Progress®



Develop an HTTP client application in ABL

OpenEdge®

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Develop an HTTP client application in ABL....9

Build an HTTP request....11

Add query and path parameters to the URI....13

Add HTTP request headers....15

Add an HTTP request body....15

Use HTTP authentication....21

Execute an HTTP request....23

Implement a stateful client using cookies....24

Process an HTTP response....26

String response....28

JSON response....28

XML response....29

Binary response....30

Multipart response....30

Configure TLS security settings....31

Preface

Purpose

This document provides an overview of the tasks that are required to develop an HTTP client in ABL. It describes how to use some of the key classes and interfaces in the `OpenEdge.Net` procedure library to make REST calls over HTTP.

Audience

This document is intended for ABL developers who want to develop an HTTP client application in ABL. To be able to benefit from this document, you need to be familiar with object oriented programming in ABL.

Organization

This document comprises three major parts:

- Build an HTTP request on page 11, which describes how to build an HTTP request object, with a request body, headers, and other properties such as the HTTP method and Authentication.
- Execute an HTTP request on page 23, which describes how to execute the HTTP request and configure TLS settings.
- Process an HTTP response on page 26, which describes how to handle response messages returned by the REST API or server.

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

1

Develop an HTTP client application in ABL

HTTP and HTTP(S)

Hypertext Transfer Protocol (HTTP) enables client applications to communicate with server applications over the Internet in a standard way. A client application initiates the communication by sending a request message to a Uniform Resource Identifier (URI). The server application processes the request and returns a response message.

Web browsers use HTTP to access websites and end-user web applications. HTTP is also used in programmatic communications, such as REST and SOAP, where client applications send request messages to invoke methods or functions on the server application over the Internet.

HTTP(S) extends HTTP by adding a layer of security (the 'S' in HTTPS stands for Secure). Communication over HTTP(S) happens in a way that is very similar to HTTP; however request and response messages are encrypted before being sent.

To learn more about HTTP and HTTPS, see the following resources:

- https://www.w3schools.com/whatis/whatis_http.asp
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP
- https://tools.ietf.org/html/rfc2616
- https://en.wikipedia.org/wiki/HTTPS

The OpenEdge.Net procedure library

The `OpenEdge.Net` procedure library provides classes and interfaces that enable you to make HTTP requests and process HTTP responses from within ABL code. It is included in the **PROPATH** by default when you create a new OpenEdge project in Progress Developer Studio for OpenEdge. You can also find it in `$DLC/tty/netlib` (Unix and Windows) and `$DLC/gui/netlib` (Windows only).

The OpenEdge.Net library supports all HTTP methods and all content types. It also supports HTTP authentication, Transport Layer Security (TLS) communication, the use of HTTP proxies, and the development of stateful HTTP clients using cookies.

Note: The following topics cover the key features of the OpenEdge.Net procedure library. See the OpenEdge API reference documentation to learn about all the available classes, methods, and interfaces. Additional code samples can be found on GitHub.

Main steps

Perform the following steps to develop an ABL HTTP client:

1. **Build an HTTP request**

   Use the OpenEdge.Net.HTTP.RequestBuilder class to create a request object that implements the OpenEdge.Net.HTTP.IHttpRequest interface. You must specify an HTTP method (GET, PUT, POST, etc) and a URI. For example:

```
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.

oURI =
URI:Parse("http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer?filter=CustNum=1").
oRequest = RequestBuilder:GET(oURI):Request.
```

   You can optionally add HTTP headers and a request body. To learn more about these and other options, see Build an HTTP request on page 11.

2. **Execute the HTTP request**

   To execute the request, you need a client object and a request object. Use the OpenEdge.Net.HTTP.ClientBuilder class to create a client object that implements the OpenEdge.Net.HTTP.IHttpClient interface. Then, call the Execute() method on the client object. For example:

```
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE OResponse AS IHttpResponse NO-UNDO.

oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).
```

   You can also use the client object to perform other tasks such as setting cookies or specifying TLS properties. To learn more about these tasks, see Execute an HTTP request on page 23.

3. **Process the HTTP response**

   Depending on your requirements, you may want to pass the HTTP response to another part of your ABL program, write it to a log file, or use it to query a database. The OpenEdge.Net library supports a wide variety of content types in an HTTP response. To learn more, see Process an HTTP response on page 26. In this example, the HTTP response is displayed through the MESSAGE statement.

```
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

Example

This example sends a GET request and processes a JSON response:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.


DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE OResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.

//Build a request
oURI =
URI:Parse("http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer?filter=CustNum=1").
oRequest = RequestBuilder:GET(oURI):Request.

//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

Additional resources

- Samples on GitHub
- OpenEdge.Net API Reference

For details, see the following topics:

- Build an HTTP request
- Execute an HTTP request
- Process an HTTP response
- Configure TLS security settings

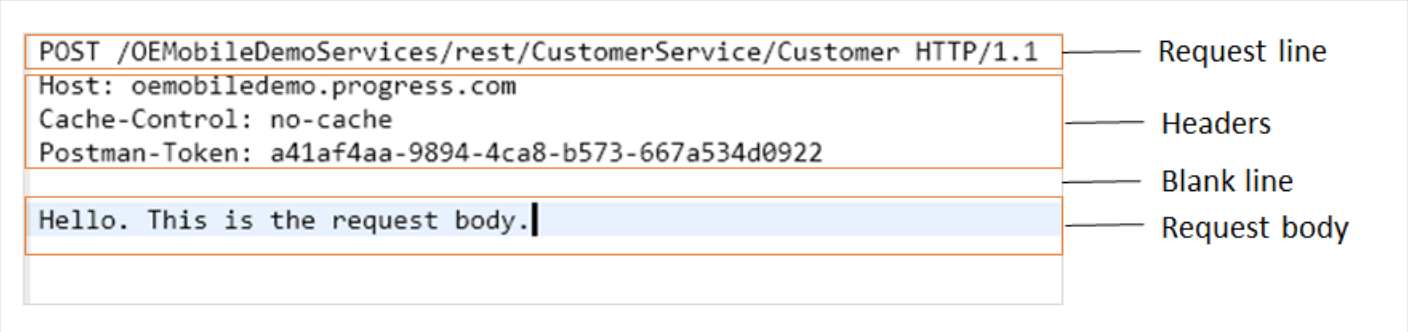
Build an HTTP request

Parts of an HTTP request message

An HTTP request message consists of the following parts:

- A request line, which specifies an HTTP method (GET, PUT, POST, DELETE, etc), a URI to a resource on the server, and the HTTP protocol version.
- A set of headers that provide various kinds of information to the server. Each header is a name-value pair. For example, `Content-Type: text/plain`.

- A blank line that separates the header section from the request body.
- A request body that consists of the payload or data that is to be sent to the server.



Create an HTTP request object

To construct an HTTP request message, create a request object in your ABL program using the `OpenEdge.Net.HTTP.RequestBuilder` class, which implements the `IHttpRequest` interface. At minimum, you need a **URI** and an **HTTP verb**. You can optionally add a request body and headers.

The following example builds a request with the HTTP `GET` verb:

```
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.

oURI =
URI:Parse("http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer?filter=CustNum=1").
oRequest = RequestBuilder:GET(oURI):Request.
```

To build requests that use other HTTP verbs, replace GET with the appropriate verb name, as shown in the following example.

```
oRequest = RequestBuilder:POST(oURI, oRequestBody):Request.
oRequest = RequestBuilder:PUT(oURI, oRequestBody):Request.
oRequest = RequestBuilder:DELETE(oURI):Request.
oRequest = RequestBuilder:HEAD(oURI):Request.
oRequest = RequestBuilder:OPTIONS(oURI):Request.
oRequest = RequestBuilder:TRACE(oURI):Request.
```

Another way to build an HTTP request is to use the `Build()` method, where you pass the HTTP verb as a parameter.

```
oRequest = RequestBuilder:Build('GET', oURI):Request.
oRequest = RequestBuilder:Build('POST', oURI):WithData(oRequestBody):Request.
```

Based on your requirements, you may also need to perform the following tasks:

- Add query and path parameters to the URI on page 13
- Add HTTP request headers on page 15
- Add an HTTP request body on page 15
- Use HTTP authentication on page 21

Other HTTP request methods

`RequestBuilder` is a fluent interface that enables you to invoke multiple methods in a method chain. The `RequestBuilder:GET | PUT | POST | ..():Request` is the minimum required to build an HTTP request. Here are a few other methods that you can add in the method chain.

```
oRequest = RequestBuilder
   :GET(<URI>)
    /* Other options include PUT | POST | DELETE | HEAD | PATCH | OPTIONS | TRACE */
   :AddHeader(<HttpHeader>)
            :HttpVersion("HTTP/1.1")
   :WithData(Progress.Lang.Object)
   :ContentType(<char>)
   :AuthCallback(<callback object> | <callback procedure>)
   :UsingBasicAuthentication(<Credentials>)
    /* Other options include UsingDigestAuthentication(<Credentials>) |
UsingCredentials(<Credentials>) */
   :ViaProxy(<url>)
   :WithTransferEncoding(<encoding>)
   :Request.
```

Note: Currently, the `OpenEdge.Net` library supports only HTTP 1.1, so if you need to use the `HttpVersion()` method, you should pass the string `"HTTP/1.1"`—`HttpVersion("HTTP/1.1")`.

API reference documentation

To view the complete list of methods that you can use on an HTTP request object, refer to the RequestBuilder documentation or see the OpenEdge.Net API Reference documentation.

Add query and path parameters to the URI

The `OpenEdge.Net.URI` class enables you to create HTTP URIs. The `URI` class constructor is overloaded, so there are a number of different parameters that you can pass. Typically though, you want to pass the following parameters:

- The **scheme** (http or https)
- The **hostname** or **sitename**
- Optionally, a **port** number

The following example defines a base URI:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.

DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.

oClient = ClientBuilder:Build():Client.
oURI = new URI('http', 'oemobiledemo.progress.com').
```

To append additional path segments to the base URI (which you need to do for most REST API endpoints), use the `Path` property:

```
oURI = new URI('http', 'oemobiledemo.progress.com').
oURI:Path = "/OEMobileDemoServices/rest/CustomerService/Customer".
```

Alternatively, you can use the `Parse()` method to turn a string into a URI object:

```
oURI =
URI:Parse('http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer').
```

Add query strings

A query string is a set of one or more parameters that, when passed to a REST API, filters the dataset returned by the API.

For example, to get records for customers in Finland, your URI path must include the parameter `country=Finland` in a query string:

```
http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer?filter=Country='Finland'
```

The ? in the URI path indicates that a query string follows. Multiple queries can be passed using the & separator. For example, to select customers from Finland, who have 'DKP' as their sales representative, you would need a URI like this:

```
http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer?filter=Country='Finland'&SalesRep='DKP'
```

To add a query to your URI object, choose from one of the following options:

- Add a query string (using the `AddQueryString()` method). Use this option if you have a string comprising all the query parameters.

  ```
  oURI:AddQueryString("filter=Country='Finland'", false).
  ```

  The `AddQueryString()` method takes two parameters—the query string, which is a character string, and a logical (true or false) value that indicates whether the query string should be URL decoded[1].

- Add one or more queries by calling `AddQuery()` for each parameter. Use this option if you have many queries, some of which are populated by variable values.

  ```
  oURI:AddQuery("filter", "'Country=Finland'").
  oURI:AddQuery("SalesRep", "DKP").
  ```

Add path parameters

A path parameter is separated by slashes (`/`) in the URI. Unlike a query, which requires a key-value pair (`Country=Finland`), a path parameter requires only a value. The following is an example of a customer ID (`1`) being passed as a path parameter in the URI:

```
http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer/1
```

To add a path parameter to a URI object, use `AddPathSegment()`:

```
oURI = new URI('http', 'oemobiledemo.progress.com').
oURI:Path = "/OEMobileDemoServices/rest/CustomerService/Customer".
oURI:AddPathSegment("1").
```

[1] URLs are encoded as per the IETF specification. URL decoding replaces '%' escape characters in the URL, if any exist.

API reference documentation

To view the complete list of methods that you can use on a URI object, refer to the URI documentation or see the OpenEdge.Net API Reference documentation.

Add HTTP request headers

To add custom headers to an HTTP request object, use the `AddHeader()` method. You can use this method multiple times to add multiple headers. For example:

```
oRequest = RequestBuilder:Build('GET', oURI)
           :AddHeader('MyCustomHeaderName','MyCustomHeaderValue')
           :AddHeader('MySecondHeader','MySecondHeaderValue')
           :Request.
```

Content type headers

While the `RequestBuilder` attempts to infer the content type of the request from the request body, you can set the Content Type header, as follows:

```
oRequest = RequestBuilder:Build('GET', oURI):ContentType('application/json').
```

To set an `Accept` header that specifies the content type to be returned by the server, select from the following methods:

```
oRequest = RequestBuilder:Build('GET', oURI):AcceptAll(). //Accepts all content types
oRequest = RequestBuilder:Build('GET', oURI):AcceptContentType(CHARACTER). //Specify a
 content type
oRequest = RequestBuilder:Build('GET', oURI):AcceptHtml(). //Accepts HTML content
oRequest = RequestBuilder:Build('GET', oURI):AcceptJson(). //Accepts Json content
oRequest = RequestBuilder:Build('GET', oURI):AcceptXml(). //Accepts XML content
```

API reference documentation

To view the complete list of methods that you can use on an HTTP request object, refer to the RequestBuilder documentation or see the OpenEdge.Net API Reference documentation.

Add an HTTP request body

The `OpenEdge.Net` library supports a wide variety of content types in the request body, including the following:

- String requests on page 16
- JSON requests on page 16
- XML requests on page 17
- Binary (for images and files)
- Form data in requests on page 19
- Multipart request messages on page 19

API reference documentation

To view the complete list of methods that you can use on an HTTP request object, refer to the RequestBuilder documentation or see the OpenEdge.Net API Reference documentation.

String requests

To add a plain string message, create a string and pass the string as the request body.

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.

DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.
DEFINE VARIABLE oRequestBody AS OpenEdge.Core.String NO-UNDO.
DEFINE VARIABLE oResponseBody AS OpenEdge.Core.String NO-UNDO.

//Build a request
oURI = URI:Parse("http://httpbin.org/post").
oRequestBody = new OpenEdge.Core.String("This is a test string").
oRequest = RequestBuilder:POST(oURI, oRequestBody):Request.


//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

JSON requests

To add a JSON message, create a JSON object and pass the object as the request body.

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.

DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE oRequestBody AS JsonObject NO-UNDO.
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oResponseBody AS OpenEdge.Core.String NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.

//Build a request
oURI = URI:Parse("http://httpbin.org/post").
oRequestBody = new JsonObject().
oRequestBody:Add("CustNum", "200").
oRequestBody:Add("Name", "Excellent Sports Apparel").
oRequest = RequestBuilder:POST(oURI, oRequestBody):Request.
```

```
//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

Alternatively, you can pass the JSON object using the `AddJsonData()` method. For example:

```
oRequest = RequestBuilder:POST(oURI):AddJsonData(oRequestBody):Request.
```

XML requests

To add an XML message, create an XML document, save it to a `MEMPTR` variable, and then pass it as an `OpenEdge.Core.Memptr` object. The following example loads XML content from a file:

```
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE hXmlDoc AS HANDLE NO-UNDO.
DEFINE VARIABLE mData AS MEMPTR NO-UNDO.
DEFINE VARIABLE oDocument AS CLASS Memptr NO-UNDO.

//Build a request
oURI = URI:Parse("http://httpbin.org/post").
CREATE X-DOCUMENT hXmlDoc.
hXmlDoc:LOAD("FILE", "C:/Samples/sample_xml_doc.xml", FALSE).
hXmlDoc:SAVE('MEMPTR', mData).
oDocument = new Memptr(mData).
oRequest = RequestBuilder:POST(oURI, oDocument):Request.
```

You can also create the XML document in your ABL program. For example:

```
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE hXmlDoc AS HANDLE NO-UNDO.
DEFINE VARIABLE hRootNode AS HANDLE NO-UNDO.
DEFINE VARIABLE hCustomerName AS HANDLE NO-UNDO.
DEFINE VARIABLE hCustomerNameValue AS HANDLE NO-UNDO.
DEFINE VARIABLE hCustomerID AS HANDLE NO-UNDO.
DEFINE VARIABLE hCustomerIDValue AS HANDLE NO-UNDO.
DEFINE VARIABLE mData AS MEMPTR NO-UNDO.
DEFINE VARIABLE oDocument AS CLASS Memptr NO-UNDO.

//Create the XML document
CREATE X-DOCUMENT hXmlDoc.
CREATE X-NODEREF hRootNode.
CREATE X-NODEREF hCustomerName.
CREATE X-NODEREF hCustomerNameValue.
CREATE X-NODEREF hCustomerID.
CREATE X-NODEREF hCustomerIDValue.
hXmlDoc:CREATE-NODE(hRootNode, "Customer", "ELEMENT").
hXmlDoc:CREATE-NODE(hCustomerName, "Name", "ELEMENT").
```

```
hXmlDoc:CREATE-NODE(hCustomerNameValue, "", "TEXT").
hXmlDoc:CREATE-NODE(hCustomerID, "ID", "ELEMENT").
hXmlDoc:CREATE-NODE(hCustomerIDValue, "", "TEXT").
hCustomerNameValue:NODE-VALUE = "Excellent Sports Apparel".
hCustomerIDValue:NODE-VALUE = "1".
hCustomerName:APPEND-CHILD(hCustomerNameValue).
hCustomerID:APPEND-CHILD(hCustomerIDValue).
hRootNode:APPEND-CHILD(hCustomerName).
hRootNode:APPEND-CHILD(hCustomerID).
hXmlDoc:APPEND-CHILD(hRootNode).

//Save the XML document to a Memptr
hXmlDoc:SAVE('MEMPTR', mData).
oDocument = new Memptr(mData).

//Build a request
oURI = URI:Parse("http://httpbin.org/post").
oRequest = RequestBuilder:POST(oURI, oDocument):Request.
```

Binary content in requests

To send a file, such as an image or a PDF file, load the file into memory using a `MEMPTR` variable, pass it to an `OpenEdge.Core.Memptr` object, and then attach the `Memptr` object to the request using the `WithData()` method.

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.
USING OpenEdge.Core.Memptr FROM PROPATH.

DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE mData AS MEMPTR NO-UNDO.
DEFINE VARIABLE oPic AS CLASS Memptr NO-UNDO.
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oResponseBody AS OpenEdge.Core.String NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.

//Build a request
oURI = URI:Parse("http://httpbin.org/post").
COPY-LOB FROM FILE "C:/Samples/smile.png" TO mData.
oPic = new Memptr(mData).
oRequest = RequestBuilder:POST(oURI):WithData(oPic):Request.

//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

Form data in requests

HTML forms are a popular way to capture and transmit sets of data. Take, for example, a registration page that includes fields such as name, email, and contact number. Each field has a name and a value. When a user clicks Submit on the page, data from the page is sent to the backend service as form data.

Form data is a set of key-value pairs. The value of a key can be a string, a JSON object, or an image. To add form data to an HTTP request, use the `AddFormData()` method. For example:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.

DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.
DEFINE VARIABLE oResponseBody AS OpenEdge.Core.String NO-UNDO.

//Build a request
oURI = URI:Parse("http://httpbin.org/post").
oRequest = RequestBuilder:POST(oURI):AddFormData('First Name', 'John'):AddFormData('Last
 Name', 'Doe'):Request.


//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

To send a JSON object in the form data, convert it to string. For example:

```
USING Progress.Json.ObjectModel.*.

DEFINE VARIABLE CustomerJsonObject AS JsonObject.

CustomerJsonObject = new JsonObject().
CustomerJsonObject:Add("CustNum", "200").
CustomerJsonObject:Add("Name", "Excellent Sports Apparel").

oRequest = RequestBuilder:POST(oURI):AddFormData('Customer Record',
string(CustomerJsonObject)):Request.
```

Multipart request messages

A multipart message contains a body with many component body parts. Each body part has a content type, a body, and headers (which are optional). Multipart messages are typically used in order to avoid multiple request/responses, or to provide different representations of the same data.

The `OpenEdge.Net` library supports the transmission of multipart messages through the `OpenEdge.Net.MultipartEntity` and the `OpenEdge.Net.MessagePart` classes.

To create a request body consisting of multiple parts, create multiple `MessagePart` objects, add them to a `MultipartEntity` object, and pass the `MultipartEntity` as the request body.

Each `MessagePart` can have two parameters—the content type (string) and an object. The object can be string, a JSON object, or a file (a binary).

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.
USING OpenEdge.Net.MessagePart.
USING OpenEdge.Net.MultipartEntity.

DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.
DEFINE VARIABLE oRequestBody AS OpenEdge.Core.String NO-UNDO.
DEFINE VARIABLE oResponseBody AS OpenEdge.Core.String NO-UNDO.
DEFINE VARIABLE oPart1 AS MessagePart NO-UNDO.
DEFINE VARIABLE oPart2 AS MessagePart NO-UNDO.
DEFINE VARIABLE oEntity AS MultipartEntity NO-UNDO.


//Create first message part - a string
DEFINE VARIABLE oText AS OpenEdge.Core.String NO-UNDO.
oText = new OpenEdge.Core.String("Some sample text").
oPart1 = new MessagePart('text/plain', oText).

//Create second message part - a JSON object
DEFINE VARIABLE oJsonPart AS JsonObject NO-UNDO.
oJsonPart = new JsonObject().
oJsonPart:Add("CustNum", "200").
oJsonPart:Add("Name", "Excellent Sports Apparel").
oPart2 = new MessagePart('application/json', oJsonPart).

//Create the multipart entity
oEntity = new MultipartEntity().
oEntity:AddPart(oPart1).
oEntity:AddPart(oPart2).

//Build a request
oURI = URI:Parse("http://httpbin.org/post").
oRequest = RequestBuilder:POST(oURI, oEntity):Request.

//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

Use HTTP authentication

As you develop your HTTP client, you may find that you need to pass user security credentials to an HTTP server for the purpose of authentication. The `OpenEdge.Net` library enables you to pass security credentials through one of the following authentication mechanisms:

- HTTP basic authentication
- HTTP digest authentication
- Authentication callback
- On-demand authentication

In all of these mechanisms, you only need to create and pass credentials to the appropriate authentication method when building an HTTP request. Any required encryption or encoding is done automatically by the `OpenEdge.Net` procedure library.

HTTP basic authentication

In HTTP basic authentication, the HTTP client passes credentials preemptively in an Authorization header along with the request. Because this authentication method passes credentials preemptively, it is useful only when you know the user's credentials in advance and when you are sure that the server requires credentials to process the request.

Note that credentials are encoded using the Base64 encoding scheme, but not encrypted, so this mechanism is secure only when you are using HTTPS.

To implement HTTP basic authentication, create a `Credentials` object and pass it to the `UsingBasicAuthentication()` method. The `Credentials` class constructor takes three parameters: domain, username, and password.

```
USING OpenEdge.Net.HTTP.Credentials.

DEFINE VARIABLE oCredentials AS Credentials NO-UNDO.
DEFINE VARIABLE httpUrl AS CHARACTER NO-UNDO.

httpUrl = "http://localhost:9090/oem/resources".

oCredentials = new Credentials('application', 'admin', '4admin').

oRequest = RequestBuilder:Get(httpUrl):UsingBasicAuthentication(oCredentials):Request.
```

HTTP digest authentication

In HTTP digest authentication, the HTTP client creates a hash key by encrypting certain fields, such as username and password, using a nonce (a random number generated for one-time use) provided by the server. The client then sends the hash key back to the server. The server produces its own hash key from the same fields (username, password, etc) and compares it with the hash key that is sent by the HTTP client. If the hash keys match, the authentication succeeds and the server provides access to the requested resource.

HTTP digest is more secure than HTTP basic authentication.

To implement HTTP digest authentication, create a `Credentials` object and pass it to a `UsingDigestAuthentication()` method when building an HTTP request. Other steps, such as obtaining a nonce from the server and creating a hash key, are performed internally by the `OpenEdge.Net` library.

```
oCredentials = new Credentials('application', 'admin', '4admin').

oRequest = RequestBuilder:Get(httpUrl):UsingDigestAuthentication(oCredentials):Request.
```

Authentication callback

In the authentication callback mechanism, the HTTP client responds to a server's demand for authentication by invoking an ABL class or procedure that is responsible for providing the required credentials (typically through a login form or widget presented to a user).

To implement the authentication callback mechanism, use the `AuthCallback()` method as shown in the following examples.

Adding Auth Filter Callback as a class

```
USING OpenEdge.Net.HTTP.Filter.Auth.IAuthFilterEventHandler.
USING OpenEdge.Net.HTTP.Credentials.


oRequest = RequestBuilder:Get('http://localhost:9090/oem/resources')
           :AcceptJson()
          :AuthCallback(new AuthStatusListener())
           :Request.

/* The AuthStatusListener class is defined as: */
CLASS AuthStatusListener implements IAuthFilterEventHandler:

  METHOD PUBLIC VOID AuthFilter_HttpCredentialRequestHandler(
      INPUT poSender as Object,
      INPUT poEventArgs as AuthenticationRequestEventArgs ):

  poEventArgs:Credentials = new
  Credentials('domain','admin','4admin').
  END METHOD.

END CLASS.
```

Adding Auth Filter Callback as a procedure

```
USING OpenEdge.Net.HTTP.AuthenticationRequestEventArgs.
USING OpenEdge.Net.HTTP.Credentials.

oRequest = RequestBuilder:Get('http://localhost:9090/oem/resources')
                :AcceptJson()
                :AuthCallback(this-procedure)
                :Request.

PROCEDURE AuthFilter_HttpCredentialRequestHandler:
    DEFINE INPUT PARAMETER poSender as Object.
    DEFINE INPUT PARAMETER poEventArgs as AuthenticationRequestEventArgs.

    poEventArgs:Credentials = new
    Credentials('domain','admin','4admin').

END PROCEDURE.
```

On-demand authentication

This method is similar to HTTP basic authentication; however, instead of sending credentials preemptively, the HTTP client sends credentials only when required by the server. To implement this authentication mechanism, use the `UsingCredentials()` method as shown in the following example.

```
oCredentials = new Credentials('application', 'admin', '4admin').

oRequest = RequestBuilder:Get('http://localhost:9090/oem/resources')
                :AcceptJson()
                :UsingCredentials(oCredentials)
                :Request.
```

Authorization header

In addition to using any of the authentication mechanisms listed in this topic, you can use the `AddHeader()` method to specify the **Authorization** header. This is useful in certain situations, such as when you need to pass OAuth bearer tokens.

```
lcToken = oJson:GetJsonText("access_token").
cToken = 'Bearer ' + STRING(lcToken).

oRequest = RequestBuilder
            :Post('<URL>',oRequestBody)
            :AddHeader('Authorization', cToken)
            :AcceptJson()
            :ContentType('application/json')
            :Request.
```

API reference documentation

To view the complete list of methods that you can use on an HTTP request object, refer to the RequestBuilder reference documentation or see the OpenEdge.Net API Reference documentation. To learn more about methods that you can invoke on a Credentials object, see the Credentials reference documentation.

Execute an HTTP request

To execute an HTTP request, you need a client object and a response object. Use the `OpenEdge.Net.HTTP.ClientBuilder` class to create a client object that implements the `OpenEdge.Net.HTTP.IHttpClient` interface. Then, call the `Execute()` method on the client object. The response returned by the server application or service is copied to the response object. For example:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.

DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE OResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.

//Build a request
oURI =
URI:Parse("http://oemobiledemo.progress.com/OEMobileDemoServices/rest/CustomerService/Customer?filter=CustNum=1").
oRequest = RequestBuilder:GET(oURI):Request.
```

```
//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).


//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

The `ClientBuilder:Build():Client` method is the minimum you need to build an HTTP client; however, `IHttpClient` is a fluent interface, so you can add other methods in a method chain as shown in the following example:

```
oClient = ClientBuilder:Build()
   :ViaProxy(URI)
   :KeepCookies(ICookieJar)
   :SetRequestTimeout(DECIMAL)
   :SetNumRetries(INTEGER)
   :Client.
```

Use the `SetRequestTimeout()` method to set a timeout value in seconds for the HTTP connection to be established. Use the `SetNumRetries()` method to set the number of attempts at making a connection.

You can also use the client object to:

- Configure TLS security settings on page 31
- Implement a stateful client using cookies on page 24

API reference documentation

To view the complete list of methods that you can use on a Client object, refer to the ClientBuilder reference documentation or see the OpenEdge.Net API Reference documentation.

Implement a stateful client using cookies

HTTP is a stateless protocol. Each HTTP request from a client must contain all the information that is required by the server to process the request. The key advantage of this approach is that it frees up server resources that would otherwise be used in maintaining state information.

Instead, state information is maintained through the use of cookies. A cookie is a set of name-value pairs that the server sends back in response to an initial client request. A cookie could include information such as a session ID or an authentication token. Each subsequent client request must include the cookie, thus preventing the need for end-users to enter information such as login credentials for each request.

The `OpenEdge.Net` procedure library supports the use of cookies, enabling you to implement a stateful HTTP client. Cookies that are received from a server during a message transaction are stored in JSON format on a local disk on the HTTP client's host.

Each cookie has a domain and a path property that is mapped to a URI. When your HTTP client makes a request to a URI, all cookies associated with the URI are attached to the request. To enable this behavior, append the `KeepCookies()` method when building the HTTP client.

```
oClient = ClientBuilder:Build():KeepCookies():Client.
```

How cookies are stored

Cookies are stored in an instance of an `OpenEdge.Net.HTTP.ICookieJar` class. The following points summarize how these cookies are stored:

- Cookies must be associated with a domain. This domain may be prefixed with a wildcard '.' .
- If the cookie itself has no domain, the request's host domain is used.
- The cookie domain and the request host domain must be the same.
- Cookies that have no expiration date are considered session cookies and are deleted when the CookieJar is deleted/cleaned up. Session cookies can be cleaned using the `CleanSessionCookies()` method.

Cookies are retrieved from the cookie jar and attached to an HTTP request automatically based on the following criteria:

- The request's host is identical to cookie's domain, or the host matches the cookie's domain.
- The request's URL path begins with the cookie path.

How to read and write cookies

To read/write cookies on requests and responses, use the methods from the `IHttpMessage` interface as shown in the following code sample.

```
/** Adds a cookie to this message
    @param Cookie The cookie to add. */

METHOD PUBLIC VOID SetCookie(INPUT poCookie AS Cookie).

/** Returns all the cookies for this message
    @param Cookie[] An array of cookies
    @return integer The number of cookies returned */

METHOD PUBLIC INTEGER GetCookies(OUTPUT poCookies AS Cookie extent).

/** Removes a cookie from this message
    @param Cookie The cookie to remove. */

METHOD PUBLIC VOID RemoveCookie(INPUT poCookie AS Cookie).

/** Indicates whether a cookie exists for this message
    @param Cookie The cookie to check
    @return logical True if this message contains the cookie */

METHOD PUBLIC LOGICAL HasCookie(INPUT poCookie AS Cookie).

/** Removes all cookies from this message */

METHOD PUBLIC VOID ClearCookies().
```

Note: You can inspect an individual cookie by casting the cookie to a string (using `ToString()`, `STRING(oCookie)`, or `MESSAGE oCookie`).

API reference documentation

To view the list of methods that you can use on a Cookie object, refer to the OpenEdge.Net.HTTP.Cookie reference documentation.

Process an HTTP response

The response returned by the server to the HTTP client is copied to the HTTP response object. The response object contains the following fields:

- `INTEGER StatusCode`—the HTTP response status code
- `CHARACTER StatusReason`—the HTTP response status reason
- `CHARACTER ContentType`—the response message content type (for example, `text/xml`, `application/json`)
- `INTEGER ContentLength`—the length (in bytes/characters) of the response message
- `CHARACTER CharacterEncoding`—the response message's content type encoding (for example, `charset='utf-8'`)
- `RAW ContentMD5`—an MD5 hash of the response message content
- `Progress.Lang.Object Entity`—an ABL object containing the response message content
- `CHARACTER TransferEncoding`—the transfer encoding used in the response message
- `CHARACTER Version`—the HTTP version

The following example demonstrates how to process a response. In this example, the response message is written to a log file if the status code that is returned is 200. The response message content is extracted from the `Entity` property of the HTTP response object.

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.

DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE OResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.

oJsonObject = new JsonObject().
LOG-MANAGER:LOGFILE-NAME = "C:\mylog.log".

oClient = ClientBuilder:Build():Client.
oURI = new URI('http', 'oemobiledemo.progress.com').
oURI:Path = "/OEMobileDemoServices/rest/CustomerService/Customer".
oURI:AddQueryString("filter=Country='Finland'").

oRequest = RequestBuilder:GET(oURI):Request.
oResponse = oClient:Execute(oRequest).

IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oJsonObject = CAST(oResponse:Entity, JsonObject).
    oJsonObject:Write(JsonString, true).
    LOG-MANAGER:WRITE-MESSAGE(STRING(JsonString)).
    LOG-MANAGER:WRITE-MESSAGE(oResponse:ContentType).
```

Response message content types

The following table lists the object types that the response message's `Entity` field may contain, with the corresponding MIME and ABL object types:

Content type	MIME type(s)	ABL object type
`TEXT`	`text/*`	`OpenEdge.Core.String`
`JSON`	`application/json`	`Progress.Json.ObjectModel.JsonConstruct`
`XML`	`text/xml`, `text/xml-external-parsed-entity`, `application/xml`, `application/xml-external-parsed-entity`, `application/xml-dtd`	Implementation of `Ccs.Common.Support.IHandleHolder`
`Binary` (including media and PDF files)	`application/octet-stream`, `application/pdf`, `application/zip`, `application/gzip`, `audio/*`, `image/*`, `video/*`	`OpenEdge.Core.ByteBucket`, `OpenEdge.Core.Memptr`
`Multipart`	`multipart/*`	`OpenEdge.Net.MultipartEntity`

Examples of how to process different content types are provided in the following topics:

- String response on page 28
- JSON response on page 28
- XML response on page 29
- Binary response on page 30
- Multipart response on page 30

Note:

- Response messages with error codes such as 404 or 501 do not cause the ABL HTTP client to throw the errors. They are processed just like any other response.
- If the HTTP client is unable to transform the response body to an object type that represents the Content-Type header (as listed in the previous table), the data is returned as bytes of type `OpenEdge.Core.Memptr`. The content type is changed to `application/octet-stream`. The original content type header returned by the server is set to `X-Orig-Content-Type`. You should, therefore, check the response's ContentType header as well as the ABL object type of the response's `Entity` property, when deciding how to handle the response.

API reference documentation

To view the list of methods that you can use on a Response object, refer to the OpenEdge.Net.HTTP.IHttpResponse reference documentation.

String response

To process a string in the response message, cast the `Entity` field in the response object to an `OpenEdge.Core.String` object. For example:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.


DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE OResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE ResponseString AS OpenEdge.Core.String NO-UNDO.

//Build a request
oURI = URI:Parse("http ://httpbin.org/robots.txt").
oRequest = RequestBuilder:GET(oURI):Request.

//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    ResponseString = CAST(oResponse:Entity, OpenEdge.Core.String).
    MESSAGE ResponseString.
```

JSON response

To process JSON content in the body of the response message, cast the `Entity` field of the response message to a `JsonObject`, as shown in this example:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.


DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE OResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.

DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.

//Build a request
oURI = URI:Parse("http://httpbin.org/json").
oRequest = RequestBuilder:GET(oURI):Request.

//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
```

```
    oJsonObject = CAST(oResponse:Entity, JsonObject).

    oJsonObject:Write(JsonString, true).
    MESSAGE string(JsonString).
```

XML response

If the content type in the response message is set to `text/xml` or `application/xml`, the XML content can be accessed as an ABL `X-DOCUMENT` object. To process the XML content, cast the `Entity` field to an `OpenEdge.Core.WidgetHandle` object and copy the resulting value to an `X-DOCUMENT` handle. For example:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.
USING OpenEdge.Core.WidgetHandle FROM PROPATH.

DEFINE VARIABLE oLib AS IHttpClientLibrary NO-UNDO.



DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE hXml AS HANDLE NO-UNDO.
DEFINE VARIABLE hNodeChild AS HANDLE NO-UNDO.
DEFINE VARIABLE jx AS INTEGER NO-UNDO.


//Build a request
oURI = URI:Parse("http://httpbin.org/xml").
oRequest = RequestBuilder:GET(oURI):Request.


//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    MESSAGE oResponse:ContentType.
    CREATE X-DOCUMENT hXml.
    CREATE X-NODEREF hNodeChild.
    hXml = CAST(oResponse:Entity, WidgetHandle):Value.
    REPEAT jx = 1 TO hXml:NUM-CHILDREN:
        hXml:GET-CHILD(hNodeChild, jx).
        MESSAGE hNodeChild:NODE-VALUE.
    END.
```

Binary response

To process images or files that are returned in an HTTP response message, cast the `Entity` field in the response message to a `ByteBucket` as shown in this example:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING Progress.Json.ObjectModel.*.
USING OpenEdge.Core.Memptr FROM PROPATH.
USING OpenEdge.Core.ByteBucket FROM PROPATH.
USING OpenEdge.Net.HTTP.Lib.ClientLibraryBuilder.

DEFINE VARIABLE oLib AS IHttpClientLibrary NO-UNDO.



DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE mData AS MEMPTR NO-UNDO.
DEFINE VARIABLE oPic AS CLASS Memptr NO-UNDO.
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.


DEFINE VARIABLE oResponseBody AS OpenEdge.Core.String NO-UNDO.
DEFINE VARIABLE JsonString AS LONGCHAR NO-UNDO.
DEFINE VARIABLE oJsonObject AS JsonObject NO-UNDO.
DEFINE VARIABLE oImageInResponse AS CLASS ByteBucket NO-UNDO.
DEFINE VARIABLE oImage AS CLASS Memptr NO-UNDO.

//Build a request
oURI =
URI:Parse("https://raw.githubusercontent.com/PeterJudge-PSC/http_samples/master/http_client/multipart_messages/smile.png").
oRequest = RequestBuilder:GET(oURI):Request.


//Execute a request
oLib = ClientLibraryBuilder:Build():sslVerifyHost(NO):Library.

oClient = ClientBuilder:Build():UsingLibrary(oLib):Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oImageInResponse = CAST(oResponse:Entity, ByteBucket).
    oImage = oImageInResponse:GetBytes().
    COPY-LOB FROM oImage:Value TO FILE "C:/Samples/NewSmile1.png".
```

Multipart response

To process a multipart response message, cast the `Entity` field of the response message to a `MultipartEntity` object, as shown in this example:

```
USING OpenEdge.Net.HTTP.*.
USING OpenEdge.Net.URI.
USING OpenEdge.Net.MessagePart.
USING OpenEdge.Net.MultipartEntity.
```

```
DEFINE VARIABLE oClient AS IHttpClient NO-UNDO.
DEFINE VARIABLE oURI AS URI NO-UNDO.
DEFINE VARIABLE oRequest AS IHttpRequest NO-UNDO.
DEFINE VARIABLE oResponse AS IHttpResponse NO-UNDO.
DEFINE VARIABLE oResponseEntity AS MultipartEntity NO-UNDO.
DEFINE VARIABLE msgPart AS MessagePart NO-UNDO.
DEFINE VARIABLE partBody AS Progress.Lang.Object NO-UNDO.
DEFINE VARIABLE loop AS INTEGER NO-UNDO.

//Build a request
oURI = URI:Parse("http://httpbin.org/get").
oRequest = RequestBuilder:POST(oURI):Request.


//Execute a request
oClient = ClientBuilder:Build():Client.
oResponse = oClient:Execute(oRequest).

//Process the response
IF oResponse:StatusCode <> 200 THEN
    RETURN ERROR "Request Error: " + STRING(oResponse:StatusCode).
ELSE
    oResponseEntity = CAST(oResponse:Entity, MultipartEntity).
    DO loop = 1 to oResponseEntity:SIZE:
        ASSIGN msgPart  = oResponseEntity:GetPart(loop)
               partBody = msgPart:Body.
    END.
```

Configure TLS security settings

Transport Layer Security (TLS) is a security protocol that specifies the use of digital certificates to verify identity and encrypt messages transmitted between client and server applications.

Note: TLS is the new version of SSL.

HTTPS is a combination of HTTP and TLS. The TLS protocol specifies that a client and server application must first perform a TLS 'handshake' before exchanging request or response messages. The TLS handshake is a process wherein the client and server:

- Prove their identity to each other through public key certificates.
- Agree upon a set of cipher suites (encryption algorithms).
- Generate a session key that will be used to encrypt messages.

In most cases, only the server proves its identity to the client. However, an organization's security policy may require a client application to also authenticate itself. In such situations, the client application must submit a public key certificate to the server.

If the handshake is successful, the client and server begin sending each other encrypted request or response messages. If the handshake fails, no further communication can happen between the client and the server.

The ABL HTTP client library supports two-way authentication. You can use it to verify a server certificate as well as send a client certificate to the server. The ABL HTTP client library performs the steps described in the handshake process under the covers.

Server authentication

To establish a successful HTTPS connection, a server sends its public key certificate to the client. The certificate must be signed by a Certificate Authority (CA) that is trusted by the client.

An ABL HTTP client verifies the identity of a server by checking the server's root certificate against the CA certificates installed in the OpenEdge certificate store. A root certificate is the public key certificate of the CA that has signed the server's certificate.

Tip: The HTTP client does *not* automatically install root certificates. To retrieve the root certificate for a site, use a browser to navigate to the URL. Modern browsers indicate an TLS connection with a padlock or some similar icon. This icon is usually clickable and includes a means to inspect and export the certificates for that site. Make sure you export all of the certificates for the site and import them into the OpenEdge certificate store.

To learn more about installing root certificates, see 'Install trusted CA/root certificates' in the *Manage OpenEdge Keys and Certificates* guide.

Client authentication

A server may have security policies that require a client to submit a client certificate during the TLS handshake. This process is called two-way TLS authentication or mutual TLS authentication.

If your ABL HTTP client needs to participate in mutual TLS authentication, you must, in addition to installing the server's root certificate in the OpenEdge certificate store, obtain a public/private key pair and a digital certificate containing the public key to prove the client's identity. You can do this in one of the following ways:

- You can create a self-signed certificate, where you act as your own CA. A self-signed certificate is often used in development and testing environments.
- Generate a public/private key pair, create a certificate signing request (CSR) containing the public key, and submit the CSR to a Certificate Authority. The CA will return a signed public key certificate for you to use. A public key certificate signed by a widely trusted CA is typically required in a production system.

Tip: You must ensure that the client-side root CA certificate is added to the server's trusted certificate store. This is especially true for self-signed certificates. If your client public key certificate is signed by a well-known CA, it is likely that the CA's root certificate is already in the server's certificate store. If not, you must work with the server administrator to ensure that the root CA certificate is included in the server's certificate store.

OpenEdge provides a utility called `pkiutil` that enables you to create certificates as well as generate CSRs.

To learn more the `pkiutil` utility, see 'pkiutil' in the *Manage OpenEdge Keys and Certificates* guide.

To learn about creating self-signed certificates, see https://knowledgebase.progress.com/articles/Knowledge/000027719.

The ABL HTTP client requires access to a PEM file that contains both the public and the private key. You can generate a PEM file using the `pkiutil` utility.

Once you have a PEM file ready, perform the following steps in your ABL HTTP client code:

1. Create a `TlsClientCredentials` object containing two properties:
   - A path to the PEM file. You can specify either an absolute path or a path relative to a directory listed in the PROPATH.
   - An encoded passphrase to enable the HTTP client to access the private key in the PEM file.

```
var EncodedString oEncSt.
var TlsClientCredentials oCred.
```

```
// Create a passphrase for the client certificate using plain-text encoding.
oEncSt = new EncodedString().
oEncSt:encoding = EncodingTypeEnum:nopr0.
oEncSt:value = "password".

// Create credentials which consists of certificate path and encoded passphrase.
oCred = new TlsClientCredentials().

oCred:identity = new OpenEdge.Core.String("PAS-default-cert.pem").

oCred:secret = oEncSt.
```

2. Pass the `TlsClientCredentials` object to the `UsingClientCredentials()` method during the construction of the HTTP request object.

```
...
// Create credentials which consists of certificate path and encoded passphrase.
oCred = new TlsClientCredentials().
oCred:identity = new
OpenEdge.Core.String("C:\OpenEdge\WRK\oepas1\bin\PAS-default-cert.pem").
oCred:secret = oEncSt.

// Create the request with stated credentials, calling the built-in REST ping service.
oRequest = RequestBuilder:Get("https://httpbin.org/get")
                          :UsingClientCredentials(oCred)
                          :Request.
```

Here is the complete code example:

```
using OpenEdge.Core.EncodedString.
using OpenEdge.Core.EncodingTypeEnum.
using OpenEdge.Net.HTTP.*.
using OpenEdge.Net.ServerConnection.TlsClientCredentials.

BLOCK-LEVEL ON ERROR UNDO, THROW.

var IHttpRequest oRequest.
var IHttpResponse oResponse.
var IHttpClient oClient.
var EncodedString oEncSt.
var TlsClientCredentials oCred.

// Create a passphrase for the client certificate using plain-text encoding.
oEncSt = new EncodedString().
oEncSt:encoding = EncodingTypeEnum:nopr0.
oEncSt:value = "password".

// Create credentials which consists of certificate path and encoded passphrase.
oCred = new TlsClientCredentials().
oCred:identity = new OpenEdge.Core.String("PAS-default-cert.pem").
oCred:secret = oEncSt.

// Create the request with stated credentials, calling the built-in REST ping service.
oRequest = RequestBuilder:Get("https://httpbin.org/get")
                          :UsingClientCredentials(oCred)
                          :Request.

// Build the client using the library/options previously defined.
oClient = ClientBuilder:Build()
                       :Client.
```

```
// Execute the client request.
oResponse = oClient:Execute(oRequest).

// Display the server's response status code.
message oResponse:StatusCode.
```

Use a callback to get client credentials

In the previous code example illustrating client-side TLS authentication, the `TlsClientCredentials` object was constructed inside the ABL HTTP client code. This approach works well if the same client certificate is used for every TLS connection to the server. However, if you need to use different client certificates—for example, if you have multiple users connecting to the server and each user has their own client certificate—you should:

- Delegate the task of constructing the `TlsClientCredentials` object to an ABL class that implements `ISocketConnectionCredentialsHandler`.
- Use a callback from within the ABL HTTP client to obtain the `TlsClientCredentials` object.

Here is an example of a callback class that implements `ISocketConnectionCredentialsHandler`:

```
class Callback implements ISocketConnectionCredentialsHandler:

    method public void ClientSocket_ConnectionCredentialsHandler (input pSender as
Progress.Lang.Object, input pEventArgs as CredentialsEventArgs):
        var EncodedString oEncSt.
        var TlsClientCredentials oCred.

        // Create a passphrase for the client certificate using plain-text encoding.
        oEncSt = new EncodedString().
        oEncSt:encoding = EncodingTypeEnum:nopr0.
        oEncSt:value = "password".

       // Create credentials which consists of certificate path and encoded passphrase.

        oCred = new TlsClientCredentials().
        oCred:identity = new OpenEdge.Core.String('PAS-default-cert.pem').
        oCred:secret = oEncSt.

        assign pEventArgs:Credentials = oCred.

    end method.

end class.
```

The callback class must implement the `ClientSocket_ConnectionCredentialsHandler()` method. This method takes two input parameters—a `Progress.Lang.Object` object and a `CredentialsEventArgs` object. Inside the method, you must construct the `TlsClientCredentials` object and assign it to the `Credentials` property of the `CredentialsEventArgs` object.

Then, from within the ABL HTTP client program, you create an object of the callback class and specify the callback object when building the HTTP request:

```
...
VAR ISocketConnectionCredentialsHandler oCallback = NEW Callback().
...
oRequest = RequestBuilder:Get("https://httpbin.org/get")
                         :UsingClientCredentials(oCallback)
                         :Request.
```

Use multiple callbacks to get client credentials from different sources

You can chain multiple client credential callbacks when building an HTTP request object. This is useful when credentials are stored in different locations, such as a database and a file system, and your code must search each location for a credential.

To use multiple callbacks, perform the following steps:

1. Create a callback handler class for each source of credentials. For example, you might create a `DatabaseCredentialsHandler` class and a `FileCredentialsHandler` class to get credentials from a database and a file system respectively. The handler classes should implement `ISocketConnectionCredentialsHandler`.

```
class DatabaseCredentialsHandler implements ISocketConnectionCredentialsHandler:

...
...
end class.
```

2. In each handler class, write the logic to obtain the credential and implement the `ClientSocket_ConnectionCredentialsHandler()` method, assigning the credential to the `CredentialsEventArgs:Credentials` property.

Note: You can also set the `CredentialsEventArgs:Cancel` property to `true` to prevent other callbacks from executing if the credential is found in the current callback.

```
class DatabaseCredentialsHandler implements ISocketConnectionCredentialsHandler:
   ...
   method public void ClientSocket_ConnectionCredentialsHandler (input pSender as
Progress.Lang.Object, input pEventArgs as CredentialsEventArgs).
    // Get credentials from the DB
              var TlsClientCredentials oCred.
     ...
    pEventArgs:Credentials = oCred.
    pEventArgs:Cancel = yes.
   end method.
...
end class.
```

3. In the ABL HTTP client program, instantiate the handler classes and chain them as shown in this example:

```
var IHttpRequest oRequest.
var DatabaseCredentialsHandler dbCallback = new
DatabaseCredentialsHandler('sports2000').
var FileCredentialsHandler fileCallback = new FileCredentialsHandler('/path/to/certs').

oRequest = RequestBuilder:Get('http://localhost/')
                 :UsingClientCredentials(dbCallback)  // first check in the database

    :UsingClientCredentials(fileCallback)// then check in the files
                    :Request.
```

Turn off host verification

If HTTPS is used as the communication protocol, the HTTP client verifies the server certificate by default. However, based on your needs, you may want to turn off this verification. To turn off host verification, create a custom `ClientLibrary` and set the `sslVerifyHost` property to `NO`. For example:

```
USING OpenEdge.Net.HTTP.IHttpClientLibrary.
USING OpenEdge.Net.HTTP.Lib.ClientLibraryBuilder.

DEFINE VARIABLE oLib AS IHttpClientLibrary NO-UNDO.

oLib = ClientLibraryBuilder:Build():sslVerifyHost(NO):Library.

oHttpClient = ClientBuilder:Build():UsingLibrary(oLib):Client.
```

Set TLS ciphers and protocols

An HTTP client begins communication with a server application over HTTPS by negotiating security settings. This includes the selection of a cipher suite — a set of algorithms that determine how keys are exchanged and how messages are encrypted.

If you need to specify TLS ciphers and protocols in your ABL HTTP client, create a custom `ClientLibrary` and use the `SetSSLProtocols()` and `SetSSLCiphers()` methods as shown in the following example.

```
USING OpenEdge.Net.HTTP.IHttpClient.
USING OpenEdge.Net.HTTP.IHttpClientLibrary.
USING OpenEdge.Net.HTTP.ClientBuilder.
USING OpenEdge.Net.HTTP.Lib.ClientLibraryBuilder.

/* *************************  Main Block  ************************** */
DEFINE VARIABLE oLib          AS IHttpClientLibrary NO-UNDO.
DEFINE VARIABLE oClient       AS IHttpClient        NO-UNDO.
DEFINE VARIABLE cSSLProtocols AS CHARACTER EXTENT   NO-UNDO.
DEFINE VARIABLE cSSLCiphers   AS CHARACTER EXTENT   NO-UNDO.

// the size and values of the TLS protocols and ciphers depend on the server
EXTENT(cTLSProtocols) = 2.
EXTENT(cTLSCiphers) = 10.


ASSIGN cTLSProtocols[1] = 'TLSv1.2'
       cTLSProtocols[2] = 'TLSv1.1'
       cTLSCiphers[1]  = 'AES128-SHA256'
       cTLSCiphers[2]  = 'DHE-RSA-AES128-SHA256'
       cTLSCiphers[3]  = 'AES128-GCM-SHA256'
       cTLSCiphers[4]  = 'DHE-RSA-AES128-GCM-SHA256'
       cTLSCiphers[5]  = 'ADH-AES128-SHA256'
       cTLSCiphers[6]  = 'ADH-AES128-GCM-SHA256'
       cTLSCiphers[7]  = 'ADH-AES256-SHA256'
       cTLSCiphers[8]  = 'AES256-SHA256'
       cTLSCiphers[9]  = 'DHE-RSA-AES256-SHA256'
       cTLSCiphers[10] = 'AES128-SHA'
       .
oLib = ClientLibraryBuilder:Build():
                    :SetSslProtocols(cSSLProtocols)
                    :SetSslCiphers(cSSLCiphers)
                    :Library.

oClient = ClientBuilder:Build()
                :UsingLibrary(oLib)
                :Client.
```

For a list of supported ciphers and protocols, see **Supported protocols, ciphers, and certificates for Progress OpenEdge clients and servers** in the *Learn About Security and Auditing* guide.

API reference documentation

To view the complete list of methods that you can use in a custom Client Library, refer to the ClientLibrary documentation or see the OpenEdge.Net API Reference documentation.