same command multiple times.

A prepared command is a database command that is compiled by the database. By preparing the command, you specify that the database need only perform these compilation tasks once for the command. The state of the command is saved after these tasks are done. From this point on, whenever the command runs, it has a head start. Without prepared commands, the database engine must recompile each command every time you run it, which is less efficient.

The iPlanet Application Server supports prepared commands through the `IGXPreparedQuery` interface.

To run a prepared database command

Before preparing a database command, you must write it using the techniques described elsewhere in this chapter. Then perform the following steps:

1. Open a connection to a data source.

2. Pre-compile the command by declaring a variable of type IGXPreparedQuery and calling the `PrepareQuery( )` method in the `IGXDataConn` interface. This method prepares all types of database commands, not just queries. For example:

```
IGXPreparedQuery *pPQuery=NULL;

hr=pConn->PrepareQuery(0, pQuery, NULL, NULL, &pPQuery);
```

3. If the command contains parameters, instantiate an IGXValList object and use `SetValString( )` to set the parameter values you want to pass into the command. For example:

```
IGXValList *pList=GXCreateValList();

GXSetValListString(pList, ":userName", pUserName);
```

4. Run the command by calling the prepared query object's `Execute( )` method. Pass the IGXValList object, if any, as a parameter. For example:

```
IGXResultSet *pRset=NULL;

hr=pPQuery->Execute(0, pList, NULL, NULL, &pRset);

pRset->Release();
```

Note that, when Sybase prepared statements are executed, either an empty result set is returned (through the `Execute( )` method of `IGXCallableStmt`) or a null pointer for a result set is returned. The application writer is required to test for both cases. This behavior is different from that of other databases in that they return an empty result set.

Also, when using prepared queries with the Sybase native driver, problems might be experienced when inserting a column of type money. The money type is not supported by the Sybase native driver. As a workaround, use a fixed insert statement with the method `ExecuteQuery( )`.

Using Parameters in Database Commands

When you are writing a database command, you can use parameter markers instead of values for parts of the command. This technique is useful for making commands more flexible and reusable.

For example, a static INSERT command is of limited usefulness. More typically, you use parameters to set up a prepared INSERT, then run the command in a loop, passing in a different IGXValList object each time.

You can use parameters in flat queries, query files, or SQL commands that you pass to `SetSQL( )`. You cannot use parameters when using method calls such as `AddRow( )` and `DeleteRow( )` to specify INSERT, UPDATE, or DELETE commands.

Before you run a database command that contains parameters, you must first prepare it, as described in “Using Prepared Database Commands” on page 109. Then, when you run the command, you pass an IGXValList object to it. Each item in the list corresponds to one of the parameters. iPlanet Application Server replaces each parameter name with a value from the `IGXValList` object.

To place a parameter in a database command, use one of the following types of parameter markers:

- A question mark (?). When you use this technique, the names (also called keys) of the items in the `IGXValList` object must consist of numbers, which can optionally be preceded by colons ("1", "2", or ":1", ":2", etc.). When you run the command, iPlanet Application Server substitutes values into the command in the order in which they are numbered in the `IGXValList` object.
- A name or number preceded by a colon (":1", ":2", ":city", ":max", etc.). When you run the command, iPlanet Application Server matches the parameter names to the names of the items in the `IGXValList` object. This technique gives you more control over the order in which the parameters appear in the command. Even if you use numbers in this technique, they need not appear in order in the command.

You can include up to 1,024 parameters in a single database command, unless your database software allows fewer parameters than this.

Examples

Suppose the code for a DELETE command contains the following line:

```
deleteCmd->SetSQL("DELETE FROM products WHERE color = :1");
```

The following `IGXValList` object sets the parameter value to be passed into the command:

```
IGXValList *pList=GXCreateValList();

GXSetValListString(pList, ":1", "green");
```

The same `IGXValList` object could be used for the following DELETE command:

```
deleteCmd->SetSQL("DELETE FROM products WHERE color = ?");
```

The next example uses code from the OBLogin AppLogic in the Online Bank sample application. It shows a prepared flat query with two parameters. The parameter values for the query are obtained from input parameters, which are passed into the AppLogic from user input on the login form.

```
// Get input parameters
LPSTR userName=GXGetValListString(m_pValIn, "userName");
LPSTR password=GXGetValListString(m_pValIn, "password");

// Write query
IGXQuery *pQuery=NULL;
hr=CreateQuery(&pQuery);
pQuery->SetTables("OBUser, OBCustomer");
pQuery->SetFields("OBUser.userName, userType, ssn, lastName,
   firstName");
pQuery->SetWhere("OBUser.userName *= OBCustomer.userName AND
   OBUser.userName= :userName AND password= :password");

// Prepare the query
IGXPreparedQuery *pPrepQuery=NULL;
hr=pConn->PrepareQuery(0, pQuery, NULL, NULL,
   &pPrepQuery);
IGXValList *pList=GXCreateValList();
GXSetValListString(pList, ":userName", userName);
GXSetValListString(pList, ":password", password);
// Run the query
IGXResultSet *pResultSet=NULL;
hr=pPrepQuery->Execute(0, pList, NULL, NULL,
   &pResultSet);
```

Parts of Syntax in Which Parameters are Not Allowed

You cannot use parameters in a hierarchical query that is written using method calls. That is, none of the flat queries that make up the hierarchical query can contain parameters. However, you can use parameters in a hierarchical query that is written in a query file (`.gxq`).

Outside of hierarchical queries, you can use parameters to replace any single word or value in a database command, except for the following parts of SQL command syntax:

- Field name in a SELECT list.
- Operand of a unary + or - operation.
- Argument of a SET operation.
- Both expressions in a comparison predicate. Only one expression in a comparison can be a parameter.
- Both operands of a binary operator. Only one operand can be a parameter.
- Both the first and second operands of a BETWEEN operation. Either the first or second operand can be a parameter, but not both.
- Both the first and third operands of a BETWEEN operation. Either the first or third operand can be a parameter, but not both.
- Both the expression and the first value of an IN operation. Either the expression or the first value can be a parameter, but not both.

For more information about the items referred to in this list, see your SQL documentation.

Using Parameters in a Flat Query

iPlanet Application Server provides three techniques for writing flat queries:

- Write the query in SQL and pass it to the `SetSQL( )` method.
- Use a series of method calls to set up the query clauses.
- Write the query in a query file.

You can use parameters no matter which technique you use to write the query. However, if the flat query is to be included in a hierarchical query, it cannot contain parameters.

For more information about hierarchical queries, see , “Types of Queries.”

Example

The following code shows a flat query with one parameter, :1, which is a placeholder for a minimum salary value:

```
IGXQuery *qry;
CreateQuery(&qry);
qry->SetTables("employee");
qry->SetFields("empSalary, empName");
qry->SetWhere("empSalary > :1");
```

Using Parameters in an INSERT, UPDATE, or DELETE Command

iPlanet Application Server provides two techniques for writing INSERT, UPDATE, and DELETE commands:

- Write the command in SQL and pass it to the `SetSQL( )` method. This is the only technique in which you can use parameters.

  For more information, see “Using Pass-Through Database Commands” on page 108.

- Use a series of method calls. You cannot use parameters with this technique.

Examples

The following code shows an INSERT command with four parameters for the values to be placed in a new row:

```
pUserQuery->SetSQL("INSERT INTO OBUser(userName, password,
   userType, eMail) VALUES (:userName, :password, :userType,
   :eMail)");
```

The following code shows an UPDATE command with three parameters:

```
pUserQuery->SetSQL("UPDATE OBUser SET password = :password,
   eMail = :eMail WHERE userName = :userName");
```

The following code shows a DELETE command with a parameter for the user name:

```
pQuery->SetSQL("DELETE FROM OBUser WHERE userName =
   :userName");
```

Using Stored Procedures

A stored procedure is a block of statements written in SQL or programmatic SQL and stored in a database. You can use stored procedures to perform any type of database operation, such as modifying, inserting, or deleting records. The use of stored procedures improves database performance by reducing the amount of information that is sent over a network.

AppLogics can call stored procedures by using the `IGXCallableStmt` interface. The `IGXCallableStmt` interface provides a standard way to call stored procedures in any database server. Methods in the `IGXCallableStmt` interface let you

- execute a stored procedure
- pass parameter values to the stored procedure, if required
- retrieve values of the stored procedure's out parameters and return value, if any

Getting the Return Value of a Stored Function

Some stored procedures are functions with return values, and others are procedures with no return value. You run these two types of stored procedures using slightly different syntax with the `SetSQL( )` method. When the stored procedure is a function, you can use the following syntax to get the return value of the function:

```
q->SetSQL("{:ret = call func(:arg1, :arg2, ...)}");
```

When the stored procedure is not a function, or when you do not care about the return value, omit the `:ret =` portion of the syntax. If the return value syntax is used, the space between the return-value parameter and the equal sign is required.

Be careful to use this syntax only with functions. If the stored procedure is not defined as a function in the database, and has no return value, a runtime error occurs.

Creating a Stored Procedure

To write a stored procedure and store it in your database, you can use the techniques supported by your database software. You can also write the stored procedure from code in your application.

To create a stored procedure in code

1. Open a connection to the data source. For example:

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
&conn);
```

2. Write a stored procedure and store it in the database using `SetSQL( )`. For example:

```
LPSTR myStoreP;
myStoreP =
"create procedure myProc1 "
"(v1land in int, v2pop out int)"
" as"
" begin"
" select COUNTIES.POP into v2pop from COUNTIES"
```

```
    " where COUNTIES.LAND = v1land;"
    " end;";

    IGXQuery *qry = NULL;
    hr = CreateQuery(&qry);
    qry->SetSQL(myStoreP);

    // Run the query to store the procedure in the database
    IGXResultSet *rs = NULL;
    hr = conn->ExecuteQuery(0, qry, NULL, NULL, &rs);
```

Running a Stored Procedure

After writing and storing a procedure in the database, you can run it using the following steps:

To run a stored procedure

1. Open a connection to the data source. For example:

```
    IGXValList *conn_params;
    conn_params = GXCreateValList();
    conn_params->SetValString("DSN",  "salesDB");
    conn_params->SetValString("DB",   "salesDB");
    conn_params->SetValString("USER", "steve");
    conn_params->SetValString("PSWD", "pass7878");
    IGXDataConn *conn = NULL;
    HRESULT hr;
    hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
&conn);
```

2. Write a pass-through database command that calls the stored procedure. In the SQL statement, use the following syntax:
   - The `call` keyword to call the stored procedure.
   - The `:ret` keyword to get the stored function's return value, if needed.

- If the stored procedure accepts parameters, use the following conventions: When accessing a stored procedure on Sybase or MS SQL Server, input parameter names specified in the call must be prefixed with the ampersand (&) character, for example, &param1. Other database drivers accept the ampersand, as well as, the colon (:) character. For all database drivers, input/output and output parameter names are prefixed with the colon (:) character, for example, :param2.

The following is an example of how you call a stored procedure:

```
IGXQuery *qry = NULL;

hr = CreateQuery(&qry);

qry->SetSQL("{:ret = call myFunction(&param1)}");
```

3. Prepare a callable statement. For example:

```
IGXCallableStmt *s;

HRESULT hr = conn->PrepareCall(0, qry, null, null, &s);
```

4. If the stored procedure requires parameters, set up an IGXValList object with the parameters. If the stored procedure is a function, you must set the return-value parameter `:ret` to some initial value in this IGXValList object. For example:

```
IGXValList *params;

params = GXCreateValList();

params->SetValInt(":ret", 9999);

params->SetValInt("&param1", 20);
```

5. To execute the stored procedure, run the callable statement object's `Execute( )` method. For example:

```
IGXResultSet *rs = NULL;

hr = s->Execute(0, params, NULL, NULL, &rs);
```

6. If the stored procedure's output parameters are of interest to you, use the `GetParams( )` method to retrieve them. For example:

```
IGXValList *paramsOut = NULL;

hr = s->GetParams(0, &paramsOut);
```

For Informix stored procedures, output parameters are returned in the result set that `Execute( )` returns.

7. Free the resources used. For example:

```
qry->Release();
s->Release();
params->Release();
paramsOut->Release();
resultSetOut->Release();
```

Supported Stored Procedure Operations

Different databases provide different support for stored procedure operations, such as retrieving output parameters, and getting the return value of a stored function. Similarly, the iPlanet Application Server supports some of these operations differently for different databases.

The following table lists some of the common operations and indicates if the operation is supported for each database. If an operation is not supported, either the database does not provide the support or the iPlanet Application Server does not.

Operation	Oracle	Informix	DB2	Sybase & MS SQL Server
Retrieve output parameters using GetParams( )	Yes	No	Yes	Yes
Retrieve output parameters through a result set returned by Execute( )	No	Yes	Yes	Yes
Retrieve multiple result sets	No	No	No	No
Retrieve a single return value from a stored function	Yes	No	No	No
Get the stored procedure's code execution status	Yes	Yes	Yes	Yes
Get the error code returned by the stored procedure	Yes	Yes	Yes	Yes

Note that the following types of DB2 stored procedures cannot be executed:

- Where OUT parameters come before all IN parameters

- Where INOUT parameters come before OUT parameters

The stored procedures need to pass parameters in the correct sequence. To preserve preexisting stored procedures, write wrapper stored procedures that take these parameters in the correct order. Note that this parameter sequence problem is not seen with MS SQL server, but can happen with the other ODBC-compliant data sources.

Sample Stored Procedure

The following code writes and runs a stored procedure:

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

LPSTR myStoreP;
myStoreP =
"create procedure myProc1 "
"(v1land in int, v2pop out int)"
" as"
" begin"
" select COUNTIES.POP into v2pop from COUNTIES"
" where COUNTIES.LAND = v1land;"
" end;";

IGXQuery *qry = NULL;
```

```
hr = CreateQuery(&qry);
qry->SetSQL(myStoreP);

// Run qry to store the procedure in the database
IGXResultSet *rs = NULL;
hr = conn->ExecuteQuery(0, qry, NULL, NULL, &rs);

rs->Release();
qry->Release();
conn->Release();
conn_params->Release();

// Now write a new query to run the stored procedure
IGXQuery *qry = NULL;
hr = CreateQuery(&qry);
qry->SetSQL("{:ret = call myFunction(&param1)}");
IGXCallableStmt *s = NULL;
hr = conn->PrepareCall(0, qry, NULL, NULL, &s);

// Set up parameters for the stored procedure
IGXValList *params;
params = GXCreateValList();
params->SetValInt(":ret", 9999);
params->SetValInt("&param1", 20);

// Run the stored procedure
IGXResultSet *rs = NULL;
hr = s->Execute(0, params, NULL, NULL, &rs);

// This output vallist contains the output values.
IGXValList *paramsOut = NULL;
```

```
hr = s->GetParams(0, &paramsOut);// More code to use the results of
the procedure ...
qry->Release();
s->Release();
params->Release();
paramsOut->Release();
resultSetOut->Release();
```

Using Triggers

A trigger is a stored block of SQL or programmatic SQL statements with the following characteristics:

- It is associated with a table.
- It runs in response to an INSERT, UPDATE, or DELETE operation.
- It runs only under certain specified conditions.

For example, you can set a trigger that runs whenever an UPDATE command is executed in a particular table, with the additional condition that the data being written into a certain field is NULL. If the user attempts to insert a NULL value in the field, the trigger runs and displays an error message or takes other remedial action.

Creating a Trigger

For each trigger, you specify the following characteristics:

- The database table with which it is associated
- The name of the trigger
- The condition that determines when the trigger is executed
- Which type of command activates the trigger (INSERT, UPDATE, or DELETE)
- What action occurs when the trigger is activated (specified in SQL)

To create a trigger

1. Open a connection to a data source. For example:

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);
```

2. Set up the trigger by calling `CreateTrigger( )`. For example:

```
hr = conn->CreateTrigger("employees", "ProcessNew",
"FOR EACH ROW WHEN(title='Director')",
"AFTER INSERT", sqlString);
```

3. If your database requires it, call `EnableTrigger( )`. For example:

```
conn->EnableTrigger("employees", "ProcessNew");
```

Disabling and Enabling Triggers

To temporarily stop the trigger from executing, call `DisableTrigger( )`. To re-enable the trigger, call `EnableTrigger( )`. For example:

```
conn->DisableTrigger("employees", "ProcessNew");
// Perform tasks without risk of executing the trigger.
// ...
conn->EnableTrigger("employees", "ProcessNew");
conn->Release();
```

In some cases the enable and disable commands might enable or disable all triggers that are defined on a certain table, not just the named trigger. For example, Oracle databases behave in this manner.

Deleting a Trigger

To remove a trigger from the database permanently, call DropTrigger( ). For example:

```
conn->DropTrigger("employees", "ProcessNew");
```

Using Sequences

A sequence is a sequential number generator which exists in a database. Some database vendors refer to a sequence as a serial, identity, or autoincrement.

A sequence is useful for generating transaction-safe numbers for database transaction applications. A single application can use several sequences to generate incremental numbers for various purposes. In some cases, you use sequences to generate numbers that are guaranteed to be unique, rather than being concerned with the order of the numbers as such.

For example, you might have an online catalog application through which customers can purchase products. When customers access the application for the first time, you assign each customer a unique, incremental ID number. You can use a sequence to generate this number. In addition, you might want to generate a unique, incremental purchase order number for each customer order. You would create another sequence to generate this number.

iPlanet Application Server supports creating and using sequences with applications through the following parts of the iPlanet Application Server Foundation Class Library:

- IGXSequence interface
- Sequence methods in the IGXDataConn interface

Creating a New Sequence

For each sequence, you specify the following characteristics:

- A name
- The corresponding column in the database (if the database implements sequences as autoincrement fields, rather than as separate objects)
- Starting value

- Increment interval
- Additional database-specific options, if any

To create a new sequence

1. Open a connection to the data source. For example:

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);
```

2. Set up the sequence by calling `CreateSequence( )`. For example:

```
IGXSequenceMgr *seqmgr;
hr = conn->QueryInterface(IID_IGXSequenceMgr,
   (LPVOID *) &seqmgr);
IGXSequence *seq = NULL;
hr = seqmgr->CreateSequence("mySeq", "orders.ID", 100, 1,
   NULL, &seq);
```

   In databases that do not support autoincrement fields, the second parameter to `CreateSequence( )` is null. For example, in an Oracle database, a sequence is implemented as an object rather than as a field in a table.

3. To start the sequence, call `GetNext( )`. For example:

```
DWORD seqVal = 0;
hr = seq->GetNext(&seqVal);
```

4. Use the sequence number to perform the task for which you created it. For example, use the sequence in an INSERT statement:

```
IGXQuery *qry;
CreateQuery(&qry);
```

```
char tmp[512];
sprintf(tmp, "INSERT into orders (ID) values (%d), (cust)"
   "values (%s)",
   seqVal,
   custName);
qry->SetSQL(tmp);
```

Using An Existing Sequence

After setting up a sequence, you can get access to it as follows.

To use an existing sequence

1. Open a connection to the data source. For example:

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);
```

2. Retrieve a reference to the sequence by calling `GetSequence( )` in the `IGXSequenceMgr` interface. For example:

```
IGXSequenceMgr *seqmgr;
hr = conn->QueryInterface(IID_IGXSequenceMgr,
   (LPVOID *) &seqmgr);
IGXSequence *seq = NULL;
hr = seqmgr->GetSequence("mySeq", "orders.ID", &seq);
```

3. If you need to find out the current value of the sequence for any reason, call `GetCurrent( )`. For example:

```
int seqVal = seq->GetCurrent();
```

4. To generate the next number in the sequence, call GetNext( ). For example:

```
int seqVal = seq->GetNext();
```

5. Use the sequence number to perform the task for which you created it. For example, use the sequence in an INSERT statement:

```
IGXQuery *qry;
CreateQuery(&qry);
char tmp[512];
sprintf(tmp, "INSERT into orders (ID) values (%d), (cust)"
   "values (%s)",
   seqVal,
   custName);
qry->SetSQL(tmp);
```

Deleting a Sequence

To permanently remove a sequence from the database, call the `Drop( )` method in the `IGXSequence` interface. However, you should exercise caution when using this method. If the database implements the sequence as a field in a table, the call to `Drop( )` will delete the entire table, not just the sequence field. If the database implements the sequence as an object, as does Oracle for example, the call to `Drop( )` deletes only the sequence object.

Typically, once you start a sequence there is no reason to delete it. The sequence is normally used to create a permanent, unique numbering system for data in a database. However, you might use `Drop( )` if you are using the sequence mechanism to generate unique sequential numbers for some short-lived programmatic purpose.

Managing Database Transactions

A database transaction is a set of database commands that succeed or fail as a group. The necessity of grouping commands into a transaction is determined by business logic. For example, when a bank customer moves money from a savings account to a checking account, two operations are involved:

- Deduct the money from the savings account.
- Add the money to the checking account.

If one of these operations is performed without the other, the transaction is not complete and the accounts will not balance. Both operations must succeed for the entire transaction to be correct.

An iPlanet Application Server application can process several transactions simultaneously. Each transaction works with one or more different database connection objects. Each transaction is made up of several method calls, each of which runs a database command. Each method call is associated with its own connection object, so that the transaction can include commands on more than one database.

Transactions are supported through the IGXTrans interface.

Setting Up a Transaction

A transaction is represented by a transaction object, which is passed to several database commands. The commands in a transaction are united by the fact that they all have the same transaction object as a parameter.

To group several database commands into a transaction

1. Open one or more connections to data sources.
2. Instantiate a transaction object. For example:

   ```
   IGXTrans *pTx=NULL;
   hr=CreateTrans(&pTx);
   ```

3. Start the transaction by calling `Begin( )`. For example:

   ```
   pTx->Begin();
   ```

4. Call the methods that are involved in the transaction. To identify which transaction the commands belong to, you pass the transaction object as a parameter to each method. For example:

   ```
   // Set up an INSERT command ...
   // Then call AddRow()
   pTable->AddRow(0, pTx);
   // Set up another INSERT command ...
   // Then call AddRow()
   ```

```
pTable->AddRow(0, pTx);
```

5. To make the changes to the database permanent, call `Commit( )`. For example:

```
pTx->Commit(0, NULL);
pTx->Release();
```

Example

The following code, from the Transfer AppLogic in the Online Bank sample application, sets up a transaction that transfers funds between accounts. The first command in the transaction withdraws funds from one account, and the second commands adds the funds to another account.

```
// Create a transaction
IGXTrans *pTx=NULL;
hr=CreateTrans(&pTx);
pTx->Begin();
// Allocate a new row for the withdrawal half of the
// transaction
pTable->AllocRow();
pTable->SetValueString(acctNumCol, pFromAcct);
pTable->SetValueInt(transTypeCol, OB_TRANSTYPE_WITHDRAWAL);
pTable->SetValueDateString(postDateCol, dateStr);
pTable->SetValueDouble(amountCol, amount*-1.0);

// Add the row using the transaction
if(pTable->AddRow(0, pTx)==GXE_SUCCESS) {

   // Allocate a new row for the deposit half of the
   //transaction
   pTable->AllocRow();
   pTable->SetValueString(acctNumCol, pToAcct);
   pTable->SetValueInt(transTypeCol, OB_TRANSTYPE_DEPOSIT);
   pTable->SetValueDateString(postDateCol, dateStr);
   pTable->SetValueDouble(amountCol, amount);
```

```
   // Add the second row using the transaction
   if(pTable->AddRow(0, pTx)==GXE_SUCCESS)
      // If both commands succeeded, commit all changes
      pTx->Commit(0, NULL);
   else {
   // If the deposit command failed, roll back
      pTx->Rollback();
      HandleOBSystemError("Could not insert transaction");
   }
else {
// If the withdrawal command failed, roll back
   pTx->Rollback();
   HandleOBSystemError("Could not insert transaction");
}
pTx->Release();
```

Committing a Transaction

When a transaction is committed, all the database commands in the transaction are finalized and changes are saved in the database. The transaction overwrites or deletes the data that was in the database previously and was affected by the commands in the transaction.

To commit a transaction, call the `Commit( )` method. For example:

```
IGXTrans *trx;
CreateTrans(&trx);
// ... series of data operations ...
trx->Commit(0, NULL);
trx->Release();
```

Rolling Back aTransaction

When a transaction is rolled back, all the database commands in the transaction are discarded, and any changes are abandoned. The actual data stored in the database remains unchanged by any of the commands in the transaction.

If a database server is interrupted in the middle of a transaction, such as by a power outage, all uncompleted transactions are automatically rolled back by the database engine. You can also roll back a transaction programmatically if you want to abandon the changes that were proposed by the commands in the transaction.

To roll back a transaction, call the `Rollback( )` method.

Example

The following code rolls back a transaction if a test condition is not met:

```
IGXTrans *trx;
CreateTrans(&trx);
// ... series of data operations ...
if (testCondition == 0)
   trx->Commit(0, NULL);
else
   trx->Rollback();
trx->Release();
```

Chapter 6

Querying a Database

This chapter describes queries, which are statements that specify a set of data to be retrieved from a database.

The following topics are included in this chapter:

- Introduction to Queries
- Using Flat Queries
- Using Hierarchical Queries
- Buffering Result Sets From Queries
- Creating Database Reports
- Working with Query Files
- Running Asynchronous Queries

Introduction to Queries

The data that a query retrieves from a database is called a result set.

Typically, the results of a query are displayed in a report. Queries can also be used to dynamically populate forms. For example, you can dynamically populate a list box with selections, such as city names, based on one of the user's previous selections, such as a state name.

Types of Queries

iPlanet Application Server applications can contain two types of queries:

- Flat queries provide tabular result sets.
- Hierarchical queries combine the result sets from several flat queries in a tree structure.

For more information, see “Using Hierarchical Queries” on page 149.

Using Flat Queries

A flat query is the simplest type of query. It is called flat because its result set is not divided into levels or groups, but is simply a raw listing of data values in a tabular format. Every row contains values from the same set of tables and columns in a single database. Relational database users know this type of query as a simple SELECT statement.

CustID	CustName
00345	Allendale, I
00670	Beauchamp,B
01499	Smith, C
01760	Wallaby, A

You can use an individual flat query to retrieve a flat result set. You can also place one or more flat queries inside a hierarchical query when you want to merge the query’s result set with a template to create dynamic output, such as a report.

A flat query is an instance of the IGXQuery interface, and its result set is an instance of the IGXResultSet interface. Flat queries are handled by the Data Access Engine service of iPlanet Application Server.

For more information about the Data Access Engine, see the Administration and Deployment Guide.

Writing Flat Queries

You can use iPlanet Application Builder to create queries quickly, without writing code. For more information, see the .

Alternatively, this section describes how to write queries programmatically if you prefer to write the code yourself. You can use any of the following techniques to write a flat query programmatically:

- Use a series of method calls to write the query. The rest of this section describes how to write a query using this technique.

- Write your own query file.

For more information, see “Working with Query Files” on page 178.

- Write a SQL SELECT statement and pass it to the `SetSQL( )` method. Use this technique only if you are very familiar with SQL syntax.

For more information, see , “Working with Databases.”

Writing a Flat Query with Method Calls

You can specify the clauses of a flat query by calling a series of methods designed for that purpose. The query-writing methods are easy to understand if you are familiar with Structured Query Language (SQL), which is a commonly used language for accessing information in relational database management systems. The method calls correspond very closely to the SELECT, FROM, WHERE, ORDER BY, GROUP BY, and HAVING clauses of a SQL SELECT statement.

When using method calls to write a flat query, you can specify aliases or parameters. An alias provides a more meaningful alternate name for a table, column, or field, and is especially useful in the case of a calculated field. Parameters provide a technique for dynamically modifying the query itself at runtime.

For more information, see “Sample Multi-Child Hierarchical Query” on page 157 and “Using Parameters in Database Commands” on page 110 of , “Working with Databases.”

To write a flat query using method calls:

1. Instantiate the flat query. For example:

```
IGXQuery *pQuery=NULL;

hr = CreateQuery(&pQuery);
```

2. Select tables or views. This step corresponds to the FROM clause in SQL. For example:

```
pQuery->SetTables("OBAccountType, OBAccount, OBCustomer");
```

   For more information, see “Specifying Tables” on page 137.

3. Select columns. This step corresponds to the SELECT clause in SQL. For example:

```
pQuery->SetFields("OBAccountType.acctDesc as
   OBAccountType_acctDesc, OBAccount.acctNum as
   OBAccount_acctNum, OBAccount.balance as
```

```
   OBAccount_balance, OBCustomer.custName as cust");
```

For more information, see “Specifying Columns and Computed Fields” on page 137.

Optional Steps

1. Put conditions on row retrieval. This step corresponds to the WHERE clause in SQL. For example:

   ```
   char tmpStr[300];
   sprintf(tmpStr, "OBAccountType.acctType = OBAccount.acctType
      and OBCustomer.ssn = OBAccount.ssn and (OBCustomer.ssn =
      '%s')", pSsn);
   pQuery->SetWhere(tmpStr);
   ```

   For more information, see “Specifying Conditions on Row Retrieval” on page 139.

2. Specify row sorting in the result set. This step corresponds to the ORDER BY clause in SQL. For example:

   ```
   pQuery->SetOrderBy("OBAccount.acctNum asc");
   ```

   For more information, see “Sorting Data” on page 139.

3. Summarize the data by creating aggregate rows. This step corresponds to the GROUP BY clause in SQL. For example:

   ```
   pQuery->SetGroupBy("cust");
   ```

   For more information, see “Summarizing Data” on page 140.

4. Put conditions on aggregate rows. This step corresponds to the HAVING clause in SQL. For example:

   ```
   pQuery->SetHaving("OBAccount_balance > 100");
   ```

   For more information, see “Specifying Conditions on Aggregate Rows” on page 144.

Example Query Using Methods to Set Clauses

The following code shows a flat query that retrieves product information. The code first instantiates the query, then sets up the query clauses.

```
IGXQuery *qry;
HRESULT hr = CreateQuery(&qry);
qry->SetTables("invoices, products");
```

```
qry->SetFields("invID, invDate, invProd");
qry->SetWhere("invProd=prodName and prodPrice > 10");
qry->SetOrderBy("invProd");
```

Specifying Tables

To specify which database tables or views contain the data you want to retrieve, use the `SetTables( )` method. Every query must contain a call to SetTables( ). For example, the following code specifies two tables, invoices and customers.

```
pQry->SetTables("invoices, customers");
```

If you call `SetTables( )` more than once in the same query, each subsequent call replaces the settings specified in the previous call. To add more tables to a query, list all of them in a single `SetTables( )` call.

You can use the same table several times in a query by using aliases. For example, you might want to use a table containing customer data to obtain both a Bill To address and a Ship To address. Specify an alias name each time you repeat the same table name. For example:

```
pQry->SetTables("customers, customers as cust2");
```

The information in the `SetTables( )` call corresponds to the FROM clause of a SQL SELECT statement. For more information about the FROM clause, refer to your SQL documentation.

Specifying Columns and Computed Fields

To specify which database columns contain the data you want to retrieve, use the `SetFields( )` method. This method call includes a comma-separated list of the columns and computed fields you want to appear in the query's result set. Any column name you use in this method's parameter list must belong to a column in one of the tables you specified in the `SetTables( )` call. If any of the columns contains a BLOB data type, that column must come last in the list.

If the query does not include a call to `SetFields( )`, then the result set includes all fields in the tables listed in the `SetTables( )` call. If you call `SetFields( )` more than once in the same query, each subsequent call replaces the settings specified in the previous call. To add more columns and computed fields to a query, list all of them in a single `SetFields( )` call.

The information in the `SetFields( )` call corresponds to the SELECT clause of a SQL SELECT statement. For more information about the SELECT clause, refer to your SQL documentation.

Specifying Columns

To retrieve data from a column in the database, list the column name in the SetFields( ) method call. For example, the following code lists three column names.

```
pQry->SetFields("invID, invDate, invProd");
```

You can include columns from more than one table in a single SetFields( ) call. For example, the following code lists columns from two tables, invoices and customers.

```
pQry->SetFields("invoices.invDate, customers.custName");
```

To specify that you want to retrieve all columns from all the tables listed in the SetTables( ) call, use the asterisk (*). For example:

```
pQry->SetFields("*");
```

You can specify an alias name for a column. The alias name becomes the name for the corresponding column in the result set. For more information, see “Using Aliases in a Query” on page 145.

Specifying Computed Fields

A computed field is a field in a result set that contains the result of an expression, rather than a value taken directly from a database column. You can specify two types of computed fields:

- Use a mathematical expression to combine the values from several database columns. You can use any mathematical expression allowed by SQL syntax. For example, the following code specifies a computed field that multiplies the values in the ProdPrice and ProdQty columns.

  ```
  pQry->SetFields("ProdID, ProdPrice * ProdQty as Total");
  ```

- Use an aggregate function to summarize the values for a particular column over a group of rows. If you are planning to use a GROUP BY clause in the query, you typically specify one or more computed fields using aggregate functions. The aggregate functions available depend on your database server, but those typically supported are Min( ), Max( ), Count( ), Avg( ), Sum( ), First( ), and Last( ). For example, the following code uses the Sum( ) function to create a computed column with the total of all values in the salary field.

  ```
  pQryCTY->SetFields("city, Sum(salary) as TotalSalaries");
  ```

For more information about the GROUP BY clause and for an example of how aggregate functions work, see “Summarizing Data” on page 140.

In the previous examples, the computed fields are given the aliases Total and TotalSalaries. It is advisable to specify an alias name for every computed field. This ensures that the result set contains a meaningful name for the computed field. However, if you do not specify an alias, you must refer to the field by its ordinal number (position) when subsequently processing data in the result set.

For more information, see “Using Aliases in a Query” on page 145.

Specifying Conditions on Row Retrieval

To specify conditions that must be met by the rows in the database, use the `SetWhere( )` method. For example, rather than retrieving information about all customers, the following code retrieves only information about customers in a certain city.

```
pQry->SetWhere("City='Bombay'");
```

If you call `SetWhere( )` more than once in the same query, each subsequent call replaces the settings specified in the previous call. To add more conditions to a query, list all of them in a single `SetWhere( )` call, separating each condition with AND or OR. For example:

```
pQry->SetWhere("City='Bombay' AND Sector='Northwest'");
```

The information in the `SetWhere( )` call corresponds to the WHERE clause of a SQL SELECT statement. Use the SQL WHERE syntax to specify conditions. The method call can include any valid SQL syntax, including nested SELECT statements. For more information about the WHERE clause, refer to your SQL documentation.

Sorting Data

To change the order of the rows retrieved by the query, use the `SetOrderBy( )` method. In most reports, the data is organized in a particular sequence. This sequence is determined by a sort key, which is a group of one or more column names. Use `SetOrderBy( )` to list the columns and computed fields that you want to use as a sort key.

For example, in a sales report, the data might be sorted alphabetically by city. Within each city, the local customers might be sorted by customer ID. For each customer, in turn, the invoices might be sorted by invoice number. The following code shows how to specify the sort key for this sales report.

```
pQry->SetOrderBy("CityName,CustomerID,InvoiceNum");
```

The sorted data would look something like the following:

```
Atlanta, Customer01, Inv01
```

```
Atlanta, Customer01, Inv02
Atlanta, Customer04, Inv16
Bombay, Customer03, Inv20
Bombay, Customer03, Inv23
Bombay, Customer10, Inv05
```

Any column name or alias you use in the parameter list of `SetOrderBy( )` must belong to one of the columns or computed fields you specified in the SetFields( ) call.

You can sort any column in either ascending (1-100, A-Z) or descending (100-1, Z-A) order. Ascending order is the default. To sort in descending order, add the keyword `Desc` after the column name. For example, the following code sorts rows from a weather database so that the data for the hottest days appears first.

```
pQry->SetOrderBy("Temperature Desc");
```

If you do not call `SetOrderBy( )`, the order of the returned rows is unpredictable. If you call `SetOrderBy( )` more than once in the same query, each subsequent call replaces the settings specified in the previous call. To add more columns to a sort key, list all of them in a comma-separated list in a single `SetOrderBy( )` call.

The information in the `SetOrderBy( )` call corresponds to the ORDER BY clause of a SQL SELECT statement. For more information about the ORDER BY clause, refer to your SQL documentation.

You can use parameters in this method call to allow parts of the clause to be set when the query runs. For more information, see “Using Parameters in Database Commands” on page 110 of , “Working with Databases.”

Summarizing Data

Data can be summarized into aggregate rows. An aggregate row is a single row in a result set that combines the data from a group of database rows. These rows have one or more column values in common.

For example, suppose a customer places several orders over a period of months. In the database, the customer orders table contains one row for each of these orders. In each of these rows, the column CustName has the same value. You can summarize all the rows for a particular customer and create one aggregate row that contains the total number and average dollar amount of all that customer’s orders.

Example Rows Before Summarizing

CustName	OrderID	OrderTotal
Customer1	Order01	200
Customer1	Order20	400
Customer2	Order55	100
Customer2	Order60	300
Customer2	Order75	200

Example Code to Summarize These Rows

```
pQry->SetFields("CustName, Count(OrderID) as Orders,
   Avg(OrderTotal) as AverageAmt");
pQry->SetGroupBy("CustName");
```

Example Aggregate Rows After Summarizing

CustName	Orders	AverageAmt
Customer1	2	300
Customer2	3	200

You can also use aggregate rows to eliminate duplicates. If you do not include any computed fields when you set up the aggregate row, the row will not contain summary data, such as the number of orders and average amount in the previous example. Instead, the row contains only the values in the columns that are the same in several rows, such as the customer names in the previous example.

To summarize rows

1. In the query's `SetFields( )` call, specify one or more columns to define the groups of rows. These are the columns that have common values in several rows, such as CustName in the example earlier in this section. For more information, see "Specifying Columns" on page 138.

2. In the same `SetFields( )` call, you typically specify one or more computed fields using aggregate functions, such as Orders and AverageAmt in the example earlier in this section. This step is optional if you only want to eliminate duplicates, not create summary data. For more information about computed fields, see “Specifying Computed Fields” on page 138.

3. Call the `SetGroupBy( )` method, and list all the column names you specified in step 1. That is, list the column names from the database table, but not the computed field aliases.

If you call `SetGroupBy( )` more than once in the same query, each subsequent call replaces the settings specified in the previous call. To add more columns to the GROUP BY clause, list all of them in a comma-separated list in a single `SetGroupBy( )` call.

The information in the `SetGroupBy( )` call corresponds to the GROUP BY clause of a SQL SELECT statement. For more information about the GROUP BY clause, refer to your SQL documentation.

You can use parameters in this method call to allow parts of the clause to be set when the query runs. For more information, see , “Working with Databases.”

Example

The following code constructs a query with aggregate rows to show the number of invoices submitted by each sales representative. The query returns one aggregate row for each sales representative. The code then constructs a segment of HTML to display the aggregate rows.

```
// Open connection
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);
```

```
// Write query
IGXQuery *qry;
CreateQuery(&qry);
qry->SetTables("invoices");

// Use COUNT() aggregate expression in SetFields()
// and GroupBy() to create aggregate rows
qry->SetFields("salesRep, COUNT(invoiceID) as numInvoices");
qry->SetGroupBy("salesRep");

// Run query
IGXResultSet *rs = NULL;
hr = conn->ExecuteQuery(0, qry, NULL, NULL, &rs);

// Construct report
StreamResult("Sales Report<p>");
ULONG col_salesRep;
rs->GetColumnOrdinal("salesRep", &col_salesRep);
ULONG col_numInvoices;
rs->GetColumnOrdinal("numInvoices", &col_numInvoices);
do
{
   char buffer[512];
   rs->GetValueString(col_salesRep, buffer, sizeof(buffer));
   StreamResult(buffer);
   StreamResult(" ");
   int numInvoices;
   rs->GetValueInt(col_numInvoices, &numInvoices);
   sprintf(buffer, "%d", numInvoices);
   StreamResult(buffer);
   StreamResult("<br>");
```

```
} while (rs->FetchNext() == NOERROR);


// Release resources
rs->Release();
qry->Release();
conn->Release();
conn_params->Release();
```

Specifying Conditions on Aggregate Rows

You can impose two kinds of conditions as part of a query:

- Conditions on the rows retrieved from the database. These conditions are specified in the `SetWhere( )` method. See “Specifying Conditions on Row Retrieval” on page 139.
- Conditions on aggregate rows. These conditions are specified in the `SetHaving( )` method, which is described in this section.

When a query runs, it follows the steps described in the following illustration and list.

1. The query first retrieves rows from the database as specified in the SELECT and FROM clauses. Only rows that meet the conditions in the WHERE clause are retrieved at this point.

2. If the query includes a GROUP BY clause, the query's result set contains aggregate rows. Each aggregate row summarizes the data in several of the rows from the database. For information about how to create aggregate rows, see "Summarizing Data" on page 140.

3. The query finally applies the conditions in the HAVING clause to the aggregate rows.

To specify conditions on aggregate rows, use the `SetHaving( )` method. This method can refer to any fields in the aggregate rows to set up criteria. Before you call `SetHaving( )`, you must call `SetGroupBy( )` to create the aggregate rows. For example:

```
pQry->SetFields("CustName, Avg(OrderTotal) as AverageAmt");
pQry->SetGroupBy("CustName");
pQry->SetHaving("AverageAmt > 1000");
```

If you call `SetHaving( )` more than once in the same query, each subsequent call replaces the settings specified in the previous call. To add more conditions on aggregate rows, list all of them in a comma-separated list in a single `SetHaving( )` call.

The information in the `SetHaving( )` call corresponds to the HAVING clause of a SQL SELECT statement. Use the SQL HAVING syntax to specify conditions. The method call can include any valid SQL syntax, including nested SELECT statements. For more information about the HAVING clause, refer to your SQL documentation.

Using Aliases in a Query

An alias is an alternate name. You can specify aliases when writing a flat query. To specify an alias, use the following syntax:

sourceName as *aliasName*

You can define two types of aliases:

- A table alias, which is specified in the SetTables( ) call, provides an alternate name for a database table. This technique is useful when you want to use the same table more than once in a query, or when you need to refer to two tables that have the same name but belong to different users within a database. You can use a table alias elsewhere in the same query to refer to the table.

- A field alias, which is specified in the SetFields( ) call, provides an alternate name for a database column or computed field in the result set. You can not use these aliases elsewhere in the same query. However, field aliases are useful in other ways. For example, if you do not use an alias for a computed field, you must refer to the field by ordinal number, which is less convenient. You can also use a field alias in the join expression of a hierarchical query.

Example

The following code gives aliases to two tables owned by different database users.

```
pQry->SetTables("jim.accounts as Jim, ann.accounts as Ann");
```

The following code specifies that you can use prodID, which is an alias, instead of using the longer field name invoiceProductID.

```
pQry->SetFields("invoiceProductID as prodID");
```

Aliases are especially useful when you specify a formula or expression in the SetFields( ) method. For example, in the following code, Total is an alias.

```
pQry->SetFields("invoiceID,
   invoiceCount * prodPrice as Total");
```

Each row in the result set from this query contains two fields, invoiceID and Total. The Total field contains the result of the computation invoiceCount * prodPrice.

Running Flat Queries

To run a flat query, call the `ExecuteQuery( )` method in the IGXDataConn interface. You can run a flat query multiple times. Each time you call `ExecuteQuery( )`, the query returns a new result set object, which might contain different data if the contents of the database have changed.

To run a flat query

1. Open a connection to a data source. For example:

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
```

```
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);
```

2. Declare a variable to hold the output result set object. For example:

```
IGXResultSet *rs = NULL;
```

3. Call the ExecuteQuery( )method in the IGXDataConn interface. For example:

```
hr = conn->ExecuteQuery(0, qry, NULL, NULL, &rs);
```

Example

The following code opens a connection, writes a query, and runs the query, with the results being retrieved in a result set called rs.

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

IGXQuery *qry;
CreateQuery(&qry);
qry->SetTables("author");
qry->SetFields("first_name, last_name");

IGXResultSet *rs = NULL;
hr = conn->ExecuteQuery(0, qry, NULL, NULL, &rs);
```

```
// Use resultset here. When done, release the objects
rs->Close(0);
rs->Release();
qry->Release();
conn->Release();
```

Getting Data From a Flat Query's Result Set

When a flat query runs, it returns rows of data in an instance of IGXResultSet interface. In order to use the data from a flat query, you need to iterate through the rows in the result set and retrieve data from each row. To move from row to row, use the `FetchNext( )` method. To retrieve data from a particular row, use the `GetValue**( )` methods in the IGXResultSet interface.

When you are finished using a result set, release it by calling `Release( )`. This method releases the database connection so that it is available for use by other application code. Do not release the result set or close the database connection until you are finished using the result set. Just because the query has run and returned a result set interface, that doesn't mean all the data is there. Typically the result set is buffered, and live database cursors may still be open. Therefore, when you reach the last row in the buffer, the result set object still needs the connection to get the next batch of rows into the buffer.

Example

The following code, from the OBShowTransferPage AppLogic in the Online Bank sample application, copies data from rows in a result set into a template map.

```
char pAcctDesc[200];
char pAcctNum[200];
// Pull the column ordinals for the account description and
// accout num
ULONG acctDescCol=0;
pRset->GetColumnOrdinal("OBAccountType_acctDesc",
   &acctDescCol);
ULONG acctNumCol=0;
pRset->GetColumnOrdinal("OBAccount_acctNum", &acctNumCol);
char tmpStr[300];
```

```
do {
   pRset->GetValueString(acctDescCol, pAcctDesc, 200);
   pRset->GetValueString(acctNumCol, pAcctNum, 200);
   sprintf(tmpStr, "acctDesc=%s;acctNum=%s", pAcctDesc,
      pAcctNum);
   pAcctsTempDB->RowAppend(tmpStr);
   pAcctsTempDB2->RowAppend(tmpStr);
} while(pRset->FetchNext()==GXE_SUCCESS);
```

Using Hierarchical Queries

A hierarchical query is a query that combines one or more flat queries to generate a result set with multiple nested levels of data. A hierarchical query returns data similarly to a database join or nested query, although its output is actually a collection of separate, flat results sets that are related to each other in specified ways. The hierarchical query object in an iPlanet Application Server application is designed to be merged with a template by the Template Engine to produce dynamic output.

Each of the flat queries in a hierarchical query retrieves a different set of data and can even use a different database connection. The flat queries are related to each other in a series of nested levels that compose the hierarchical query.

The outer level of information is called the parent level of information, and the query that retrieves this information is called the parent query. The inner level of information is called the child level of information, and the query that retrieves this information is called the child query. This parent-child relationship can be repeated for any number of levels, as shown in the following illustration:

The parent level of information determines the grouping of information in its child levels. Each child query is run multiple times, once for each row in the parent query's result set, as shown in the following illustration.

Hierarchical queries are used in conjunction with HTML templates to create reports. If the desired output is a tabular report, the hierarchical query contains only one flat query. If the desired output is a grouped report, the hierarchical query contains two or more flat queries. For more information, see “Types of Reports” on page 164.

A hierarchical query is an instance of the IGXHierQuery interface, and its result set is an instance of the IGXHierResultSet interface. For more information, see the iPlanet Application Server Foundation Class Reference.

Example

The following report shows the results of a hierarchical query that contains two flat queries, one for city data and one for employee data. The outer-level rows show city summary data, taken from the result set of the parent query. The inner-level rows show individual employee data. Each set of employee data represents the result set of the child query being run once for the corresponding row from the city query.

```
Berkeley        1000
   Isaac        300
   Ken          700
Paris           600
   Steve        300
   Tim          300
```

The two levels in this example, without data, can be abstractly pictured as follows.

Parent query:	city	SUM(salary)
Child query:	name	salary

Writing Hierarchical Queries

You can use iPlanet Application Builder to create queries quickly, without coding. For more information, see User’s Guide. This section describes how to write hierarchical queries programmatically. Use these techniques if you prefer to write the code yourself.

You can use two techniques to write a hierarchical query programmatically:

- Use a series of method calls to write the query. This section describes how to write a hierarchical query this way.
- Write a query file.

  For more information, see “Working with Query Files” on page 178.

To write a hierarchical query using a series of method calls

1. Write the flat queries that you plan to use in the hierarchical query. For every level of data in a hierarchical query, you need to define one flat query. Each query can have a different database connection.

   For more information, see “Writing Flat Queries” on page 134.

2. Instantiate a hierarchical query object. For example:

```
IGXHierQuery *pHq=NULL;
hr=CreateHierQuery(&pHq);
```

3. Construct the hierarchical query by using the AddQuery( ) method to place the flat queries in relation to each other. Each query you add, except the first, is a child query. There is no practical limit to the number of nested parent-child levels. For example, in the following code, SelCustAccts is the parent and SelAcctTrans is the child:

```
pHq->AddQuery(pQuery, pConn, "SelCustAccts", "", "");
pHq->AddQuery(pQ, pConn, "SelAcctTrans", "SelCustAccts",
   "SelAcctTrans.OBAccount.acctNum =
   'SelCustAccts.OBAccount_acctNum'");
```

In the AddQuery( ) call, you list the flat query object that you are adding, its database connection, and its name. In the case of a child query, you also specify the name of its parent query and a join expression that shows how the two queries are related. In this example, the two queries are joined on the acctNum fields. For more information, see “Joins in Hierarchical Queries” on page 153.

Typically, each parent query has a single child query, so each AddQuery( ) call represents a new level of nesting. For an example of this type of hierarchical query, see “Example Two-Level Hierarchical Query” on page 155.

A parent query can also have several child queries, as shown in the following illustration:

In a multi-child hierarchical query, several AddQuery( ) calls refer to the same parent query. The result is parallel subreports whose results are displayed one after the other. For an example of this type of hierarchical query, see “Sample Multi-Child Hierarchical Query” on page 157.

Joins in Hierarchical Queries

In an iPlanet Application Server application, join syntax is used to connect the flat queries that make up a hierarchical query. The join is specified for every flat query in the hierarchical query except the first, which is the outermost parent query. You specify the join in the last parameter of the AddQuery( ) method.

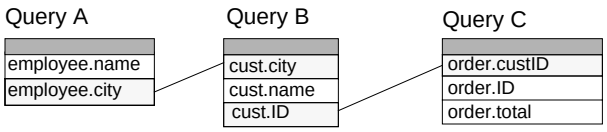
When you write a join expression, you are specifying a relationship between a field in the child query and a field in the parent query. A join expression uses the following syntax:

```
"child.table.col = ']parent.colOrAlias[']"
```

The single quotes in the parent portion of the syntax are required only if the field is a String or Date/Time data type. For example:

```
hq->AddQuery(qryEMP, conn, "EMP", "CTY",
   "EMP.employees.city = 'CTY.city'");
```

The following illustration shows an example of joins in a three-level hierarchical query:



Inter-query joins in a hierarchical query

Avoid queries with more than two or three joins, as this will degrade performance. To improve performance, consider denormalizing the database. Denormalization results in duplicate data in the database, but simplifies queries and improves performance.

Multi-Field Joins

A join expression can set up relationships among multiple fields. This is useful when you cannot set up a unique relationship using a single field.

For example, suppose your database contains information about products from several vendors. Each product has an ID number which is unique for that vendor, but which might be the same as an ID number used by a different vendor for a totally different product. The database might contain the following rows:

VendorID	ProductID	Description
Vendor01	prod1111	Sweater
Vendor02	prod1111	Hard disk

In this case, you need to use both the vendor name and product ID to make a unique join expression.

Sample Hierarchical Query with Multi-field Join

Suppose you have a database with two tables, one for authors (first_name, last_name) and one for books (name, author_first, author_last). You want to construct a report that shows the titles of novels grouped by author. For example:

```
Bill Smith
   No Road Ahead
Sandra Smith
   Sunshine in May
```

```
James Worthington
   King of Hearts
   After the Supper
```

You need a hierarchical query object with two query levels, one for author information and the other for book information. The relationship between the two queries involves multiple fields. Both the first name and last name of the author must be matched, because two authors might have the same last name. The following code shows how to construct a hierarchical query to produce this report:

```
IGXQuery *qryAuth;
CreateQuery(&qryAuth);
qryAuth->SetTables("author");
qryAuth->SetFields("first_name, last_name");

IGXQuery *qryBook;
CreateQuery(&qryBook);
qryBook->SetTables("book");
qryBook->SetFields("name, author_first, author_last");

IGXHierQuery *hqry;
CreateHierQuery(&hqry);
hqry->AddQuery(qryAuth, conn, "AUTHOR", "", "");
hqry->AddQuery(qryBook, conn, "BOOK", "AUTHOR",
   "BOOK.author_firstname = 'AUTHOR.first_name'
   and
   BOOK.author_lastname = 'AUTHOR.last_name'");
```

Example Two-Level Hierarchical Query

The following report shows salary figures for employees in various cities. The total salary amount for each city is shown as summary data next to each city name:

```
Berkeley     100
   Isaac     30
   Ken       70
```

```
Paris        60
   Steve     30
   Kim       30
```

The following code produces the report above:

```
// Begin by defining two flat queries, one for city
// information and the other for employee information.
// The example assumes the database contains an employeee
// table with columns called name, salary, and city.
IGXQuery *qryCTY;
CreateQuery(&qryCTY);
qryCTY->SetTables("employee");
qryCTY->SetFields("city, SUM(salary) as salarysum");
qryCTY->SetGroupBy("city");

IGXQuery *qryEMP;
CreateQuery(&qryEMP);
qryEMP->SetTables("employee");
qryEMP->SetFields("name, salary, city");

// Next, the code instantiates a hierarchical query object.
IGXHierQuery *hqry;
CreateHierQuery(&hqry);

// The following code adds the first flat query to the
// hierarchical query.  The flat query is given the name CTY.
hqry->AddQuery(qryCTY, conn, "CTY", "", "");

// The following code adds the second flat query.  This flat
// query is given the name EMP in the third parameter to
// AddQuery().  The next parameter specifies that the EMP
// query is a child of the CTY query.  That is, results from
```

```
// the EMP query are grouped and nested within results from
// the CTY query.  The last parameter specifies that the two
// flat queries are joined by their city fields.
hqry->AddQuery(qryEMP, conn, "EMP", "CTY",
   "EMP.employees.city = 'CTY.city'");
```

Sample Multi-Child Hierarchical Query

In the following code, the CITY query has two child queries, OFFICE and EMP:

```
hqry->AddQuery(qryCTY, conn, "CITY", "", "");
hqry->AddQuery(qryOFC, conn, "OFFICE", "CITY",
   "OFFICE.cities.city = 'CITY.city'");
hqry->AddQuery(qryEMP, conn, "EMP", "CITY",
   "EMP.employee.city = 'CITY.city'");
```

This query can be used to generate the following type of report. Under each row from the CITY query, the results of the OFFICE query are printed, followed by the results of the EMP query. The two child queries, OFFICE and EMP, are parallel, not grouped or joined to each other. Therefore, the report shows all the offices in a city, followed by all the employees in that city, regardless of which office they work in.

```
San Francisco
   Financial District Office
   Haight Office
   Anderson, M
   Chen, S
   Myers, P
San Jose
   Santa Clara Street Office
   Bellows, R
   Franklin, M
```

Running Hierarchical Queries

Typically, hierarchical queries are constructed in order to be merged with templates, and they are run automatically when you call `EvalTemplate( )` or `EvalOutput( )` to merge the data. You can also run a hierarchical query in a standalone fashion by calling the `Execute( )` method in the IGXHierQuery interface. When you call a hierarchical query in this manner, a hierarchical result set is returned.

To run a hierarchical query without merging it with a template

1. Declare a variable to hold the output result set object. For example:

   ```
   IGXHierResultSet *rs = NULL;
   ```

2. Call the . For example:

   ```
   HRESULT hr = hqry->Execute(0, 0, NULL, &rs);
   ```

Getting Data From a Hierarchical Query's Result Set

To use the data in a hierarchical query's result set, you typically call `EvalOutput( )` or `EvalTemplate( )` to merge the data with a template. However, when you do not wish to return results to the client but instead need to access data in a hierarchical result set programmatically, you can use the methods in the IGXHierResultSet interface.

When you are finished using a result set, release it by calling `Release( )`. This method releases the database connection so that it is available for use by other application code. Do not release the result set or close the database connection until you are finished using the result set. Just because the query has run and returned a result set interface, that doesn't mean all the data is there. Typically the result set is buffered, and live database cursors may still be open. Therefore, when you reach the last row in the buffer, the result set object still needs the connection to get the next batch of rows into the buffer.

Example

The following code example shows processing a hierarchical result set and retrieving fixed length values in the result set rows:

```
STDMETHODIMP

TestGxq::TestResultSetProcessing(IGXHierQuery *pHierQuery)
```

```
{
   HRESULT hr = GXE_SUCCESS;
   char cityName[300];
   cityName[0] = '\0';
   double population = 0;
   ULONG rowNumber = 0;
   IGXHierResultSet *hrs = NULL;
   hr = pHierQuery->Execute(0, 0, NULL, &hrs);
   if ((hr == GXE_SUCCESS) && hrs &&
   (hrs->GetRowNumber("cityQuery", &rowNumber)==GXE_SUCCESS)
   &&
   rowNumber) {
      hrs->GetValue("cityQuery", "NAME", cityName, 300);
      hrs->GetValue("cityQuery", "POP", (LPSTR) &population,
         sizeof(population));
   }
   cout << "City Name: " << cityName << endl;
   cout << "Population: " << population << endl;
   return hr;
}
```

Buffering Result Sets From Queries

You can retrieve result sets from flat or hierarchical queries into a memory buffer. This technique offers the following advantages:

- Backward (as well as forward) movement through the rows.
- Multiple passes through the rows, such as to perform two-pass calculations.

The buffer exists only within the scope of the code that created it. For example, if an AppLogic uses a buffer, the buffer lasts only as long as the AppLogic runs, and is deallocated when the AppLogic returns.

To buffer a result set, you pass a buffering flag and a set of buffer parameters when you run a query using any of the following techniques:

- When calling the ExecuteQuery( ) method in the IGXDataConn interface to run a flat query.
- When calling the Execute( ) method in the IGXPreparedQuery interface to run a prepared query.
- When calling the Execute( ) method in the IGXHierQuery interface to run a hierarchical query.

Avoid buffering too many rows, because buffering can use large amounts of virtual memory. To keep control of the buffer size, use buffer parameters, as described in the next section.

Setting Buffer Parameters

You can customize buffering by setting the following optional parameters:

- Initial number of rows in buffer (`RS_INIT_ROWS`). The default is 10.
- Maximum number of rows in buffer (`RS_MAX_ROWS`). The default is 100.
- Maximum size of the buffer in bytes (`RS_MAX_SIZE`). The default is 12,800.

By specifying a maximum number of rows or byte size for the buffer, you conserve memory space. However, this does not limit your access to the data returned by the query.

For example, you might have a maximum buffer size of 100 rows, but the query might actually return 200 rows. The buffer stores 100 rows to begin with, but if you attempt to access the 101st row by calling `FetchNext( )`, iPlanet Application Server adds another row to the buffer and returns an informational message to let you know that the buffer has been exceeded.

The same holds true if you call `MoveTo( )` and specify any row number greater than the buffer size. iPlanet Application Server adds the necessary number of rows to the buffer.

If you specify both `RS_MAX_ROWS` and `RS_MAX_SIZE`, the actual limit is the smaller of the two settings, that is, whichever is exceeded first.

To buffer a result set

1. Define an IGXValList object that contains an item with the name `RS_BUFFERING` and the value `TRUE`. For example:

   ```
   IGXValList *props;
   ```

```
props = GXCreateValList();
props->SetValString("RS_BUFFERING", "TRUE");
```

2. To set the initial size of the buffer, add an item with the name `RS_INIT_ROWS` and a value that indicates the initial number of rows. For example:

```
props->SetValInt("RS_INIT_ROWS", 50);
```

3. To set the maximum number of rows you want to include in the buffer, add an item with the name `RS_MAX_ROWS`. For example:

```
props->SetValInt("RS_MAX_ROWS", 100);
```

4. To set the maximum size of the buffer in bytes, use an item named `RS_MAX_SIZE`. For example:

```
props->SetValInt("RS_MAX_SIZE", 500000);
```

5. Declare a variable to hold the results of the query. For example, the following code is for a flat query:

```
IGXResultSet *prs;
```

6. Pass the IGXValList object when you run the query. In addition, pass GX_DA_RS_BUFFERING as the flags parameter. The exact text to use for this flag varies depending on the type of query you are running. For example, the following code runs a flat query with buffering:

```
HRESULT hr = conn->ExecuteQuery(GX_DA_RS_BUFFERING, qry,
   null, props, &prs);
```

   The following code runs a prepared query with buffering:

```
HRESULT hr = pqry->Execute(GX_DA_RS_BUFFERING, cmdIn,
   null, null, &prs);
```

Example

The following code shows how to define properties for buffering the result set of a flat query.

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
```

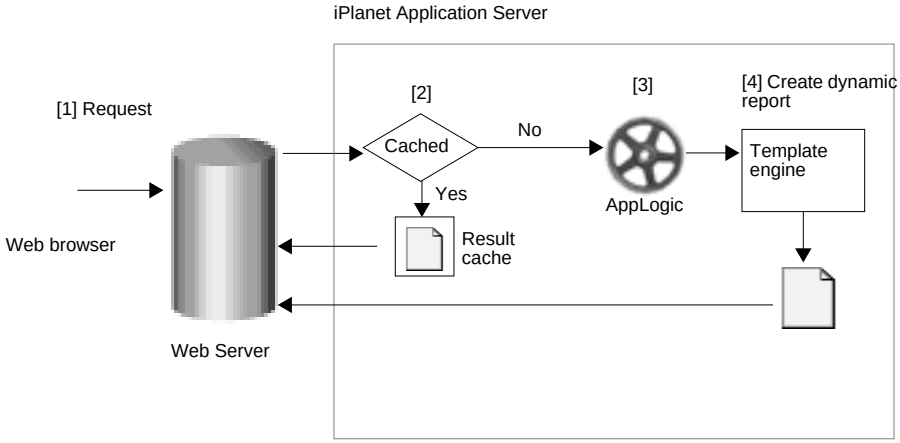
```
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

IGXValList *props;
props = GXCreateValList();
// Turn on result set buffer.
props->SetValString("RS_BUFFERING", "TRUE");
// Specify the maximum number of rows to buffer.
props->SetValInt("RS_MAX_ROWS", 50);

IGXQuery *qry;
CreateQuery(&qry);
// . . . define query properties . . .
// Execute query with result set buffer.
IGXResultSet *rs = NULL;
hr = conn->ExecuteQuery(GX_DA_RS_BUFFERING, qry, NULL,
   props, &rs);

// Use resultset here. When done, release objects.
rs->Close(0);
rs->Release();
qry->Release();
props->Release();
conn->Release();
conn_params->Release();
```

Creating Database Reports

A report is a formatted presentation of data. In an iPlanet Application Server application, a report is an HTML page presented to the user in response to a request for information. In order to create a report, an iPlanet Application Server application combines the following elements:

- A hierarchical result set, which comes from either a hierarchical query or an instance of the `GXTemplateDataBasic` class.
- An HTML template to format and present the data. The HTML template can include images, applets, input fields, buttons, hyperlinks, and any other HTML features. The template also contains special GX markup tags that specify where to merge data from the hierarchical query.
- An AppLogic object that specifies the hierarchical query and runs the report. When the AppLogic runs the report, it specifies which HTML template to use.

The following illustration and list summarize the typical sequence of events in an iPlanet Application Server reporting application.



1. A user requests a report by submitting a form in a Web browser. The request is routed through the Web server to iPlanet Application Server.
2. iPlanet Application Server checks to see whether the report has already been created and cached as a result of a previous request for the same report. If so, iPlanet Application Server returns the cached report, which is routed back through the Web server to the user's Web browser.

3. If the report is not cached, iPlanet Application Server runs the AppLogic identified in the request.
4. The AppLogic calls iPlanet Application Server's Template Engine service, which runs the query (unless you are using a GXTemplateDataBasic object), merges the data with the specified HTML template, and streams the resulting dynamically-generated HTML page to the user's Web browser.

Types of Reports

iPlanet Application Server applications can create both tabular and grouped reports. Tabular reports, sometimes called listings, simply print all the records retrieved from the database. Grouped reports can show the records in logical groups, such as sales grouped by geographic region, with summary data for each group.

Tabular reports and grouped reports both obtain their data from hierarchical queries, but the hierarchical query for a tabular report contains only a single flat query with no join expression. The hierarchical query object is used for tabular reports, even though the data in tabular reports does not appear to be hierarchical. This is because the `EvalTemplate( )` method, which you use to run reports, requires a hierarchical query as a parameter.

Sample Tabular Report

Isaac	300	Berkeley
Julie	300	Paris
Ken	700	Berkeley
Steve	200	Paris

Sample Grouped Report

Berkeley		1000
	Isaac	300
	Ken	700
Paris		500

	Julie	300
	Steve	200

Creating Tabular Reports

To create a tabular report, use the following techniques:

- Write a flat query.

For more information, see “Writing Flat Queries” on page 134.

- Place the flat query in a hierarchical query object, and do not specify a join expression.

For more information about how to construct a hierarchical query, see “Writing Hierarchical Queries” on page 151.

- Write an HTML template that uses a single level of GX `tile` and `cell` markup tags. These tags refer to the fields in the hierarchical query’s result set, and when the report runs they are dynamically replaced with values in the result set.

  For more information, see , “Working with Templates.”

- To add standard headers and footers to the report, you can use the GX `include` tag in the HTML template.

  For more information, see “GX Markup Tag Syntax” on page 193 of , “Types of Queries.”

Creating Grouped Reports

To create a grouped report, you need:

- A hierarchical query or a GXTemplateDataBasic object. Both of these provide data in a hierarchical result set.
- An HTML template. The HTML template contains text and tags that format the data from the hierarchical result set.

For example, in a report that shows sales generated by employees in various cities, one query is required for city data and another query is required for employee data. The two queries are combined into one hierarchical query, and the results are merged with the HTML template to create the output report.

To create a grouped report, use the following techniques:

- Write a hierarchical query or a GXTemplateDataBasic object.

  For more information, see “Writing Hierarchical Queries” on page 151 or “Constructing a Hierarchical Result Set with GXTemplateDataBasic” on page 212 of , “Creating Grouped Reports.”

- Write an HTML template that uses several nested levels of GX `tile` and `cell` markup tags. These tags refer to fields in the hierarchical result set.

  For more information, see , “Working with Templates.”

- To add standard headers and footers to the report, you can use the GX `include` tag in the HTML template.

  For more information, see “GX Markup Tag Syntax” on page 193 of , “Working with Templates.”

Running Reports

Running a report means dynamically creating a new version of the report with current data.

When you run a report, iPlanet Application Server performs the following tasks:

1. If a hierarchical query is used, iPlanet Application Server runs the report’s query to get current data from a database. This step might be skipped if you are caching results, because the iPlanet Application Server will first check to see whether a cached result is available before incurring the overhead of running a query. For more information, see , “Writing Server-Side Application Code.”
2. iPlanet Application Server then merges the data with an HTML template that specifies the report’s appearance.
3. Finally, iPlanet Application Server returns the merged HTML page to the user.

To run a report

You run both tabular and grouped reports using the same technique:

- Call `EvalTemplate( )`. Pass the hierarchical query object (or `GXTemplateDataBasic` object) and the HTML template file name to `EvalTemplate( )` as parameters.

Sample Reports

This section provides annotated examples that show how to work with reports in an iPlanet Application Server application. The examples illustrate how to perform the following tasks:

- Sample Tabular Report
- Sample Grouped Report
- Sample Three-Level Grouped Report

For more complete examples, see the demo applications supplied with your iPlanet Application Server package.

Sample Tabular Report

This example application retrieves data about the salaries of employees and provides the data in a tabular report. The application consists of an AppLogic called TabRept, and an HTML template called tabrept.html.

Tabrept AppLogic Output

The TabRept AppLogic creates a report that looks like the following:

```
Isaac   300
Ken     700
Steve   300
Vasu    300
```

Tabrept AppLogic Code

The following AppLogic generates the employee salary report. The example assumes the data source is an employee table which has name and salary columns.

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "salesDB");
conn_params->SetValString("DB",   "salesDB");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
```

```
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

// Create a flat query to retrieve employee data.
IGXQuery *qry;
CreateQuery(&qry);
qry->SetTables("emp");
qry->SetFields("name, salary");

// Create a hierarchical query to contain the employee
// query so that it can be passed to EvalTemplate().
IGXHierQuery *hqry;
CreateHierQuery(&hqry);

// The query is added to the hierarchical query under
// the name EMP.
hqry->AddQuery(qry, conn, "EMP", "", "");

// Call EvalTemplate to merge the data from the
// hierarchical query with the template and return
// the resulting report HTML page.
EvalTemplate("GXApp/EmpTrack/Templates/tabrept.html",
   hqry, NULL, NULL, NULL);

hqry->Release();
qry->Release();
conn->Release();
conn_params->Release();
```

HTML Template tabrept.html

The following HTML template formats the data in this example report. The template contains a `tile` tag to repeat a set of data for each employee. In this example, comments are provided before each GX markup tag. For information about the other tags in the template, refer to your HTML documentation.

```
<HTML>

<BODY>

<!-- The following GX tag sets up a loop that repeats for each
employee. The type=tile attribute specifies that this is a looping
marker. The id=EMP attribute specifies that the loop repeats for
each row of the result set from the EMP query.

-->

%gx type=tile id=EMP%

<!-- The following GX tags display the name and salary for each
employee. The type=cell attribute specifies that the body text of
each marker is to be replaced with a dynamic data value. The
id=EMP.name and id=EMP.salary attributes specify the fields in the
result set that contain the dynamic values.

-->

   %gx type=cell id=EMP.name%%/gx%

   %gx type=cell id=EMP.salary%%/gx%

   <BR>

%/gx%

</BODY>

</HTML>
```

Sample Grouped Report

This example application retrieves data about the salaries of employees in various cities and presents that data in a grouped report. The application consists of an AppLogic called DoReport and an HTML template called report.html.

DoReport AppLogic Output

The DoReport AppLogic creates a report that looks like the following:

```
Berkeley      1000

   Isaac      300

   Ken        700
```

```
Paris          600
   Steve       300
   Kim         300
```

DoReport AppLogic Code

The following AppLogic generates the city-employee summary report. The example assumes the data source is an employee table which has name, city, and salary columns.

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "emp");
conn_params->SetValString("DB",   "emp");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

// Begin by defining two flat queries, one for city
// information and the other for employee information.
// The example assumes the database contains an employeee
// table with columns called name, salary, and city.
IGXQuery *qryCTY;
CreateQuery(&qryCTY);
qryCTY->SetTables("employee");
qryCTY->SetFields("city, SUM(salary) as salarysum");
qryCTY->SetGroupBy("city");

IGXQuery *qryEMP;
CreateQuery(&qryEMP);
qryEMP->SetTables("employee");
```

```
qryEMP->SetFields("name, salary, city");

// Next, the code instantiates a hierarchical query object.
IGXHierQuery *hqry;
CreateHierQuery(&hqry);

// The following code adds the first flat query to the
// hierarchical query.  The flat query is given the name CTY.
hqry->AddQuery(qryCTY, conn, "CTY", "", "");

// The following code adds the second flat query.  This flat
// query is given the name EMP in the third parameter to
// AddQuery().  The next parameter specifies that the EMP
// query is a child of the CTY query.  That is, results from
// the EMP query are grouped and nested within results from
// the CTY query.  The last parameter specifies that the two
// flat queries are joined by their city fields.
hqry->AddQuery(qryEMP, conn, "EMP", "CTY",
   "EMP.employees.city = 'CTY.city'");

// Call EvalTemplate to merge the data from the
// hierarchical query with the template and return
// the resulting report HTML page.
EvalTemplate("GXApp/EmpTrack/Templates/report.html",
   hqry, NULL, NULL, NULL);

hqry->Release();
qryCTY->Release();
qryEMP->Release();
conn->Release();
conn_params->Release();
```

HTML Template report.html

The following HTML template, report.html, formats the data in this example report. The template contains two nested `tile` tags, one to loop over the cities and the other to loop over the employees within each city. In this example, comments are provided before each GX markup tag. For information about the other tags in the template, refer to your HTML documentation.

```
<HTML>

<BODY>

<!-- The following GX tag sets up a loop that repeats for each city.
The type=tile attribute specifies that this is a looping marker. The
id=CTY attribute specifies that the loop repeats for each row of the
result set from the CTY query.

-->

%gx type=tile id=CTY%

<!-- The following GX tags display the name and total salary figure
for each city. The type=cell attribute specifies that the body text
of each marker is to be replaced with a dynamic data value. The
id=CTY.city and id=CTY.sumsalary attributes specify the fields in
the result set that contain the dynamic values.

-->

%gx type=cell id=CTY.city%%/gx%

%gx type=cell id=CTY.sumsalary%%/gx%

<BR>

<!-- The following GX tags set up a loop that prints the name and
salary of each employee. The tags are similar to those used to print
the city data.

-->

   %gx type=tile id=EMP%

   %gx type=cell id=EMP.name%%/gx%

   %gx type=cell id=EMP.salary%%/gx%

   <BR>

   %/gx%

%/gx%

</BODY>

</HTML>
```

Sample Three-Level Grouped Report

This example retrieves data about the populations of cities from various continents. The report in this example has three levels of nested data. The application consists of one AppLogic called Cities, and an HTML template called cityrept.html.

Cities AppLogic Output

The Cities AppLogic creates a report that looks like the following:

```
Asia
   China
      Bejing     300
      Shanghai   700
   Japan
      Tokyo      250
      Osaka      250
Europe
   France
      Paris      300
      Nice       300
   Spain
      Madrid     200
```

Cities AppLogic Code

To generate the three-level hierarchical report, the AppLogic uses a hierarchical query that consists of three flat queries: one each for continent, country, and city information.

Abstractly, without data, the hierarchical query for this example can be pictured as follows.

```
continent
   country
      city      population
```

The following example code assumes that the database contains the following tables:

- The continents table has a field called `name`.

- The countries table has `name` and `continent` fields.
- The cities table has `city` and `pop` fields.

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "geo");
conn_params->SetValString("DB",   "geo");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

// Specify the three flat queries.
//
IGXQuery *qryCONTINENT;
CreateQuery(&qryCONTINENT);
qryCONTINENT->SetTables("continents");
qryCONTINENT->SetFields("name");

IGXQuery *qryCOUNTRY;
CreateQuery(&qryCOUNTRY);
qryCOUNTRY->SetTables("countries");
qryCOUNTRY->SetFields("name");

IGXQuery *qryCTY;
CreateQuery(&qryCTY);
qryCTY->SetTables("cities");
qryCTY->SetFields("city, pop");
```

```
// Next, the code instantiates a hierarchical query object.
IGXHierQuery *hqry;
CreateHierQuery(&hqry);

// Add the first flat query to the hierarchical query,
// and name the query CONTINENT.
hqry->AddQuery(qryCONTINENT, conn, "CONTINENT", "", "");

// Add the second flat query to the hierarchical query,
// and name the query COUNTRY.  The fourth paramter
// specifies that the COUNTRY query's parent query is
// CONTINENT so that the data for each country is nested
// within the data for the corresponding continent.
// The last parameter is a join expression.  This expression
// specifies that the result sets of the COUNTRY and
// CONTINENT queries are joined on the continent name.
// This means that the value of the continent field in
// the COUNTRY query's result set matches the value of the
// name field in the result set of the parent query,
// CONTINENT.
hqry->AddQuery(qryCOUNTRY, conn, "COUNTRY", "CONTINENT",
   "COUNTRY.countries.continent = 'CONTINENT.name'");

// Add the second flat query to the hierarchical query,
// and name the query CITY.  The fourth parameter specifies
// that the CITY query's parent query is COUNTRY.  The result
// sets of the CITY and COUNTRY queries are joined on the
// country name.  This means that the value of the country
// field in the CITY query's result set matches the value of
// the name field in the result set of the parent query,
// COUNTRY.
```

```
hqry->AddQuery(qryCTY, conn, "CITY", "COUNTRY",
   "CITY.cities.country = 'COUNTRY.name'");

// Call EvalTemplate to merge the data from the
// hierarchical query with the template and return
// the resulting report HTML page.
EvalTemplate("GXApp/EmpTrack/Templates/cityrept.html",
   hqry, NULL, NULL, NULL);

hqry->Release();
qryCTY->Release();
qryCOUNTRY->Release();
qryCONTINENT->Release();
conn->Release();
conn_params->Release();
```

HTML Template cityrept.html

The following HTML template, cityrept.html, formats the data in this example report. The template contains three nested `tile` tags: one to loop over the continents, another to loop over the countries, and a third to loop over the cities within each country. In this example, comments are provided before each GX markup tag. For information about the other tags in the template, refer to your HTML documentation.

```
<HTML>
<BODY>
<!-- The following GX tag sets up a loop that repeats for each
continent. The type=tile attribute specifies that this is a looping
marker. The id=CONTINENT attribute specifies that the loop repeats
for each row of the result set from the CONTINENT query.
-->
%gx type=tile id=CONTINENT%
```

```
<!-- The following GX tag displays the name of each continent. The
type=cell attribute specifies that the body text of each marker is
to be replaced with a dynamic data value. The id=CONTINENT.name
attribute specifies the field in the result set that contains the
dynamic value.

-->

%gx type=cell id=CONTINENT.name%%/gx%

<BR>

<!-- The following GX tags sets up a loop that repeats for each
country within each continent. The tags are similar to those used
for the CONTINENT query.

-->

   %gx type=tile id=COUNTRY%

   %gx type=cell id=COUNTRY.name%%/gx%

   <BR>

<!-- The following GX tags sets up a loop that repeats for each city
within each country. The tags are similar to those used for the
CONTINENT and COUNTRY queries.

-->

   %gx type=tile id=CITY%

      %gx type=cell id=CITY.city%%/gx%

      %gx type=cell id=CITY.pop%%/gx%

      <BR>

      %/gx%

   %/gx%

%/gx%

</BODY>

</HTML>
```

Working with Query Files

A query file is a file is a file with a .gxq extension that contains the specifications for one or more queries. Each query in the file can be flat or hierarchical. Query files are generated automatically when you use iPlanet Application Builder to build queries.

You can also write query files yourself using any text editor. This technique is useful for running SQL SELECT statements that you already have on hand before you begin programming for iPlanet Application Server. You can also write new queries in a query file if you prefer this to using the iPlanet Application Builder or the query-writing method calls provided in the iPlanet Application Server Foundation Class Library.

Writing a Flat Query in a Query File

The specification for a flat query in a query file begins with the following lines:

```
/* optional comments */

query queryName using (driverCode, DSN, UserName) is
```

After these lines, write a SQL SELECT statement (compliant with ANSI SQL89). The statement can include parameters, but do not type any statement terminators. These characters vary depending on your database. For example, the SQL Server statement terminator is GO.

Example

The following lines, from the `SelCustTrans.gxq` file in the Online Bank sample application, specify a flat query named `SelCustAccts`.

```
query SelCustAccts using (ODBC, ksample, kdemo) is

select OBAccountType.acctDesc as OBAccountType_acctDesc
   /* DATATYPE_STRING */,
   OBAccount.acctNum as OBAccount_acctNum
   /* DATATYPE_STRING */,
   OBAccount.balance as OBAccount_balance /* DATATYPE_LONG */
from OBAccount /* (157, 2) */,
   OBAccountType /* (329, 19) */,
   OBCustomer /* (15, 5) */
```

```
where OBAccountType.acctType = OBAccount.acctType
   and OBCustomer.ssn = OBAccount.ssn
   and (OBCustomer.userName = ':userName'
      /* DATATYPE_STRING */ )
order by OBAccount.acctNum asc
```

Running a Flat Query in a Query File

To run a flat query in a query file, you specify the file name and query name, load the file, then run it like any other flat query.

To run a flat query in a query file

1. Declare a variable of type IGXQuery. For example:

   ```
   IGXQuery *qry;
   ```

2. If the query requires parameters, set up an IGXValList object with the parameter values to be passed to `LoadQuery( )`.

   For more information, see , “Working with Databases.”

3. Load the query file into the query object by calling `LoadQuery( )`. For example:

   ```
   LoadQuery("queryFile", query1, 0, params, &qry);
   ```

4. Open a connection to the data source that corresponds to the query you want to run.

5. Declare a variable to reference the output from the query. For example:

   ```
   IGXResultSet *rs;
   ```

6. Call the `ExecuteQuery( )` method in the IGXDataConn interface. For example:

   ```
   HR = conn->ExecuteQuery(0, qry, NULL, NULL, &rs);
   ```

Writing a Hierarchical Query In a Query File

The specification for a hierarchical query in a query file contains several SQL SELECT statements (compliant with ANSI SQL89) with the following additions:

- Each flat 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[']
```

  For more information about the syntax in the where clause, see “Joins in Hierarchical Queries” on page 153.

In a query file, do not type any statement terminators. These characters vary depending on your database. For example, the SQL Server statement terminator is GO.

Example

The following lines, from the SelCustTrans.gxq file in the Online Bank sample application, specify a hierarchical query.

```
/* SelCustAccts:  */
query SelCustAccts using (ODBC, ksample, kdemo) is
select OBAccountType.acctDesc as OBAccountType_acctDesc,
   OBAccount.acctNum as OBAccount_acctNum ,
   OBAccount.balance as OBAccount_balance
from OBAccount, OBAccountType, OBCustomer
where OBAccountType.acctType = OBAccount.acctType
   and OBCustomer.ssn = OBAccount.ssn
   and (OBCustomer.userName = ':userName' )
order by OBAccount.acctNum asc
/* SelAcctTrans:  */
query SelAcctTrans using (ODBC, ksample, kdemo) is
select OBAccount.acctNum as OBAccount_acctNum,
   OBTransaction.postDate as OBTransaction_postDate,
   OBTransactionType.transDesc as
   OBTransactionType_transDesc,
   OBTransaction.amount as OBTransaction_amount
from OBAccount ,
```

```
   OBTransaction,
   OBTransactionType
where OBTransactionType.transType = OBTransaction.transType
   and OBAccount.acctNum = OBTransaction.acctNum
order by OBTransaction.postDate desc

/* Join expression */
join SelAcctTrans to parent SelCustAccts
   where SelAcctTrans.OBAccount.acctNum =
   'SelCustAccts.OBAccount_acctNum'
```

Running a Hierarchical Query in a Query File

To run a hierarchical query in a query file, you specify a set of database connections that are needed by the queries in the file. To do so, you use an instance of the IGXDataConnSet interface. Then you load the file and run it like any other hierarchical query.

To run a hierarchical query file

1. Declare a pointer variable of type IGXHierQuery. For example:

   ```
   IGXHierQuery *hqry;
   ```

2. Declare a pointer variable of type IGXDataConnSet and create a connection set for the connections used by all the queries in the file. For example:

   ```
   IGXDataConnSet *connSet;

   CreateDataConnSet(0, &connSet);
   ```

3. Populate the connection set by calling the AddConn( ) method from the IGXDataConnSet interface. Each call to AddConn( ) specifies a query name and the connection to be used for that query. For example:

   ```
   connSet->AddConn("COUNTIES", conn1);

   connSet->AddConn("STATES", conn2);
   ```

4. If the hierarchical query requires parameters, set up an IGXValList object with the parameter values to be passed to LoadHierQuery( ).

   For more information, see , “Working with Databases.”

5. Load the query file into the hierarchical query object by calling `LoadHierQuery( )`. For example:

   ```
   LoadHierQuery("queryFile", connSet, 0, params, &hqry);
   ```

6. Declare a variable to reference the output from the query. For example:

   ```
   IGXHierResultSet *hrs;
   ```

7. Run the hierarchical query by calling `Execute( )` on the hierarchical query object. For example:

   ```
   hr = hqry->Execute(0, 0, NULL, &hrs);
   ```

   Alternatively, if you want to send the output to the end user, call `EvalOutput( )` or `EvalTemplate( )`. For example:

   ```
   EvalOutput("templateReport", hqry, NULL, NULL, NULL);
   ```

Example

The following code shows how to use a query file to run a hierarchical query.

```
IGXValList *conn_params;
conn_params = GXCreateValList();
conn_params->SetValString("DSN",  "geo");
conn_params->SetValString("DB",   "geo");
conn_params->SetValString("USER", "steve");
conn_params->SetValString("PSWD", "pass7878");
IGXDataConn *conn = NULL;
HRESULT hr;
hr = CreateDataConn(0, GX_DA_DRIVER_ODBC, conn_params, NULL,
   &conn);

IGXDataConnSet *connSet;
CreateDataConnSet(&connSet);
connSet->AddConn("COUNTIES", conn);
connSet->AddConn("STATES", conn);

IGXValList *params;
params = GXCreateValList();
```

```
params->SetValString("pop", "100000");

IGXHierQuery *hqry = NULL;
hr = LoadHierQuery("file.gxq", connSet, 0, params, &hqry);

// Call EvalTemplate to merge the data from the
// hierarchical query with the template and return
// the resulting report HTML page.

EvalTemplate("states_report.html", hqry, NULL, NULL, NULL);
hqry->Release();
```

Running Asynchronous Queries

You can run queries asynchronously so that your application can do other work while the database server is processing the query. Your application can detect when the query is finished so it can process the result set. The iPlanet Application Server supports asynchronous queries through the IGXOrder interface.

To run asynchronous queries

1. Create the query or queries that you want to run asynchronously.
2. Execute each query, passing the GX_DA_EXEC_ASYNC constant to `ExecuteQuery( )`. For example:

   ```
   IGXResultSet *rs0;
   hr = conn0->ExecuteQuery(GX_DA_EXEC_ASYNC,
         qry0, NULL, NULL, &rs0);
   IGXResultSet *rs1;
   hr = conn1->ExecuteQuery(GX_DA_EXEC_ASYNC,
         qry1, NULL, NULL, &rs1);
   ```

3. Declare and allocate an array of IGXOrder pointers, with one array element for each query. For example:

   ```
   IGXOrder *orders[2];
   ```

4. Call `GetOrder( )` in the `IGXResultSet` interface for each result set and assign the result of each call to an element in the array. For example:

```
hr = rs0->GetOrder(&orders[0]);
hr = rs1->GetOrder(&orders[1]);
```

5. Call `GXWaitForOrder( )` to wait for an order to come back, indicating that one of the queries has finished its result set. For example:

```
hr = GXWaitForOrder(orders, 2, &nOrder, m_pContext, 7200);
```

6. Process the result set. If desired, use the array index to determine which result set is finished and perform processing that is specific to that query. For example:

```
if (nOrder == 0) {
   // . . . process result set rs0 . . .
```

7. }Release the resources used. For example:

```
orders[0]->Release();
```

8. Reset the array element that was just processed to null. For example:

```
orders[0] = NULL;
```

Example

The following code uses IGXOrder to track the progress of several asynchronous flat query commands.

```
HRESULT hr;
// Define the flat queries
IGXResultSet *rs0;
hr = conn0->ExecuteQuery(GX_DA_EXEC_ASYNC,
      qry0, NULL, NULL, &rs0);
IGXResultSet *rs1;
hr = conn1->ExecuteQuery(GX_DA_EXEC_ASYNC,
      qry1, NULL, NULL, &rs1);
IGXOrder orders[2];
hr = rs0->GetOrder(&orders[0]);
hr = rs1->GetOrder(&orders[1]);
while(orders[0] != NULL || orders[1] != NULL) {
```

```
   int nOrder;
   hr = GXWaitForOrder(orders, 2, &nOrder, m_pContext, 7200);
   if (nOrder == 0) {
      // . . . process result set rs0 . . .
   }
   else if (nOrder == 1) {
      // . . . process result set rs1 . . .
   }
   else
      //. . . break & return error condition on waitOrders()
orders[0]->Release();
orders[1]->Release();
}
```

Chapter 7

Working with Templates

This chapter describes templates, which are text files that can be merged with dynamic data to produce formatted output.

The following topics are included in this chapter:

- What are Templates?
- How to Write a GXML Template
- How to Write an HTML Template
- Calling an AppLogic Object From an HTML Page
- GX Markup Tag Syntax
- Using a Template Map
- Constructing a Hierarchical Result Set with GXTemplateDataBasic
- Using Conditionals in an HTML Template
- Example HTML Template
- Example GXML Template

What are Templates?

Your application can include two types of templates, depending on the types of results being returned from the AppLogic objects in the application:

- GXML templates are used to return self-describing, formatted data to other AppLogics. GXML templates are used only when you are using the `EvalOutput( )` method to return client-independent results.

- HTML templates are used to return HTML pages to Web browsers. HTML templates are used when you are using the `EvalTemplate( )` method to explicitly return HTML results, or when you are using `EvalOutput( )` to return client-independent results and the client happens to be a Web browser.

What is a GXML Template?

A GXML template is a definition for a dynamically-generated set of output data. GXML templates are made up of special GX markup tags that specify how dynamic data is to be sent back to the client.

AppLogics use GXML templates in conjunction with the `EvalOutput( )` method to return client-independent results. Data retrieved from a database or other data source at runtime is sent back to the client in a self-describing stream of output. This self-describing data stream is created according to the specifications in the GXML template. The client receiving these results then processes the output and puts it to use in reports, calculations, UI controls, or any desired task.

What is an HTML Template?

An HTML template is a definition for a dynamically-generated HTML page. HTML templates are similar to HTML pages, but they also include special GX markup tags that are specific to iPlanet Application Server applications. The GX markup tags in the template specify how dynamic data is merged with the page. Dynamic data is the added feature that makes a page an HTML template rather than just an ordinary, static HTML page.

AppLogic objects use HTML templates to format their output and present dynamically generated HTML pages to a Web browser. Data retrieved from a database or other data source at runtime is merged with the HTML template to create one of the following types of output:

- The data can be displayed in the HTML page to present a database report. For example, a customer might request a report of currently available products and prices.
- The data can be used to dynamically modify the HTML page itself, changing the HTML tags, images, sounds, applets, or other features. For example, you might dynamically change which commands or AppLogic objects the user can choose from next by modifying the returned URL.

An HTML template can contain static elements, such as introductory text and logos. Its GX markup tags provide the placeholders in which a variety of data values can be used.

HTML templates provide a modular technique for designing HTML pages. AppLogics can share the same HTML templates, and the templates can easily be updated or translated and localized without affecting application code and business rules.

Example: Report

In a sales support application, a user's request for customer sales data causes an AppLogic to get the latest data from the database. The AppLogic merges the data with an HTML template and returns a sales report to the user. The sales report is an HTML page with standard headings, graphics, and other elements merged with the dynamic sales data.

Example: Dynamically Modified HTML Tag

An AppLogic queries a geographic database to create a report about the countries of the world. The data includes country names and populations. The country name is used to dynamically modify the filename in an HTML `<img>` tag inside a loop, so that a different illustration is displayed for each country.

Example: Dynamically Populated Form

A chain of hardware stores has an online inquiry application in which users can get information about store locations. The user types a home address into a form, then submits the form. The application looks up the stores in the customer's home town, then uses an HTML template to create a form that contains a list box with the street addresses of the selected stores. The user can then select one store and submit a request for more information about that store.

Parts of an HTML Template

Like any HTML page, an HTML template can include features implemented using the normal HTML markup tags, such as the following:

- Text and graphics
- Interactive features such as buttons and hyperlinks
- Calls to applets

By using the GX markup tags, an HTML template can also include the following:

- Dynamically populated data fields.

For more information, see “Using the Cell Attribute in a GX Markup Tag” on page 199.

- Dynamically modified HTML tags.

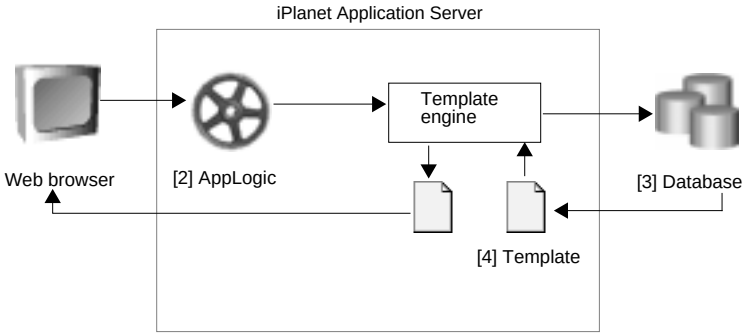
  For more information, see “Using the Replace Attribute in a GX Markup Tag” on page 205.

- Placeholders for function calls or other data.

  For more information, see “Using a Template Map” on page 207.

Runtime Behavior of HTML Templates

The following illustration and list summarize how an HTML template is typically used at runtime.



1. A user makes a request through a Web browser.
2. In response to the request, an AppLogic runs. The AppLogic specifies a query and passes the query and the name of an HTML template file to iPlanet Application Server’s Template Engine.
3. The Template Engine queries the database and retrieves dynamic data.
4. The Template Engine merges the data with the HTML template and streams the resulting dynamically-generated HTML page to the Web browser.

How to Write a GXML Template

To create a GXML template, you can use either of the following techniques:

- Use the `khtml2gxml` tool to convert an existing HTML template into a GXML template. For more information, see “Converting HTML Templates to GXML Templates” on page 191.
- Write the GXML file yourself, using any text editor. To do so, you need to understand the GX markup tags. For more information, see “GX Markup Tag Syntax” on page 193.

By convention, GXML templates are kept in one of the following directories:

- [HTTPDIR]/GXApp/ProjectName/Templates, where [HTTPDIR] is the document root directory of your HTTP server and ProjectName is the name of your application or package.
- [GXINSTALL]/Apps/ProjectName/Templates, where [GXINSTALL] is the directory in which your iPlanet Application Server software is installed.

In addition, administrators can specify a search path for templates. For more information, see the Administration and Deployment Guide.

Converting HTML Templates to GXML Templates

You can convert an existing HTML template into a GXML template by using the command-line utility `khtml2gxml`. This tool removes all HTML from the template, leaving only the GX markup tags.

To convert HTML to GXML

Type the following command at the command line prompt:

```
khtml2gxml filename.html
```

A new GXML template file, `filename.gxml`, is created.

How to Write an HTML Template

To write HTML templates, you can use iPlanet Application Builder. This tool gives you a head start on creating some commonly-used types of templates.

You can also code the HTML tags yourself, using any text editor or HTML authoring tool. In either case, you need to understand the GX markup tags.

For more information, see “GX Markup Tag Syntax” on page 193.

By convention, HTML templates are kept in one of the following directories:

- [HTTPDIR]/GXApp/ProjectName/Templates, where [HTTPDIR] is the document root directory of your HTTP server and ProjectName is the name of your application or package.
- [GXINSTALL]/Apps/ProjectName/Templates, where [GXINSTALL] is the directory in which your iPlanet Application Server software is installed.

Calling an AppLogic Object From an HTML Page

When a user makes an action request from a Web browser, such as by clicking a button on an HTML page, iPlanet Application Server runs the appropriate AppLogic object to handle the request. To create an HTML page that calls an AppLogic, you embed a special URL link in the HTML page. For example, the URL can be activated through a hyperlink or a button. Use the normal HTML technique for linking to a URL, and use a URL with one of the following syntaxes:

To call the AppLogic using a GUID, use the following syntax (all on one line):

```
[http://www.company.com]/cgi-bin/gx.cgi/
   GUIDGX-{E5CA1000-6EEE-11cf-96FD-0020AFED9A65}[?]
   [param=value[&paramN=valueN] ...]
```

To call an AppLogic by name, use the following syntax (all on one line):

```
[http://www.company.com]/cgi-bin/gx.cgi/AppLogicName[?]
   [param=value[&paramN=valueN] ...]
```

- The URL prefix (`http://www.company.com`) is required if the URL is used in a hyperlink.
- The code `cgi-bin/gx.cgi` runs the `gx.cgi` program, which forwards the request to the iPlanet Application Server. The server loads and runs the AppLogic registered with the given name or GUID. If you are using a Netscape Web connector, such as NSAPI or ISAPI, the connector automatically intercepts any gx.cgi requests. Thus you can write your application with a CGI environment and deploy in an NSAPI environment.
- In the first syntax, the `GUIDGX-` prefix indicates that a GUID is the next part of the URL.
- In the first syntax, the last part of the URL, between and including the brace characters { }, is the GUID that identifies which AppLogic to run.

- The question mark is required if parameters are included in the URL.
- You can optionally specify one or more named parameters at the end of the URL.

Use these special URLs throughout your application's HTML interface, wherever you want to run an AppLogic in response to a user clicking a button or other control.

Examples

The following HTML form, from the Online Bank sample application, runs the FindCust AppLogic. The AppLogic request is coded in the `action` attribute. When the user clicks the Search button, the Web browser forwards this request to the Web server. The Web server, in turn, sends the request to iPlanet Application Server. If an AppLogic has been compiled and registered with this name before the request is received, iPlanet Application Server runs that AppLogic.

```
<FORM method="POST" action="/cgi-bin/gx.cgi/AppLogic+FindCust">


<!-- Body of HTML form here -->


<input type="submit" name="go" value="Search">
```

The following portion of an HTML page shows how to call an AppLogic from a hyperlink:

```
<a href="/cgi-bin/gx.cgi/AppLogic+CShowNewCustPage">
Add New Customer </a>
```

GX Markup Tag Syntax

The GX markup tag is a matched tag. Every opening marker, `%gx%`, must be matched by a closing marker, `%/gx%`. The syntax of the GX markup tag is as follows:

```
%gx {TagAttributes} %
   [textBlock]
%/gx%
```

You can also use angle brackets instead of percent signs, as follows:

```
<gx {TagAttributes} >
```

```
   [textBlock]
</gx>
```

TextBlock

The `TextBlock` portion of a GX markup tag can include the following:

- Plain text
- HTML tags
- Nested GX markup tags

For example, in the following GX markup tag, the second line is the text block, including both a nested GX markup tag and an HTML tag (`<br>`):

```
%gx type=tile id=CONTINENT%
   %gx type=cell id=CONTINENT.NAME%%/gx%<br>
%/gx%
```

In the following GX markup tag, the second line is the text block, including both plain text and an HTML tag (`<img>`):

```
%gx type=replace id=CONTINENT.NAME value=PlaceHolder%
   Selected Continent: <img src="images/PlaceHolder.gif">
   %/gx%
```

TagAttributes

The `TagAttributes` portion of a GX markup tag can be a combination of the following items:

- type=TypeCode
- id=Name
- visible={True | False}
- min=MinVal
- max=MaxVal
- value=ReplaceVal

The rest of this section describes each item in more detail. You can specify these items in any order within the GX markup tag.

type=TypeCode

Required. Indicates what action is to be performed for this tag when an AppLogic merges data with the template. The TypeCode is one of the following:

- Cell
- Tile
- Replace
- Include
- User-Defined Tag

Cell

Replaces the entire GX marker, from %gx% to %/gx%, with a dynamic data value. Used with the id attribute, which specifies a field in the result set that contains the dynamic value to be used. For example:

```
%gx type=cell id=CTY.sumsales%%/gx%
```

For a more detailed example, see “Using the Cell Attribute in a GX Markup Tag” on page 199.

Tile

Repeats the TextBlock. The tile tag can be used in two ways: repeating a fixed number of times, or repeating for each row in a result set. When you use nested tile tags along with a hierarchical query, the result is a grouped master-detail report.

To repeat the TextBlock a specified number of times, use the min attribute to set the number. Do not use the id attribute. For example:

```
%gx type=tile min=5% . . . %/gx%
```

To repeat the TextBlock for every row in a query’s result set, use the id attribute to specify the name of the query. A query is given a name when you add a flat query to a hierarchical query, or when you name a query in a query file. For more information about using the id attribute in this way, see “Meaning Of id when Type is Tile” on page 197.

You can also use the max attribute, which specifies a limit on how many times the tile can be repeated. This is useful for limiting the length of the generated HTML page if the size of the result set is potentially large. For example:

```
%gx type=tile id=COUNTRY max=1000% . . . %/gx%
```

For more detailed examples of both techniques, see “Using the Tile Attribute in a GX Markup Tag” on page 203.

Replace

Searches `TextBlock` for a string and substitutes a dynamic data value for that string. Used with the `value` attribute, which specifies the search string, and with the `id` attribute, which specifies where to find the replacement value. For example:

```
%gx type=replace id=CUST.name value=CustName%
   Dear CustName: %/gx%
```

For a more detailed example, see “Using the Replace Attribute in a GX Markup Tag” on page 205.

Include

Replaces `TextBlock` with HTML output created by evaluating another template. Used with the `id` attribute, which specifies the path of the template. For example:

```
%gx type=include
   id="GXApp/OnlineBooks/Templates/header1.html"%
```

For a more detailed example, see “Using the Include Attribute in a GX Markup Tag” on page 206.

Do not specify the same template name as the current template. A template cannot include itself.

User-Defined Tag

Performs a user-defined action. For example, you could write the following GX markup tag:

```
%gx type=trigger name=Clock arg1=hello%%/gx%
```

To use this type of tag, you must write a customized template map class. When the Template Engine encounters an unknown type of GX tag, it calls the template map’s `Get( )` method and passes it the unknown tag. For example, the trigger tag shown above results in the following string being passed to `Get( )`:

```
trigger:name="Clock";arg1="hello"
```

The `Get( )` method in your template map subclass must be able to parse and respond to this string.

For more information about implementing a custom GX markup tag, see “Creating a User-Defined Tag” on page 207.

id=Name

The meaning of the `id` attribute varies depending on the value of the `type` attribute. The `id` attribute is required in all GX markup tags, except when the `min` attribute is used to repeat a `tile` a specified number of times.

Meaning Of id when Type is Tile

When the `type` attribute is `tile`, the `id` attribute specifies the name of a query. The marker's `TextBlock` repeats for the number of rows in the query's result set. For an example, see "Using the Tile Attribute in a GX Markup Tag" on page 203.

The value you can use in the `id` attribute is set when a query is added to a hierarchical query, or when you name a query in a query file. For example, suppose the AppLogic contains the following code, which adds a query named CTY to a hierarchical query:

```
pHq->AddQuery(pQuery, pConn, "CTY", "", "");
```

The HTML template that displays data from this query can use the name CTY in the `id` attribute of a `tile` tag. For example

```
%gx type=tile id=CTY%
```

This is how `tile` tags are normally implemented. However, if you are using the `min` attribute to control the number of times the `tile` repeats, you do not use the `id` attribute.

Meaning of id when Type is Cell or Replace

When the `type` attribute is `cell` or `replace`, the `id` attribute specifies a field in the result set. The field contains the data value to be displayed. If the AppLogic is using a hierarchical query, the field is specified using dot notation. A query name (derived in the same way as that used with the `tile` tag) comes before the dot, and a column name or alias comes after the dot. For example:

```
%gx type=cell id=CTY.sumsales%%/gx%
```

For a more detailed example, see "Using the Cell Attribute in a GX Markup Tag" on page 199.

The `id` attribute in a `cell` tag can include a format string to specify a numeric or character format.

For more information, see "Formatting Data in a Cell Tag" on page 200.

If the data specified in the `id` attribute is not found in the result set, the static text, if any, in the `TextBlock` is displayed instead of the dynamic data that would have replaced it. For example, in the following `tile` tag, the text `Customer name here` appears if the custName field is not found or is empty:

```
%gx type=tile id=CUST%
   %gx type=cell id=CUST.custName%Customer name here
   %/gx%<br>
%/gx%
```

Meaning of id when Type is Include

When the `type` attribute is `include`, the `id` attribute specifies the path of an HTML template. You can use a literal path enclosed in double quotes or a field in a result set. For example:

```
%gx type=include
   id="GXApp/OnlineBooks/Templates/header1.html"%
```

For a more detailed example, see “Using the Include Attribute in a GX Markup Tag” on page 206.

visible={True | False}

Optional. Determines whether the portion of the page enclosed by the GX markup tag appears in the final HTML page that results when an AppLogic merges data with the template. Default is `True`. Set to `False` to hide a marked-off portion of the page, including nested GX tags and dynamic data.

min=MinVal

Optional. Use only when the `type` attribute is `tile`. Specifies the smallest number of times the `tile` can be repeated when an AppLogic merges data with the template. When used alone, specifies a static number of times for the `tile` to repeat. Default is `0`.

max=MaxVal

Optional. Use only when the `type` attribute is `tile`. Specifies the greatest number of times the `tile` can be repeated. Default is 2^{32}. For an example, see “Using the Tile Attribute in a GX Markup Tag” on page 203.

value=ReplaceVal

Required when the `type` attribute is `replace`. Specifies a string to search for in `TextBlock`. This string is replaced with dynamic data when an AppLogic merges data with the template. For an example, see “Using the Replace Attribute in a GX Markup Tag” on page 205.

Using the Cell Attribute in a GX Markup Tag

The `cell` type of GX markup tag is the lowest-level building block in a report-style HTML template. Each `cell` tag, including any `TextBlock`, is replaced by a single value from one row of data from the database. For example, suppose you want to generate a dynamic sales letter from a customer database. Your HTML template could include the following `cell` tag:

```
Dear %gx type=cell id=CustName%

   Customer name here   %/gx%,<br>

Thank you for buying the SmartSurf 2000 Web robot . . .
```

When an AppLogic merges data with this template, the entire text from `%gx%` to `%/gx%`, including `Customer name here,` is replaced with a customer name from the database field CustName. For example:

```
Dear M. Smith,<br>

Thank you for buying the SmartSurf 2000 Web robot . . .
```

Using Cell with Tile

The `cell` tag is commonly used in combination with the `tile` tag, which causes the `cell` to repeat for a group of data rows. The following example contains both `tile` and `cell` tags. The `cell` tag is a placeholder for the individual continent name. The `tile` tag causes the `cell` tag to be repeated for all the continents in the database.

```
%gx type=tile id=CONTINENT%

   %gx type=cell id=CONTINENT.NAME%%/gx%<br>

%/gx%
```

Remember that all text inside the `cell` tag is deleted and replaced with dynamic data. Take care that the `cell` tag does not contain text you want to appear in the finished HTML page. In the previous example, the HTML tag `<br>` is outside the `cell` tag, so it is passed through without change. If the `<br>` were accidentally placed before the `%/gx%` that marks the end of the `cell` tag, the `<br>` would be deleted.

When an AppLogic merges data with this HTML template, text like the following replaces the GX tags:

```
AFRICA<br>

ANTARCTICA<br>

ASIA<br>

AUSTRALIA<br>
```

```
EUROPE<br>
N_AMERICA<br>
OCEANIA<br>
S_AMERICA<br>
```

You can include several `cell` tags within one `tile` tag. For example, the following tags print two fields, the county name and number of customers, for each county in a state. Because the `max` attribute is set, the tag will not generate entries for more than 100 counties in a single state.

```
%gx type=tile id=DETAILS MAX=100%
   %gx type=cell id=DETAILS.COUNTYNAM%%/gx%
   %gx type=cell id=DETAILS.CUSTS%%/gx%<br>
%/gx%
```

When an AppLogic merges data with this HTML template, text like the following replaces the tags:

```
San Mateo  100<br>
Santa Clara  300<br>
Sonoma  400<br>
```

Formatting Data in a Cell Tag

You can include a format specification in the `id` attribute of a `cell` tag. To do so, place the attribute value in quotes, place a comma after the data field name, and call the format( ) function. You use this function to specify a format string, which determines how the data appears in the HTML output page.

For example, the following `cell` tag specifies that the sales figure is to start with a dollar sign, include a thousands separator, and show two digits to the right of the decimal point:

```
%gx type=cell id="CTY.sumsales, format($0,000.00)"%%/gx%
```

The argument to the format( ) function is a format string. This string is made up of ordinary text, such as the dollar sign, and special characters that influence how data is presented. The following tables describe the special characters you can use in a format string. The set of characters you can use varies depending on the type of data. You cannot mix characters from the different types in a single format string.

Numeric Format Characters

Character	Meaning	Example Data	Example String	Example Result
#	Unfilled digit placeholder. Replaced by numeric digits in the data.	679.649	#####.##	679.64
		700	$#,###.00	$700.00
	If the data to format has fewer digits than the format string, the empty places are not filled. The output can be shorter than the format string.			
0	Zero-filled digit placeholder. Replaced by numeric digits in the data.	679.649	0000.00	0679.65
		679.649	00###	00679
	If the data to format has fewer digits than the format string, the empty places are filled with zeros. The output is always at least as long as the format string.	700	$000.00	$700.00
	When placed to right of decimal point, indicates precision. Data is rounded if necessary.			
,	Thousands separator.	1234	0,000	1,234
	A separator character will appear between every three digits to the left of the decimal point in the output data.			
.	Decimal separator.	679.649	000.00	679.65
	A decimal point character will appear between the whole and fractional parts of the output data.	700	$#,###.00	$700.00
;	Separates a pair of formats. The first format is used for positive numbers, the second for negative numbers.	23	##;(##)	23
		-66	##;(##)	(66)

Character	Meaning	Example Data	Example String	Example Result
Literals such as $ + - ( ) and space characters	Any character in the format string that is not a special character will appear in the output data exactly as typed. If using parentheses, always use matched pairs.	5552365	###-####	555-2365
		69100	$0,000	$69,100
		123	0 0 0	1 2 3

Date/Time Format Characters

Character	Meaning	Example Data	Example String	Example Result
D	Day of the month, with no leading zero.	1/1/2020	M-D-YY	1-1-20
DD	Day of the month, with leading zero.	1/1/2020	MM-DD-YY	01-01-20
DDD	Day of the week, abbreviated.	7/4/1996	DDD	Thu
DDDD	Full day of the week.	7/4/1996	DDDD	Thursday
M	Number of the month, with no leading zero.	1/1/2020	M-D-YY	1-1-20
MM	Number of the month, with leading zero.	1/1/2020	MM-DD-YY	01-01-20
MMM	Name of the month, abbreviated.	1/1/2020	MMM	Jan
MMMM	Full name of the month.	1/1/2020	MMMM	January
Y	Number of the day in the year (1-366).	1/1/2020	Y	1
YY	Last 2 digits of the year.	1/1/2020	M-D-YY	1-1-20
YYY or YYYY	All 4 digits of the year.	1/1/2020	M-D-YYYY	1-1-2020

Character	Meaning	Example Data	Example String	Example Result
h	Hour from 1-12, with no leading zero.	8:05 pm	h:mm	8:05
hh	Hour from 1-12, with leading zero.	8:05 pm	hh:mm	08:05
H	Hour from 1-24, with no leading zero.	8:05 pm	H:mm	20:05
HH	Hour from 1-24, with leading zero.	3:00 am	HH:mm	03:00
m	Minute, with no leading zero.	3:09 am	h:m	3:9
mm	Minute, with leading zero.	3:09 am	h:mm	3:09
s	Seconds, with no leading zero.	3:09 am	h:m:s	3:9:0
ss	Seconds, with leading zero.	5:07:02 am	hh:mm:ss	05:07:02
AM/PM	Adds letters AM or PM after the date/time data to indicate morning or afternoon/evening hours.	5:07:02 am	h:mm AM/PM	5:07 AM
Literals such as / - : and space characters	Any character in the format string that is not a special character will appear in the output data exactly as typed.	1/1/2020	m-d-yy	1-1-20
		8:05 am	h:m:s	8:5:0

Using the Tile Attribute in a GX Markup Tag

The `tile` type of GX markup tag can be used to repeat portions of an HTML template in two ways:

- Repeating for Each Row in a Result Set
- Repeating a Specified Number of Times

Repeating for Each Row in a Result Set

The `tile` tag is typically used to nest levels of data in an HTML template for a grouped report. There is no practical limit to the levels of nesting. Each `tile` tag specifies a group of data rows. The text block nested in the `tile` tag is repeated once for each row in the group. The group of data rows is the result set from one of the queries in a hierarchical query object.

In a grouped report, the `tile` tag is used in combination with the `cell` tag. The `tile` tag specifies the repeating, and the `cell` tag specifies what data value to display for each repetition.

The following example shows two levels of nested tags:

```
%gx type=tile id=CONTINENT%
   - %gx type=cell id=CONTINENT.NAME%%/gx%<br>
   %gx type=tile id=COUNTRY%
      --- %gx type=cell id=COUNTRY.NAME%%/gx%<br>
   %/gx%
%/gx%
```

When an AppLogic merges data with this HTML template, text like the following replaces the tags:

```
- ASIA<br>
--- China<br>
--- Japan<br>
- EUROPE<br>
--- France<br>
--- Germany<br>
- N_AMERICA<br>
--- Canada<br>
--- Mexico<br>
```

Repeating a Specified Number of Times

You can use the `min` attribute to repeat a `tile` a specified number of times. For example, the following example repeats a decorative graphic five times:

```
%gx type=tile min=5%
   <IMG SRC="/GXApp/MyApp/Images/smiley.gif" ><br>
```

```
%/gx%
```

When an AppLogic merges data with this HTML template, text like the following replaces the tags:

```
<IMG SRC="/GXApp/MyApp/Images/smiley.gif" ><br>
<IMG SRC="/GXApp/MyApp/Images/smiley.gif" ><br>
<IMG SRC="/GXApp/MyApp/Images/smiley.gif" ><br>
<IMG SRC="/GXApp/MyApp/Images/smiley.gif" ><br>
<IMG SRC="/GXApp/MyApp/Images/smiley.gif" ><br>
```

Using the Replace Attribute in a GX Markup Tag

The `replace` type of GX markup tag is used to dynamically modify part of the text block immediately following the GX tag. You can use the `replace` tag to dynamically modify HTML tags. Each `replace` tag specifies a string to search for in the text block and a field from which to retrieve the dynamic replacement value.

The field referred to in a `replace` tag can be either a database field or a template map field. For more information about template maps, see “Using a Template Map” on page 207.

The following example uses the `replace` tag to modify an illustration. The HTML tag <img> specifies the filename of an illustration:

```
%gx type=replace id=CONTINENT.NAME value=PlaceHolder%
   <img src="images/PlaceHolder.gif">
%/gx%
```

To make this tag more interesting, consider what happens if it is placed inside a `tile` tag. In the following example, the illustration is repeated for each data row in the CONTINENT query’s result set. The HTML tag `<img>` is displayed once for each data row. The GX `replace` tag dynamically changes the filename each time the text repeats, so that a different illustration is displayed for each continent.

```
%gx type=tile id=CONTINENT%
   %gx type=replace id=CONTINENT.NAME value=PlaceHolder%
   <img src="images/PlaceHolder.gif">
   %/gx%
%/gx%
```

When an AppLogic merges data with this HTML template, text like the following replaces the tags:

```
<img src="images/ASIA.gif">
<img src="images/EUROPE.gif">
<img src="images/N_AMERICA.gif">
```

Using the Include Attribute in a GX Markup Tag

The `include` type of GX markup tag is used to insert output from another template into the current template. For example, you can use the `include` tag to add commonly-used elements such as headers and footers. You can maintain a library of reusable HTML templates for such tasks to avoid repetitive HTML coding.

Each `include` tag specifies the pathname of an external template. The included template is merged with the same result set being used for the calling template.

The following examples use the `include` tag to insert a header. In the first example, a literal path is given for the template file:

```
%gx type=include
   id="/GXApp/OnlineBooks/Templates/header1.html"%
   Text block to be replaced by output from header1.html
%/gx%
```

In the second example, a field from a result set is used to create a dynamic path for the template file.

```
%gx type=include id=CUSTOMER.Preferred_Header%
   Text block to be replaced by output from a template
%/gx%
```

When an AppLogic merges data with the HTML template that contains one of these tags, the tag is replaced with the output from the included template. The effect is that the AppLogic merges data with several HTML templates using one result set.

Creating a User-Defined Tag

Instead of using one of the predefined types of GX markup tag, such as tile or include, you can use the GX markup syntax to write your own customized tag. Use the `type` attribute to give the tag a name, and follow it with other attributes that you name and define. For example, you could write the following GX markup tag:

```
%gx type=trigger name=Clock arg1=hello%Default text%/gx%
```

To use this type of tag, you must write a customized template map class (that is, subclass the `GXTemplateMapBasic` class and override the `Get( )` method). When the Template Engine encounters an unknown type of GX tag, it calls the template map's `Get( )` method and passes the tag attributes in the following format:

```
userDefinedType:attr1="val1";attr2="val2"
```

For example, the trigger tag shown above results in the following string being passed to `Get( )`:

```
trigger:name="Clock";arg1="hello"
```

The code you write when you override **Get( )** in your subclass must be able to parse and respond to this string. In the simplest case, your `Get( )` method can return a null string, in which case the system defaults to using the text block of the gx tag. In the above example, if `Get( )` returns a null string, the system replaces the gx tags with `Default text`.

For more information about subclassing GXTemplateMapBasic, see "Using Your Own Template Map Class for Special Processing" on page 210.

Using a Template Map

A template map is an object that maps fields in a template to the data used to replace those fields. Template maps are instances of the `GXTemplateMapBasic` class.

A template map is useful when the template uses data from:

- Multiple data sources with different column names
- A data source whose schema changes over time
- Dynamic calculations made at runtime
- Memory-based data sources defined using an implementation of the`IGXTemplateData` interface, such as the GXTemplateDataBasic class in the iPlanet Application Server Foundation Class Library.

A template map is useful when the template refers to fields in a particular database, and you want to use the same template with data from another database in which the field names are different. Typically, this occurs when you move from a test database to a production database.

With a template map, you can assign values to special placeholders that will be evaluated at runtime. For example, you can use a placeholder to indicate where you want the current date to appear. You can use these placeholders anywhere you would use database field names inside a GX markup tag. These placeholders need not be preceded by the dollar sign.

You can also use a template map to link column names in a table to field names that you have used in a template. A template map allows your application to use the same template file with data from different data sources.

For example, an application might obtain invoice information from several accounting databases with slight variations in the name of the invoice number column (such as invNum, InvoiceNumber, invId, and so on). With a template map, you can simply change the links between table columns and template fields without rewriting your queries.

To ensure that the template map works, the two database schemas must be the same except for the field names. Also, each field name that is to be mapped must be preceded by a dollar sign ($) in the template.

A single template map can contain values for both placeholders and fields. When data is merged with a template, iPlanet Application Server uses the template map to find the meaning of each field name or placeholder in the template. For example, the following lines from an HTML template contain a placeholder, `CURDATE`, and a field name, `CUST.custname`.

```
%gx type=cell id=CURDATE%%/gx% <br>

Dear %gx type=cell id=$CUST.custname% :
```

To construct a template map

1. Instantiate an object from the `GXTemplateMapBasic` class. For example:

   ```
   GXTemplateMapBasic *map;

   map = GXCreateTemplateMapBasic();
   ```

   Use the *`Put( )`* method to specify the field name pairs or placeholder and value pairs. The first item in each pair is the field name or placeholder used in the HTML template. For example:

   ```
   IGXBuffer *buffer;

   buffer = GXCreateBufferFromString(curr_time):
   ```

```
map.put("CURTIME", buffer);
```

You can place the `Put( )` method call inside a loop to construct the template map iteratively. For example, you could use this technique to read the map from a file line by line.

2. Pass the template map to `EvalTemplate( )` or `EvalOutput( )`. For example:

```
result = EvalTemplate("template.html", data, map);
```

Example

The following code constructs a template map for the following three items in an HTML template called template.html:

- The placeholder `LOGIN` is mapped to the value of the `login` parameter, which is one of the AppLogic's input parameters.
- The placeholder `CURTIME` is mapped to a function call which evaluates the current time at the moment the HTML template is merged with data.
- The database field name `STATES.state` is mapped to the new field name `ST.name`. In the HTML template, the original field name is preceded by a dollar sign (`$STATES.state`).

```
char login[256];
m_pValIn->GetValString("login", login, sizeof(login));
GXTemplateMapBasic *map;
map = GXCreateTemplateMapBasic();
IGXBuffer *buffer;
buffer = GXCreateBufferFromString(login):
map.put("LOGIN", buffer);
buffer->Release();
buffer = GXCreateBufferFromString(curr_time):
map.put("CURTIME", buffer);
buffer->Release();
buffer = GXCreateBufferFromString("ST.name"):
map.put("$STATES", buffer);
buffer->Release();
HRESULT hr;
result = EvalTemplate("template.html", data, map);
```

```
map->Release();
```

The following lines from the HTML template show how these mapped items appear in the template:

```
User %gx type=cell id=LOGIN%%/gx%
accessed the system at %gx type=cell id=CURTIME%%/gx%
from the following state:
%gx type=cell id=$STATES.state%%/gx%
```

Using Your Own Template Map Class for Special Processing

A template map is an object instantiated from any class that implements the `IGXTemplateMap` interface. The iPlanet Application Server Foundation Class Library provides one such class, called the `GXTemplateMapBasic` class. You can also implement your own template mapping class.

The simplest technique for implementing your own template mapping class is to subclass GXTemplateMapBasic and override the `Get( )` method. By doing so, you gain the opportunity to run your own code as part of the template generation process. This is a useful technique for providing special template map processing that is specific to your application.

For example, you can perform the following types of special processing:

- Filter data from the database before allowing the Template Engine to merge the data with the template. This technique is illustrated in the example code later in this section.
- Perform special character formatting.
- Add counters.
- Insert large, dynamically generated HTML fragments into an output HTML page.
- Evaluate placeholders multiple times per template. The default implementation of `Get( )` in the GXTemplateMapBasic class evaluates the placeholders just once for the entire template.

Suppose the placeholder CURTIME is mapped to a function that returns the current time. When you set the time by calling Put( ) in the GXTemplateMapBasic class, the time is calculated once, and if CURTIME appears in several places in the template, the same time is printed in each place. Instead, you might want to re-evaluate the placeholder each time it is encountered in the template, so you can implement the Get( ) method in your template map subclass to recalculate CURTIME every time it is called.

Example

The following example from an include file (templatemap.h) shows deriving a template map class (MyTemplateMap) from the GXTemplateMapBasic class. It also declares two overloaded versions of the Put( ) method.

The syntax of the first Put( ) method is identical to the Put( ) method in the parent GXTemplateMapBasic class. The syntax of the second Put( ) method accepts a string parameter (LPSTR) instead of the IGXBuffer object.

```
class MyTemplateMap : public GXTemplateMapBasic {

public:

            MyTemplateMap();

    virtual ~MyTemplateMap();

    STDMETHOD(Put) (LPSTR szKey, IGXBuffer *pBuff);

    STDMETHOD(Put) (LPSTR szKey, LPSTR szValue);

};
```

The following sample code, from a program file (templatemap.cpp), shows the definitions of the Put( ) methods in the MyTemplateMap class. The first version merely calls the Put( ) method in the parent GXTemplateMapBasic class. The second version copies the szValue string parameter to a newly-created IGXBuffer, then calls the first version of the Put( ) method.

```
MyTemplateMap::MyTemplateMap()

{

}

MyTemplateMap::~MyTemplateMap()

{

}

// Adds a mapping to a template map

STDMETHODIMP
```

```
MyTemplateMap::Put(LPSTR szKey, IGXBuffer *pBuff)
{
    return GXTemplateMapBasic::Put(szKey, pBuff);
}
// Adds a mapping to a template map
STDMETHODIMP
MyTemplateMap::Put(LPSTR szKey, LPSTR szValue)
{
   if (!szKey || !szValue) return GXE_INVALID_ARG;
   // Create an IGXBuffer object
   IGXBuffer *pBuff = GXCreateBufferFromString(szValue);
   if (!pBuff) return GXE_ERROR;
   // Call the first Put() version
   hr = Put(szKey, pBuff);
   pBuff->Release();
   }
   return hr;
}
```

Constructing a Hierarchical Result Set with GXTemplateDataBasic

Normally, you use `EvalTemplate( )` or `EvalOutput( )` with the result set from a hierarchical query. You pass a hierarchical query object and the path to a template as parameters to the method, which then runs the query and merges the hierarchical result set with the template automatically.

However, you can bypass this automatic procedure and construct a hierarchical result set programmatically, rather than running a query to get the result set. In this way, you can pass data that is from a source other than a database.

For example, the AppLogic might display a list of numbers generated from a formula, or it might display a list of processors available on the server machine and their CPU loads.

To implement a hierarchical result set

1. Create an instance of the `GXTemplateDataBasic class` for the first parent level of data in the hierarchy.
2. Create another instance of GXTemplateDataBasic for a group of data at the child level.
3. Call the child GXTemplateDataBasic object's `RowAppend( )` method one or more times to specify the data in the first group.
4. Call the parent GXTemplateDataBasic object's `RowAppend( )` method to add a row for the child GXTemplateDataBasic object.
5. Call the parent GXTemplateDataBasic object's `GroupAppend( )` method to add the child group of data.
6. Repeat Step 2 through Step 5 for each group of data in the result set.
7. After constructing the GXTemplateDataBasic object, pass it to `EvalTemplate( )` or `EvalOutput( )` instead of passing in the name of a hierarchical query. Each of these methods provides an alternative syntax for this purpose.

Examples

The following code constructs a flat result set with a single level of data:

```
GXTemplateDataBasic *tdbSalesRev;
tdbSalesRev = GXCreateTemplateDataBasic("salesOffices");
tdbSalesRev->RowAppend("office=New York;revenue=150M");
tdbSalesRev->RowAppend("office=Hong Kong;revenue=130M");
tdbSalesRev->RowAppend("office=Singapore;revenue=105M");
// Use tdbSalesRev, like with EvalTemplate...
tdbSalesRev->Release();
```

The following code constructs a hierarchical result set with two child levels of data, one for Asia and one for Europe:

```
/* Create Data Object */
GXTemplateDataBasic *tdbContinents;
tdbContinents = GXCreateTemplateDataBasic("continents");
/* Create the Asia group */
GXTemplateDataBasic *tdbAsia;
```

```
tdbAsia = GXCreateTemplateDataBasic("countries");
tdbAsia->RowAppend("country=China;currency=yuan");
tdbAsia->RowAppend("country=Japan;currency=yen");
tdbAsia->RowAppend("country=South Korea;currency=won");
tdbContinents->RowAppend("name=Asia");

/* Link child records to continents group */
tdbContinents->GroupAppend(tdbAsia);

/* Create the Europe group */
GXTemplateDataBasic *tdbEurope;
tdbEurope = GXCreateTemplateDataBasic("countries");
tdbEurope->RowAppend("country=France;currency=franc");
tdbEurope->RowAppend("country=Germany;
   currency=deutsche mark");
tdbEurope->RowAppend("country=Italy;currency=lire");
tdbContinents->RowAppend("name=Europe");

/* Link child records to continents group */
tdbContinents->GroupAppend(tdbEurope);
EvalTemplate("salesByContinent.html", tdbContinents,
   NULL, NULL, NULL);
tdbContinents->Release();
tdbEurope->Release();
tdbAsia->Release();
```

Improving Performance When Using GXTemplateDataBasic

If you are using an IGXTemplateData object, such as GXTemplateDataBasic, as the source of data for a call to EvalTemplate( ) or EvalOutput( ), you can increase the perceived performance of the call by using the following technique. Instead of populating the IGXTemplateData object by calling RowAppend( ) repeatedly, implement the IGXTemplateData interface yourself and call EvalTemplate( ) or EvalOutput( ) much earlier in the AppLogic code. In this way, the Template Engine can call the IGXTemplateData object as it needs data and return results as they are available, keeping the user waiting much less time for a response.

The Template Engine service of iPlanet Application Server automatically calls the MoveNext( ) method in the IGXTemplateData interface each time it needs a new row of data; for example, when it has completed one pass in a tile tag and is ready to start the next iteration of that tile. If you have implemented your own MoveNext( ) method, you can use that code to retrieve data as needed. This takes the place of calling RowAppend( ) repeatedly to populate the IGXTemplateData object all at once. After MoveNext( ) is called, Get( ) is called to retrieve the values in that row.

For example, the following code shows how the AppLogic code looks when you use RowAppend( ):

```
// Populate the in-memory template data.  The number
// of calls to RowAppend() is unlimited.  Meanwhile,
// the user is waiting for an unknown length of time
// until the full template data set is populated.
//
GXTemplateDataBasic *td;

td = new GXTemplateDataBasic("offices");
td->RowAppend("office=New York;revenue=150");
td->RowAppend("office=Hong Kong;revenue=130");
// ... add more records here.
// Pass the finished data set to EvalTemplate().
HRESULT hr;
```

```
hr = EvalTemplate("salesReportByOffice.html",
   (IGXTemplateData *) td, NULL, NULL, NULL);
td->Release();
return hr;
```

Now suppose you create your own implementation of the `IGXTemplateData` interface or subclass from the `GXTemplateDataBasic class`. The following code is in the header file:

```
class MyTemplateDataBasic : public GXTemplateDataBasic
{
public:
   MyTemplateDataBasic(LPSTR group) :
      GXTemplateDataBasic(group)
   {
      // Prepare the retrieval of the offices records here.
      // We don't have to get all the data yet, just
      // the first record data.
   }
   STDMETHOD(IsEmpty) (
      LPSTR group,
      BOOL *empty
   );

   STDMETHOD(MoveNext) (
      LPSTR group
   );
   STDMETHOD(GetValue) (
      LPSTR       szExpr,
      IGXBuffer **ppBuff
   );
};
```

The following code is in the source file:

```
STDMETHODIMP
MyTemplateDataBasic::GetValue(LPSTR field,
   IGXBuffer **ppBuff)
{
   if (strcmp(field, "offices.office") == 0)
   {
      IGXBuffer *office;
      // ... retrieve current office field value here.
      *ppBuff = office;
      return NOERROR;
   }
   if (strcmp(field, "offices.revenue") == 0)
   {
      IGXBuffer *revenue;
      // ... retrieve current revenue field value here.
      *ppBuff = revenue;
      return NOERROR;
   }
   return GXTemplateDataBasic::GetValue(field, ppBuff);
}
STDMETHODIMP
MyTemplateDataBasic::IsEmpty(LPSTR group, BOOL *empty)
{
   if (strcmp(group, "offices") == 0)
   {
      boolean isOfficeRecordSetEmpty;
      // ... determine if the data set is empty.
      *empty = isOfficeRecordSetEmpty;
      return NOERROR;
   }
   return GXTemplateDataBasic::IsEmpty(group, empty);
```

```
}

STDMETHODIMP
MyTemplateDataBasic::MoveNext(LPSTR group)
{
   if (strcmp(group, "offices") == 0)
   {
      HRESULT noMoreRecords;
      // Move to next record in offices data set here.
      // This is where we can dynamically compute
      // the next record.
      //
      // Return NOERROR (0) if next record is available.
      // Return non-zero if no more records.
      return noMoreRecords;
   }
   return GXTemplateDataBasic::MoveNext(group);
}
```

The following code shows how the AppLogic looks when you let the Template Engine retrieve the data through `MoveNext( )`:

```
// Use our own GXTemplateDataBasic subclass, which is
// smart enough to dynamically retrieve office records
// when called back by the template engine.  This allows
// data to be streamed back to the user as it becomes
// available, instead of waiting for the entire
// data set to be created first in memory.
//
MyTemplateDataBasic *td;

td = new MyTemplateDataBasic("offices");
// MyTemplateDataBasic retrieves office records
```

```
// as necessary, so we do not prepopulate it here.
// Pass the MyTemplateDataBasic object to EvalTemplate().
HRESULT hr;
hr = EvalTemplate("salesReportByOffice.html",
   (IGXTemplateData *) td, NULL, NULL, NULL);
td->Release();
return hr;
```

Using Conditionals in an HTML Template

You can vary the output from a template depending on specified conditions. For example, in order to have different content depending on the user's browser, you could use the following tags in your template:

```
text for all browsers.
text for all browsers.

%gx type=cell id=netscapeOnly%
   text for netscape browser only.
   text for netscape browser only.
%/gx%
text for all browsers.
%gx type=cell id=microsoftOnly%
   text for internet explorer browser only.
   <blink>you're using
    MS!</blink>
   text for internet explorer browser only.
%/gx%
text for all browsers.
text for all browsers.
```

In your AppLogic, set up a template map with values for the keys used in your conditional tag. You might want to put this code is in your application base class or a helper function.

```
BOOL isNetscapeBrowser;
BOOL isMicrosoftBrowser;
// ... Compute browser type here, not shown ...
// Populate template map.
//
GXTemplateMapBasic *m;
m = new GXTemplateMapBasic();
IGXBuffer *buff;
buff = GXCreateBufferFromString("");
m->Put("netscapeOnly",  isNetscapeBrowser  ? NULL : buff);
m->Put("microsoftOnly", isMicrosoftBrowser ? NULL : buff);
buff->Release();
// ... Use map here, such as with EvalTemplate, not shown ...
m->Release();
```

Provide values for the boolean variables by consulting the input parameters in your AppLogic's input ValList.

Example HTML Template

Several example applications are shipped with iPlanet Application Server software, and the HTML templates used by these applications are available for you to view. The following example presents one of those templates with explanatory comments:

This example application displays a report on states in a given geographic region. For each state in the region, an illustration of the state flag and population figures for each county are displayed.

HTML Template for Region & States Report

This template contains two nested `tile` tags, one to loop over the states and another to loop over the counties within each state. In this example, comments are provided before each GX markup tag. For information about the other tags in the template, refer to your HTML documentation.

```
<HTML>
<HEAD>
```

```
<TITLE>Region And States Report</TITLE>
</HEAD>
<BODY BACKGROUND="/GXApp/Demo/States/Backgrounds
   /logo30pctbkglight.gif">
<H3>Region And States Report</H3>
<TABLE BORDER=1>
   <TR>
      <TH COLSPAN=2>Counties by State</TH>
   </TR>
```

The following GX tag sets up a loop that repeats for each state. The `type=tile` attribute specifies that this is a looping marker. The `id=STATES` attribute specifies that the loop will repeat for each row of the result set from the STATES query. The `MAX=50` attribute specifies that the loop can repeat no more than 50 times.

```
%gx type=tile id=STATES MAX=50%
   <TR>
      <TD ALIGN=CENTER>
```

The following GX tag displays the name of a state at the beginning of each `tile` repetition. The `type=cell` attribute specifies that the text block of this marker is replaced with a dynamic data value. The `id=STATES.STATE` attribute specifies the field in the result set that contains the dynamic value. The text block “Name of a state” is acting as a comment within the tag. When an AppLogic merges data with this template, the comment is replaced with dynamic data.

```
<b>%gx type=cell id=STATES.STATE%Name of a state%/gx%
</b>
</TD>
<TD ALIGN=RIGHT>
```

The following GX tag dynamically updates the filename of the illustration so that a different state flag is displayed for each state. The `type=replace` attribute specifies that a value in the text block is to be replaced dynamically. The `id=STATES.STATE` attribute specifies the field in the result set that contains the dynamic value. The `value=ABBR` attribute specifies the placeholder string that is to be replaced in the text block.

```
      %gx type=replace id=STATES.STATE value=ABBR%
         <IMG SRC="/GXApp/Demo/States/Images/ABBR.gif"
```

```
            width=100 height=60>
      %/gx%
   </TD>
</TR>
<TR ALIGN=RIGHT>
   <TD><b>County</b></TD>
   <TD><b>Population</b></TD>
</TR>
```

The following GX tag sets up a loop that will repeat for each county in a state. The `type=tile` attribute specifies that this is a looping marker. The `id=DETAILS` attribute specifies that the loop will repeat for each row in the result set from the DETAILS query. The `MAX=100` attribute specifies that the loop can repeat no more than 100 times. This prevents the report from getting too long if a state has an unusually large number of counties.

```
%gx type=tile id=DETAILS MAX=100%
   <TR ALIGN=RIGHT>
```

The following GX tags display the name and population of each county in the state. The `type=cell` attribute specifies that the text block of each marker is to be replaced with a dynamic data value. The `id=DETAILS.COUNTYNAM` and `id=DETAILS.POP` attributes specify the fields in the result set that contain the dynamic values. The text block strings `Name of a county` and `Its population` are acting as comments within the `cell` tags.

```
            <TD>%gx type=cell id=DETAILS.COUNTYNAM%
               Name of a             county%/gx%</TD>
            <TD>%gx type=cell id=DETAILS.POP%
               Its population
            %/gx%</TD>
         </TR>
      %/gx%
   %/gx%
</TABLE>
</BODY>
</HTML>
```

Example GXML Template

The following GXML template formats an output data stream containing the best game statistics for a sports application:

```
<gx type=tile id="DETAIL" max=10>
<gx type=cell id="DETAIL.player">
</gx>
<gx type=cell id="DETAIL.games">
</gx>
<gx type=cell id="DETAIL.fstr">
</gx>
<gx type=cell id="DETAIL.mstr">
</gx>
</gx>
```

Chapter 8

Managing Session and State Information

This chapter describes sessions, which are made up of a continuous series of interactions between a user and an iPlanet Application Server application.

The following topics are included in this chapter:

- What is a Session?
- Starting a Session
- Using an Existing Session
- Removing a Session and Its Related Data
- Example AppLogic Using Sessions
- Using Custom Sessions
- Viewing the Number of Active Sessions
- Using the State Layer

What is a Session?

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 an iPlanet Application Server application. The user's session with a Web browser or other client software might start before the iPlanet Application Server application begins tracking the user, and could continue after the application stops tracking the user.

Why Use Sessions?

You need not implement sessions if your application has no need to keep track of users or session-related data. Sessions are useful when you want to store information about each user's interaction with the application. For example:

- Increase security by requiring the user to log in to a secured session before running certain portions of the application.
- Record a history of which pages the user has visited during the session.
- In an online shopping application, keep track of items in the user's shopping cart.

How Sessions Work

Each session that you track in an application has a session ID, which is typically assigned automatically by the iPlanet Application Server. The session ID enables the application to keep track of the session. Each time the user submits an action from the client, the session ID accompanies the request.

Instead of using the automatically-generated session IDs, you can take control of the session ID mechanism. For more information, see "Assigning Your Own Session IDs" on page 240.

Each session is associated with a set of data, such as the contents of the user's shopping cart. The session data is stored in an IGXValList object. As the session continues, this data is updated as needed by the AppLogics in the application. By using the session ID, the application is able to match up the correct data with the user session every time an AppLogic in the application needs to access the session data.

A session can also have security information associated with it. Often, the first screen in an application is a login screen. When the user clicks the Login button, an AppLogic verifies the user's security level, then starts a secured session. For more information about using sessions to secure an application, see "Secure Sessions" on page 258 of , "Writing Secure Applications."

Avoid storing too much data in a session. Every time you save or retrieve the session-related data, the whole IGXValList object is involved. This can impact the performance of the application.

For each session, you can determine whether the session data is made available locally, within a cluster, or throughout the enterprise. You can also specify a timeout value. There are three different styles of timeout:

- Destroy the session if the user does not interact with the application for a given number of seconds. This style of timeout is the default.
- Destroy the session a given number of seconds from the time the session was created.
- Destroy the session at a given date and time, specified in seconds.

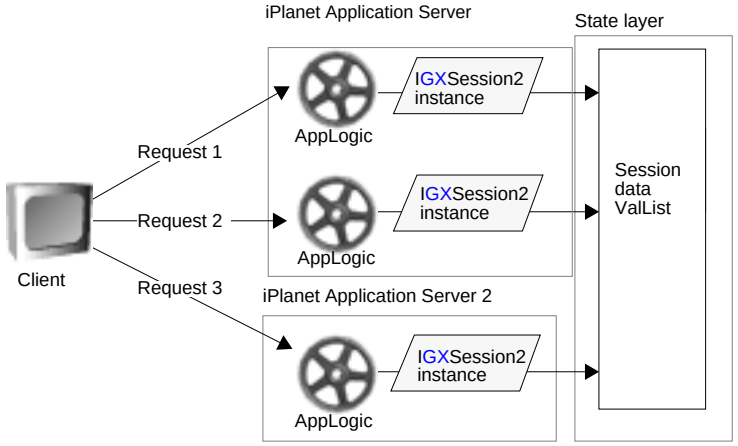
For more details about how to specify distribution and timeouts, see the entry for `CreateSession( )` in the iPlanet Application Server Foundation Class Reference.

Sessions and the iPlanet Application Server Foundation Class Library

The following parts of the iPlanet Application Server Foundation Class Library support sessions:

- IGXSession2 interface
- GXSession2 class
- IGXSessionIDGen interface
- Session-related functions in the GXAppLogic class

To track session information, the AppLogic code uses an instance of the IGXSession2 interface. Each time an AppLogic in the application is executed during the session, the AppLogic instantiates the IGXSession2 interface, as shown in the following illustration. This instance is a view of the actual session information. The session ID ensures that all these IGXSession2 instances in fact point to the same data.



Example

Suppose an iPlanet Application Server application is distributed to many machines. A user logs in through the login screen of the application. In response, an AppLogic runs on the iPlanet Application Server. This AppLogic creates a session, using an instance of IGXSession2, and creates the session ID. The AppLogic then sets some session-related data, and displays a main menu page to the user.

The user selects an action from the menu page. In response, another AppLogic is executed on one of the iPlanet Application Servers. This AppLogic could execute on any of the machines to which the application is distributed, depending on how load balancing has been configured for that application. The AppLogic retrieves the session that was created earlier, returning it in another instance of IGXSession2. Through this instance, the AppLogic can get the session data, which is the same data that was set earlier. The AppLogic manipulates the session data, saves it, and returns the next page to the user.

As the user continues with the application, the AppLogics in the application access the session as needed. Eventually, the user is finished with the application and logs off. In response, an AppLogic explicitly destroys the session that was created for this user. An instance of IGXSession2 is used to access the session, then the actual session data and the session ID are deleted, not just the IGXSession2 instance.

At this point, the session has ended, even though the user's session with their client software might still be active.

Starting a Session

You can start the user session at any point in the application. You can track all users from the moment they begin using the application. However, it is not necessary to track all users all the time, and doing so may be wasteful of system resources. It often makes sense to start a session only in response to a particular user action. For example:

- The user browses through an online catalog looking at items available for sale. The first time the user asks to purchase an item, the purchasing AppLogic starts the user session to begin tracking which items the user wishes to buy.
- An application might display information about a company, with different types of information available to different users. When a user requests to see sensitive information, such as employee records, the application requires a password from the user, then starts a secured session.

To start a session

1. Call `CreateSession( )` to create an IGXSession2 instance that refers to the session data for this application session. In the parameter list of CreateSession( ), you can specify the session timeout, distribution level, and name of the application with which the session is associated. For example:

```
hr = CreateSession(GXSESSION_DISTRIB,/* distributed */
0,         /* no timeout, will be explicitly killed */
"sessiontest",                          /* appname */
NULL,              /* not generating the sessionid */
NULL,              /* not generating the sessionid */
&m_pSession);        /* sess obj ref returned here */
```

2. Use the `GetSessionData( )` method to gain access to the IGXValList object that stores the data for this session. For example:

```
hr = m_pSession->GetSessionData(&m_pVL);
```

3. Use the methods of the IGXValList interface to retrieve or modify the data in the IGXValList object returned by `GetSessionData( )`. For example:

```
GXSetValListInt(m_pVL, "execcounter", counter);
```

4. If you made changes, call `SetSessionData( )` to update the session data. For example:

```
hr = SetSessionData(m_pVL);
```

5. If you want the session to be available to other application components (servlets, for example), you must call SetSessionVisibility( ) before going on to the next step. See “Setting the Session’s Visibility”Setting the Session’s Visibility below for details. For example:

```
hr = SetSessionVisibility("netscape.com", "/NASApp/myApp", true);
```

6. Call `SaveSession( )` in the GXAppLogic class to save the new data. For example:

```
hr = SaveSession(NULL);
```

If you call `SetSessionData( )` several times before calling `SaveSession( )`, only the value from the last `SetSessionData( )` call is saved.

The method called SaveSession( ) exists in both the IGXSession2 interface and the GXAppLogic class. The method in the GXAppLogic class is a wrapper that calls the method in the interface, and performs some tasks that ensure that the session is accessible to future AppLogics. The SaveSession( ) method in the interface saves session data only. Therefore, be sure to call SaveSession( ) in the GXAppLogic class at least once after a session is created.

Setting the Session's Visibility

Since sessions are transmitted in a cookie, they are only available within the same URL name space where they were created, which disables their use by servlets, as servlets use a different URL addressing scheme.

The method signature for `setSessionVisibility( )` is as follows:

```
HRESULT SetSessionVisibility(
    LPSTR domain,
    LPSTR path,
    BOOL isSecure)
```

domain. The domain in which the session is visible.

path. The path to which this session must be visible.

isSecure. If TRUE, the session is visible only to secure servers (HTTPS).

If you call `SetSessionVisibility( )` before `SaveSession( )`, you can control the attributes of the session cookie to allow it to be transmitted to other name spaces, particularly for servlets. By default, the session is visible only to the URL that created the cookie. Use the path parameter to specify different URLs that will be visible. For example, the path /phoenix would match "/phoenixbird" and "/phoenix/bird.html". To make the entire server root visible, specify a path of "/", the most general value possible.

You must be part of the domain to set the domain. For example, if the domain is set to .netscape.com, then the session is visible to foo.netscape.com, bar.netscape.com, and so on. Domains must have at least two periods (.) in them.

Using an Existing Session

Once you have created and saved a session, you will need to gain access to the session data repeatedly as the user continues to interact with the application.

To gain access to an ongoing user session

1. Call `GetSession( )`. Use the return value of `GetSession( )` to check whether a session has already been created. If no session exists, this method will return null. For example:

```
IGXSession2 *m_pSession;

hr = GetSession(0,  /* flags */
     "sessiontest", /* appname */
     NULL,          /* session id gen */
     &m_pSession);  /* sess object ref returned here */
```

2. Use the `GetSessionData( )` method to gain access to the IGXValList object that stores the data for this session. This method retrieves the contents that were last saved in the distributed store with `SaveSession( )`.

```
hr = m_pSession->GetSessionData(&m_pVL);
```

3. Use the methods of the `IGXValList interface` to retrieve and set the data in the IGXValList object returned by GetSessionData( ). For example:

```
GXSetValListInt(m_pVL, "execcounter", counter);
```

4. If you have changed the data, call `SetSessionData( )` to update the session data. For example:

```
hr = SetSessionData(m_pVL);
```

5. Call `SetSessionVisibility( )` to change or set any visibility settings, if necessary. For more information, see “Setting the Session’s Visibility” on page 230`Setting the Session’s Visibility`. For example:

```
hr = SetSessionVisibility("netscape.com", "/NASApp/myApp",
true);
```

6. Call `SaveSession( )` to save the new data. For example:

```
hr = SaveSession(NULL);
```

If you call `SetSessionData( )` several times before calling `SaveSession( )`, only the value from the last `SetSessionData( )` call is saved.

Removing a Session and Its Related Data

If you set a timeout for the session when you create it, you need not delete the session explicitly. It will be deleted automatically when its timer expires, that is, when the user has not interacted with the session for a given length of time.

However, in order to increase security and conserve system resources, you might want to delete a session as soon as possible after the user is finished with it. For example, if your application includes a Logout button, you can delete the session when the user clicks that button.

To destroy a session

To remove a session and its related data, call `DestroySession( )`. For example:

```
HRESULT hr;
hr = DestroySession(NULL);
```

Example AppLogic Using Sessions

The following AppLogic code uses a session to count the number of times this AppLogic executes. This code demonstrates how to create a session, access an existing session, retrieve and update session data, and destroy a session.

```
STDMETHODIMP
ExecCounter::Execute()
{
   HRESULT       hr;
   CHAR          SessID[128];
   CHAR          AppName[128];
   ULONG         Timeout;
   DWORD         Flags;
   CHAR          msg[512];
   // If the session is already underway, get the session
   hr = GetSession(0, /* flags */
      "sessiontest", /* appname */
      NULL, /* session id gen */
      &m_pSession); /* sess object ref returned here */
```

```
// If there is no existing session, create a new one
if (hr != GXE_SUCCESS)
   hr = CreateSession(GXSESSION_DISTRIB,
   /* not local,not limited to cluster */
   0,         /* no timeout, will be explicitly killed */
   "sessiontest",                          /* appname */
   NULL,           /* I'm not generating the sessionid */
   NULL,           /* I'm not generating the sessionid */
   &m_pSession);         /* sess obj ref returned here */

// Retrieve session data
hr = m_pSession->GetSessionData(&m_pVL);

// Update the count of AppLogic executions
LONG counter = GXGetValListInt(m_pVL, "execcounter");
if (counter != 0)
   counter++;
else
   counter = 1;

if (counter % 10)
{
   // On executions 1-9, set and save the new data
   GXSetValListInt(m_pVL, "execcounter", counter);
   m_pSession->SetSessionData(&m_pVL);
   hr = SetSessionVisibility("mysite.com", "/myApp", false);
   hr = SaveSession(NULL);
}
else
// Destroy the session every 10 times
{
```

```
        hr = DestroySession(NULL);
        Log("session destroyed");
    }
    return 0;
}
```

Using Custom Sessions

You can customize sessions to track application-specific information in each session, such as shopping cart contents, multiple database logins, pages visited, and so on. To do so, declare a subclass of the GXSession2 class. In your subclass, you can define simple accessor methods to read and write information in the IGXValList object associated with each session.

To create a custom session class:

1. Subclass the GXSession2 class.
2. In your AppLogic, implement the `CreateSession( )` and `GetSession( )` methods to use the custom class instead of GXSession2.
3. Pass in the IGXSession2 interface in the subclass constructor.
4. Implement other methods as necessary to perform the custom processing you desire.

Example

This example declares a customized session class, MySession, and an AppLogic that uses the custom class.

The following code is from the header file:

```
// Define my own session class that inherits from GXSession2.
class MySession : public GXSession2
{
    public:
        // Constructor and destructor
        MySession(IGXSession2 *pSess);
        virtual ~MySession();
        // Declare two custom methods in this subclass
```

```
   public:
      LONG    IncCounter();
      LONG    GetCounter();
};
// AppLogic that keeps track of a counter but uses its own
// session class
class MySessionExecCounter : public GXAppLogic
{
   // ... constructor, destructor, and Execute() methods
   // ...

   public:
      // In order to use my own session class, I define and
      // use these methods to get and create sessions
      STDMETHOD(GetSession) (
         DWORD dwFlags,
         LPSTR pAppName,
         IGXSessionIDGen *pIDGen
      );
      STDMETHOD(CreateSession) (
         DWORD dwFlags,
         ULONG dwTimeout,
         LPSTR pAppName,
         LPSTR pSessionID,
         IGXSessionIDGen *pIDGen
      );
      // I can also define simpler versions of GetSession and
      // CreateSession if some of the arguments are well-
      // known for my application, such as appname, timeout
      STDMETHOD(GetSession) (
      );
```

```
        STDMETHOD(CreateSession) (
        );
};
```

The following code is from the .cpp file, showing the implementation of the custom class and the AppLogic that uses it:

```
MySession::MySession(IGXSession2 *pSess) :
    GXSession2(pSess)
{
}// Implementation of the custom method IncCounter()
// to count the number of executions of the session AppLogic.
LONG MySession::IncCounter()
{
   IGXValList *pVL;
   LONG Counter = -1;
   GetSessionData(&pVL);
   if (pVL)
   {
      counter = GXGetValListInt(pVL, "execcounter");
      if (counter != 0)
         counter++;
      else
         counter = 1;

      // The counter is stored in the session data GXValList
      // in an item named "execcounter"
      GXSetValListInt(pVL, "execcounter", counter);
      SetSessionData(pVL);
      pVL->Release();
   }
   return counter;
}
```

```
// Implementation of the custom method GetCounter()
// to retrieve the counter.
LONG
MySession::GetCounter()
{
   IGXValList   *pVL;
   LONG counter = -1;
   GetSessionData(&pVL);
   if (pVL)
   {
      counter = GXGetValListInt(pVL, "execcounter");
      if (!counter)
         counter = -1;
      pVL->Release();
   }
   return counter;
}

// Implementation of the AppLogic that uses the custom
// session class. This AppLogic counts how often it executes.
// ...
// constructor and destructor here ...

// Simple version of my GetSession, which calls the
// more complex version of my GetSession.
STDMETHODIMP
MySessionExecCounter::GetSession()
{
   return GetSession(0, "sessiontest", NULL);
}
```

```
// Simple version of my CreateSession calls the more complex
// version of my CreateSession.
STDMETHODIMP
MySessionExecCounter::CreateSession()
{
   return CreateSession(GXSESSION_DISTRIB, 0,
      "sessiontest", NULL, NULL);
}

// Complex version of my GetSession.
STDMETHODIMP
MySessionExecCounter::GetSession(DWORD dwFlags,
   LPSTR pAppName, IGXSessionIDGen *pIDGen)

{
   HRESULT hr;
   IGXSession2 *pSession = NULL;
   hr = GXAppLogic::GetSession(dwFlags, pAppName, pIDGen,
      &pSession);

// Here is where the custom class MySession is used.
   m_pSession = new MySession(pSession);
   pSession->Release();
}

// Complex version of my CreateSession.
STDMETHODIMP
MySessionExecCounter::CreateSession(DWORD dwFlags,
    ULONG dwTimeout, LPSTR pAppName, LPSTR pSessionID,
   IGXSessionIDGen *pIDGen)
```

```
{
   HRESULT hr;
   IGXSession2 *pSession = NULL;
   hr = GXAgent::CreateSession(dwFlags, dwTimeout,
        pAppName, pSessionID, pIDGen, &pSession);

// Here is where the custom class MySession is used.
   m_pSession = new MySession(pSession);
   pSession->Release();
}
STDMETHODIMP

// Main task of the AppLogic. Gets or creates the session,
// then updates the counter using custom methods from the
// custom session class.
MySessionExecCounter::Execute()
{
   HRESULT       hr;
   CHAR          SessID[128];
   CHAR          AppName[128];
   ULONG         Timeout;
   DWORD         Flags;
   CHAR          msg[512];

   // Check whether a session already exists
   hr = GetSession();

   // If no session exists, create one
   if (hr != GXE_SUCCESS)
   {
      hr = CreateSession();
```

```
    }

    // Update the counter to indicate the AppLogic just
    // executed one more time. Uses the IncCounter() method
    // from the custom session class.
    LONG counter = m_pSession->IncCounter();
    SaveSession(NULL);
    // Display result
    sprintf(msg, "<HTML> <BODY> Applogic execution count = %d
              </BODY></HTML>", counter);
    return Result(msg);
}
```

Assigning Your Own Session IDs

Normally, iPlanet Application Server automatically generates a unique ID for each session as it is created. In most situations, you will find this automatic mechanism serves your needs. However, if you prefer to assign the session IDs yourself, you can do so using one of the following techniques:

- Implement the IGXSessionIDGen interface. This technique is illustrated in the code example below. Or,
- Generate IDs using some programmatic mechanism, then pass the generated IDs whenever you call CreateSession( ).

Morphing Session IDs

You can change the session ID each time it is passed between the client and the iPlanet Application Server. This is called morphing the session ID. A continuously changing session ID helps to avoid unauthorized access to the session.

To morph the session ID, implement the GenerateVariantID( ) and MapToBaseID( ) methods in the IGXSessionIDGen interface. Every time you call SaveSession( ), iPlanet Application Server automatically calls GenerateVariantID( ) to create a morphed ID to be sent out to the browser. Every time you call GetSession( ), iPlanet Application Server automatically calls MapToBaseID( ) in order to match the morphed ID to the original ID and find the correct existing session.

Example

This example shows a customized session ID generation class, MySessIDGen, which generates and morphs session IDs.

The following code is from the header file:

```
// Class that implements the IGXSessionIDGen interface
class MySessIDGen : public IGXSessionIDGen
{
// ...
// Below are the three IGXSessionIDGen methods of interest
// in custom session ID generation and morphing.
public:
   STDMETHOD(GenerateSessID) (
      /* [in] */  DWORD   dwFlags,
      /* [in] */  ULONG   nSessID,
      /* [out] */ LPSTR   SessID
   );

   STDMETHOD(GenerateVariantID) (
      /* [in] */  LPCSTR  pBaseID,
      /* [in] */  DWORD   dwFlags,
      /* [in] */  ULONG   nVariantID,
      /* [out] */ LPSTR   pVariantID
   );

   STDMETHOD(MapToBaseID) (
      /* [in] */  LPCSTR  pVariantID,
      /* [in] */  DWORD   dwFlags,
      /* [in] */  ULONG   nBaseID,
      /* [out] */ LPSTR   pBaseID
   );
};
```

The following code is from the .cpp file, showing the implementation of the custom class and an AppLogic object that uses it:

```
// IGXSessionIDGen methods
// GenerateSessID() creates unique session IDs
STDMETHODIMP
MySessIDGen::GenerateSessID(DWORD dwFlags, ULONG nSessID,
   LPSTR pSessID)
{
   if (!pSessID)
      return GXE_INVALID_ARG;
      // Simple scheme: just use a high resolution counter
      // as the id. This example is Solaris-specific.
      // The base session id is "<counter>"
   WORD64 HiResCounter;
   HiResCounter = gethrtime();
   CHAR id[64];
   sprintf(id, "%lld", HiResCounter);
   strncpy(pSessID, id, nSessID);
   pSessID[nSessID-1] = '\0';
   return NOERROR;
}

// GenerateVariantID() creates the morphed ID to be passed
// to the Web browser. The variant session id for
// "<counter>" will be "<newcounter>.<counter>"
STDMETHODIMP
MySessIDGen::GenerateVariantID(LPCSTR pBaseID, DWORD dwFlags,
                                ULONG nVariantID, LPSTR pVariantID)
{
   if (!pBaseID || !pVariantID ||
       nVariantID <= GXStrLen(pBaseID))
```

```
        return GXE_INVALID_ARG;
   CHAR id[64];
   WORD64 HiResCounter;
   HiResCounter = gethrtime();
   sprintf(id, "%lld.%s", HiResCounter, pBaseID);
   strncpy(pVariantID, id, nVariantID);
   pVariantID[nVariantID-1] = '\0';
   return NOERROR;
}

// MapToBaseID() finds the original session ID to correspond
// to the morphed one which is passed back from the Web
// browser
STDMETHODIMP
MySessIDGen::MapToBaseID(LPCSTR  pVariantID, DWORD   dwFlags,
                         ULONG nBaseID, LPSTR pBaseID)
{
   if (!pVariantID || !pBaseID)
      return GXE_INVALID_ARG;
   int founddot = 0;
   LPSTR p = strchr(pVariantID, '.');
   if (!p)
      p = pVariantID;
   else
   {*p++ = '\0';
     founddot = 1;
   }
   CHAR id[64];
   sprintf(id, "%s", p);
   strncpy(pBaseID, id, nBaseID);
   pBaseID[nBaseID-1] = '\0';
```

```
   If (founddot) *--p = '.';
      return NOERROR;
}

// AppLogic that uses the custom ID generation class. This
// AppLogic maintains a count of the number of times it has
// executed.
IDGenExecCounter::IDGenExecCounter() {
//...
STDMETHODIMP
IDGenExecCounter::Execute()
{

// Check whether there is already a session.
// The custom session ID generation class is passed in the
// third parameter to GetSession().
hr = GetSession(0,            /* flags */
                "sessiontest", /* appname */
                m_pIDGen,     /* session id gen */
                  &m_pSession);   /* sess object reference */
                                /* returned here */

// If there is no existing session, create one.
if (hr != GXE_SUCCESS)
{
// The custom session ID generation class is passed in the
// third parameter to CreateSession().
hr = CreateSession(GXSESSION_DISTRIB,
                 0,              /* no timeout */
                 "sessiontest",   /* appname */
                 NULL,           /* sessionid */
```

```
                m_pIDGen,        /* session id gen */
                  &m_pSession);  /* sess object reference */
                                 /* returned here */
}
m_pSession->GetSessionData(&m_pVL);
// Increment the execution counter
LONG counter = GXGetValListInt(m_pVL, "execcounter");
   if (counter != 0)
      counter++;
   else
      counter = 1;

GXSetValListInt(m_pVL, "execcounter", counter);
m_pSession->SetSessionData(m_pVL);
SaveSession(NULL);

// Display result
sprintf(msg, "<HTML> <BODY> Applogic execution count = %d
</BODY></HTML>", counter);

return Result(msg);
}
```

Viewing the Number of Active Sessions

You can programmatically retrieve the current number of active sessions in the server. This could be used to write this number to a log along with a date in order to map application usage, for example, or to display on an application management screen.

To view the number of active sessions at a given instant, you use the method GXContext`GetSessionCount()` from the class `GXContext`.

Context is a member of the AppLogic class; it's available thru "m_pContext" member variable in C++ AppLogics. You can pass it directly to `GXContextGetSessionCount()`. The prototype is as follows:

```
GXContextGetSessionCount(
    IGXContext *pContext,
    DWORD dwFlags,
    LPSTR pAppName,
    ULONG *pCount);
```

pContext . Pointer to the current server context object

dwFlags . Currently not used

pAppName . Application's name

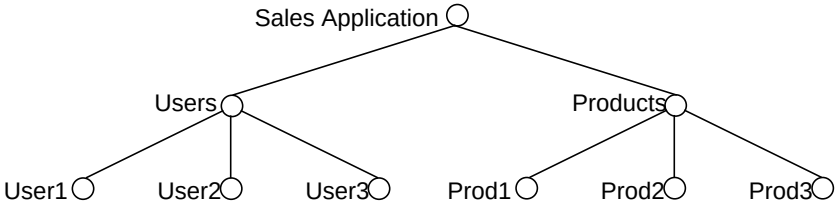
pCount . Where count is returned

For more information, see the section on the `GXContext` class in the *Foundation Class Reference*.

Using the State Layer

The state layer is a distributed data storage mechanism which you can use to store the state of an application. The application state is a collection of application variables whose scope is global within the application. In contrast, a session is a collection of application variables whose scope is a user session.

Information in the state layer can be organized in a hierarchical structure, or tree, as shown in the following illustration:



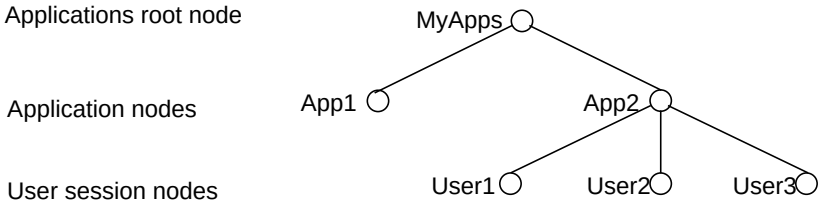
Each node on a tree consists of the following:

- An IGXValList object containing data for that node.

- Characteristics of the node itself. The characteristics you can specify for each node include whether its contents expire after a specified time and whether it is distributed enterprise-wide, within a cluster, or local to one process.

Each application running in an iPlanet Application Server system can have its own state tree. The application data is accessible to any code in the application, no matter which server or machine that code happens to be running on, because the state is distributed. The state data is accessible as long as the code is within the scope of distribution that you specified when setting up the state node (from one process up to enterprise-wide). iPlanet Application Server takes care of synchronizing access to data in the state layer so that read and write conflicts do not occur when several distributed processes are trying to work with the same data.

For example, iPlanet Application Server uses the state layer to store session-related data. Each session has one node in the tree structure. The session nodes are grouped under the application that those users are using, as shown in the following illustration:



By storing session data in the state layer, multiple iPlanet Application Servers can access the same session data, depending on the scope of distribution specified when the session was created.

The state layer is preferable to using global application variables when you need distributed data. Unlike the state layer, a global variable is bound to a particular process on one machine. In a load-balanced installation, the next time the user invokes the same AppLogic, the AppLogic might run on a different machine and would not have access to the same value of the global variable.

The iPlanet Application Server supports creating and using the state layer through the following parts of the iPlanet Application Server Foundation Class Library:

- `IGXState2 interface`. Each node on the state tree is an instance of this interface, which provides methods for creating and deleting nodes and getting the name and other characteristics of a node.
- The `GetStateTreeRoot( )` method in the `GXAppLogic class`. This returns an IGXState2 instance which is the starting point for navigating a given state tree.

Example

The following AppLogic code updates the information in a state node:

```
STDMETHODIMP
SetState::Execute()
{
   HRESULT     hr;
   CHAR        msg[256];
   // Get the state tree. Its name is "grammy" and it
   // should be a distributed tree.
   // This state tree is available to any AppLogic in any
   // session belonging to the "grammy" application.
   hr = GetStateTreeRoot(GXSTATE_DISTRIB,
                         "grammy", &m_pStateRoot);
   IGXState2  *pState = NOERROR;
   hr = m_pStateRoot->GetStateChild("Best Female Vocal",
                                    &pState);
   if (hr != NOERROR || !pState)
   {
      // the "grammy/Best Female Vocal" state node
      // doesn't exist; let's create it
      hr = m_pStateRoot->CreateStateChild("Best Female
                Vocal",            /* name */
                120,                /* timeout in secs */
                GXSTATE_DISTRIB, /* flags */
                    &pState);          /* return node pointer */
   }
   if (hr == NOERROR && pState)
   // the "grammy/Best Female Vocal" state node
   // exists; get the data.
   {
      IGXValList *pVL = NULL;
```

```
      pState->GetStateContents(&pVL);
      GXSetValListString(pVL, "winner",
                         "Whitney Houston");
      GXSetValListString(pVL, "runner up",
                         "Barbara Streisand");
      pState->SaveState(0);
      pVL->Release();
      sprintf(msg, "<HTML> <BODY> SetState applogic set
         this info in the \"grammy/Best Female Vocal\" state
         node: winner-Whitney Houston, runner up-Barbara
         Streisand </BODY></HTML>");
   }
   pState->Release();
}
return Result(msg);
}
```

The following AppLogic code gets information from an existing state node:

```
STDMETHODIMP
GetState::Execute()
{
   HRESULT     hr;
   CHAR        msg[256];
   // get the state tree
   hr = GetStateTreeRoot(GXSTATE_DISTRIB,
                         "grammy", &m_pStateRoot);
   IGXState2  *pState = NOERROR;
   hr = m_pStateRoot->GetStateChild("Best Female Vocal",
                                    &pState);
   IGXValList *pVL = NULL;
   pState->GetStateContents(&pVL);
   LPSTR w, r;
```

```
   w = GXGetValListString(pVL, "winner");
   r = GXGetValListString(pVL, "runner up");
   sprintf(msg, "<HTML> <BODY> GetState applogic got this
      info in the \"grammy/Best Female Vocal\" state node:
      winner-%s, runner up-%s </BODY></HTML>", w, r);
   pVL->Release();
   }
pState->Release();
return Result(msg);
```

Adding a Node to a State Tree

When you create a new state node, you specify its characteristics, including a timeout value. There are three different styles of timeout:

- Destroy the node if it is not accessed for a given number of seconds. This style of timeout is the default.
- Destroy the node a given number of seconds from the time the node was created.
- Destroy the node at a given date and time, specified in seconds.

You can assign a non-zero timeout value only to child nodes that are leaf nodes. Parent nodes can only have a timeout value of 0.

For more details about how to specify timeouts, see the entry for `CreateStateChild( )` in the iPlanet Application Server Foundation Class Reference.

To add a node to a state tree

1. If you already have data to store in the new node, place the data in an IGXValList object. For example:

```
IValList val = GX.CreateValList();
val.setValString("winner", "Whitney Houston");
val.setValString("runner up", "Barbara Streisand");
IGXValList *pVL = GXCreateValList();
pVL->SetValString("winner", "Whitney Houston");
```

```
pVL->SetValString("runnerup", "Barbara Streisand");
```

2. Call `GetStateTreeRoot( )` for the tree to which you want to add a node. For example, the following code gets the root node for the Grammy tree:

```
hr = GetStateTreeRoot(GXSTATE_DISTRIB,
                      "grammy", &m_pStateRoot);
```

3. Call `GetStateChild( )` to traverse to the next node.

```
hr = m_pStateRoot->GetStateChild("Best Female Vocal",
                                 &pState);
```

4. If necessary, repeat Step 3 until you reach the node under which you want to place a new node.

5. Call `CreateStateChild( )` to set up the name, timeout, and other characteristics of the new node.

```
hr = m_pStateRoot->CreateStateChild("Best Female Vocal",
          120,               /* timeout in secs */
          GXSTATE_DISTRIB,   /* flags */
          &pState);          /* return node pointer */
```

6. If you have data for the node, call `SetStateContents( )` to store the data in the new node.

```
hr = pState->SetStateContents(pVL);
```

```
child.setStateContents(val);
```

7. Save the data in the distributed store.

```
pState->SaveState(0);
```

If you call `SetStateContents( )` several times before calling `SaveState( )`, only the value from the last `SetStateContents( )` call is saved.

Storing Data in an Existing Node in a State Tree

After you have created a state node, you can access it and update the values in it by using the following procedure:

To store data in a node

1. Call `GetStateTreeRoot( )`.

```
hr = GetStateTreeRoot(GXSTATE_DISTRIB,
```

```
                                "grammy", &m_pStateRoot);
```

2. Call `GetStateChild( )` to traverse to the next node in the hierarchy.

```
hr = m_pStateRoot->GetStateChild("Best Female Vocal",
                                        &pState);
```

3. If necessary, repeat Step 2 until you have traversed to the node in which you want to store the data.

4. Call `GetStateContents( )` to retrieve a reference to the session data, and modify the values. For example:

```
IGXValList *pVL = NULL;
pState->GetStateContents(&pVL);
GXSetValListString(pVL, "winner",
                        "Whitney Houston");
GXSetValListString(pVL, "runner up",
                        "Barbara Streisand");
```

5. Call SaveState( ) to replace the old values with the new. For example:

```
pState->SaveState(0);
```

Deleting a Node From a State Tree

When you no longer need a state node, you can delete it by using the procedure described in this section. You can delete a parent node only after deleting its child nodes.

To delete a node

1. Call GetStateTreeRoot( ).

```
hr = GetStateTreeRoot(GXSTATE_DISTRIB,
                                "grammy", &m_pStateRoot);
```

2. Call GetStateChild( ) to traverse to the next node.

```
hr = m_pStateRoot->GetStateChild("Best Female Vocal",
                                        &pState);
```

3. If necessary, repeat Step 2 until you reach the parent of the node you want to delete. (Don't traverse to the node you want to delete. Stay on its parent.)

4. Call DeleteStateChild( ).

```
pState->DeleteStateChild("Streisand");
```

Chapter 9

Writing Secure Applications

This chapter describes how to implement iPlanet Application Server security features.

The following topics are included in this chapter:

- Introduction to iPlanet Application Server Security
- Providing Application Security in Code
- Secure Sessions
- Writing a Login AppLogic Object
- Validating Input to AppLogic Objects
- Secure Caching

Introduction to iPlanet Application Server Security

During the testing phase and initial deployment of your application, you are responsible for implementing application security by performing the following steps:

1. Register security information, such as users and access control lists, with iPlanet Application Server. To register security information, edit the .gxr file, as described in , “Running and Debugging Applications.”
2. In your application code, implement secured sessions. When you start a secured session, iPlanet Application Server uses the security information, which you previously registered, to ensure that unauthorized users are not granted access to sensitive portions of code. For more information, see “Secure Sessions” on page 258.

The iPlanet Application Server offers application security in the following ways:

- Authentication by user
- Authentication by roles
- Access Control List authorization
- AppLogic access authorization

The rest of this section explains the differences between users and roles, the type of security each provides, and how they are used with Access Control Lists.

About User Authentication

User-based security allows access to an application by authenticating a user's username and password. You must set up each user of the application with a username and password. A user is anyone who will be accessing the application, including developers, special user-types, administrators, and so on.

Once a user is logged into the system, his or her access to specific AppLogic objects can be managed programmatically by code in each AppLogic. Alternatively, users can be managed by explicitly mapping them to the AppLogic objects to which they have access. This is a more tedious approach to application security, but it might be useful when first deploying and testing the application.

When you install an iPlanet Application Server, two default usernames and passwords are stored in the registry, kdemo/kdemo for sample application users, and the administrator username and password you specify during install.

How the iPlanet Application Server Stores Users

The information you specify for each user you create is stored in the local registry, which is managed by the Global Directory Service (GDS). The local registry is not held in a database, but rather a local file system that is typically cached in active memory. The information held in the registry is shared between all iPlanet Application Servers when you have multiple servers supporting an application.

Because each user is stored in the local registry, this level of application security is most useful for applications where the number of expected users is limited. For example, for an Internet online shopping with potentially tens of thousands of customers, the local registry is not the best place to manage those users. Instead, you should use roles. For more information about roles, see , "Writing Secure Applications."

About Role Authentication

Role security is based on the type of user accessing an application, and not directly on the user's exact identity. A role is based on the usage characteristic of a group of users, such as Application_Admin, Application_Vendor, or Application_Employee roles.

In contrast to security based on strict user identity, where an iPlanet Application Server authenticates a user by their username and password, role security only authenticates the existence of a role and the permissions of that role. Role security does not authenticate the individual user.

You must manage how users are authenticated and determine what role a user has. This is typically done by referencing a database table that would keep track of actual user-identity information. You are responsible, in this case, for managing and accessing the user records in the database.

For example, you could construct a database table that stores the user's username and password, obtained from a login screen, along with other pertinent information, such as credit card numbers, shipping address, and so on. Along with this information is the user's role, assigned the first time the user logs in. When the user logs in, the application can verify the user and their password, and link this information with their role for further use within the application. For first-time users, the application can quickly create a new record and add that user to the database.

As the user traverses through the application, certain AppLogic objects might require a secure login. Before putting the user session into secured mode, your code would obtain the user's role from the database table, also verifying that the user is valid.

You determine the roles for your application. For example, if the application is for purchasing products over the Web, and a user entered through the publicly available access page, that user's role might be Public_User. This role might change later, if, for example, the application determines that the user is actually a vendor or a partner. The updating and assignment of roles is under application logic control and, as long as the role is defined with the iPlanet Application Server, the user obtains the appropriate privileges.

How the iPlanet Application Server Stores Roles

A role and a user are stored and verified by the iPlanet Application Server in the same way, using the local registry. Implementing role security requires less administration maintenance than explicit user security because the system administrator does not have to make updates for every user. Mapping users to roles is the responsibility of the application developer and is done in the database through application code. In addition, the number of roles you create for an application is typically much less than the number of users that use the application.

Role security is the most scalable way to provide security. It requires more planning from the application developer to create the user database table. Once the database table is created, however, all further maintenance of users is handled by the application. Role security is a must for Intranet and Internet applications with a large number of users where tracking the actual users in a database table is required for scalability.

About Access Control List Authorization

The Access Control List, or ACL, allows you to set specific permissions for users and roles. A permission relates to an action the user is allowed to perform, such as a read or write.

The iPlanet Application Server comes with default permissions, but you can also create your own application specific permissions and ACLs. You name the ACL when you create it, so you can later refer to the ACL in code.

For example, a segment of an AppLogic object might perform a read action. Before this segment of code is executed, the application developer can request iPlanet Application Server to verify that the current user or role has read permissions. iPlanet Application Server checks the local registry to verify that the current ACL name has this user as a member and that the user has read permission.

If a user does not have a certain permission, the application developer can proceed to the next logical step for either exiting the user from the application, allowing them to re-login, or directing them to a different part of the application.

About Groups

Rather than adding individual users and roles as members to the ACL, it is recommended that you create groups to which users and roles belong and add only groups to the ACL. This is especially useful if you are using individual user-based security rather than role-based security.

This saves the maintenance of updating users and roles in the ACL when users and roles change. For example, if you have created users for an Intranet application, and a user leaves the company, you only need to remove that user from the appropriate group or groups, as opposed to removing the user from the groups and any ACLs.

Providing Application Security in Code

Before users gain access to your application, they must pass through several types of security tests that are provided by the Web server and the iPlanet Application Server. The Web server provides security between the Web browser and the iPlanet Application Server. The iPlanet Application Server itself provides features, such as event logging and user groups, that you can use to ensure that the server is secure.

Once a user has access to an iPlanet Application Server, controlling access to the applications must be done at the application level. You can use several techniques to ensure that users running AppLogics do not gain unauthorized access or perform intentional or unintentional harmful actions:

- Use secure sessions.

  For more information, see “Secure Sessions” on page 258.

- Make a single AppLogic the only valid entry point to the application.

  For more information, see “Writing a Login AppLogic Object” on page 261.

- Verify all incoming AppLogic parameters, especially from forms filled out by users.

  For more information, see “Validating Input to AppLogic Objects” on page 265.

- Make sure that unauthorized users cannot access cached AppLogic results.

  For more information, see “Secure Caching” on page 266.

- Implement referrer checking or history checking so that the only users who can run the AppLogic are either those viewing or submitting from a certain HTML page, or those who have viewed a certain HTML page sometime during their session.

  For more information, see , “Managing Session and State Information.”

Secure Sessions

You can set up user IDs and groups and specify their security permissions, by registering them through a .gxr file passed to the `kreg` utility. In code, you can refer to this security information to make sessions secure. In order to make use of the security features of iPlanet Application Server, your application must include sessions.

This section describes how to make sessions secure. Before reading this section, it would be advisable to have an understanding of sessions themselves. For more information about creating and using sessions, see , "Managing Session and State Information."

In applications with a large number of users, such as an Internet online catalog, normally you do not set up a user ID for each user. Instead, the users' individual records are kept in a database and, as part of each record, a user ID is assigned. Each user ID can be used for multiple actual end users. In an application with a large user base, the user IDs normally represent groups of users, or *roles*, rather than individual users.

Before using secure sessions, you must register the security information with the iPlanet Application Server. For more information, see , "Running and Debugging Applications."

Starting a Secured Session

To make a session secure, call `LoginSession( )` (after calling `CreateSession( )` or `GetSession( )`) and pass in one of the user IDs that has already been set up and registered. The `LoginSession( )` call puts the session into secured mode. If this method returns successfully, the iPlanet Application Server begins to automatically check whether the user is authorized to run each AppLogic as it is requested during the user's session.

Example

The following code starts a secured session:

```
// Get login parameters
m_pValIn->GetValString("NAME", bufferName, sizeof(bufferName);
m_pValIn->GetValString("PASSWORD", bufferpw, sizeof(bufferpw);


IGXSession2 *mySess = NULL;
```

```
HRESULT hr;
hr = GetSession(0, NULL, NULL, &mySess);
if (hr != NOERROR || !mySess)
   // If no session, create one
   hr = CreateSession(0, 60000, NULL, NULL, NULL, &mySess);

// Here, lookup user NAME/PASSWORD in database
// and see what role the user has.  The database
// should have a user table which tracks all the
// users of the online shop application.
//
LPSTR role;
role = /* Database lookup here. */ "Shop_Customer";
// The role might be Shop_Customer, Shop_Admin,
// Shop_Accounting, or Shop_Supplier
//
// Setup the session with that role.  Future requests
// to AppLogics in this session will now operate under
// the right role.
//
LoginSession(role, "");
SaveSession(NULL);
if (mySess)
   mySess->Release();
```

Checking a User's Authorization

To check access to other resources, such as databases and files, use the `IsAuthorized( )` method. Unlike AppLogics, iPlanet Application Server does not automatically check access to these resources.

Example

The following code checks to see whether the currently logged-in user is authorized to perform some of the more advanced operations, and returns the appropriate main menu page:

```
DWORD auth_result = 0;
if ((IsAuthorized("Shop_Inventory", "READ", &auth_result)
   == NOERROR &&
   auth_result == (DWORD) GXACL_ALLOWED) ||
   (IsAuthorized("Shop_Daily_Forecast", "READ",
      &auth_result) == NOERROR
   &&
   auth_result == (DWORD) GXACL_ALLOWED) ||
   (IsAuthorized("Shop_Weekly_Forecast", "READ",
      &auth_result) == NOERROR
   &&
   auth_result == (DWORD) GXACL_ALLOWED))

   // If the user is authorized, return this menu
   return EvalOutput("kivaapp/shop/mainmenu_advanced",
      (IGXTemplateData *) NULL,
      (IGXTemplateMap *)  NULL, NULL, NULL);

// If the user is not authorized, return a different menu
return EvalOutput("kivaapp/shop/mainmenu_regular",
   (IGXTemplateData *) NULL,
   (IGXTemplateMap *)  NULL, NULL, NULL);
```

Stopping a Secured Session

When the user exits the application, or exits the secured portion of it, you can remove the security enforcement on the session by calling `LogoutSession( )`. For example:

```
LogoutSession(0);
```

After this method call, iPlanet Application Server stops validating the user for each AppLogic. In your AppLogic code, perform all security operations between a `LoginSession( )` call and its corresponding `LogoutSession( )` call.

Writing a Login AppLogic Object

To secure an application, the you can write an authentication AppLogic through which all users access the application. This AppLogic could use any combination of the following techniques:

- Prompting for ID and Password
- Writing Login Attempts to the Event Log
- Starting a Secured Session

Example

The following code is the login AppLogic from the Online Bank sample application:

```
HRESULT hr=GXE_SUCCESS;
OBSession *pSession=NULL;
(hr=GetOBSession(&pSession);
LPSTR userName=GXGetValListString(m_pValIn, "userName");
LPSTR password=GXGetValListString(m_pValIn, "password");

// Validate the username and password
if(ValidateString(userName, "User Name", TRUE, 10,
    FALSE)&& ValidateString(password, "Password", TRUE,
   10, FALSE)) {
   // Validate existence of user in database
    // Get database connection
    IGXDataConn *pConn=NULL;
    hr=GetOBDataConn(&pConn);
```

```
 // Create a query
 IGXQuery *pQuery=NULL;
hr=CreateQuery(&pQuery);
pQuery->SetTables("OBUser, OBCustomer");
pQuery->SetFields("OBUser.userName, userType, ssn,
   lastName, firstName");
pQuery->SetWhere("OBUser.userName *=
   OBCustomer.userName AND OBUser.userName= :userName AND
   password= :password");
IGXPreparedQuery *pPrepQuery=NULL;
hr=pConn->PrepareQuery(0, pQuery, NULL, NULL,
   &pPrepQuery);
IGXValList *pList=GXCreateValList();
GXSetValListString(pList, ":userName", userName);
GXSetValListString(pList, ":password", password);

IGXResultSet *pResultSet=NULL;
hr=pPrepQuery->Execute(0, pList, NULL, NULL,
   &pResultSet);
ULONG count=0;
pResultSet->RowCount(&count);
if(count) {
   // Save login criteria in the session
   pSession->SetUserName(userName);
   pSession->SetPassword(password);

   // Determine what type of user this is and bring
   // up page
   ULONG ord=0;
   pResultSet->GetColumnOrdinal("userType", &ord);
   ULONG userType=0;
```

```
            pResultSet->GetValueInt(ord, &userType);
            pSession->SetUserType(userType);

            // Save session before streaming. This is because the
            // session is initally transmitted as a cookie and
            // needs to go in the http header
            SaveSession(NULL);

            ord=0;
            pResultSet->GetColumnOrdinal("ssn", &ord);
            char ssn[50];
            pResultSet->GetValueString(ord, ssn, 50);
            pSession->SetSSN(ssn);
            pSession->SaveSession();

            // Show the correct menu page
            NewRequest("AppLogic CShowMenuPage", m_pValIn,
               m_pValOut, 0);
         }
         else
         {
            Result("<HTML><BODY>Incorrect user name and password.
               </BODY></HTML>");
            pResultSet->Release();
         }
      else
      {
         Result(NULL);
         pSession->Release();
      }
      pList->Release();
```

```
pPrepQuery->Release();
pQuery->Release();
pConn->Release();
pSession->Release();
return GXE_SUCCESS;
```

Prompting for ID and Password

The login AppLogic can prompt the user for a login ID and password, then verify that the entries are valid. If the login ID and password are invalid, the user is not granted access to the application. In addition, valid logins can be assigned access levels based on business rules.

For example, suppose an employee is accessing a human resources application to change existing employee data and add new employees. It is crucial that this sensitive employee data is accessible only to authorized personnel in the human resources department.

A second employee is using another application to maintain the company's supply of office products such as paper, pens, and diskettes. To avoid unnecessary orders being made, it is important that only authorized personnel access this application.

Because the company does not want all employees to have access to all applications, the applications must verify who is attempting to gain access and admit only those users who are authorized. Therefore, the login AppLogics of these applications prompt all users for IDs and passwords, and do not allow them to continue using the application until valid IDs and passwords are given.

If the user did not type a user ID and password, you can return from the AppLogic at that point, displaying a prompt that asks the user to type the required information. The login AppLogic will not continue executing until the correct user ID and password are supplied. Once they are, the login AppLogic can create a session and call `LoginSession( )`.

For more information about securing user sessions, see , "Writing Secure Applications."

You can save the user's login ID and password as part of the session data. At any point in AppLogic code, you can refer to the user ID and password for other operations, such as logging into a database.

You can also use the user security information to choose from several alternatives in code. For example, the AppLogic might choose from several HTML templates to format reports differently for different users.

Writing Login Attempts to the Event Log

The login AppLogic can record all login attempts in the event log by using the `Log( )` method. The iPlanet Application Server system administrator can then view the log to monitor who is attempting to use the application.

For example, suppose the system administrator is monitoring the iPlanet

Application Server event log and notices an attempted security breach. The system administrator could take the login AppLogic off line for a period of time. Without the login AppLogic, there is no access to the application.

Validating Input to AppLogic Objects

An application should always validate the incoming parameters to an AppLogic. Don't trust the input coming from the user. HTML forms, in particular, cannot adequately enforce the type and range of input. Be sure to check for the following cases:

- Although your HTML input form might provide a drop-down SELECT list, the value sent by the Web browser during a request might not necessarily be a value from your selection list.
- Your HTML input form might specify the maximum length for input control fields, but the Web browser might send more characters than are specified. For example, your AppLogic might receive a user name 4000 characters long.
- Check for special characters that might be embedded in the incoming input.
- Check ranges when converting strings to numbers.

Example

The following code, from the BaseAppLogic in the Online Bank sample application, validates an input social security number (SSN). The BaseAppLogic also contains several other methods to validate input such as phone numbers and other strings.

```
STDMETHODIMP_(BOOL)
```

```
OBBaseAppLogic::ValidateSSNString(LPSTR pTestString, LPSTR pName,
BOOL isRequired)
{
   if(!((pTestString)&&(strlen(pTestString)))) {
      if(isRequired) {
         char tmpStr[50];
         sprintf(tmpStr, "%s is a required field.", pName);
         HandleOBValidationError(tmpStr);
         return FALSE;
      }
   }
   else if(strlen(pTestString)!=9) {
      char tmpStr[50];
      sprintf(tmpStr, "%s must be of format ###-##-####",
         pName);
      HandleOBValidationError(tmpStr);
      return FALSE;
   }
   return TRUE;
}
```

Secure Caching

If your AppLogic is used in an environment in which high security is of critical importance, make sure that you attach security information to all cached AppLogic results. If an AppLogic caches its results, subsequent requests for the same

AppLogic will cause iPlanet Application Server to check for cached results first to avoid running the AppLogic unnecessarily. Because the security checking code is in the AppLogic, which might not run, subsequent unauthorized users might be able to get the cached results.

To make sure caching is secure, include security information as part of the cache criteria. For example, the criteria could include the user ID and password parameters or other security information, such as the session ID. If the incoming request contains a user ID and password that match those stored with the cached results, then the user is authorized to get the results and the cache can be used.

iPlanet Application Server passes the session ID in the input IGXValList object. The name uses the following syntax:

`gx_session_id_`*`appName`*

where *appName* is the name of the application that you passed in the call to `GetSession( )` or that was registered using the `kreg` utility. For example, the input IGXValList object for AppLogics in an application called Catalog might be an item with the following name:

```
gx_session_id_Catalog
```

The cache criteria for AppLogics in this application might look like the following example:

```
SetCacheCriteria(3600, 1, "gx_session_id_Catalog");
```

For more information about cache criteria, see , “Writing Server-Side Application Code.”

Chapter 10

Integrating Applications with Email

This chapter describes how to write applications that both send and receive electronic mail (email).

The following topics are included in this chapter:

- Introduction to Email in iPlanet Application Server Applications
- Receiving Email
- Sending Email

Introduction to Email in iPlanet Application Server Applications

The iPlanet Application Server Foundation Class Library supports email through the IGXMailBox interface.

In order for email applications to work, you must have access to one or both of the following types of email servers:

- SMTP server if you want to send email. Not required if you only want to receive email.
- POP server if you want to receive email. Not required if you only want to send email.

Security in Email

Security is often a concern when sending or receiving email. If the application generates and sends email using user input to set the address or content, then there is a risk of propagating inappropriate messages or mailing to incorrect recipients. Be sure to validate all user input before incorporating it in email.

Receiving Email

To receive email, your application must have access to a POP server.

Before retrieving messages, you can use RetrieveCount( ) to see how many messages are waiting in the specified inbox on the mail server. By checking first, you can avoid attempting to retrieve messages if the mailbox is empty. You can also use this technique when you need to know how many messages are waiting in order to construct a loop that will iterate through them one by one.

To retrieve messages, call Retrieve( ). Depending on the parameters you pass to this method, you can customize the retrieval process in the following ways:

- Retrieve all messages
- Retrieve only unread messages
- Delete messages from the mail server as they are retrieved

Only those messages received before the last call to Open( ) will be retrieved. You can not open a mailbox session, leave it open, and continuously receive emails. Instead, you must open a new session each time you want to retrieve new email.

After retrieving messages, you can return the mailbox to its original state by calling RetrieveReset( ). This method undeletes and unmarks any messages that were affected by the previous Retrieve( ) call.

To receive email

3. Create an instance of IGXMailbox by calling CreateMailbox( ). In this call, you specify valid user information and the name of the POP server you want to access. For example:

   ```
   hr = pMMBox->CreateMailbox("smtp", NULL, NULL,
     "sid@blm.com", (IGXObject **)&pMbox);
   ```

4. Open a session on your POP server by calling Open( ) with the OPEN_RECV flag. For example:

```
hr = pMBox->Open(OPEN_RECV);
```

5. To find out whether you have messages, call RetrieveCount( ). For example:

```
LONG res;

res = pMbox->RetrieveCount();
```

6. To retrieve messages, instantiate an IGXValList object to contain the email messages, then call Retrieve( ). For example, the following code retrieves the latest unread messages and does not delete them from the mailbox:

```
IGXValList *msgs = NULL;

hr = pMbox->Retrieve(TRUE, FALSE, &msgs);
```

   Only the messages received before the call to Open( ) are retrieved.

7. To undo changes, call RetrieveReset( ). For example:

8. `hr = pMbox->RetrieveReset();`

9. To close the session, call Close( ). For example:

```
pMBox->Close();
```

You can have only one mail server session open at a time. For example, suppose you open a session with the OPEN_RECV flag, then want to send email. You must first close the existing session, then open another one with the OPEN_SEND flag.

Example

The following code retrieves email:

```
LONG res;
pMMbox->CreateMailbox("smtp","sdas","sdas",
   NULL, (IGXObject **)&pMbox);
pMbox->Open(OPEN_RECV);
if ((res=pMbox->RetrieveCount()) > 0) {
   IGXValList *msgs = NULL;
   printf("Mailbox has %d messages\n", res);
   hr = pMbox->Retrieve(FALSE, FALSE, &msgs);

   CHAR msgno[16];
   msgs->ResetPosition();
```

```
    while (msgs->GetNextKey(msgno, 16) == NOERROR) {
        GXVAL val;
        hr = msgs->GetValByRef(msgno, &val);
        printf(Msg %s:\"%s"\n", msgno, val.u.pstrVal);
    }
    msgs->Release();
pMbox->Close();
pMbox->Release();
```

Sending Email

To send email, your application must have access to an SMTP server. Construct the email address and message separately, then use the Send( ) method to send the email out through the server.

You can send email to a single recipient or to a group of recipients. To send email to a group, use one of the following techniques:

- Pass the email addresses to Send( ) as an array.
- Use a loop to send a series of messages one at a time.

You can populate an address array dynamically using the results of a query, in which each row returned by the query is one email address. Use a loop to iterate through the rows in the query's result set and assign the data to successive elements of the array.

For example, this multiple-recipient technique would be useful for sending email to all customers to notify them of changes on your Web site.

To send email

1. Create an instance of IGXMailbox by calling CreateMailbox( ). In this call, you specify valid user information and the name of the POP server you want to access. For example:

   ```
   hr = pMMBox->CreateMailbox("smtp", NULL, NULL,
     "sid@blm.com", (IGXObject **)&pMbox);
   ```

2. Open a session on your SMTP server by calling Open( ) with the OPEN_SEND flag. For example:

```
hr = pMBox->Open(OPEN_SEND);
```

3. To send the message, call Send( ). Pass a single email address or an array of addresses to this method, along with the text of the message. For example:

4. pMbox->Send(ppTo, "Testing, please ignore");

5. To close the session, call Close( ). For example:

   ```
   pMBox->Close();
   ```

   You can have only one mail server session open at a time. For example, suppose you open a session with the OPEN_SEND flag, then want to retrieve your email. You must first close the existing session, then open another one with the OPEN_RECV flag.

Example

The following code sends email:

```
LONG res;
LPSTR ppTo[2];
pMMbox->CreateMailbox("smtp","sdas","sdas",
   NULL, (IGXObject **)&pMbox);
pMbox->Open(OPEN_SEND);
ppTo[0] = "sdas@kello.com";
ppTo[1] = NULL;
pMbox->Send(ppTo, "Testing, please ignore");
pMbox->Close();
pMbox->Release();
```

Chapter 11

Running and Debugging Applications

This chapter describes how to set up a test version of your application, as well as how to use the debugging tools in your C++ development toolkit.

The following topics are included in this chapter:

- Getting Ready to Run an Application
- Debugging with Third-Party Tools

Getting Ready to Run an Application

Before you can execute the AppLogic objects in your application, you must set up a test version of your application. This involves copying your application files to the appropriate locations and registering the AppLogics, other code modules, and security information needed in the application. For the purposes of testing, you will probably set up the application on a single iPlanet Application Server.

This manual does not describe application deployment in detail. In general, the procedure for setting up an application for testing is similar to that for deploying production applications. This section describes only those procedures that are specific to setting up an application for testing.

Alternatively, if you are using iPlanet Application Builder, you can deploy applications using its deployment feature. This procedure is described in User's Guide.

Compiling Applications

This section assumes that you are already familiar with creating makefiles and compiling source files on your development platform. It also assumes that you have already installed the entire Calhoops sample application on your development system using the iPlanet Application Server installation procedure described on the product CD.

This section lists relevant libraries to link to and describes creating a makefile for your AppLogic, using examples of makefiles from the Calhoops sample application for the Sun Solaris, Hewlett-Packard HP-UX, and Microsoft Windows NT platforms.

Setting the GX_ROOTDIR Environment Variable

Before compiling, you need to set the $GX_ROOTDIR environment variable to point to the root directory for the iPlanet Application Builder. For instructions on how to do this, see the system documentation for your particular development platform.

Creating a Makefile

Use your C++ application development tools to generate the makefile for your project. The following examples show the makefile for the Calhoops sample application on the Sun Solaris, Hewlett-Packard HP-UX, and Microsoft Windows NT platforms.

Sun Solaris

```
#
# Sample makefile for C, C++ applogics
# Run 'make debuggable' or simply 'make'
# to build shared library with debug info.
# Run 'make release' to build optimized shared library
#
OBJS = calhoops.o # list of all object files to be generated
SHO  = libcalhoops.so   # name of shared object file to generate
DEBUG  = -g
debuggable := TARGET = debuggable
release := TARGET = release
release := DEBUG = -O
```

```
debuggable: all

release: all

PLATFORM = SOLARIS

CC   = cc

CPP   = CC

GX_ROOT  = $$GX_ROOTDIR

LIBDIR  = $(GX_ROOT)/gxlib

INCDIR  = $(GX_ROOT)/include

INCLUDES = -I. -I$(INCDIR)

CPPFLAGS = -KPIC -mt -DUNIX -DGXPUBLIC_BUILD -D$(PLATFORM) $(DEBUG)
$(INCLUDES)

CFLAGS  = -KPIC -mt -DUNIX -DGXPUBLIC_BUILD -D$(PLATFORM) $(DEBUG)
$(INCLUDES)

SHOFLAGS = -G -Bsymbolic -mt -z text -L $(LIBDIR)

.SUFFIXES :

.SUFFIXES : .c .h .cpp .s .o

.cpp.o:

   $(CPP) $(CPPFLAGS) -c $<

.c.o:

   $(CC) $(CFLAGS) -c $<

.s.o:

   $(CPP) $(CPPFLAGS) -c $<

all : install

install: $(SHO)

   [ -d $(LIBDIR) ] && cp $(SHO) $(LIBDIR)

$(SHO): $(OBJS)

   $(CPP) $(SHOFLAGS) -o $(SHO) $(OBJS) -lgxagent -lgxidl -lgxutil

clean :

   @rm -f $(OBJS)

clobber : clean

   @rm -f $(SHO) $(LIBDIR)/$(SHO)
```

Hewlett-Packard HP-UX

```
#
# Sample makefile for C, C++ applogics
# Run 'make debuggable' or simply 'make'
# to build shared library with debug info.
# Run 'make release' to build optimized shared library
#
OBJS = calhoops.o  # list of all object files to be generated
SHO  = libcalhoops.sl # name of shared object file to generate
DEBUG = -g
debuggable := TARGET = debuggable
release := TARGET = release
release := DEBUG = -O
debuggable: all
release: all
PLATFORM = HPUX
CC   = cc
CPP   = CC
GX_ROOT  = $$GX_ROOTDIR
LIBDIR  = $(GX_ROOT)/gxlib
INCDIR  = $(GX_ROOT)/include
INCLUDES = -I. -I$(INCDIR)
CPPFLAGS = +Z +a1 -DUNIX -D$(PLATFORM) -DGXPUBLIC_BUILD $(DEBUG)
$(INCLUDES)
CFLAGS  = +Z -Ae -DUNIX -D$(PLATFORM) -DGXPUBLIC_BUILD $(DEBUG)
$(INCLUDES)
SHOFLAGS = -b -q -Wl,+s,+b:,+vshlibunsats $(CPPFLAGS) -L $(LIBDIR)
.SUFFIXES :
.SUFFIXES : .c .h .cpp .s .o
.cpp.o:
 $(CPP) $(CPPFLAGS) -c $<
.c.o:
```

```
   $(CC) $(CFLAGS) -c $<
.s.o:
   $(CPP) $(CPPFLAGS) -c $<
all : install
install: $(SHO)
   [ -d $(LIBDIR) ] && cp $(SHO) $(LIBDIR)
$(SHO): $(OBJS)
   $(CPP) $(SHOFLAGS) -o $(SHO) $(OBJS) -lgxagent -lgxidl -lgxutil
clean :
   @rm -f $(OBJS)
clobber : clean
   @rm -f $(SHO) $(LIBDIR)/$(SHO)
```

Microsoft Windows NT

```
PROJECT=calhoops.dll
PROJECT_OBJS=calhoops.obj
#################################################################
KIVA_ROOT=C:\KIVA\KDS
MYCFLAGS=
MYLFLAGS=
CC_FLAGS    =$(MYCFLAGS) /c /nologo /MTd /W3 /GX -DWIN32 -DWINDOWS
-D_WINDOWS $(MYCFLAGS)
CC_FLAGS_REL=/Ox
CC_FLAGS_DBG=/Od /Zi /Gm -DDEBUG -D_DEBUG
LINK_FLAGS    =$(MYLFLAGS) /MAP /subsystem:windows /dll
/incremental:no
LINK_FLAGS_REL=
LINK_FLAGS_DBG=/DEBUG:FULL /DEBUGTYPE:BOTH
INCLUDE=$(KIVA_ROOT)\include;$(INCLUDE)
LIB    =$(KIVA_ROOT)\lib\c;$(LIB)
LIBS=gxagent.lib gxutil.lib gxidl.lib kernel32.lib user32.lib
uuid.lib
```

```
!if "$(DEBUG)" == "YES"
CC_FLAGS=$(CC_FLAGS) $(CC_FLAGS_DBG)
LINK_FLAGS=$(LINK_FLAGS) $(LINK_FLAGS_DBG)
!else
CC_FLAGS=$(CC_FLAGS) $(CC_FLAGS_REL)
LINK_FLAGS=$(LINK_FLAGS) $(LINK_FLAGS_REL)
!endif
CC=CL
LINK=LINK
.SUFFIXES: .cpp
.cpp.obj:
   $(CC) $(CC_FLAGS) $<
all: project
project: $(PROJECT_OBJS)
   $(LINK) $(LINK_FLAGS) $(PROJECT_OBJS) /OUT:$(PROJECT) $(LIBS)
clean:
   del *.obj
   del *.map
   del *.dll
   del *.pdb
```

Placing Files on the iPlanet Application Server

This section uses the following abbreviations to refer to directories on your file system:

- $GX_ROOTDIR is the root directory where you installed the iPlanet Application Builder.
- $APP_ROOTDIR is the root directory where you store AppLogic files for a specific application or project. We recommend that you make this a subdirectory under the GXApp directory:
  - Unix: $GX_ROOTDIR/APPS/GXApp/<Italic>AppName
  - Windows: $GX_ROOTDIR\APPS\GXApp\<Italic>AppName

CAUTION If you reinstall or uninstall the iPlanet Application Builder, this directory might be overwritten by the installation program. Be sure to move or copy these files to a different location before reinstalling or uninstalling your software.

When placing shared libraries and HTML template files (for HTML-based clients) on your iPlanet Application Server, consider storing them in the following locations, where $GX_ROOTDIR is the Netscape Application Server root installation directory:

Component	Suggested Location
Sun Solaris	
shared library files	`$GX_ROOTDIR/gxlib/*.so.`
html templates (.html)	`$APP_ROOTDIR/templates`
Hewlett-Packard HP-UX	
shared library files	`$GX_ROOTDIR/gxlib/*.sl`
html templates (.html)	`$APP_ROOTDIR/templates`
Microsoft Windows NT	
shared library files	`$GX_ROOTDIR\bin\*.dll`
html templates (.html)	`$APP_ROOTDIR\templates`

For example, the Calhoops sample application shared library files might be stored in the following locations:

Platform	Shared Library File
Sun Solaris	`$GX_ROOTDIR/gxlib/libcalhoops.so`
Hewlett-Packard HP-UX	`$GX_ROOTDIR/gxlib/libcalhoops.sl`
Microsoft Windows NT	`$GX_ROOTDIR\bin\calhoops.dll`

Placing Files on the Web Server (HTML Client)

When placing web pages and graphics files on your Web server, consider storing them in the following locations, where <Code>$WS_DOCROOTDIR is the document root directory of the web server:

Component	Sample Location
html pages (.html)	`<Code>$WS_DOCROOTDIR/GXApp/<Italic>AppName`
html graphics (.gif, .jpeg)	`<Code>$WS_DOCROOTDIR/GXApp/<Italic>AppName`

For example, the query selection form for the Calhoops sample application (index.html) might be in the following path:

```
<Code>$WS_DOCROOTDIR/GXApp/CCalhoops/index.html
```

Registering Code And Security Information

Before you can execute your application, you must set up a test version. As one of the steps in this procedure, you must register certain application items with the iPlanet Application Server. When setting up an application for testing, you might need to register any of the following types of items with the iPlanet Application Server:

- AppLogic objects
- Other code modules
- Users
- User groups
- Access control lists (ACLs)

By registering users, user groups, and ACLs, you can set up security for the application. During testing and initial deployment, you must define these items yourself if your application includes security features, such as calls to LoginSession( ). During production deployment, the system administrator can also manage the security information, using the iPlanet Application Server Administrator tools.

Using Utilities to Register Application Information

The iPlanet Application Server maintains a record of registered application items in local registry storage. You can use the kreg utility to populate the registry. To do so, edit one or more files with a .gxr extension. These files are the input to kreg.

To register code or security information using utilities

6. For an AppLogic object or other code module, you must generate a unique GUID. To do so, run the kguidgen utility from a command line or window by typing the following command:

   ```
   kguidgen
   ```

   This utility returns a random, unique GUID which you can copy and paste.

7. Add lines to the .gxr file for the items you want to register. The syntax for each type of item is given later in this section. Each item uses four lines.

   - Make sure you assign a unique name to the item. Unlike the GUIDs, which are sure to be unique since they are generated automatically, there is a possibility for duplicates when you assign names yourself. For example, prefix the names of your AppLogics with the application name to reduce the possibility of name collisions.
   - For an AppLogic object or other code module, paste in the GUID that was generated by kguidgen.

8. Run the kreg utility from a command line or window using the following syntax:

   `kreg` *fileName.gxr*

   For example:

   ```
   kreg OnlineBank.gxr
   ```

AppLogic .gxr Syntax

The entry in a .gxr file to register an AppLogic object uses the following syntax (the portion from `AppLogic` to `group` is all on one line):

```
AppLogic name [:type=c][:enable=y|n][:encrypt=y|n]
              [:lb=y|n][:descr[:AppName;group;...]]
GUID [:server;...[acl=user,[!]EXECUTE;...]]
Blank line
Path to AppLogic
```

The items in the syntax are as follows:

- *name*: Name of the AppLogic object.
- :type=c: Optional letter c to indicate that the AppLogic object is written in C++.
- :enable=y|n: Optional flag to indicate whether the AppLogic is enabled.
- :encrypt=y|n: Optional flag to indicate whether the communications to the AppLogic are encrypted.
- :lb=y|n : Optional flag to indicate whether sticky load balancing is set.
- :*descr*: Optional description of the AppLogic, which will appear in the Application Administrator.
- :*AppName*;*group*;. . .: Optional semicolon-separated list of groups to which the AppLogic belongs. The first group should be the name of the application.
- Globally unique identifier (GUID) that was generated by kguidgen.
- :*server*;. . .: Optional semicolon-separated list of server descriptions. Each server description uses the following syntax:

  <SERVER_IP_ADDR>:<SERVER_IP_PORT>[=<SERVER_FLAGS>]

  SERVER_IP_ADDR is a decimal-dotted IP address, such as 192.23.43.15. SERVER_IP_PORT is the Executive Server port number, in decimal. SERVER_FLAGS is an optional decimal number. The flags, in hexadecimal, are 0x8000000 to set sticky load balancing and 0x00000001 to set the enable flag. Note that you can turn these on using the lb and enable flags on the first line.

- acl=*user*,[!]EXECUTE;. . .: Optional semicolon-separated list of access control list (ACL) entries, to specify which users can execute the AppLogic. Each ACL entry uses the following syntax:

  user,[!]permission

  User is the name of a User or UserGroup, as specified by the system administrator. Permission is an operation name. For AppLogics, EXECUTE is the only operation that is checked automatically by the system. An exclamation point in front of EXECUTE means the user or user group can not run the AppLogic.

- Path to AppLogic object. For cross-platform compatibility, you can omit the extension of the file (such as .dll or .so) and, on UNIX platforms, you can also omit the lib prefix of the file. For increased portability, it is advisable to use relative paths in the .gxr file.

- (Optional) Comments. Each line can end with a comment preceded by the vertical bar character |.

Example

The following lines show the .gxr file entry for an AppLogic object (the portion from `AppLogic` to `digibanker` is all on one line):

```
AppLogic CalculateBalancesLogic:Updates account balance
   :lb=y:type=c::digibanker
{e248ecd0-9bfd-bc32-00a024d1709f}
|Blank
libbanking
```

Code Module .gxr Syntax

The entry in a .gxr file to register a code module (that is not an AppLogic object or an iPlanet Extension) uses the following syntax:

```
Module name [:type=c][:descr[:AppName;group;...]]
GUID
Blank line
Path to code module
```

The items in the syntax are as follows:

- *name*: Name of the module.
- `:type=c`: Optional letter `c` to indicate that the module is written in C++.
- `:`*descr*: Optional description which will appear in the Application Administrator.
- `:`*AppName*`;`*group*`;...`: Optional semicolon-separated list of groups to which the module belongs. The first group should be the name of the application.
- Globally unique identifier (GUID) that was generated by kguidgen.
- Relative path to the code module. For cross-platform compatibility, you can omit the extension of the file (such as .dll or .so). For increased portability, it is advisable to use relative paths in the .gxr file.
- (Optional) Comments. Each line can end with a comment preceded by the vertical bar character |.

Example

The following lines show the .gxr file entry for a code module:

```
Module DigiBank CICS Integration:type = c
{e248ecd0-9bfd-bc32-00a024d1709f}
|Blank
digicics
```

User .gxr Syntax

The entry in a .gxr file to register a user has the following syntax:

```
User name
Password
[group;...]
Blank line
```

The items in the syntax are as follows:

- *name*: Name of the user.
- *Password*: The user's password.
- *group*;. . .: Optional semicolon-separated list of groups to which the user belongs. These groups are defined elsewhere in the .gxr file, using the syntax described later in this section.
- (Optional) Comments. Each line can end with a comment preceded by the vertical bar character |.

Example

The following lines show the .gxr file entry for a user:

```
User DigiBankAdmin
tiger
DigiBankUsers;DigiBankAdministrators
| Blank
```

User Group .gxr Syntax

The entry in a .gxr file to register a user group has the following syntax:

```
UserGroup name
```

```
Blank line
Blank line
Blank line
```

The items in the syntax are as follows:

- *name*: Name of the user group.
- (Optional) Comments. Each line can end with a comment preceded by the vertical bar character |.

Example

The following lines show the .gxr file entry for a user group:

```
UserGroup DigiBank Users
| Blank
| Blank
| Blank
```

ACL .gxr Syntax

The entry in a .gxr file to register a named access control list (ACL) uses the following syntax:

```
ACL name
[user,[!]permission;...]
Blank line
Blank line
```

The items in the syntax are as follows:

- *name*: Name of the ACL.
- *user*,[!]*permission*;...: Optional semicolon-separated list of ACL entries, to specify which users can perform operations. Each ACL entry uses the following syntax:

  ```
  user,[!]permission
  ```

  User is the name of a user or user group, as specified by the system administrator. Permission is an operation name, such as EXECUTE, READ, or WRITE . An exclamation point in front of the permission means the user or user group can not perform that operation.

- (Optional) Comments. Each line can end with a comment preceded by the vertical bar character |.

Example

The following lines show the .gxr file entry for an ACL:

```
ACL DigiBank.monthlyforecast
DigiBankAdmin,ADMIN;DigiBankPartner,!READ
| Blank
| Blank
```

Saving and Restoring Registry Configurations

As described in the previous section, the iPlanet Application Server maintains a record of registered application items in local registry storage. You can save and restore snapshots of the registry by using the kreg utility. You might want to do this for the following reasons:

- Back up registry settings.
- More easily bring up secondary, replicated servers.
- More easily bring up replicated engines or control sets.
- Save, replicate, and manually synchronize registry information about users, user groups, ACLs, AppLogics, and iPlanet Extensions across multiple machines.
- Debug administration settings.
- Remotely view and debug a configuration.
- Allow various groups in your organization to swap between saved configurations, especially if certain registry configurations are required for certain tests, such as load balancing, partitioning, and stress testing.
- Replicate AppLogic information to the Web server tier. This supports better AppLogic load balancing from the Web server. It also allows encryption of AppLogic requests on a per-AppLogic basis between a Web server and Executive Server.

To Save the Registry

To save a snapshot of the current registry settings, run the kreg utility from a command line or window using the following syntax:

`kreg -save` *fileName key1 [key2...]*

The `fileName` is the file in which the registry settings are saved. The `key` is the path to a registry key, starting from the root key. For example:

```
kreg -save mytest.data SOFTWARE\KIVA\Enterprise\2.0\CCS0\DAE
```

To Restore the Registry

To save a snapshot of the current registry settings, run the kreg utility from a command line or window using the following syntax:

`kreg -load` *fileName [fileName2...]*

The `fileName` is the file in which the registry settings were saved. For example:

```
kreg -load mytest.data
```

Debugging with Third-Party Tools

You can test and debug AppLogic objects using the debugging tools in your C++ development toolkit. The tools available vary depending on the machine and operating system you are using. iPlanet Application Builder integrates with several debugging tools.

The following steps give a general outline of the debugging process. Specific procedures for each debugging tool follow.

General steps to debug AppLogic

1. Compile the AppLogic using the debug option.

   For example, in many compilers, the `-g` flag enables the debug option.

2. Start your debugging tool.

   For example, on Microsoft Windows NT, you might use msdev. On UNIX, you might use dbx.

1. Determine the process ID of the C++ Server process you want to debug.

   For example, on Microsoft Windows NT, use the Task Manager. On UNIX, use the `ps` command.

2. Using this process ID, attach to the C++ Server process from within your debugging tool.

   For example, if you are using msdev on Microsoft Windows NT, use the menus. If you are using dbx on UNIX, use the command `attach` *PID*.

Note: The AppLogic you are trying to debug may not yet be loaded into the C++ Server. Most debugging tools will still allow you to set the breakpoint in the AppLogic before it has been loaded. If not, execute the AppLogic once so that it is loaded.

3. Set a breakpoint in your AppLogic.

   For example, the beginning of the Execute( ) method is often a logical place for a breakpoint.

4. Execute your AppLogic.

5. At the breakpoint, begin stepping through your AppLogic code.

Debugging with MSVC (Version 4.2 or Higher)

To debug code using Microsoft Visual C++, in addition to any projects that you use to build your AppLogic code, you must create a project of type "Console Application" that contains no code files at all, but simply specifies the C++ Server .EXE file of your Netscape Application Server installation as its executable. After invoking your AppLogic once, you will be able to open its source files in the MSVC's editor and set breakpoints in these files.

Create the Project

1. File – New– Project Workspace.
2. Console Application; any name you like; Location can be any directory, but it will be convenient to use the root directory of your AppLogic tree.

Set Up the Project

1. Build – Settings (Alt-F7)
2. Select the "Win32 Debug" configuration.
3. Go to the Debug tab.
4. In "Category", select General.
5. As "Executable for debug session" supply the path to the C++ Server .EXE file in your Runtime Environment.
6. "Program arguments": -debug (may not be necessary).
7. In "Category", select Additional DLLs.
8. In the Modules list, add one or more DLLs from your AppLogic and check the boxes to the left of their names.

Run the Project

1. Start the Netscape Application Builder Runtime Environment and shut down the C engine once initialization is complete.
2. Build – Debug – Go or F5 key.
3. "One or more files are out of date ...?": choose No.
4. "One or more breakpoints cannot be set ....": choose OK.
5. "First-chance exception ....": choose OK.
6. Debug – Go or F5.
7. "Pass exception on to the program being debugged?": choose Yes.
8. Repeat steps 5-7 once more.
9. File – Open and open one or more source files from the additional DLLs you specified in "Set up" above.
10. Set breakpoints in these files, as needed.
11. Bring up a suitable form in a Web browser or somehow cause your AppLogic to execute, and debug normally.

Chapter 12

Sample Code Walkthrough

The following code samples are included in this chapter:

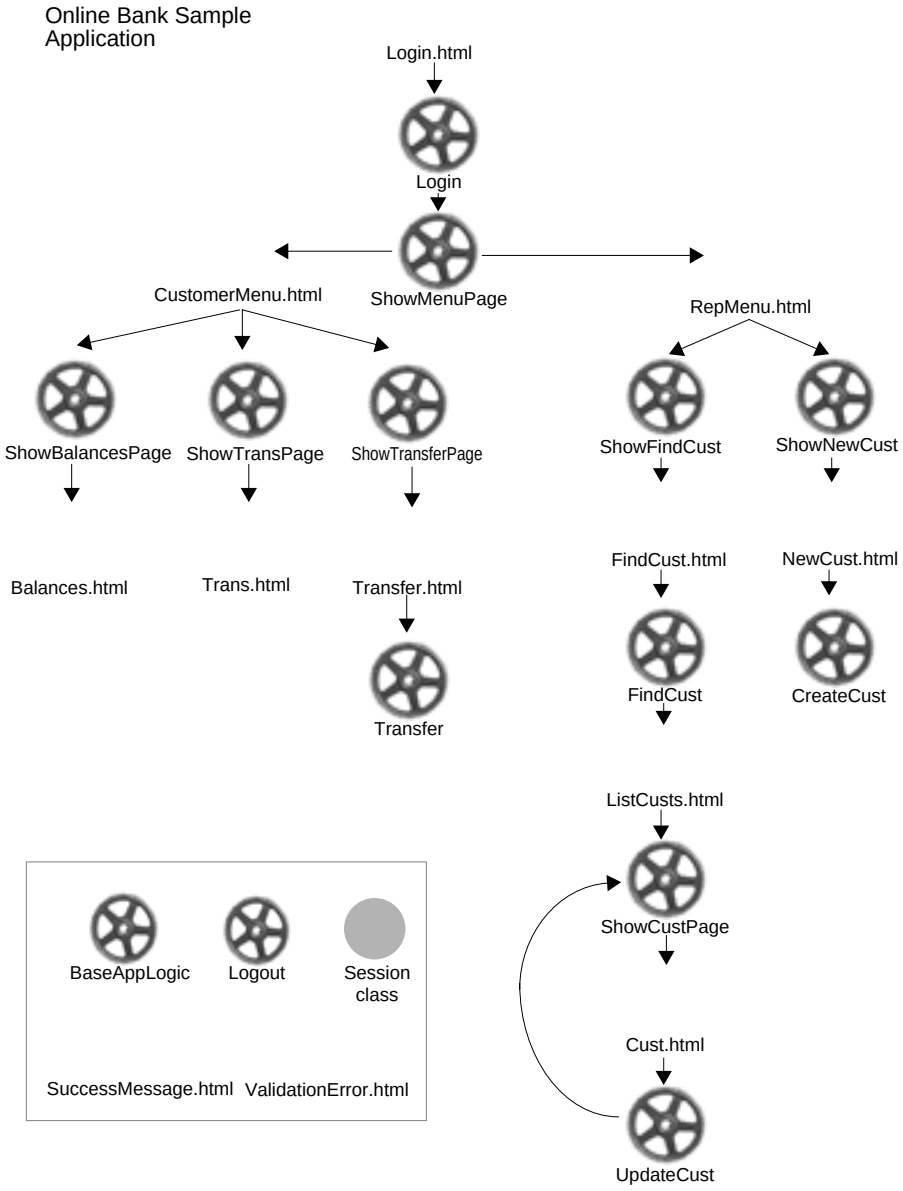
- About the Online Bank Sample Application
- The Online Bank Base AppLogic
- Detailed Walk Through of Funds Transfer Functionality
- Online Bank Registration File

About the Online Bank Sample Application

The Online Bank sample application, which is shipped with the iPlanet Application Server, provides online banking and customer management. In this application, the user interface is HTML pages displayed in Web browsers. Customers can log in to the application over the Internet, view account information, and move funds between accounts. Representatives of the bank can display information about customers, add new customers, and update or delete existing customer data.

You should have already installed the Online Bank sample application on your development system using the iPlanet Application Server installation procedure described on the product CD. If you followed the instructions found there, you should find these files in your sample directory.

The illustration on the following page summarizes the design of Online Bank. The application consists of fourteen AppLogic objects, twelve HTML templates, and assorted additional code modules, including a custom session class.


Online Bank Sample Application
Login.html
Login
ShowMenuPage
CustomerMenu.html
RepMenu.html
ShowBalancesPage
ShowTransPage
ShowTransferPage
ShowFindCust
ShowNewCust
Balances.html
Trans.html
Transfer.html
FindCust.html
NewCust.html
Transfer
FindCust
CreateCust
ListCusts.html
ShowCustPage
BaseAppLogic
Logout
Session class
Cust.html
SuccessMessage.html
ValidationError.html
UpdateCust

The following sequence describes a typical user session with Online Bank:

12. The user navigates their Web browser to the first HTML page of the application, COBLogin.html.
13. When the user clicks the Login button, a request to run the OBLogin AppLogic is sent to iPlanet Application Server. If the user's login information is not valid, the ValidationError.html page is displayed.
14. If the user's login information is valid, the OBLogin AppLogic creates a session for the user. A session is an object that is used to store information about a user's interaction with an application. In the Online Bank sample application, the session stores information about the user, such as user name and type. The OBLogin AppLogic then calls the OBShowMenuPage AppLogic.
15. The OBShowMenuPage AppLogic displays the appropriate menu, depending on the type of user. For example, if the user is a customer, the AppLogic displays CustomerMenu.html.
16. The CustomerMenu.html page contains several hyperlinks. Each hyperlink is coded to call a different AppLogic. When the user clicks one of the hyperlinks, the corresponding AppLogic is executed on the iPlanet Application Server. For example, if the user clicks Show Balances, the OBShowBalancesPage AppLogic is executed.
17. The OBShowBalancesPage AppLogic queries a database to get the customer's account information.
18. The result set of this query is merged with an HTML template, Balances.html, which specifies how to format the data into an output page. After the customer's account information is merged with the template, the resulting HTML page is displayed in the user's Web browser.
19. The user can continue to use hyperlinks to view other HTML pages and execute other AppLogics in the application.
20. Eventually, the user clicks the Logout hyperlink, which causes the OBLogout AppLogic to execute. This AppLogic destroys the user's session object and displays a farewell message to the user.

AppLogic Objects in Online Bank

The following list summarizes each AppLogic object in Online Bank:

- **OBBaseAppLogic** is the base AppLogic class for the Online Bank application. It contains methods to create a user session, initiate contact with the database, handle certain types of errors, and show a success message to the user.
- **OBLogin** logs the user in, starts a session, and calls the ShowMenuPage AppLogic.
- **OBShowMenuPage** displays the appropriate menu page depending on the type of user. The possible menus are CustomerMenu.html, for bank customers, and RepMenu.html, for employees of the bank.
- **OBShowBalancesPage**, called from the CustomerMenu, runs a query to get the customer's account information from a database. It then displays the results in the page Balances.html.
- **OBShowTransPage**, called from the CustomerMenu, generates a report of transactions for the customer's accounts.
- **OBShowTransferPage**, called from the CustomerMenu, displays a page in which the customer can transfer funds from one account to another. The page displayed is Transfer.html.
- **OBTransfer**, called from the Transfer page, stores the funds transfer in a transaction table. The Online Bank application is written under the assumption that the database will perform a batch update of account balances on a periodic basis, based on entries in this table.
- **OBShowFindCust**, called from the RepMenu, displays a page in which a bank employee can enter search criteria to find one or more customers in the database. The page displayed is FindCust.html.
- **OBFindCust**, called from the FindCust page, looks up a list of customers that match the search criteria, and displays the results in ListCusts.html.
- **OBShowCustPage**, called from the ListCusts page, displays detail information about a particular customer. The detail page is Cust.html. The bank employee can make changes to the customer data in this page.
- **OBUpdateCust**, called from the Cust page, changes the customer data in the database. It then calls OBShowCustPage to redisplay the customer page, showing the new data.
- **OBLogout** destroys the user session and displays a farewell message to the user.

The Online Bank Base AppLogic

The AppLogic base class for the Online Bank application, OBBaseAppLogic, contains methods to create a user session, initiate contact with the database, handle certain types of errors, show a success message to the user, and perform data validation and formatting. Below is an abbreviated listing of the code from this class.

```
// ... #include statements
// ...
// Constructor and destructor methods. These contain the
```

// required calls to GXDllLockInc() and GXDllLockDec().

```
OBBaseAppLogic::OBBaseAppLogic()
{
   // Update count of references to the AppLogic library
   GXDllLockInc();
   m_pSession=NULL;
}
OBBaseAppLogic:: ~OBBaseAppLogic()
{
   if(m_pSession) {
      m_pSession->Release();
      m_pSession=NULL;
   }
   // Update count of references to the AppLogic library
   GXDllLockDec();
}
// ********************************************
//* GetOBSession(): Return a reference to this
// application's session object
```

// required calls to GXDllLockInc() and GXDllLockDec().

```
// ********************************************
STDMETHODIMP
OBBaseAppLogic::GetOBSession(OBSession **ppSession)
{
   // See if there is already a session
   if(!m_pSession) {
      // Get the session
      IGXSession2 *pSess=NULL;
      if(((hr=GetSession(0, OB_APPNAME, NULL, &pSess))
         ==GXE_SUCCESS)&&pSess) {
         m_pSession=new OBSession(pSess);
         pSess->Release();
      }
      // If Session has not been created yet, instantiate
      // it
      else if(((hr=CreateSession(GXSESSION_DISTRIB, 600,
         OB_APPNAME, NULL, NULL, &pSess))==GXE_SUCCESS)
         &&pSess) {
         m_pSession=new OBSession(pSess);
         pSess->Release();
      }
   }
   // If we now have a session, return it
   if(m_pSession) {
      *ppSession=m_pSession;
      m_pSession->AddRef();
   }
   return hr;
}

// ********************************************
```

```
// required calls to GXDllLockInc() and GXDllLockDec().

//*  GetOBDataConn(): Return a reference to the Online
// Bank's default database
// ********************************************
STDMETHODIMP
OBBaseAppLogic::GetOBDataConn(IGXDataConn **ppConn)
{
   HRESULT hr=GXE_SUCCESS;
   // Create a ValList for the connection parameters
   // ...
      // Attempt to create the connection
      hr=CreateDataConn(0, GX_DA_DRIVER_DEFAULT, pList,
         m_pContext, ppConn);
      // We're done with the list
      pList->Release();
   }
   return hr;
}
// ********************************************
// Error handling methods
// ********************************************
STDMETHODIMP
OBBaseAppLogic::HandleOBSessionError()
{
   return StreamResult("<HTML><BODY> You must re-log in to
      perform this action </BODY></HTML>");
}

STDMETHODIMP
OBBaseAppLogic::HandleOBSystemError(LPSTR pMessage)
{
   // ... Displays message, logs the occurrence, and
```

// required calls to GXDllLockInc() and GXDllLockDec().

```
    // returns -1
}

STDMETHODIMP
OBBaseAppLogic::HandleOBValidationError(LPSTR pMessage)
{
    // ... Displays message, using ValidationError.html
    // ... logs the occurrence, and returns -1
}

STDMETHODIMP_(BOOL)
OBBaseAppLogic::IsSessionValid()
{
    HRESULT hr=GXE_SUCCESS;
    BOOL ret=FALSE;
    // There must be a user name in the session
    OBSession *pSess=NULL;
    if(((hr=GetOBSession(&pSess))==GXE_SUCCESS)&&pSess) {
        LPSTR pName=NULL;
        if((pName=pSess->GetUserName())) {
            ret=TRUE;
            delete [] pName;
        }
        pSess->Release();
    }
    return ret;
}

// ********************************************
// ShowSuccessMessage()
// ********************************************
```

```
STDMETHODIMP
OBBaseAppLogic::ShowSuccessMessage(LPSTR pMessage)
{
   // ... Displays message, using SuccessMessage.html
}

// ********************************************
// Validation methods
// ********************************************
STDMETHODIMP_(BOOL)
OBBaseAppLogic::ValidateString(LPSTR pTestString,
   LPSTR pName, BOOL isRequired, UINT len,
   BOOL useExactLength)
{
   // ... Tests a string
}

STDMETHODIMP_(BOOL)
OBBaseAppLogic::ValidateSSNString(LPSTR pTestString,
   LPSTR pName, BOOL isRequired)
{
   // ... Tests a Social Security number
}

STDMETHODIMP_(BOOL)
OBBaseAppLogic::ValidatePhoneString(LPSTR pTestString,
   LPSTR pName, BOOL isRequired)
{
   // ... Tests a phone number
}
```

// required calls to GXDllLockInc() and GXDllLockDec().

```
STDMETHODIMP_(BOOL)
OBBaseAppLogic::ValidateZipString(LPSTR pTestString,
   LPSTR pName, BOOL isRequired)
{
   // ... Tests a ZIP code
}

STDMETHODIMP_(LPSTR)
OBBaseAppLogic::FormatSSNString(LPSTR pS)
{
   // ... Formats data as a Social Security number
}

STDMETHODIMP_(LPSTR)
OBBaseAppLogic::FormatPhoneString(LPSTR pS)
{
   // ... Formats data as a phone number
}

STDMETHODIMP_(LPSTR)
OBBaseAppLogic::FormatZipString(LPSTR pS)
{
   // ... Formats data as a ZIP code
}

STDMETHODIMP_(LPSTR)
OBBaseAppLogic::GetDigitString(LPSTR pSource)
{
   // ... Formats data
}
```

Detailed Walk Through of Funds Transfer Functionality

This section walks through the code and templates involved in a transfer of funds between customer accounts. The following application components are presented with comments:

- CustomerMenu.html
- OBShowTransferPage AppLogic
- Transfer.html
- OBTransfer AppLogic
- Other Code

CustomerMenu.html

After a customer logs in, the first page they see is the CustomerMenu HTML page. This page displays a list of hyperlinks which a customer of the bank can use to display various other pages in the application. The page begins with a standard heading.

```
<html>

<head>

<title>Customer Menu</title>

</head>

<body>

<h2>Welcome!</h2>

<h4>What would you like to do today?</h4>

<p><a
```

The following lines define four hyperlinks, allowing the user to choose between four AppLogic objects: CShowBalancesPage, CShowTransPage, CShowTransferPage, and COBLogout. For the purposes of a funds transfer, the user will click the hyperlink that runs the OBShowTransferPage AppLogic.

NOTE	The names used in HTML pages are the names under which the AppLogic objects are registered in the Online Bank Registration File, not necessarily the names with which the AppLogic classes are declared in code.

The names used in HTML pages are the names under which the AppLogic objects are registered in the Online Bank Registration File, not necessarily the names with which the AppLogic classes are declared in code.

```
href="/cgi-bin/gx.cgi/AppLogic+CShowBalancesPage">View

Balances </a><br>

</p>

<p><a

href="/cgi-bin/gx.cgi/AppLogic+CShowTransPage">View Transactions
</a></p>

<p><a

href="/cgi-bin/gx.cgi/AppLogic+CShowTransferPage"> Transfer between
Accounts </a></p>

<p><a

href="/cgi-bin/gx.cgi/AppLogic+COBLogout"> Log out </a></p>

</body>

</html>
```

OBShowTransferPage AppLogic

The OBShowTransferPage AppLogic displays a page in which the customer can transfer funds from one account to another. The page displayed is Transfer.html.

Header File (ShowTransferPage.h)

The header file for the OBShowTransferPage AppLogic does the following:

- Includes C++ and iPlanet Application Builder header files.
- Declares the AppLogic object OBShowTransferPage, which is subclassed from the OBBaseAppLogic class.

The OBShowTransferPage AppLogic includes this header file at the beginning of its source file.

First, in the following code, the header file includes other header files needed for the AppLogic:

```
#ifndef __SHOWTRANSFERPAGE_H__
#define __SHOWTRANSFERPAGE_H__
#include <stdio.h>
#include <gxplat.h>
#include <gxutil.h>
#include <gxapplogic.h>
#include <gxdlm.h>
#include "BaseAppLogic.h"
```

Next, the header file declares the variable that will contain the GUID that uniquely identifies the AppLogic object in the iPlanet Application Server system.

```
extern GUID OBShowTransferPageGUID;
```

Next, the header file subclasses the OBShowTransferPage class from the OBBaseAppLogic class and declares its constructor, destructor, and Execute( ) methods. Note that the constructor and destructor methods in this subclass do not contain the required calls to GXDllLockInc( ) and GXDllLockDec( ). These calls are made in the superclass, OBBaseAppLogic.

```
class OBShowTransferPage : public OBBaseAppLogic
{public:
   OBShowTransferPage();
   virtual ~OBShowTransferPage();
   STDMETHOD(Execute) ();
};
```

Finally, the header file associates the OBShowTransferPage AppLogic with its corresponding GUID.

```
GXDLM_DECLARE( OBShowTransferPage,  OBShowTransferPageGUID);
#endif /* __SHOWTRANSFERPAGE_H__ */
```

Source File (ShowTransferPage.cpp)

The source file begins by including the header files it needs, including ShowTransferPage.h.

```
#include <stdio.h>

#include <gxplat.h>

#include <gxutil.h>

#include <gxagent.h>

#include <gxdlm.h>

#include "ShowTransferPage.h"

#include "gxval.h"

#include "common.h"
```

Next, the source file defines the GUID for the AppLogic that was declared in the header file. You must use the kguidgen utility to generate a unique GUID for each AppLogic, then paste it into the .cpp file. In addition, you must create a registration file (.gxr), using a text editor, and paste the GUID information into the .gxr file as well. For more information, see “Getting Ready to Run an Application” on page 275 of , “Running and Debugging Applications.”

```
// {23F1E8F0-6388-11D1-A1AF-006008293C54}

//

GUID OBShowTransferPageGUID =

{ 0x23F1E8F0, 0x6388, 0x11D1, { 0xA1, 0xAF, 0x00, 0x60, 0x08, 0x29,
0x3C, 0x54 } };

//////////////////////////////////////////////////////////
```

The code next defines constructor and destructor methods for the class.

```
OBShowTransferPage::OBShowTransferPage() {

};


OBShowTransferPage::~OBShowTransferPage() {

};
```

The code next begins implementation of the Execute( ) method, using the STMETHODIMP macro. For more information about this macro, see the iPlanet Application Server Foundation Class Reference.

```
STDMETHODIMP

OBShowTransferPage::Execute()

{
```

The code next declares and initializes the variable hr, which is used throughout the code for return values of calls to methods and functions.

```
HRESULT hr=GXE_SUCCESS;
```

The code next checks for a valid user session, using the method IsSessionValid( ) from the OBBaseAppLogic class. If the user logged in properly through the OBLogin AppLogic, this validation should return successfully.

```
if(!IsSessionValid())
   return HandleOBSessionError();
```

The code next opens a connection to a database, using the method GetOBDataConn( ) from the OBBaseAppLogic class.

```
// Create the data connection
IGXDataConn *pConn=NULL;
if(((hr=GetOBDataConn(&pConn))==GXE_SUCCESS)&&pConn) {
```

The code next retrieves the session data to get information about the user. The user's Social Security number is retrieved.

```
// Pull the session
OBSession *pSession=NULL;
if(((hr=GetOBSession(&pSession))==GXE_SUCCESS)&&pSession) {
LPSTR pSsn=NULL;
// Get the social security number
if((pSsn=pSession->GetSSN())) {
```

If the number was retrieved successfully, the code sets up parameters for the query which will retrieve information about the user's accounts. In this AppLogic, the query is stored in a query file.

```
// Create a vallist for loadQuery parameters
IGXValList *pList=GXCreateValList();
if(pList) {
   GXSetValListString(pList, "ssn", pSsn);
```

If the parameters were set up successfully, the code loads the query and runs it.

```
IGXQuery *pQuery=NULL;
if(((hr=LoadQuery("GXApp/OnlineBank/queries/
   SelCustAccts.gxq", "SelCustAccts", 0, pList,
```

```
   &pQuery))==GXE_SUCCESS)&&pQuery) {
   // Execute the query
   IGXResultSet *pRset=NULL;
   if(((hr=pConn->ExecuteQuery(0, pQuery, NULL,
      NULL, &pRset))==GXE_SUCCESS)&&pRset) {
```

If the query successfully retrieved some data, the code places the data into two template data objects. Each object contains the same data. The two template data objects are needed because the template that will be merged with the data has to display the same information twice (once under “Transfer from account” and again under “Transfer to account”).

```
ULONG numRows=0;
if(((hr=pRset->RowCount(&numRows))==GXE_SUCCESS)&&numRows) {
   // Fill up 2 template data basics with results
   // from the same query to avoid running the same database
   // query twice
   GXTemplateDataBasic *pAcctsTempDB =
      GXCreateTemplateDataBasic("SelCustAccts");
   GXTemplateDataBasic *pAcctsTempDB2 =
      GXCreateTemplateDataBasic("SelCustAccts2");

   if(pAcctsTempDB&&pAcctsTempDB2) {
      char pAcctDesc[200];
      char pAcctNum[200];

      // Pull the column ordinals for the account
      // description and account num
      ULONG acctDescCol=0; pRset->GetColumnOrdinal(
         "OBAccountType_acctDesc", &acctDescCol);
      ULONG acctNumCol=0; pRset->GetColumnOrdinal(
         "OBAccount_acctNum", &acctNumCol);
      char tmpStr[300];
```

The code uses a loop to copy the data from the result set into the template data objects.

```
do {
   pRset->GetValueString(acctDescCol, pAcctDesc, 200);
   pRset->GetValueString(acctNumCol, pAcctNum, 200);
   sprintf(tmpStr, "acctDesc=%s;acctNum=%s", pAcctDesc,
      pAcctNum);
   pAcctsTempDB->RowAppend(tmpStr);
   pAcctsTempDB2->RowAppend(tmpStr);
} while(pRset->FetchNext()==GXE_SUCCESS);
```

Since the data is expected to be hierarchical, an empty template data object is created to act as parent to the other two.

```
GXTemplateDataBasic *pParent=NULL;
if((pParent=GXCreateTemplateDataBasic("Parent"))) {
   // One dummy row, to contain the child groups
   pParent->RowAppend("Dummy=dummy");
```

Then the template data objects that contain the data are added to the parent.

```
   // Add the first template data object to the set
   pParent->GroupAppend(pAcctsTempDB);
   // Add the second template data object to the set
   pParent->GroupAppend(pAcctsTempDB2);
```

The code merges the template data objects with the template Transfer.html to display a page in which the user can set up the transaction.

```
   if(EvalTemplate("GXApp/COnlineBank/templates/
      Transfer.html", pParent, NULL, NULL,
      NULL)!=GXE_SUCCESS)

   // If unsuccessful...
         Result("<HTML><BODY>Unable to evaluate template.
                </BODY></HTML>");
   pParent->Release();
```

```
   }
}
// Release the template data basics, if applicable
if(pAcctsTempDB)
   pAcctsTempDB->Release();
if(pAcctsTempDB2)
   pAcctsTempDB2->Release();
```

If the account data query failed, the code displays an error message, using the HandleOBSystemError( ) method declared in the BaseAppLogic class.

```
}
else
   HandleOBSystemError("Could not retrieve account data");
pRset->Release();
```

Next, the code releases the objects used.

```
               }
               pQuery->Release();
               }
               pList->Release();
            }
            delete [] pSsn;
         }
         pSession->Release();
      }
      pConn->Release();
   }
   else
   // If data connection failed, return error
   HandleOBSystemError("Could not create data connection");
```

Finally, the code sets the return value of the Execute( ) method.

```
   return GXE_SUCCESS;
}
```

Transfer.html

Transfer.html is a page in which the customer can transfer funds from one account to another. The OBShowTransferPage AppLogic uses this template to return results.

```
<html>

<head>

<title>Account Transfer</title>

</head>

<body>
```

The following line is a hyperlink that runs the ShowMenuPage AppLogic, returning the user to the main menu.

NOTE	The names used in HTML pages are the names under which the AppLogic objects are registered in the Online Bank Registration File, not necessarily the names with which AppLogic classes are declared in code.

The names used in HTML pages are the names under which the AppLogic objects are registered in the Online Bank Registration File, not necessarily the names with which the AppLogic classes are declared in code.

```
<p><a

href="/cgi-bin/gx.cgi/AppLogic+CShowMenuPage">Back to Main
Menu</a><br>

</p>

<h2>Account Transfer</h2>
```

The following line sets the action of this page to run the Transfer AppLogic. This action will occur when the user clicks the Transfer button, coded later in the page.

```
<form action="/cgi-bin/gx.cgi/AppLogic+CTransfer" method="POST">
```

The following lines contain GX markup tags, which merge the user's own account information with the page.

```
%gx type=tile id=Parent MAX=100%

  <p><B>Which account would you like to transfer from?</B><br>
```

```
  %gx type=tile id=SelCustAccts MAX=100%
  <input type="radio" name="fromAcct"
   value="%gx type=cell id="SelCustAccts.acctNum"%%/gx%">
   %gx type=cell id="SelCustAccts.acctDesc"%%/gx%: %gx type=cell
id="SelCustAccts.acctNum"%%/gx% <br>
  %/gx%
  </p>
```

The following lines repeat the user's account information, so the user can choose the destination account:

```
<p><B>Which account would you like to transfer to?</B><br>
  %gx type=tile id=SelCustAccts2 MAX=100%
  <input type="radio" name="toAcct" value="%gx type=cell
      id="SelCustAccts2.acctNum"%%/gx%">
   %gx type=cell id="SelCustAccts2.acctDesc"%%/gx%:
      %gx type=cell id="SelCustAccts2.acctNum"%%/gx% <br>
  %/gx%
  </p>
  %/gx%
```

The following lines set up the field where the user tells how much money to transfer between the two accounts selected above:

```
<p><B>How much would you like to transfer?</B> <br>
  <input type="text" name="amount" size="20" value="0.00">
   </p>
```

Finally, the template specifies the Transfer button, which sets off the page action specified earlier, running the Transfer AppLogic.

```
<p><input type="submit" value="Transfer" name="transfer"></p>
</form>
</body>
</html>
```

OBTransfer AppLogic

The OBTransfer AppLogic stores the funds transfer in a transaction table. The Online Bank application is written under the assumption that the database will perform a batch update of account balances on a periodic basis, based on entries in this table.

Header File (Transfer.h)

The header file for the OBTransfer AppLogic does the following:

- Includes C++ and iPlanet Application Builder header files.
- Declares the AppLogic object OBTransfer, which is subclassed from the OBBaseAppLogic class.

The OBTransfer AppLogic includes this header file at the beginning of its source file.

First, in the following code, the header file includes other header files:

```
#ifndef __TRANSFER_H__
#define __TRANSFER_H__
#include <stdio.h>
#include <gxplat.h>
#include <gxutil.h>
#include <gxapplogic.h>
#include <gxdlm.h>
#include "BaseAppLogic.h"
```

Next, the header file declares the variable that will contain the GUID that uniquely identifies the AppLogic object in the iPlanet Application Server system.

```
extern GUID OBTransferGUID;
```

Next, the header file subclasses the OBTransfer class from the OBBaseAppLogic class and declares its constructor, destructor, and Execute( ) methods. Note that the constructor and destructor methods in this subclass do not contain the required calls to GXDllLockInc( ) and GXDllLockDec( ). These calls are made in the superclass, OBBaseAppLogic.

```
class OBTransfer : public OBBaseAppLogic
{
public:
```

```
    OBTransfer();
    virtual ~OBTransfer();
    STDMETHOD(Execute) ();
};
```

Finally, the header file associates the OBTransfer AppLogic with its corresponding GUID.

```
GXDLM_DECLARE( OBTransfer,  OBTransferGUID);
#endif /* __TRANSFER_H__ */
```

Source File (Transfer.cpp)

The source file begins by including the header files it needs, including Transfer.h.

```
#include <stdio.h>
#include <gxplat.h>
#include <gxutil.h>
#include <gxagent.h>
#include <gxdlm.h>
#include "Transfer.h"
#include "gxval.h"
#include "common.h"
```

Next, the source file defines the GUID for the AppLogic that was declared in the header file. You must use the kguidgen utility to generate a unique GUID for each AppLogic, then paste it into the .cpp file. In addition, you must create a registration file (.gxr), using a text editor, and paste the GUID information into the .gxr file as well. For more information, see “Getting Ready to Run an Application” on page 275 of , “Running and Debugging Applications.”

```
// {2754DA40-638C-11D1-A1AF-006008293C54}
//
GUID OBTransferGUID =
{ 0x2754DA40, 0x638C, 0x11D1, { 0xA1, 0xAF, 0x00, 0x60, 0x08, 0x29,
0x3C, 0x54 } };
////////////////////////////////////////////////////////////
```

The code next defines constructor and destructor methods for the class.

```
OBTransfer::OBTransfer() {
```

```
};
OBTransfer::~OBTransfer() {
};
```

The code next begins implementation of the Execute( ) method, using the STMETHODIMP macro. For more information about this macro, see the iPlanet Application Server Foundation Class Reference.

```
STDMETHODIMP
OBTransfer::Execute()
{
```

The code next declares and initializes the variable `hr`, which is used throughout the code for return values of calls to methods and functions.

```
HRESULT hr=GXE_SUCCESS;
```

The code next checks for a valid user session, using the method IsSessionValid( ) from the OBBaseAppLogic class. If the user logged in properly through the OBLogin AppLogic, this validation should return successfully.

```
if(!IsSessionValid())
   return HandleOBSessionError();
```

The code next retrieves the input data which the user specified in the page Transfer.html.

```
// Get parms from the input form
LPSTR pFromAcct=NULL;
LPSTR pToAcct=NULL;
LPSTR pAmountString=NULL;
pFromAcct=GXGetValListString(m_pValIn, "fromAcct");
pToAcct=GXGetValListString(m_pValIn, "toAcct");
pAmountString=GXGetValListString(m_pValIn, "amount");

// Convert amount to double
double amount=0.0;
if(pAmountString)
   amount=atof(pAmountString);
```

The code next performs some tests on the input. This is always advisable when data comes in from outside the application, such as from end users.

```
// Validate input
if((!pFromAcct)||(!pToAcct))
   return HandleOBValidationError("You must specify both an
      account to transfer from and an account to transfer
      to.");
else if(!strcmp(pFromAcct, pToAcct))
   return HandleOBValidationError("You may not transfer into
   the account from which you are transferring.");


// Check the amount
if(amount<=0.0)
   return HandleOBValidationError("You must enter a positive
      non-zero amount of money to transfer.");
```

The code then retrieves the current date and time, so that it can time-stamp the transaction.

```
// Get the current date time
GXDATETIME dt;
GXGetCurrentDateTime(&dt);
char dateStr[50];
sprintf(dateStr, "%d-%d-%d %d:%d:%d", dt.year, dt.month,
   dt.day, dt.hour, dt.minute, dt.second);
Log(dateStr);
```

The code next opens a connection to a database, using the method GetOBDataConn( ) from the OBBaseAppLogic class.

```
// Create the data connection
IGXDataConn *pConn=NULL;
if(((hr=GetOBDataConn(&pConn))==GXE_SUCCESS)&&pConn) {
```

The code next accesses the database to retrieve a handle to the OBTransactions table, and gets the ordinal numbers of the columns in the database. The column ordinals are required in later method calls.

```
IGXTable *pTable=NULL;
if(((hr=pConn->GetTable("OBTransaction", &pTable))
   ==GXE_SUCCESS)&&pTable) {
   // Look up the column ordinals for the table
   ULONG transTypeCol=0; pTable->GetColumnOrdinal(
      "transType", &transTypeCol);
   ULONG postDateCol=0; pTable->GetColumnOrdinal("postDate",
      &postDateCol);
   ULONG acctNumCol=0; pTable->GetColumnOrdinal("acctNum",
      &acctNumCol);
   ULONG amountCol=0; pTable->GetColumnOrdinal("amount",
      &amountCol);
```

The code then starts a transaction. This ensures that both parts of the funds transfer are executed together.

```
IGXTrans *pTx=NULL;
if(((hr=CreateTrans(&pTx))==GXE_SUCCESS)&&pTx) {
   pTx->Begin();
```

The code then performs an INSERT command to add a row for the withdrawal portion of the transaction.

```
// Create a new row for the withdrawal half of the
// transaction
pTable->AllocRow();
pTable->SetValueString(acctNumCol, pFromAcct);
pTable->SetValueInt(transTypeCol, OB_TRANSTYPE_WITHDRAWAL);
pTable->SetValueDateString(postDateCol, dateStr);
pTable->SetValueDouble(amountCol, amount*-1.0);

// Add the row using the transaction
if(pTable->AddRow(0, pTx)==GXE_SUCCESS) {
```

The code then performs another INSERT command for the deposit portion of the transfer.

```
pTable->AllocRow();
pTable->SetValueString(acctNumCol, pToAcct);
pTable->SetValueInt(transTypeCol, OB_TRANSTYPE_DEPOSIT);
pTable->SetValueDateString(postDateCol, dateStr);
pTable->SetValueDouble(amountCol, amount);

// Add the row using the transaction
if(pTable->AddRow(0, pTx)==GXE_SUCCESS) {
```

If both INSERT commands were performed successfully, the code commits the transaction.

```
   pTx->Commit(0, NULL);
   ShowSuccessMessage("Your transfer has been completed");
}
```

If either INSERT command failed, the code rolls back the transaction.

```
      else {
         pTx->Rollback();
         HandleOBSystemError("Could not insert transaction");
      }
   }
   else {
      pTx->Rollback();
      HandleOBSystemError("Could not insert transaction");
   }
```

The remainder of the code handles any earlier failures in creating objects, getting connections, and so on.

```
         pTx->Release();
      }
      else
         HandleOBSystemError("Could not start transaction");
      pTable->Release();
   }
```

```
   else
      HandleOBSystemError("Could not access table
         OBTransaction");
   pConn->Release();
}
else
   HandleOBSystemError("Could not create data connection");
```

Finally, the code sets the return value of Execute( ).

```
   return GXE_SUCCESS;
}
```

Other Code

The code in the file applogics.cpp establishes to the iPlanet Application Server the entry point in a dynamically loadable, shared library module (DLM) for each AppLogic object in the Online Bank application, including OBShowTransferPage and OBTransfer.

```
GXDLM_IMPLEMENT_BEGIN();
GXDLM_IMPLEMENT( OBLogin,  OBLoginGUID);
GXDLM_IMPLEMENT( OBShowMenuPage,  OBShowMenuPageGUID);
GXDLM_IMPLEMENT( OBShowBalancesPage,  OBShowBalancesPageGUID);
GXDLM_IMPLEMENT( OBShowTransPage,  OBShowTransPageGUID);
GXDLM_IMPLEMENT( OBShowTransferPage,  OBShowTransferPageGUID);
GXDLM_IMPLEMENT( OBLogout,  OBLogoutGUID);
GXDLM_IMPLEMENT( OBTransfer,  OBTransferGUID);
GXDLM_IMPLEMENT( OBShowNewCustPage,  OBShowNewCustPageGUID);
GXDLM_IMPLEMENT( OBShowFindCustPage,  OBShowFindCustPageGUID);
GXDLM_IMPLEMENT( OBFindCust,  OBFindCustGUID);
GXDLM_IMPLEMENT( OBShowCustPage,  OBShowCustPageGUID);
GXDLM_IMPLEMENT( OBCreateCust,  OBCreateCustGUID);
GXDLM_IMPLEMENT( OBUpdateCust,  OBUpdateCustGUID);
GXDLM_IMPLEMENT_END();
```

For more information, see the description of the GXDLM_IMPLEMENT, GXDLM_IMPLEMENT_BEGIN, and GXDLM_IMPLEMENT_END macros in the iPlanet Application Server Foundation Class Reference.

Online Bank Registration File

The Online Bank registration file (COnlineBank.gxr) contains the GUIDs for the AppLogics in the Online Bank sample application. Blank lines, where they occur, are significant. The names given to AppLogics in the .gxr file do not necessarily have to match the class names with which the AppLogic classes are declared in code.

For more information about .gxr files, see “Getting Ready to Run an Application” on page 275 of , “Running and Debugging Applications.”

```
AppLogic COBLogin::COnlineBank
{C1B5E720-6153-11D1-A1AE-006008293C54}

COnlineBank
AppLogic CShowMenuPage::COnlineBank
{C8899CE0-61DC-11D1-A1AE-006008293C54}

COnlineBank
AppLogic CShowBalancesPage::COnlineBank
{5AE8CE70-62D0-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CShowTransPage::COnlineBank
{B10DF490-62E7-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CShowTransferPage::COnlineBank
{23F1E8F0-6388-11D1-A1AF-006008293C54}
```

```
COnlineBank
AppLogic COBLogout::COnlineBank
{0C598E40-6389-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CTransfer::COnlineBank
{2754DA40-638C-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CShowNewCustPage::COnlineBank
{C6C91850-64F0-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CShowFindCustPage::COnlineBank
{092043B0-64F2-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CFindCust::COnlineBank
{9962F0B0-64F3-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CShowCustPage::COnlineBank
{769F1300-64FC-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CCreateCust::COnlineBank
{C9103DF0-6549-11D1-A1AF-006008293C54}

COnlineBank
AppLogic CUpdateCust::COnlineBank
{951550D0-655D-11D1-A1AF-006008293C54}
```

```
COnlineBank
```

Appendix A

Implementation Tips

This appendix includes the following tips:

- System Configuration Tips
- Memory Management Tips
- Database and Query Tips
- HTML Tips
- Session Tips
- Tips for Calling an Applogic From Another Applogic
- Streaming Tips

System Configuration Tips

Install the Web server, the iPlanet Application Server, and the database on different machines so that they will not have to compete for resources. The two servers use Web Connector to communicate.

When using the Web Connector, remember that the CGI variables or NSAPI/ISAPI variables are set on the machine where Web Connector is running, not on the iPlanet Application Server machine. When changing the configuration of Web Connector, make sure to update the registry on the correct machine.

Memory Management Tips

When resources are no longer needed, you must explicitly free them by calling the Release( ) method. However, even if you are extremely careful about reference counting, it is likely that some memory will not be released properly. Over time, such memory leaks will consume the available resources on your machine. Therefore, when using C++, it is advisable to use a tool capable of memory use analysis, such as Purify 4.1 by Pure Atria.

Database and Query Tips

To improve database performance:

- Design the database schema carefully.
- Tune the database cache sizes and the number of connections.

Query files are easier to maintain than queries written in code using method calls. You can use the Query Designer to set up queries visually, and let the Query Designer automatically generate the query file. You can also write query files using a text editor.

Avoid queries with more than two or three joins, as this will degrade performance. To improve performance, consider denormalizing the database. Denormalization results in duplicate data in the database, but simplifies queries and improves performance in the database.

Use the caching capabilities of iPlanet Application Server to improve performance. Plan on caching early in the design process, because its use affects how you design the client side of the application. All the criteria needed for caching must be present in the input parameters of the request. For example, when the clients are Web browsers, this means the caching criteria must be present as fields on an HTML form, or as arguments in the URL that calls the AppLogic.

When you are finished using a result set, release it by calling `Release( )`. This method releases the database connection so that it is available for use by other application code. Do not release the result set or close the database connection until you are finished using the result set. Just because the query has run and returned a result set interface, that doesn't mean all the data is there. Typically the result set is buffered, and live database cursors may still be open. Therefore, when you reach the last row in the buffer, the result set object still needs the connection to get the next batch of rows into the buffer.

HTML Tips

To improve the performance of any application with Web browser clients, always use the NSAPI/ISAPI plugin.

Avoid building HTML strings in code, because this is difficult to maintain and update. Keep the HTML in files and templates. HTML designers can then improve and enhance your application's Web browser presentation without having to edit business logic code.

Session Tips

Avoid storing too much data in a session. Every time you save or retrieve the session-related data, the whole IGXValList object is involved. This can impact the performance of the application.

Tips for Calling an Applogic From Another Applogic

By using the NewRequest( ) method, an AppLogic can call another AppLogic. Usually, this involves a distributed process-to-process communication. This is slower than an in-process local procedure call, but has the added benefits of allowing AppLogic location transparency, load balancing, more support for partitioning, and result caching. For truly fine-grained, often-called operations, however, the remote communication costs are not worth the benefits.

Streaming Tips

The SaveSession( ) method in the GXAppLogic class performs some processing of HTTP headers, which must be sent before the HTTP body. Therefore, if your application uses sessions, and also streams HTML results to a Web browser, be sure to call SaveSession( ) before calling any streaming methods, including EvalTemplate( ) or EvalOutput( ).

The method called SaveSession( ) exists in both the IGXSession2 interface and the GXAppLogic class. The method in the GXAppLogic class is a wrapper that calls the method in the interface, and performs some tasks that ensure that the session is accessible to future AppLogics. The SaveSession( ) method in the interface saves session data only. Therefore, be sure to call SaveSession( ) in the GXAppLogic class at least once after a session is created.

Streaming Results from EvalTemplate( ) or EvalOutput( ) Using IGXTemplateData

If you are using an IGXTemplateData object rather than a database query as the source of data for a call to EvalTemplate( ) or EvalOutput( ), you can increase the perceived performance of the call by using the following technique. Instead of populating the IGXTemplateData object by calling RowAppend( ) repeatedly, implement the IGXTemplateData interface yourself and call EvalTemplate( ) or EvalOutput( ) much earlier in the AppLogic code. In this way, the Template Engine can call the IGXTemplateData object as it needs data and return results as they are available, keeping the user waiting much less time for a response.

The Template Engine calls the MoveNext( ) method in the IGXTemplateData interface each time it needs a new row of data; for example, when it has completed one pass in a tile tag and is ready to start the next iteration of that tile. If you have implemented your own MoveNext( ) method, you can use that code to retrieve data as needed. This takes the place of calling RowAppend( ) repeatedly to populate the IGXTemplateData object all at once. After MoveNext( ) is called, Get( ) is called to retrieve the values in that row.

For example, the following code shows how the AppLogic code looks when you use RowAppend( ):

```
// Populate the in-memory template data.  The number
// of calls to RowAppend() is unlimited.  Meanwhile,
// the user is waiting for an unknown length of time
// until the full template data set is populated.
//
GXTemplateDataBasic *td;

td = new GXTemplateDataBasic("offices");
td->RowAppend("office=New York;revenue=150");
td->RowAppend("office=Hong Kong;revenue=130");
// ... add more records here.
```

```
// Pass the finished data set to EvalTemplate().
HRESULT hr;

hr = EvalTemplate("salesReportByOffice.html",
   (IGXTemplateData *) td, NULL, NULL, NULL);
td->Release();
return hr;
```

Now suppose you create your own implementation of the IGXTemplateData interface or subclass from the GXTemplateDataBasic class. The following code is in the header file:

```
class MyTemplateDataBasic : public GXTemplateDataBasic
{
public:
   MyTemplateDataBasic(LPSTR group) :
      GXTemplateDataBasic(group)
   {
      // Prepare the retrieval of the offices records here.
      // We don't have to get all the data yet, just
      // the first record data.
   }
   STDMETHOD(IsEmpty) (
      LPSTR group,
      BOOL *empty
   );
   STDMETHOD(MoveNext) (
      LPSTR group
   );
   STDMETHOD(GetValue) (
      LPSTR       szExpr,
      IGXBuffer **ppBuff
   );
```

```
};
```

The following code is in the source file:

```
STDMETHODIMP
MyTemplateDataBasic::GetValue(LPSTR field,
   IGXBuffer **ppBuff)
{
   if (strcmp(field, "offices.office") == 0)
   {
      IGXBuffer *office;
      // ... retrieve current office field value here.
      *ppBuff = office;
      return NOERROR;
   }
   if (strcmp(field, "offices.revenue") == 0)
   {
      IGXBuffer *revenue;
      // ... retrieve current revenue field value here.
      *ppBuff = revenue;
      return NOERROR;
   }
   return GXTemplateDataBasic::GetValue(field, ppBuff);
}

STDMETHODIMP
MyTemplateDataBasic::IsEmpty(LPSTR group, BOOL *empty)
{
   if (strcmp(group, "offices") == 0)
   {
      boolean isOfficeRecordSetEmpty;
      // ... determine if the data set is empty.
      *empty = isOfficeRecordSetEmpty;
```

```
      return NOERROR;
   }
   return GXTemplateDataBasic::IsEmpty(group, empty);
}

STDMETHODIMP
MyTemplateDataBasic::MoveNext(LPSTR group)
{
   if (strcmp(group, "offices") == 0)
   {
      HRESULT noMoreRecords;
      // Move to next record in offices data set here.
      // This is where we can dynamically compute
      // the next record.
      //
      // Return NOERROR (0) if next record is available.
      // Return non-zero if no more records.
      return noMoreRecords;
   }
   return GXTemplateDataBasic::MoveNext(group);
}
```

The following code shows how the AppLogic code looks when you let the Template Engine retrieve the data through MoveNext( ):

```
// Use our own GXTemplateDataBasic subclass, which is
// smart enough to dynamically retrieve office records
// when called back by the template engine.  This allows
// data to be streamed back to the user as it becomes
// available, instead of waiting for the entire
// data set to be created first in memory.
//
MyTemplateDataBasic *td;
```

```
td = new MyTemplateDataBasic("offices");
// MyTemplateDataBasic retrieves office records
// as necessary, so we do not prepopulate it here.
// Pass the MyTemplateDataBasic object to EvalTemplate().
HRESULT hr;
hr = EvalTemplate("salesReportByOffice.html",
   (IGXTemplateData *) td, NULL, NULL, NULL);
td->Release();
return hr;
```

Glossary

administrator See system administrator.

aggregate expression An expression in a query that summarizes values from one database column across several rows. You can use aggregate expressions to specify computed fields. The aggregate functions available depend on your database server, but those typically supported are `Min( )`, `Max( )`, `Count( )`, `Avg( )`, `Sum( )`, `First( )`, and `Last( )`.

alias An alternate name. In a query, an alias is a name given to a database table, column, or computed field.

API See application programming interface (API).

applet An applet is a small application written in Java that runs in a web browser. Typically, applets are called by web pages to provide special functionality.

application flow The perceived progress of activity from page to page in a browser-oriented application.

application programming interface (API) A set of instructions that a computer program can use to communicate with other software or hardware that is designed to interpret that API. In an iPlanetApplication Server application, the API consists of the iPlanet Application Server Foundation Class Library. Computer programs can use this API to communicate with iPlanetApplication Server.

application server A program that runs an application in a client/server environment, executing the logic that makes up the application and acting as middleware between a browser (client) and a data source (server).

AppLogic object A special Java class responsible for completing a well-defined, modular task within a iPlanet Application Server application. Use AppLogics to perform actions such as handling form input, accessing data, or generating data used to populate HTML templates.

banded report See grouped report.

base class See also superclass. A class from which another class is derived.

base session resource The class defined to handle the iPlanet Application Server application session variables.

binary large object (BLOB) A large block of bits that can be stored in a database. A BLOB is useful for storing any large piece of data, such as pictures or sounds, that does not need to be interpreted by the database.

BLOB See binary large object (BLOB).

browser See web browser.

browser events Actions that occur on the browser page, such as passing the cursor over a particular component, that can trigger actions specified by JavaScript objects on the page.

build project Compile all source files in the project that have been edited since the last time they were compiled.

business logic The implementation rules determined by an application's requirements and processed by AppLogics on the iPlanet Application Server.

cache See result cache.

cell tag A type of GX markup tag that displays a dynamic data value.

child query A flat query that represents an inner level of data in a hierarchical query. Each child query is nested relative to another flat query, which is its parent query. Every flat query in a hierarchical query is a child query, except the outermost.

class A named set of methods and member variables that define the characteristics of a particular type of object. The class defines what types of data and behavior are possible for this type of object.

clean project Remove all object files from the project, leaving only source.

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.

column A field in a database row.

compile To translate source code written by a programmer into object code that can run on a computer. A compiler is a program that performs this translation.

component Reusable objects that you can place on a page or template to perform a certain task. For example, an ImageLink component uses a GIF or JPG image as an anchor for an HTML hypertext link. The behavior and appearance of components are determined by their properties.

Component Object Model (COM) A specification that provides a standard way for objects and their clients to interact. COM specifies only how objects interact, not how applications are structured internally or how they are implemented.

computed field A field in a query that displays the result of an expression rather than stored data. The database engine recalculates the value each time it runs the query.

connection A database connection is a communication link with a database or other data source. AppLogics can create and manipulate several database connections simultaneously to access data.

connection validation Guarantees that an IDataConn can be built for a set of named connections listed for an AppLogic.

constructor A method that instantiates a class.

cookie A variable that your application can send to a web browser to be stored there for a specified length of time. Each time a web browser views an HTML page in your application, the cookies from that browser are sent to the application. Cookies are domain-specific and can take advantage of the same web server security features as other data interchange between your application and the server. Thus, cookies are useful for privately exchanging data between your application and the web browser.

custom property editor A dialog box that helps you determine the correct value for a complex property definition.

database forms wizard A wizard that produces an HTML page with a data input form, a data model to establish database table relationships, a query based on the data model, an AppLogic to process the query, an HTML template to display the results of the query, and an optional search form to enable user-initiated database queries.

data-bound properties Properties that enable components to be bound to a data set in an AppLogic.

data connection A logical connection between an application and a relational database.

data expression An expression containing one or more columns of data, which consist of a data set and a data field, with an optional display format.

data field One column of data in a data set.

data model An entity relationship (ER) diagram that specifies the data source tables and relationships used in your application.

data set A user-populated data source for the iPlanet Application Server template engine.

data source A collection of data electronically stored within a relational database, legacy system, or object database.

default A value that is automatically assigned by the application when the user or programmer does not specify a value.

DELETE query A statement that specifies which data to delete from a database.

deploy Create a copy of all the files in a project on one or more servers, in such a way that one or more iPlanet Application Servers and (optionally) one or more web servers can run the application.

design-time The behavior (in the test server) or appearance (in the iPlanet Application Builder windows) of an object when the application is being developed.

detail record The result of a secondary query, based on a master record.

display format See format mask.

distributed computing A collection of computers linked together. Such systems can exist on a local area network (LAN), a wide area network (WAN), or the Internet. Distributed systems make several types of advanced computing systems possible, including client/server, multi-tier, and partitioned applications.

DLM See dynamically loadable module (DLM).

dockable window A window, such as the Project or Properties window, that has the ability to "snap into place" against the workspace border. When a window is attached in this way, it cannot be overlapped by other windows.

download project Copying application files from a server back to a development machine. Note that this does not re-create the development environment if the files were filtered during development.

drag and drop Clicking an object, holding the mouse button down while moving the cursor and the object to a destination (dragging), and then releasing the button to insert the object at the destination (dropping).

dynamically loadable module (DLM) A binary executable file that can be loaded while an application is running. In Windows NT or Win95 systems, DLM is another name for a Dynamic-Link Library (DLL). In UNIX systems, DLMs are implemented as ELF shared libraries.

Editor Beans Java objects which generate and maintain components.

enable (AppLogic) Enable the application server to run registered AppLogics.

ER diagram Describes the attributes of database entities and the relationships among them.

event Named actions that you register with the iPlanetApplication Server. The event occurs either when a timer expires or when the event is called from application code at run time. Typical uses for events include periodic backups, reconciling accounts at the end of the business day, or sending alert messages.

event handler JavaScript object on an HTML page or template that handles a browser event at run time.

execute server Part of the iPlanetiPlanet Application Server that handles executive functions such as load balancing and process management.

field The smallest identifiable part of a table in a database. A field is the intersection of a row and a column.

Also, a unit of data in a result set. Each field in a result set has a name, which corresponds to either a database column or an expression. Each field contains a single data value.

filtering Removing development-oriented information from files while deploying.

flat query A query that produces a result set that is not divided into levels or groups. The result set of a flat query is like a table.

floating toolbar The state of a toolbar when it appears unattached in the center of the workspace, as opposed to being docked along the workspace borders. Floating toolbars have title bars to identify them.

format mask A mask applied to data to specifically tailor its format for display. Options include numeric formats (integer, percentage), and custom date formats. Also known as display format.

generated code JavaScript code generated by Editor Beans which should not be edited.

globally unique identifier (GUID) A unique number that identifies an AppLogic object and is used to request that iPlanetApplication Server runs that AppLogic object.

group A set of rows in a result set that have one or more field values in common.

grouped report A report that shows records in logical groups, such as sales grouped by geographic region, and can show summary data for each group.

GUID See globally unique identifier (GUID).

GX markup tag A special type of syntax used in templates to indicate where dynamic data is to be merged with the template. A GX tag is made up of two tags, <GX ...> and </GX>, and the text between them. Some GX tags are represented with % rather than < or >, as in %GX TYPE="cell" ...% %/GX%. This is equivalent to <GX TYPE="cell" ...> </GX> at run time.

.gxm file A file that keeps track of all files belonging to a project. Also called a project file.

GXML template A definition for a dynamically generated set of output data. Data retrieved from a database or other data source at run time is sent back to the client in a self-describing stream of output.

.gxr file A file containing information that allows .java files and other files in a project to be registered with the iPlanet Application Server. Also called a registration.

handle The verticle strip on the left side of a toolbar by which you can drag the toolbar.

hierarchical query A query that combines several flat queries to construct a result set with multiple nested levels of data.

HTML See Hypertext Markup Language (HTML).

HTML page A page coded in HTML and intended for display in a web browser. Many HTML pages are view-only images and text, but HTML pages can also form the interface of a web application. A user can type data in an HTML page, then click a button on the page to submit the data. The web application manipulates the data and sends a response to the user on another HTML page.

HTML template A definition for a dynamically generated HTML page. AppLogic uses HTML templates to present dynamic data. Data retrieved from a database, or otherwise generated dynamically, is merged with the template to create a database report or other type of HTML page, which is displayed to the user. Contrast with HTML page.

HTTP See Hypertext Transport Protocol (HTTP).

hyperlink A word or phrase that the user can click to display another page in an online document.

Hypertext Markup Language (HTML) The coding language used to create documents that can be displayed by web browsers. Each block of text is surrounded by codes that indicate the nature of the text, such as heading, body paragraph, or list item. Additional codes are used to create hyperlinks and call applets or AppLogic objects. HTML codes are surrounded by angle brackets < >.

Hypertext Transport Protocol (HTTP) The protocol for communicating hypertext documents across the Internet and the World Wide Web. HTTP provides a structure for communication between HTTP clients and HTTP servers.

image URL Source code for an image. The URL can be relative (local server) or absolute (local or remote server). The URL can be determined dynamically if the component that requires it is in a template.

include tag A type of GX markup tag that displays HTML output created by evaluating another template.

inheritance A mechanism 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.

input validation The set of rules which defines a valIn.

input wizard A wizard that produces a static HTML page containing an input form, an AppLogic to process the form, an HTML template to display the results, and optionally a query to retrieve data from a data source.

INSERT query A statement that specifies which data to add to a database.

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 run time.

interface Description of the services provided by an object. An interface defines a set of functions, called methods, and 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 can not be instantiated.

JavaScript A language that can run as a script in an HTML page, allowing screen actions outside the scope of HTML, including responses to browser events.

Java server Part of the iPlanet Application Server that runs and manages Java objects.

layout view An HTML editor window display that shows the HTML page similarly to how it appears in a browser at runtime.

link URL A target for a hypertext link. These include static pages, AppLogics, and other web sites. The URL can be relative (local server) or absolute (local or remote server). The URL can be determined dynamically if the component that requires it is in a template.

listing See tabular report.

load balancing A technique for distributing the user load evenly among identical AppLogic objects distributed across several computers running iPlanetApplication Server.

login wizard A wizard that produces a static HTML page containing a login form, an AppLogic to verify a user name and password, and an HTML template to display the results of a successful login.

master record The primary target of a query.

member A variable or method declared in a 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.

metadata Represents information that is passed into the run time's BaseResource constructor.

iPlanet Application Server Foundation Class Library A set of interfaces and classes provided by iPlanet that can be used to develop object-oriented iPlanetApplication Server applications. The classes in the iPlanet Application Server Foundation Class Library define many types of objects you can include in iPlanetApplication Server applications, such as AppLogic objects, data connections, queries, and result sets.

object A programmed entity with the following characteristics:

An object embodies both data and behavior.

Objects come into existence at run time through the process of instantiation.

Each object is based on a definition, which is called a class.

Many parts of an iPlanetApplication Server application, such as AppLogic objects, queries, and result sets, are objects.

object-oriented programming A method for writing programs using classes, not algorithms, as the fundamental building blocks. At run time, the classes give rise to objects, which perform the tasks of the application.

ODBC Open Database Connectivity (ODBC)

online application server A server that stores, manages, and executes dynamic Internet and intranet applications. An online application server is specifically designed to run such applications quickly and efficiently. iPlanetApplication Server is an online application server.

Open Database Connectivity (ODBC) A standard protocol used by many database vendors to provide an interface to outside applications. iPlanetApplication Server applications can interact with databases that comply with ODBC 1.0 and 2.0.

outline view An HTML editor window display that shows the structural relationships between HTML tags on the page.

override To write new code that replaces the default code of an inherited method.

palette A window containing components that you can drag and drop into HTML files to create pages and templates.

palette tabs Individual sections of the palette that contain similar elements.

parameter The data passed between methods, AppLogic objects, and other program code.

A placeholder for dynamic data that is passed into a prepared database command at run time.

parent query A flat query that represents an outer level of data in a hierarchical query. Each parent query is nested outside another flat query, which is its child query. Every flat query in a hierarchical query is a parent query, except the innermost.

prepared command A database command (INSERT, UPDATE, DELETE, or query) that is precompiled to make repeated execution more efficient. Prepared commands can contain parameters.

presentation logic The process of generating output for an iPlanet Application Server application.

project A collection of related files that, when deployed, constitute a web application.

project file See .gxm file.

project map A window that displays file dependencies. A project map is useful for visually representing the page flow of an application.

project window A directory listing of files in a project. Files can be grouped by folder or listed alphabetically.

properties window A window showing the properties for a selected object.

property Name-value pairs that indicate how an object behaves or appears. For example, a Name property might contain a name that identifies the object to other objects, while a Background Color property defines a background color for the object.

property definition Value associated with a given property that, together with the other properties for a given object, determines the object's appearance and/or behavior.

query A statement that specifies which data to retrieve from a database. Typically, the results of a query are displayed in a report.

query file A query file is a file that contains the specification for a flat or hierarchical query. Query files are useful for running legacy SQL SELECT statements. You can also use query files to write new queries.

rebuild project Remove all object files (clean project) and compile all source files in the project.

register (AppLogic) Register the Java methods that make up an AppLogic with the Java Virtual Machine running on the application server.

registered servers Servers registered to iPlanet Application Builder for the purpose of deployment. You can not deploy to a server until it is registered.

registration The process of informing iPlanetApplication Server of the existence of an AppLogic object, code module, or security information.

registry file See .gxr file.

relationship A named connection between data source tables.

replace tag A type of GX markup tag that substitutes a dynamic data value for a specified string.

report A formatted presentation of data. In an iPlanetApplication Server application, a report is an HTML page presented to the user in response to a request for information. AppLogic objects create reports by combining hierarchical result sets and HTML templates.

request A message from a client to a server, asking for data or another service.

In an iPlanetApplication Server application, a request is a message that causes an AppLogic object to run on the iPlanetApplication Server. A request uses a unique name or globally unique identifier (GUID) to identify the proper AppLogic object to handle the request. The request can include parameters to be passed to the AppLogic object. Requests can come from clients, AppLogic objects, or other code.

result cache Storage in iPlanetApplication Server that holds the output from an AppLogic object so that the output can be accessed repeatedly without the necessity of running the AppLogic object again.

results wizard A wizard that produces an AppLogic to retrieve data, optionally accessing a query to retrieve data from a data source, and an HTML template to display the results.

result set A set of data records returned by a query. A record is a set of fields. Each field in the record has a name, which corresponds to either a database column or an expression, and each field contains a single data value.

row A record in a database table. Each row is made up of several columns.

runtime The behavior (in the server) or appearance (in a browser) of an object when the application runs.

select distinct (query) A query type that retrieves only the unique instances of the requested target search items.

SELECT query A statement that specifies which data to retrieve from a database, as specified by your data model.

sequence A sequential number generator which exists in a database. Some database vendors refer to a sequence as a serial, identity, or autoincrement. A sequence is useful for generating transaction-safe numbers for database transaction applications.

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.

session A continuous series of interactions between a user and an iPlanet Application Server application.

session accessors Any method that starts with set and takes a String or starts with "get".

session validation The set of rules that guarantees that the application is in a valid state to perform the functionality of a specific AppLogic.

session variables Used by many AppLogics in an application to store and access data that is shared throughout the application.

source view An HTML editor window display that shows the source code for the page.

SQL See Structured Query Language (SQL).

standard query A query that produces a result set that is not divided into levels or groups. The result set of a standard query is like a table.

state A distributed data storage mechanism which you can use to store the state of an application. The application state is a collection of application variables whose scope is global within the application. Information in the state layer can be organized in a hierarchical structure, or tree.

stored procedure A block of statements written in SQL or programmatic SQL and stored in a database. You can use stored procedures to perform any type of database operation, such as modifying, inserting, or deleting records. The use of stored procedures improves database performance by reducing the amount of information that is sent over a network.

streaming 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.

Structured Query Language (SQL) A language commonly used in relational database applications. In an iPlanetApplication Server application, you can specify SQL SELECT, INSERT, UPDATE, and DELETE commands.

subclass A class that is derived from and is a special case of another class, called a base class or superclass.

superclass A class from which another class, called a subclass, is derived. See also base class.

system administrator The person who is responsible for installing and maintaining iPlanetApplication Server software and for deploying production iPlanetApplication Server applications.

table A named group of related data in rows and columns in a database.

tabular report A report, sometimes called a listing, that prints all the records retrieved from the database.

tag See GX markup tag or HTML.

target window Name of the window or frame that displays the results of a hypertext link.

template See GXML template or HTML template.

template engine The part of the server responsible for taking HTML template files and merging them with the data from an AppLogic.

template map An object that maps fields in a template to the data used to replace those fields. With a template map, you can assign values to special placeholders that will be evaluated at run time. You can also use a template map to link column names in a table to field names that you have used in a template. A template map allows your application to use the same template file with data from different data sources.

test server A version of the iPlanet Application Server that runs in debugging mode for local testing purposes.

tile tag A type of GX markup tag which repeats the tags and text nested within it. Tile can be used in two ways: repeating a fixed number of times, or repeating for each row in a result set.

tooltip A word or phrase that appears whenever you briefly place the mouse over a toolbar button. Tool tips are useful reminders of a button's purpose.

transaction A set of database commands that succeed or fail as a group. All the commands involved must succeed for the entire transaction to be correct.

trigger A trigger is a stored block of SQL or PL/SQL statements that is associated with a table, runs in response to an INSERT, UPDATE, or DELETE operation, and runs only under certain specified conditions.

Uniform Resource Locator (URL) An address that uniquely identifies an HTML page or other resource. web browsers use URLs to specify which pages to display.

update query A statement that specifies which data to modify within a database.

URL See Uniform Resource Locator (URL).

user A person who runs a computer application.

user validation Validation written by the user.

validation A method for ensuring that the contents of a form field are within certain parameters. If a user enters data outside the parameters, a dialog box appears notifying them of the error and what to do to correct it.

valIn An argument to an AppLogic.

valOut The return values from internal AppLogic calls.

variable A named storage location for data that can be modified while a program is running. Each variable has a unique name that identifies it within its scope. Each variable can contain a certain type of data.

Web See World Wide Web.

web application A computer program that uses the World Wide Web for connectivity and user interface. 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 server, connected to the browser through the World Wide Web.

web browser Software that is used to view resources on the World Wide Web, such as web pages coded in HTML.

web page See HTML page.

web server A computer that stores and manages HTML pages and web applications. The web server responds to user requests from web browsers.

wizard A code generator that provides a framework for creating the most commonly-used types of application development components.

World Wide Web A network of many computers linked together by their ability to understand the HyperText Transfer Protocol (HTTP). Two types of computers make up the Web: clients and servers. Clients are computers with web browsers installed on them. Servers are computers that store and manage the information requested by the clients.

workspace The main window of iPlanet Application Builder. The workspace contains the windows, toolbars, and dialog boxes that constitute the user interface.

Index

SYMBOLS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z