Adapter Development Guide

iPlanet™ Integration Server

Version 3.0

August 2001

Contents

Preface

The *iIS Adapter Development Guide* explains how to use the iIS software development kit to build an iIS adapter using the 'C' language or iPlanet UDS (TOOL). This manual discusses the design and functioning of an iIS adapter, and includes guidance on using the 'C' and TOOL adapter SDKs. This information can also be a useful reference if you build an iIS adapter with other tools, such as Java.

This preface contains the following sections:

Product Name Change

Forte Fusion has been renamed the iPlanet Integration Server. You will see full references to this name, as well as the abbreviation iIS.

Audience for This Guide

This manual assumes familiarity with iIS, knowledge of HTTP, as well as knowledge of the native API of the applications you are integrating, and how to represent application data requirements in XML. This manual builds on information contained in the *iIS Backbone System Guide*, and is supplemented by information in the *iIS Backbone Integration Guide*.

If you are new to iIS or want to familiarize yourself with the components of iIS and how they interact in an iIS system, refer to the *iIS Conceptual Overview*.

Organization of This Guide

The following table briefly describes the contents of each chapter:

Chapter	Description
Chapter 1, "iIS Adapters"	Provides an overview of the design and operation of an iIS adapter.
Chapter 2, "Building an iIS TOOL Adapter"	Explains how to construct a TOOL adapter.
Chapter 3, "Building an iIS 'C' Adapter"	Explains how to construct a 'C' adapter.

Text Conventions

This section provides information about the conventions used in this document.

Format	Description
italics	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.

Format	Description
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (iIS, JSP, iMQ). Uppercase text can also represent a constant. Type uppercase text exactly as shown.
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Other Documentation Resources

In addition to this guide, there are additional documentation resources, which are listed in the following sections. The documentation for all iIS products can be found on the iIS CD. Be sure to read "Viewing and Searching PDF Files" on page 9 to learn how to view and search the documentation on the iIS CD.

iIS documentation can also be found online at http://docs.iplanet.com/docs/manuals/iis.html.

The titles of the iIS documentation are listed in the following section.

iPlanet Integration Server Documentation

iIS Adapter Development Guide

iIS Backbone Integration Guide

iIS Backbone System Guide

iIS Conceptual Overview

iIS Installation Guide

iIS Process Client Programming Guide

iIS Process Development Guide

iIS Process System Guide

Online Help

When you are using an iIS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iIS distribution: `FORTE_ROOT/userapp/forte/cln/*.hlp`.

When you are using a script utility, such as FNscript or Cscript, type help from the script shell for a description of all commands, or help <*command*> for help on a specific command.

Documentation Roadmap

A roadmap to the iIS documentation can be found in the *iIS Conceptual Overview* manual.

iIS Example Programs

iIS example programs are shipped with the iIS product and installed in two locations, one for process development (using the process engine) and one for application integration (using the iIS backbone).

Process Development Examples Process development examples are installed at the following location:

```
FORTE_ROOT/install/examples/conductr
```

The PDF file, `c_examp.pdf`, describes how to install and run the examples in this directory. The Appendix to the *iIS Process Development Guide* also describes how to install and run the examples.

Application Integration Examples Process integration examples are installed at the following location:

```
FORTE_ROOT/install/examples/fusion
```

Each example has its own sub-directory, which contains a README file that explains how to install and run the example.

Viewing and Searching PDF Files

You can view and search iIS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

NOTE You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

➤ **To copy the documentation to a client or server**

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the `doc` directory; the location is not dependent on the iIS distribution. You may want to consolidate your iIS documentation with the documentation for your iPlanet UDS distribution.

2. Set up a directory structure that keeps the `iisdoc.pdf` and the `iis` directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

NOTE To uninstall the documentation, delete the `doc` directory.

➤ **To view and search the documentation**

1. Open the file `iisdoc.pdf`, located in the `doc` directory.
2. Click the Search button at the bottom of the page or select Edit > Search > Query.

3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

NOTE For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

   To return to the `iisdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `iisdoc.pdf` home page or select Edit > Search > Results.

Chapter 1

iIS Adapters

iIS adapters need to construct, parse and transport the XML documents being exchanged over HTTP between the iIS application and its proxy.

This chapter explains, at a high level, the design and operation of an iIS adapter independent of the APIs you use to construct the adapter.

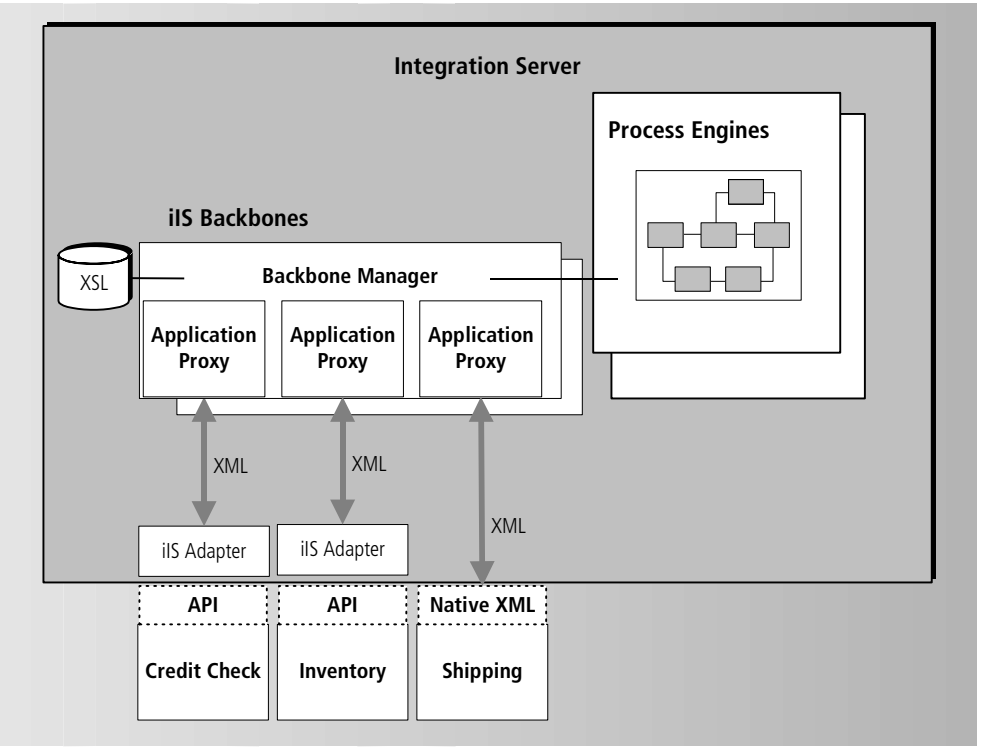
For details on iIS backbone architecture and application proxy operations, see the *iIS Backbone System Guide*.

What is an iIS Adapter?

An iIS adapter integrates an application that does not have a native XML/HTTP interface into an iIS application system. The adapter translates between the XML used by the proxy and the application's native method for interaction. The native method can be any communication protocol or linked-in library supplied by the application vendor. To perform its functions, an iIS adapter typically implements an XML message builder, an XML parser, and HTTP transport services.

Figure 1-1 illustrates an iIS backbone in which two applications use adapters to communicate with partnered proxies:

Figure 1-1 An iIS Backbone With Two Adapters



Prebuilt adapters iIS provides a number of prebuilt adapters, an adapter toolkit, as well as integration services. For current information, see the Sun web site.

iIS Adapter Operations

Adapters translate requests or responses from the proxy into interactions with the application. The XML documents sent or received by the proxy specify work requests at a high level (for example, “Perform Credit Check”), while the adapter handles the details on how its partner application should perform these requests.

From the point of view of the business process and the proxy, applications can function in one of two ways: either to request services (for example, start a business process), or to provide services (perform work requests). Since a business process typically consists of a number of work actions to be performed, an iIS backbone usually has more applications that provide, rather than request, services. The adapter assumes the same function as its application; a typical adapter can therefore be thought of as a service provider.

Both service providers and service requestors send and receive XML documents, examine these documents, and act upon the contents. What varies is the order of operations and the destination for the response document. These differences are reflected in the adapter’s HTTP registration, with a provider becoming an HTTP server, and the requestor an HTTP client.

It is important to make this distinction early in the design phase because the operations of the adapter are based on its function. Adapters representing service provider applications (HTTP servers) accept HTTP requests (with request details described in XML) and send back HTTP responses (with response details described in XML). Adapters representing service requestor applications (HTTP clients) send HTTP requests and receive HTTP responses.

NOTE A single adapter can act as both a client and a server without conflict of the APIs. However, in understanding basic adapter design, it is useful to think of only one type of operation at a time.

HTTP Requests and Responses

HTTP is a request/response protocol in which a request to a server must result in a response message. This mode of functioning is fundamental to the HTTP protocol itself, and is therefore reflected in iIS adapter HTTP operations.

Although there is always a response to a request, the response does not necessarily provide the disposition to the request; other processing may take place in the interim, and it may take time for the request to be fulfilled.

Synchronous mode When an adapter represents a service provider, its interaction with the proxy is either synchronous or asynchronous with respect to work done by the application. In either case the operation begins when the proxy makes an activity ACTIVE and sends this status to the application.

If the adapter interacts synchronously with the proxy, the response contains the disposition of the activity (for example, whether it should be completed or aborted). The proxy awaits the response and does not take any action until the adapter notifies it that the activity has been completed (or aborted or rolled back). The proxy assumes that the adapter's response refers to the current process or activity. The adapter does not need to pass back the process ID and activity ID.

Asynchronous mode If the proxy and adapter are interacting asynchronously, the initial response acknowledges receipt of the request, but it does not contain the disposition of the activity. The response frees the proxy to perform other actions before the adapter notifies it about the status of the activity. In this case, the application must send the process ID and activity ID back to the proxy. In addition, the XSL stylesheet used for this scenario must ensure that these IDs are made available, and the proxy must be configured for client/server communications with the adapter. For details about synchronous and asynchronous behavior, see the *iIS Backbone System Guide*; for details about constructing the appropriate XSL stylesheets, see the *iIS Backbone Integration Guide*.

Service Provider Operations

You register a service provider adapter as an HTTP server so that it listens for work requests.

Handling HTTP Requests

The adapter's XML parser examines the XML it receives to decode the request and any relevant details about the requested service (typically corresponding to iIS process attributes). The XSL stylesheets that are configured as "outbound" from the partner proxy determine the document format and content. For example, a document requesting a credit check might contain the customer account number and the amount of credit requested. The credit check application would indicate approval or disapproval of the request by setting the appropriate value in a process attribute and communicating this information in the HTTP response message.

Once the processing is complete, the adapter builds a response and passes it to the adapter's HTTP server, which returns it to the sender as an HTTP response message.

Service Requestor Operations

Service requestors operate differently than service providers. The requestor is probably receiving requests for services from its partner application, which might be an interactive client such as a Web-based order entry application. The adapter's responsibility in this situation is to translate these requests into XML/HTTP requests to the iIS engine.

When the application informs the adapter of a specific service request, the following actions occur:

Creating HTTP Requests

- The adapter's XML message builder constructs an XML document
- The adapter sends the document to its partner proxy
- The partner proxy applies its inbound stylesheets to process the document and applies its outbound stylesheets to return the response
- The adapter examines the result and informs the requesting partner application of the outcome of its request

For more details on proxy documents, see the *iIS Backbone System Guide*; for details on writing stylesheets, see the *iIS Backbone Integration Guide*.

Session Authentication

An iIS adapter can request or provide session authentication in order to interact with the proxy. With this feature enabled, only authorized users are able to obtain access. The HTTP request-response messages contain authorization headers that are managed automatically by cookies.

The adapter must be coded to implement session authentication, which might include the ability to send an authentication document automatically in the header. The authentication document is a type of proxy document that uses the FusionXML authentication scheme described below.

The adapter's partner proxy must be configured as either an HTTP client or HTTP server that can provide or request authentication.

NOTE If the adapter's partner proxy is process-based it can act as either an HTTP client or HTTP server; if it operates without a process engine, it can only be configured as an HTTP server and the authentication credentials must be provided on the local machine. For configuration details, see the *iIS Backbone System Guide*.

Authentication Schemes

The available authentication schemes are Basic and FusionXML. Both types of authentication strings are encoded in Base64 before transmission:

- Basic, as defined in HTPP 1.1, consists of a *name:password* combination that is returned in the authorization header.
- FusionXML, an extension defined by iIS, allows more complex user data to be transmitted, such as that contained in a user profile. FusionXML is a string that is returned in the WWW-Authenticate header that lets the application or the proxy know the authentication scheme that is required. A user profile is specified by an authentication document; for details on this document type, see the iIS Backbone online help. This scheme is only available if the adapter's partner proxy is process-based.

Authorization Header

As discussed in the preceding section, an adapter may function either as an HTTP server (to an HTTP client proxy) or HTTP client (to an HTTP server proxy). Session authentication is implemented by the component that acts as the HTTP server.

When an HTTP client makes a request to an HTTP server, session authentication proceeds as follows:

1. The server determines whether session authentication is required.
2. If so, and a session does not exist, the server returns the request with a status code of Unauthorized (401), along with a specification of the required authentication scheme (Basic or FusionXML).
3. When the client receives this information, it resends the original request, along with a completed authorization header that uses the required authentication scheme.
4. When the server receives the required information, it validates the session request and takes appropriate action, typically fulfilling the original request. If the information it receives is incorrect, it rejects the attempt by sending another Unauthorized response.

Service Provider Authentication

When an adapter is acting as a service provider and requires session authentication, it rejects requests until its client proxy returns the Authorization header with the required user information. The proxy must be configured as an HTTP client that can fulfill session authentication requests.

Service Requestor Authentication

When an adapter is acting as a service requestor and requires session authentication, the proxy must be configured as an HTTP server that can provide authentication using either Basic or FusionXML. If a server proxy is not configured with an authentication scheme, it does not require authentication of service requestors.

For sample session authentication code, see the subsequent chapters of this manual.

NOTE Client proxies are configured using the FNscript commands `SetAplSession`, `AddAplRole`, and `SetAplProfile`. Server proxies are configured using the FNscript command `SetAuthentication`. For details, see the *iIS Backbone System Guide*.

Recovery and Reliability

iIS supports a number of features for load balancing, recovery and reliability. For example, the proxy can be configured to retry communications with its adapter and can have alternative partner adapters to contact in the event of an adapter failure. For details, see the *iIS Backbone System Guide*.

Chapter 2

Building an iIS TOOL Adapter

The iIS TOOL Adapter SDK includes several APIs: two for XML parsing, and one to provide standard HTTP communications.

This chapter explains how an iIS TOOL adapter functions, how to access information on XML APIs, and how to use the HTTPSupport API.

TOOL Adapter Operations

This section describes the operations of a TOOL adapter, building on the information about general adapter operations in Chapter 1, "iIS Adapters."

HTTP Operations

HTTP operations are different for service providers and service requestors. The difference is that a server gets requests and sends (returns) a response, while a client sends a request and gets (is returned) a response. In both cases, messages are sent and received, so the adapter must know how to create as well as interpret XML messages.

Service Providers

An adapter that provides services establishes itself as an HTTP server. To do this, it must have a machine (for example, *mymachine.mycompany.com*) and port (for example, *1234*) on which it is available. The application URL property configured for the partner proxy contains this information so that the proxy can locate the adapter. While the proxy needs to know the entire URL, the adapter implicitly knows what machine it is running on, and only needs to determine the port number on which to listen for proxy requests.

Once these parameters are identified, the HTTP server can make itself available to the proxy. As messages arrive, the adapter's XML parser is called. Each XML parser invocation returns a response message to the originator of the request.

HTTP requests arrive invoking HTTP `<method>` requests. The proxy makes only Post or Get requests, sending a proxy document that includes instructions to the applications, according to the XSL stylesheets defined for the proxy. The TOOL adapter generates an error message for any other HTTP method request.

The TOOL adapter processes the XML document. Once an adapter's work is completed, it exits. After an adapter exits, the port no longer listens for messages. An error is generated if a service requestor proxy sends a request after the service provider adapter has exited.

Service Requestors

The operations of a service requestor adapter are simpler. As an HTTP client, it needs to know the destination URL for its message. This URL consists of a machine and an optional port specification. If no port is specified, the default HTTP port 80 is used. Once the destination is known, the client packages XML requests into an HTTP message and sends the message. The destination server proxy processes the message and returns a response. The adapter interprets the response, which may include generating errors.

XML Operations

As XML documents arrive, their data is extracted and the associated application operations are performed. You can use a parser or a scanner for this purpose. As message size and complexity increases, however, an XML parser is recommended.

When the adapter needs to send information, its message builder must construct the document. It can either create a document and insert node information, or construct the document directly. For details, see the DOM API specification (`http://www.w3c.org/`). The iIS Backbone online Help specifies the subset of DOM classes supported in the TOOL XMLParser library.

iIS Application Operations

Once it is determined that certain work is to be performed, the application must do that work and respond with status information when it completes the work. For example, if the application performs a credit check, the response documents would include information about whether credit is authorized.

It is up to system integrator to determine how the adapter operates in conjunction with its application.

The iIS TOOL Adapter SDK

The iIS TOOL Adapter SDK supports standard XML tools and the HTTP protocol.

XML Parser APIs For convenience, iIS supports standard XML APIs. The Document Object Model (DOM) API, as specified by the World Wide Web Consortium (W3C), is a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents. The document can be further processed and the results of that processing incorporated back into the final document. The DOM API uses an object graph to represent the XML document. In addition to parsing, the DOM API lets you create and build XML documents, as well as add, modify, and delete elements and content.

The second API is the Simple API for XML (SAX). SAX provides a simpler interface than the DOM, and is therefore useful if you only need to parse XML documents. In addition, SAX is useful if your documents are too large to be compiled into an object graph in available memory.

The iIS Backbone online Help directs you to the web pages for information on DOM and SAX, and specifies the subset of DOM classes supported in the TOOL XMLParser library.

HTTP Services API For an iIS TOOL adapter, you can use the iIS HTTPSupport API to implement standard HTTP communications.

The iIS TOOL Adapter Example

The TOOL Adapter example is a simple order processing system. Each adapter/application pair is represented by an iPlanet UDS window that presents the user with information and options. The example runs automatically, with windows timing out after a few seconds. The example also has a batch mode.

The TOOL adapter example process definition specifies several activities. When the order is entered, the first activity is a credit check. Once the credit check is complete, the process routes to a shipping activity. When the shipping activity is complete, billing takes over, followed by order verification. If the credit check is rejected, the order routes directly to order verification.

The credit check activity is the only activity with a significant role. It sets the CreditApproved attribute, using the length of the billee's name as the basis for approval. Order verification looks at this attribute to determine the message to display.

There are a number of stylesheets included with the example files:

Filename	Contents
orderin.xsl	Templates that process messages from the adapter to its proxy
orderout.xsl	Templates that process messages from the proxy to its adapter
nopein.xsl	Templates for processing messages exchanged between applications that interact with independent proxies
nopexsl.xsl	An example of using xsl:include to import another stylesheet

NOTE In the TOOL adapter example, the adapter/application pair is actually a single application that tailors its behavior based on its name, which is provided in command-line arguments. This means that the application can service any of the activities, or it can run as a service requestor, placing new orders. While not a typical configuration for a real system, it nonetheless illustrates how iIS operates.

Using the HTTPSupport API

The HTTPSupport library provides HTTP communications from a TOOL program. Like a servlet engine, HTTPSupport allows an object to send and receive simple documents using a set of HTTP user classes. This object is a service requestor, a service provider, or both. The HTTPSupport library is available in iPlanet UDS releases that are compatible with iIS.

This section provides an overview of HTTPSupport interfaces and classes useful to iIS projects. For a conceptual overview of the library, see the *iPlanet UDS Programming Guide*. For a complete HTTPSupport API reference, see the iIS Backbone online Help.

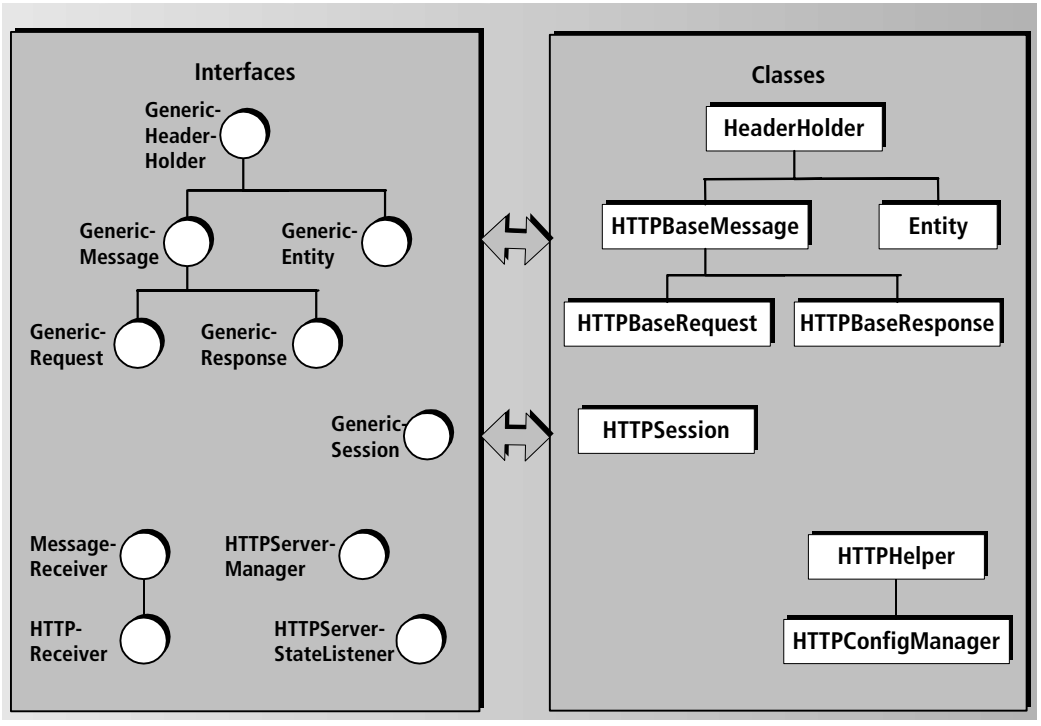
The HTTPSupport API supports Version 1.1 of the HTTP protocol (by default).

HTTPSupport API Interface and Class Hierarchy

The HTTPSupport library consists of interfaces and classes. The interfaces declare the behavior (methods) that the classes implement.

Figure 2-1 shows the hierarchy of HTTPSupport interfaces and classes that are useful in building an iIS TOOL adapter. The arrows in the diagram indicate classes that implement similarly-named interfaces, for example, HTTPSession implements GenericSession.

Figure 2-1 TOOL HTTPSupport API Interface and Class Hierarchy



HTTPSupport Interfaces

The following table indicates the HTTPSupport interfaces and the types of methods each interface declares:

Interface	Methods Declared
GenericHeaderHolder	Set, get, add, remove, and list HTTP headers and their values
GenericEntity	Write and read data; compose nested entities
GenericMessage	Set and get message body, version numbers, and other information
GenericRequest	Get protocol, remote address/host, scheme, and server port for HTTP requests
GenericResponse	Set and get status and status string for HTTP responses
GenericSession	Get application and network session, close session
MessageReceiver	Initialize, process HTTP request/build response, terminate
HTTPReceiver	DoPost, DoGet (DoDelete, DoHead, DoOptions, DoPut, DoTrace are available but not used by the iIS proxy)
HTTPServerManager	AddInstances, ActiveServers, CloseApplicationSession, FreeServers, GetApplicationSessions, GetInstances, RemoveInstances, SetTimeOut
HTTPServerStateListener	State to inform an HTTP server object of state changes.

HTTPSupport Classes

The following table indicates useful HTTPSupport classes and the methods each class defines:

Class	Methods Defined
HeaderHolder	All GenericHeaderHolder methods
Entity	All GenericEntity methods
HTTPBaseMessage	Add/get cookies; set/get message values, such as header date, content type/length, version, session; decode/encode Base 64; all GenericMessage methods
HTTPBaseRequest	Get URL address; set/get method, request URI, query string; send; all GenericRequest methods
HTTPBaseResponse	All GenericResponse methods

Class	Methods Defined
HTTPSession	All GenericSession methods
HTTPHelper	Advertise (allows an HTTP object to receive messages)
HTTPConfigManager	Get/set configuration values

Building the TOOL Adapter

This section explains in detail how to use the TOOL HTTPSupport API to construct HTTP services and includes guidance on providing XML processing. Sample code from the iIS TOOL adapter example is included to illustrate important details. For a complete HTTPSupport API reference, see the iIS Backbone online Help.

Building an HTTP Server

An adapter acting as a service provider requires an HTTP server object. The server object supplies the methods that implement the server. In the TOOL adapter example, the base class FusionServer and its subclass OrderProcessor implement the HTTP server.

Advertising the Server

As an HTTP server, the adapter uses the `HttpHelper.Advertise` method to make itself available to process messages. You provide this method with the port on which the adapter is to listen, and the object that will act as the server.

Implementing a Receiver Interface

The HTTP server object must also implement the methods declared for either `MessageReceiver` or `HTTPReceiver`. The TOOL adapter example implements `HTTPReceiver`, accepting `Post` or `Get` requests to perform the adapter's work. iIS proxies only generate `Post` or `Get` requests, as determined by the proxy's XSL stylesheets. If an adapter receives a different HTTP method (such as `Trace` or `Put`), it generates an error.

In the example, the FusionServer class is a base class that handles HTTP details. The FusionServer class implements `HTTPReceiver`. The `HTTPSetUp` method advertises the server, as shown in the following code sample:

```
optionString.ReplaceParameters('port = %1',
servicePort will pass in port value
  IntegerData(value = servicePort));

if (ConfigMgr <> nil) then
  ConfigMgr.SetConfigValue(config = HTTP_CONFIG_SESSION_VALUE,
      value = HTTP_SESSION_MAYBE);
  ConfigMgr.SetConfigValue(config=HTTP_CONFIG_TCPSESSION_VALUE,
      value = HTTP_TCPSESSION_PERSISTENT);
end if;

Advertising the object
Helper.Advertise(obj = self, param = optionString.value);
```

The real work of the TOOL adapter is performed in the OrderServer and OrderProcessor classes. When `OrderServer.DoPost` gets a message, it calls its associated `OrderProcessor.Process` with the message text, returning a message body for the response, as shown in the following code sample:

```
respData : httpdc.Entity;
xmlInput : httpdc.Entity = (httpdc.Entity)(request.body);
statusCode : integer;

MsgProcessor.Process(request = xmlInput,
    status = statusCode,
    response = respData);

response.SetStatus(statusCode = statusCode);
response.body = respData;
```

Building an HTTP Client

HTTP client processing is straightforward. An `HTTPBaseRequest` message is constructed, and its `Send` method is called. This message returns an `HTTPBaseResponse`. The response message contains any results returned by the proxy, including errors.

In the TOOL adapter example, `OrderProcessor.PlaceOrder` contains the code for HTTP client operations. The new order document is constructed and embedded in an `HTTPBaseRequest`. The request message is sent to the proxy, whose URL is stored in PartnerURL. The response is returned and processed, as long as the statusCode returned is acceptable.

The following code sample shows an XML message being formatted directly.

NOTE The carriage returns ("\n") and indentation performed by GenerateXmlList are included only to enhance human readability; they are neither required by XML nor by the proxies.

```
Format text directly
etd : TextData = new(Value = '<MsgDoc>\n');
line : TextData = new();
etd.Concat('<NewOrder>\n');
etd.Concat(GenerateXmlList(depth = 1));
etd.Concat('</NewOrder>\n');
etd.Concat('</MsgDoc>');
```

The following code shows how to construct the HTTP request to contain the XML message:

```
Construct the request
req : HTTPBaseRequest = new();
xmlEnt : XmlEntity = new();
req.body = xmlEnt;

xmlEnt.WriteText(source = etd);
```

The remaining code shows how to send the request, construct a document from the response, and parse the XML contents of the response message from that document:

```
Send the request
response : HTTPBaseResponse =
          req.Send(methodName = 'POST', url = PartnerURL.value);

statusCode : integer = response.GetStatus();

Construct document from the response
if (statusCode = SC_OK) then
  doc:Document = XmlTools().NewDocument(
NewDocument imports XML
      response.body.GetContentStream());
Process XML
  GetValue(doc, 'Cfnumber', ProcessID);
else
  -- Error received from server
  ProcessID.value = response.GetStatusString();

  errorString : TextData = new();
  response.Body.ReadText(target = errorString);
  msg : TextData = new();
  msg.ReplaceParameters('Error:  %1 : %2',
          TextData(value = response.GetStatusString()),
          errorString);
  task.lgr.putline(msg);
end if;
```

XML Processing

The preceding code sample showed how the XML response message content is imported into a document for parsing. The `NewDocument` method imports the document after removing unnecessary white space. The resulting document is passed to the `GetValue` method to find the value of the Cfnumber element. The `GetValue` method invokes the DOM to walk through the document, searching for a node with the Cfnumber element, and returns the value of this node in the ProcessID attribute. (See the TOOL adapter example for the additional code that performs this work.) Similarly, the `OrderProcessor.FillFromXml` method fills the attributes when new requests are received (that is, when the adapter is acting as a service provider).

You can also import a collection of XML text into a document using the `ImportDocument` method of the XMLParser class. Once in document form, the document methods can find the relevant parts of the XML document.

The following code shows how to use the `ImportDocument` method:

```
doc : Document = new();
doc.ImportDocument(xmlMessage);

GetValue(doc, 'ProcessID', ProcessID);
GetValue(doc,'ActivityID', ActivityID);
GetValue(doc, 'WorkType', WorkType);

FindAttValue(doc, TextData(value = 'Billee'), Billee);
FindAttValue(doc, TextData(value = 'Shippee'), Shippee);
FindAttValue(doc, TextData(value = 'ItemCount'), ItemCount);
FindAttValue(doc, TextData(value = 'CreditApproved'),
CreditApproved);
FindAttValue(doc, TextData(value = 'OrderID'), OrderID);
```

The incoming message is imported into a document for parsing. This document is then used to find the values for ProcessID, ActivityID, and WorkType. FindAttValue is specialized, in that it knows how to look for the attribute value where that attribute is named so it can parse the XML. The following code sample shows the attribute name, type, and value to be parsed:

```
<AttList>
  <Att>
    <AttName>Billee</AttName>
    <AttType>TextData</AttType>
    <AttValue>Beelzebub</AttValue>
  </Att>
</AttList>
```

The `FindAttValue` method called for Billee will return the value Beelzebub ()it understands the message structure involved.

Using CDATA Sections

XML messages are simply collections of text in which the text is formatted into XML elements using specific punctuation (for example, `<start-tag>`*some value*`</end-tag>`). If messages contain greater than, less than, and ampersand signs ("`>`", "`<`", and "`&`"), proxies and adapters might construct illegal XML responses. To avoid this type of error, you should normally escape any text value (for example, a process attribute) in a `CDATA` section. However, if the value contains

the end of CDATA marker "]]>", you must do textual replacement on the values within the message using XML internal entities (for example, you would replace "<" with "<" rather than use a CDATA section). For sample code, see OrderProcessor.MakeXMLOK() in the iIS TOOL adapter example.

Session Authentication

As explained in Chapter 1, "iIS Adapters," when an HTTP client makes a request to an HTTP server, it is up to the server to determine whether session authentication is required and, if so, which type of authentication (Basic or FusionXML). If session authentication is required, and a session does not exist, the following steps occur:

1. The server returns the request with a status code of Unauthorized (401) along with a specification of the required authentication scheme.
2. When the client receives this information, it resends the original request, along with a completed Authorization header that uses the required authentication scheme.
3. When the server receives the required information, it validates the session request and takes appropriate action, typically fulfilling the original request. If the information it receives is incorrect, it rejects the attempt by sending another Unauthorized response.

Service Requestors

The following code samples show how session authentication can be implemented when a TOOL service requestor adapter, acting as an HTTP client, initiates a work request. If the proxy does not require session authentication, the proxy proceeds with the request; if authentication is required, the adapter must provide the required information. In the first code sample, the `OrderProcessor.PlaceOrder` method is used to determine whether authentication is required or not:

```
tryCount : integer = 0;
  while (tryCount < 3) do
    response = request.Send(methodName = 'POST', url =
                PartnerURL.value);
    statusCode : integer = response.GetStatus();
Authorization not needed
    if (statusCode = SC_OK) then
      -- Proceed with work request...
      exit;
    elseif (statusCode = SC_UNAUTHORIZED) then
      -- Provide authentication...
      Authenticate(response, request);
      tryCount = tryCount + 1;)
      continue;
    end if;
  end while;
```

If the adapter receives the `Unauthorized` response, it must determine the authentication scheme the proxy requires and generate the appropriate one. In the following code sample the `OrderProcessor.Authenticate` method is used to recognize and provide the authentication scheme (Basic or FusionXML). The `HTTPBaseMessage.EncodeBase64` method provides Base64 encoding for the user information:

```
-- Figure out required authorization ...
authHeader : TextData = TextData(response.getHeader('WWW-
                Authenticate'));
authString : TextData = new();
authMechanism : TextData;
Basic authentication
if (authHeader.MoveToString('Basic', ignoreCase = TRUE)) then
          -- Basic authentication is required.  Generate Basic.
    authMechanism = TextData('Basic');
    namePw : TextData = new();
    namePw.ReplaceParameters('%1:%2',
                  lname,
                lpw);
             -- Create authorization header contents...
    authString.ReplaceParameters('Basic %1',
Base64 encoding
               request.EncodeBase64(namePw));
FusionXML authentication
elseif (authHeader.MoveToString('FusionXml', ignoreCase = TRUE))
  then
    authMechanism = TextData('FusionXml');
    authDoc : TextData = new();
The FNAuthenticate quoted string must be on one line
    authDoc.ReplaceParameters('<FNAuthenticate>
            <FNUserProfileName="%1"/><FNUser Name="%2"
                Password="%3"/></FNAuthenticate>',
              TextData('FNProxyProfile'),
              myName,
              myPassword);
             -- Create authorization header contents...
    authString.ReplaceParameters('FusionXml %1',
Base64 encoding
                  request.EncodeBase64(authDoc));
end if;
elseif (authHeader.MoveToString('FusionXml', ignoreCase = TRUE))
  then
  -- Set the authorization header...
  request.SetHeader('Authorization', authString.Value);
```

Service Providers

The following code samples show how session authentication can be implemented when a TOOL service provider adapter, acting as an HTTP server, receives a work request from a client proxy.

NOTE The TOOL service provider adapter can currently implement only Basic authentication.

```
if ((session.GetApplicationSession() = HTTP_SESSION_ESTABLISHED)
        and (not Validated)) then
    authValue : string = request.GetHeader('Authorization');

    if (authValue = nil) then
      response.SetHeader('WWW-Authenticate',
              'Basic realm="ForteFusion"');
      respData = new();
      statusCode = SC_UNAUTHORIZED;
    else

    av : TextData = TextData(authValue);

    if (not av.MoveToString(source = 'Basic', ignoreCase = TRUE))
        then
              -- Error : only Basic authentication is supported.
              msg.ReplaceParameters(
            'ERROR: Order connector supports only BASIC
                authentication. " %1" is an invalid request.',
              av);
            statusCode = SC_BAD_REQUEST;
    else
```

The following section of the code sample extracts and decodes the Base64-encoded characters that provide the user name and password.

NOTE The TOOL adapter example has a simple but unrealistic algorithm that requires the user name and password to be identical. A more sophisticated algorithm is recommended.

```
-- Extract the encoded characters after the word Basic + 1 space.
            av.MoveToString(source = 'Basic',
                GoPast = TRUE,
                ignoreCase = TRUE);
            av.MoveToNotChar(' ');
                -- Move past & delete ...
            av.CutRange(0, av.Offset);

            namePw : TextData = request.DecodeBase64(av);
            -- At this point namePw contains name:password
            namePw.MoveToChar(':');
            name : TextData = namePw.CutRange(0, namePw.Offset);
            namePw.MoveToNotChar(':');
            namePw.CutRange(0, namePw.Offset);
            -- name contains the name & namePw contains the password

            if ((name.ActualSize > 0) and (name.Compare(namePw) = 0))
            then
              -- If the name actually exists (has more than zero
              -- characters in it) and
              -- password matches the name, identity is proven.
              -- Note:  A more sophisticated algorithm is
                  recommended.
            statusCode = SC_OK;
            Validated = TRUE;
            SessionCount = SessionCount + 1;
            else
              msg.ReplaceParameters
              ('Invalid Session attempt for user"%1"', name);

              task.lgr.putline(msg);

              -- Otherwise, the user may be an imposter.
              -- Log the error.

              response.SetHeader('WWW-Authenticate',
              Basic realm="ForteFusion"');
              -- Request authorization again...

              respData = new();
              statusCode = SC_UNAUTHORIZED;
          end if;
      end if;
    end if;
end if;
if (statusCode = SC_OK) then
        -- The server proceeds with the work request...
end if;
response.SetStatus(statusCode = statusCode);
```

Building an iIS ‘C’ Adapter

The iIS ‘C’ Adapter SDK provides source code for building an iIS ‘C’ adapter.

This chapter explains how an iIS ‘C’ adapter functions, how to access information on a standard XML parser, and how to use the iIS ‘C’ API to build a service provider or service requestor adapter.

iIS ‘C’ Adapter Operations

This section describes the operations of an iIS ‘C’ adapter, building on the information about general adapter operations in Chapter 1, “iIS Adapters.”

HTTP Operations

HTTP operations are different for service providers and service requestors. In both cases, messages are sent and received, and the adapter must know how to both create and interpret XML messages. The difference is that a server gets requests and sends (returns) a response, while a client sends a request and gets (is returned) a response.

Service Providers

An adapter that provides services establishes itself as an HTTP server. To do this, it must have a machine (for example, *mymachine.mycompany.com*) and port (for example, *1234*) on which it is available. The application URL property configured for the partner proxy contains this information so that the proxy can locate the adapter. While the proxy needs to know the entire URL, the adapter implicitly knows what machine it is running on, and only needs to determine the port number on which to listen for proxy requests.

Once these parameters are identified, the HTTP server can make itself available. As messages arrive, the adapter's XML parser is called. Each parser invocation returns a response message to the originator of the request.

HTTP requests arrive invoking HTTP requests. The proxy makes only `Post` or `Get` requests, with the message including the action information. Service provider adapters need accept only `Post` or `Get` requests, according to the XSL rules defined for the proxy. The iIS 'C' adapter generates an error message for any other HTTP request.

Once an adapter's work is completed, it exits and the port no longer listens for messages. An error is generated if a requestor sends a message after the service provider adapter has exited.

Service Requestors

The operations of a service requestor adapter are simpler. As an HTTP, it needs to know the destination URL for its message. This URL consists of a machine and an optional port specification. If no port is specified, the default HTTP port 80 is used. Once the destination is known, the client packages XML-encoded requests into an HTTP message and sends the message. The destination server processes the message and returns a response. The client interprets the response, which may include generating errors.

XML Operations

XML messages contain elements. Elements can either contain other elements or text. All data in an XML message is text.

As XML messages arrive, their data is extracted and the associated application operations are performed. XML messages can either be parsed or simply scanned for information. As message size and complexity increases, however, an XML parser is recommended.

When the adapter needs to send an XML message, its message builder must construct the message.

iIS Application Operations

Once it is determined that certain work is to be performed, the application must do that work and respond with status information when it completes the work. For example, if it performs a credit check, the response messages would include information about whether credit is authorized.

It is up to the system integrator to determine how your iIS ‘C’ adapter operates in conjunction with its application and supply this code to the adapter.

The iIS ‘C’ Adapter SDK

The iIS ‘C’ Adapter SDK provides a ‘C’ implementation to build adapters for non-TOOL applications. The ‘C’ SDK supports ANSI ‘C’ and compiles under both Win32 and non-Win32 environments. The iIS ‘C’ implementation is fully independent of iIS and the iPlanet UDS runtime system.

The iIS ‘C’ SDK is distributed as source code that you can modify as needed.

NOTE Be sure to copy modified code to a different directory so that future iIS installations do not overwrite it.

The following functions are provided in the code:

- a simple HTTP client, HTTPSend
- a simple HTTP server, HTTPListen or HTTPListenEx
- supporting functions for creating and manipulating HTTP messages
- extended message generation functions, BuildHTTPRequestEx and BuildHTTPResponseEx, for returning request and response messages
- session authentication, which includes message header creation and Base64 encoding and decoding

HTTPSend is used to implement a service requestor, and HTTPListen to implement a service provider.

The code for the HTTPListen, HTTPListenEx, HTTPSend, and supporting functions is documented in fnconect.h and fnconect.c. The functions are described in fnconect.h and fnconect.htm.

All the source files are located in `FORTE_ROOT/install/fusion/ccon/`. The following table indicates the contents of each file in the directory:

Filename	Contents
fnconect.htm	iIS ‘C’ API reference
fnconect.h	The #defines and function prototypes for fnconect.c
fnconect.c	The implementation of HTTPListen, HTTPSend, and supporting functions
fnsocket.h	The #defines and #includes for support non-Win32 sockets
base64.h	Header file for Base-64 encoder / decoder
base64.c	Implementation of Base-64 encoder / decoder
uuid.h	Header file for Universally Unique Identifier (UUID) generator
uuid.c	Implementation of UUID generator

XML processing The iIS ‘C’ SDK does not directly support XML processing. The iIS ‘C’ adapter example uses James Clark’s Expat XML parser. For the source code and information on the parser, see `http://www.jclark.com/xml/expat.html`.

The iIS ‘C’ Adapter Example

The iIS ‘C’ SDK includes a demo client, `democlnt.h/democlnt.c`, and a demo server, `demosvr.h/demosvr.c`.

The demo client/server is a very simple credit check system. The client submits a request for a credit check. The billee, item count, and order identification are encoded for the credit check. The server accepts requests for credit checks, and responds with the results of the credit check.

The demo client/server uses `xpiface.c/xpiface.h` and the Expat XML Parser Toolkit to process XML. Expat is a low-level XML parser; you need to provide the callback functions used during XML parsing. The demo client/server XML callback functions are contained in `xpiface.h/xpiface.c`.

All example files are located in `FORTE_ROOT/install/examples/fusion/ccon/`. The following table describes the iIS ‘C’ adapter example files:

Filename	Contents
democlnt.h	The #defines and function prototypes for democlnt.c
democlnt.c	The implementation of the demo client
demosvr.h	The #defines and function prototypes for demosvr.c
demosvr.c	The implementation of the demo server
xpiface.h	The #defines, structure declaration, and function prototype for Expat callback functions
xpiface.c	The implementation for the Expat callback functions
creditck.h	Information on the credit check XML tags and error messages

Building the ‘C’ Adapter

This section explains how to build an iIS ‘C’ adapter. Sample code from the iIS ‘C’ adapter example is included to illustrate important details.

The iIS ‘C’ SDK allows you to construct a simple HTTP client or server, as well as construct and manipulate XML messages and headers. The iIS files needed to build a ‘C’ adapter are fnconect.h and fnconect.c. For a non-Win32 environment, you also need fnsocket.h. To implement session authentication you need base64.h and base64.c.

All the files you need to build an iIS ‘C’ adapter are located in `FORTE_ROOT/install/ccon/`. Included in that directory is the file fnconect.htm, which contains the iIS ‘C’ SDK reference.

Building an HTTP Server

An adapter acting as a service provider calls HTTPListenEx to start a simple HTTP server. In the iIS ‘C’ adapter example, the `demosvr.c/demosvr.h` and `creditck.h` files implement the credit check service provider.

As an HTTP server, the adapter uses the HTTPListenEx function to make itself available to process messages. The service provider adapter must also implement a callback function for message processing. This function is passed to HTTPListenEx. For details, see the typedef for FNMsgProcessor in `fnconect.htm`. The adapter parses each message received, and calls `BuildHTTPResponse` to construct the response message. The adapter also supports the user authentication functions Authenticator and FusionXMLAuthenticator.

The following code shows how to call HTTPListenEx. The parameters specify:

- the port number on which the server should listen for requests (nPortNum)
- whether the server should process only one message (bListenOnce=TRUE) or process messages until shutdown (bListenOnce=FALSE)
- the echo level (nEchoLvl)
- a pointer to the processor function (MsgProcessor), and
- user authentication functions

HTTPListenEx

```
HTTPListenEx( nPortNum, bListenOnce, MsgProcessor, Authenticator,
             FusionXmlAuthenticator , "Example Server" , nEchoLvl);
```

The following code sample shows how to use the processor callback function provided by demosvr.c. The code validates that the HTTP message is supported by the adapter. It first verifies that the request is an HTTP Post, then validates that the message is XML. It generates errors if a different message type is received or if the message body cannot be found:

```
/* The following section of code validates the HTTP message and that
   the method and message types are supported by this adapter.
*/

szMethod = GetHTTPMethod( szHTTPMsg );

if ( strcmp( HTTP_METHOD_POST, szMethod ) != 0 ) {
    /* Method not supported; create error response */
    bStillSuccessful = FALSE;
    szErrorMsg       = NO_METHOD_SUPPORT_ERROR;
}

free( szMethod );
pHeaders = CreateHTTPHeaders();

/* Get Message Type and validate that it is XML */
if ( bStillSuccessful ) {
  /* Get Message Type */
  char * szMsgType = GetHTTPFieldValue( szHTTPMsg,
                                        HTTP_FN_CONTENT_TYPE );

  if ( szMsgType == NULL ) {
    /* Message type is NULL: create error response */
    bStillSuccessful = FALSE;
    szErrorMsg       = NO_MSG_TYPE_ERROR;
  } else {
    if ( strcmp( szMsgType, HTTP_CT_TEXT_XML ) != 0 ) {
      /* Message type is wrong type: create error response */
      bStillSuccessful = FALSE;
      szErrorMsg       = WRONG_MSG_TYPE_ERROR;
    }
    free( szMsgType );
  }
}

if ( bStillSuccessful ) {
  szMsgBody = GetHTTPMsgBody( szHTTPMsg );

  if ( NULL == szMsgBody ) {
    /* Message body can't be found: create error response */
    bStillSuccessful = FALSE;
    szErrorMsg       = NO_MSG_BODY_ERROR;
  }
}

/* *** End of HTTP Message Validation *** */
```

The following code shows how the request message body is to be parsed. The XML parser is invoked to create the XML tree and get the XML elements and values needed to process the request. XML error messages are generated if any errors occur during parsing. Finally, the response message is returned.

```
if ( bStillSuccessful ) {
  if ( IsShutDownMsg( szMsgBody ) ) {
    /* It is a shutdown message: create shutdown response */
    szResponseMsg = malloc( strlen
                            ( SVR_SHUTDOWN_RESPONSE_MSG ) + 1 );
    if ( szResponseMsg == NULL ) {
      free( szMsgBody );

      ECHO_ALLOC_ERROR( "malloc()", "szResponseMsg", __LINE__,
             __FILE__ );

      return NULL;
    }

    strcpy( szResponseMsg, SVR_SHUTDOWN_RESPONSE_MSG );
    free( szMsgBody );
  } else {
    szXmlErrorMsg[ 0 ] = '\0';

    pRoot = ParseXML( szMsgBody, szXmlErrorMsg );
    free( szMsgBody );
    if ( pRoot == NULL ) {
      bStillSuccessful = FALSE; /* bad XML */
    }
    if ( bStillSuccessful ) {
      /* GetElementPtrXtn() will return element EN_NEW_WORK in the
        XML tree, any errors will be returned in szXmlErrorMsg
      */
      pNewWorkElem = GetElementPtrXtn( pRoot, EN_NEW_WORK,
                                       szXmlErrorMsg );
      bStillSuccessful = pNewWorkElem != NULL;

    }

    if ( bStillSuccessful ) {
      /* Find WorkType and validate that it is CreditCheck */
      szWorkType = GetElementValueXtn( pNewWorkElem, EN_WORK_TYPE,
                                       szXmlErrorMsg );
      bStillSuccessful= szWorkType != NULL;
      }
      if ( bStillSuccessful ) {
        /* Check to see that WorkType is a credit check */
        if ( strcmp( szWorkType, WT_CREDIT_CHECK ) != 0 ) {
          bStillSuccessful = FALSE;
          szErrorMsg = WRONG_WORKTYPE_ERROR;
        }
      }
    }
    if ( bStillSuccessful ) {
```

```
      /* Get Process ID */
      szProcessID = GetElementValueXtn( pNewWorkElem, EN_PROCESS_ID,
                                        szXmlErrorMsg);
      bStillSuccessful = szProcessID != NULL;
    }
    if ( bStillSuccessful ) {
    /* Get Activity ID */
    szActivityID = GetElementValueXtn( pNewWorkElem,
                                    EN_ACTIVITY_ID, szXmlErrorMsg );

    bStillSuccessful = szActivityID != NULL;
  }
    if ( bStillSuccessful ) {
      /* Get AttValue for AttName = "Billee" */
      szBillee = GetAttValueForAttName( pNewWorkElem, IB_AT_BILLEE,
                                        szXmlErrorMsg );
      bStillSuccessful = szBillee != NULL;
    }

    if ( bStillSuccessful ) {
      /* Get AttValue for AttName = "ItemCount" */
      szItemCnt = GetAttValueForAttName( pNewWorkElem,
                                      IB_AT_ITEM_CNT, szXmlErrorMsg );
      bStillSuccessful = szItemCnt != NULL;
    }

    if ( szXmlErrorMsg[ 0 ] != '\0')
      szErrorMsg = szXmlErrorMsg;

    if ( bStillSuccessful ) {
      /* Perform Credit Check */
      if ( CreditCheck( szBillee, szItemCnt ) ) {
        szAprv = CC_APRV_YES;
      }
      else {
        szAprv = CC_APRV_DEADBEAT;
      }
    }
    if ( bStillSuccessful ) {
      /* Create Response XML with the results of the credit check.
      CC_XML_RESPONSE_MSG is a an XML block that is defined in
      creditck.h
      */
    szResponseMsg = malloc( strlen( CC_XML_RESPONSE_MSG ) +
                            strlen( szProcessID         ) +
                            strlen( szActivityID        ) +
                            strlen( szBillee            ) +
                            strlen( szItemCnt           ) +
                            strlen( szAprv              ) + 1 );

    if ( szResponseMsg == NULL ) {
      ECHO_ALLOC_ERROR( "malloc()", "szResponseMsg", __LINE__,
          _FILE__ );
      return NULL;
    }
```

```
    sprintf(
      szResponseMsg,
      CC_XML_RESPONSE_MSG,
      szProcessID,
      szActivityID,
      szBillee,
      szItemCnt,
      szAprv );
    }
  }
}

if ( !bStillSuccessful ) {
  /* Somewhere along the way an error has occurred: create XML error
     response.
  */
    if ( szErrorMsg == NULL )
      szErrorMsg = SOMETHING_FAILED_ERROR;

    szResponseMsg = malloc( strlen( PROXY_ERROR_TEMPLATE ) +
      strlen( szErrorMsg ) + 1 );
    if ( NULL == szResponseMsg ) {
      ECHO_ALLOC_ERROR( "malloc()", "szResponseMsg", __LINE__,
                                                     __FILE__ );
      return NULL;
    }

    sprintf( szResponseMsg, PROXY_ERROR_TEMPLATE, szErrorMsg );
  }
  pSession = GetHTTPSession( szHTTPMsg );

  /* Create HTTP Response */
  szRetHTTPMsg = BuildHTTPResponseEx(
    szStatusCode,
    "Forte Fusion Example Server",
    HTTP_CT_TEXT_XML,
    szResponseMsg,
    pSession,
    pHeaders );

free( szResponseMsg );
FreeHTTPSession( pSession );
FreeHTTPHeaders( pHeaders );

/* Return results to HTTPListenEx() */
return szRetHTTPMsg;
```

Building an HTTP Client

A service requestor adapter uses BuildHTTPRequestEx to construct an HTTP request message. The message, along with the host name and port number of the proxy, is passed to the HTTPSendEx function. For additional parameters, see the iIS ‘C’ SDK reference (fnconect.htm).

The HTTP response message contains any results returned by the proxy, including errors and authentication challenges.

The `democlnt.h/democlnt.c` and `creditck.h` files implement the credit check service requestor.

The following code shows a simple service request using HTTPSend. First an XML block is created for the request. The request message is then built and sent. When the host receives the message, it calls the XML parser to process it. Errors are generated as appropriate.

```
HTTPSession * pSession = CreateHTTPSession();

SetHTTPSessionApplicationSession( pSession, HTTP_SESSION_REQUIRED );

/* N Orders using one network and application session */
for ( idx = 1; idx <= N_ORDERS; idx++ ) {

  /* Create XML Block for Request */
  sprintf( szBillee, "B_%d", idx );
  sprintf( szOrderNum, "%d", idx );

  szXML = GetOrderXML( szOrderNum, szBillee, "1000" );

  if ( NULL == szXML ) {
    printf( "Failed to create XML Order block!" );
    exit ( 1 );
  }
  if ( idx == N_ORDERS ) {
    /* last order; terminate session & network connections */

  SetHTTPSessionApplicationSession( pSession, HTTP_SESSION_CLOSE );
  SetHTTPSessionNetworkSession( pSession, HTTP_TCPSESSION_MESSAGE );
}

szHTTPReq = BuildHTTPRequestEx(
  HTTP_METHOD_POST,
  "democlnt",
  HTTP_CT_TEXT_XML,
  szXML,
  pSession,
  NULL );

FREE( szXML );
```

```
/* Send HTTP Request to host */
szHTTPRep = HTTPSend( szHost, nPortNum, szHTTPReq, nEchoLvl );

/* Our server will establish a session with unauthenticated clients,
  so we'll update out session information after getting the first
  response.
*/
if ( idx == 1 ) {
  FreeHTTPSession( pSession );
  pSession = GetHTTPSession( szHTTPRep );

  if ( pSession == NULL ) {
    printf( "Out of memory updating session data\n" );
    exit( 1 );
  }
}
/* Check for authentication challenge from host */
szHTTPRep = CheckAuthentication( szHTTPReq, szHTTPRep, szHost,
        nPortNum, nEchoLvl );

FREE( szHTTPReq );

/* Process Response */
if ( NULL ==  szHTTPRep ) {
  printf( "HTTPSend() Failed to return response!" );
  exit ( 1 );
}
/* Get Message */
szXML = GetHTTPMsgBody( szHTTPRep );
FREE( szHTTPRep );

if ( NULL == szXML ) {
  printf( "Failed to find XML Message!" );

  exit ( 1 );
}

/* ... process XML response from proxy */

FREE( szXML );
}
 FreeHTTPSession( pSession );
```

XML Processing

XML processing can take any number of forms; iIS imposes no restrictions. In the iIS ‘C’ example, `xpiface.h/xpiface.x` provide the functionality to parse an XML text into a tree of elements. As shown in the preceding code samples, you can use the ParseXML function to build the tree, and GetElementPtr to get a pointer to an element in the tree. GetElementValue gets the value of an element in the tree.

To build an outgoing message, you can create an XML string/buffer using 'C' string and file handling functions.

Using CDATA Sections

XML messages are simply collections of text in which the text is formatted into XML elements using specific punctuation (for example, `<tagName>`*some value*`</tagName>`). If messages contain greater than, less than, and ampersand signs ("`>`", "`<`", and "`&`"), proxies and adapters might construct illegal XML responses. To avoid this type of error, you should normally escape any text value (for example, a process attribute) in a `CDATA` section. However, if the value contains the end of CDATA marker "`]]>`", you must do textual replacement on the values within the message with XML internal entities (for example, you would replace "`<`" with "`&lt;`" rather than use a `CDATA` section).

Session Authentication

As explained in Chapter 1, "iIS Adapters," when an HTTP client makes a request to an HTTP server, it is up to the server to determine whether session authentication is required and if so, which type of authentication (Basic or FusionXML). If session authentication is required, and a session does not exist, the following steps occur:

1. The server returns the request with a status code of Unauthorized (401) along with a specification of the required authentication scheme.
2. When the client receives this information, it resends the original request, along with a completed authorization header that uses the required authentication scheme.
3. When the server receives the required information, it validates the session request and takes appropriate action, typically fulfilling the original request. If the information it receives is incorrect, it rejects the attempt by sending another Unauthorized response.

NOTE The 'C' adapter supports both Basic and FusionXML authentication, regardless of whether it is a service provider or a service requestor. However, if the adapter is a service requestor and its partner proxy is independent (it operates without the iIS process engine), only Basic authentication is possible. For details on configuring the adapter and proxy appropriately, see the *iIS Backbone System Guide*.

Service Requestors

The following code samples show how session authentication can be implemented when a ‘C’ service requestor adapter, acting as an HTTP client, initiates a work request. If session authentication is not required, the server proxy proceeds with the request; if it is required, the adapter must provide the required information.

In the following code sample, the CheckAuthentication function is called to check for authentication challenges.

```
/*****************************************************************
    CheckAuthentication
    Check for authentication challenges. If server asks for
    authorization, an appropriate header line is returned. Insert it
    into the request, and send it to the server again. The caller
    must free() the returned host response.
*/
static char *
CheckAuthentication(
  char      * szHTTPRequest,   /* [IN] request sent to host        */
  char      * szHTTPResponse,  /* [IN] response received from host */
  CONST char   * szHost,       /* [IN] host name                   */
  int       nPortNum,          /* [IN] host port number            */
  int       nEchoLvl )         /* [IN] HTTP echo level             */
{
  char      * szResult = NULL;

  if ( szHTTPResponse != NULL ) {
    char   * szStatusCode = GetHTTPStatusCode( szHTTPResponse );

    if ( strcmp( szStatusCode, HTTP_SC_UNAUTHORIZED ) == 0 ) {
      /* The host is challenging our authorization */
      char * szChallenge = GetHTTPFieldValue( szHTTPResponse,
                                              "WWW-Authenticate" );

      if ( szChallenge != NULL ) {
          char    * szAuthorization;

GetUserAuthorization creates the response to the challenge
          szAuthorization = GetUserAuthorization( szChallenge );

          if ( szAuthorization != NULL ) {
            /* Insert authorization into request */
            char        * szNewReq;
            int           ncNewReq = strlen( szHTTPRequest ) +
                                     strlen( szAuthorization ) + 1;

            szNewReq = (char *) malloc( ncNewReq );

            if ( szNewReq != NULL ) {
              char * pcRover  = strstr( szHTTPRequest, "\r\n" ) + 2;
              char * pcRover2;
```

```
            strcpy( szNewReq, szHTTPRequest );

            pcRover2 = strstr( szNewReq , "\r\n" ) + 2;
            *pcRover2 = '\0';

            strcat( szNewReq, szAuthorization );
            strcat( szNewReq, pcRover );

            free( szHTTPResponse );

            szResult = HTTPSend( szHost, nPortNum, szNewReq,
                                 nEchoLvl );

            free( szNewReq );
          } else {
              ECHO_ALLOC_ERROR( "malloc", "szNewReq", __LINE__,
                                                      __FILE__ );
          }
        }
      }
      free( szChallenge );
    } else {
           /* No challenge -- pass the result along unchanged */
           szResult = (char *) szHTTPResponse;
    }
    free( szStatusCode );
    }
    return szResult;
  }
```

The GetUserAuthorization function, in addition to composing the response message to an authentication challenge, also encodes the user information in Base64:

```
  char * font = "Authorization: Basic %s\r\n";
  ncResult = Base64Encode( szEncoded,(CONST unsigned char *)
        szBuffer, sprintf( szBuffer, "%s:%s", szName, szPassword ) );

  ncResult += strlen( fmt );
  szResult  = malloc( ncResult );

  if ( szResult != NULL ) {
    sprintf( szResult, fmt, szEncoded );

  } else {
      ECHO_ALLOC_ERROR( "malloc", "szResult", __LINE__, __FILE__ );
  }

  return szResult;
}
```

Service Providers

The service provider adapter listens for requests, using HTTPListenEx to set up the listener loop. Once the session is authenticated, the application session does not have to be authorized again. The service provider adapter supports both Basic and FusionXML authentication. The following code samples illustrate both schemes.

Basic Authentication

Basic authentication is implemented by the function type FNAuthenticateUser. The following code sample shows how Basic authentication can be implemented when a ‘C’ service provider adapter, acting as an HTTP server, receives a work request from a client proxy.

```
/*****************************************************************
A simplistic example of an authentication function.  In this case,
everyone except the user "EvilDoer" may have access.

This is Basic authentication, an implementation of
FNAuthenticateUser().
*/
BOOL
Authenticator(   /* returns TRUE if the user is authenticated */

    CONST char * szUserId,     /* [IN] User ID */

    CONST char * szPassword )  /* [IN] Password */
{
    return strcmp( szUserId, "EvilDoer" ) != 0;
}
```

FusionXML Authentication

FusionXML authentication is implemented by the function type FNAuthenticateUser2. The parameter sxXml provides the authentication document. The following code sample shows how FusionXML session authentication can be implemented when a ‘C’ service provider adapter, acting as an HTTP server, receives a work request from a client proxy:

```
/*************************************************************
Authenticate user with the FusionXml scheme.
*/
static BOOL
FusionXMLAuthenticateUser(
    CONST char * szXml ) /* [IN] Fusion authentication document
*/
{
    BOOL             bResult = FALSE;
    struct Element * pRoot;
    char             szErrorMsg[ 1024 ];
    pRoot = ParseXML( szXml, szErrorMsg );

    if ( pRoot != NULL ) {
        char    * szName;
        char    * szPassword;
        char    * szRole;

      /* Extract authentication information of interest from DOM
         tree, then authenticate the identified entity.
         ...
      */
      bResult = UserIsRighteous( szName, szPassword, szRole );
    }
    return bResult;
}
```

Index

S

X