MAKING HTTP & REST REQUESTS WITH OPENEDGE.NET.PL

TABLE OF CONTENTS

INTRODUCTION..... 2
BASIC OPERATION 3
SECURITY CONSIDERATIONS 9
SUPPORTED OPENEDGE VERSIONS..... 10
INTERNALS 11

INTRODUCTION

OpenEdge 11.4.0 provides the ability to send ("push") notifications to mobile devices via the `OpenEdge.Net.Mobile.PushNotificationService` and helper classes. This service provides a simplified API around the REST calls used to actually call the mobile notification server.

Those REST calls are made with an HTTP Client Library that is in the `OpenEdge.Net.HTTP` namespace. This page describes using that client library to make more generic HTTP and REST calls.

BASIC OPERATION

The Http Client uses a fairly common request-response design with 4 main types being used: `URI`, `HttpRequest`, `HttpClient` and `HttpResponse`. There are a number of helper classes (builders and enumerations) that are also used.

OPENEDGE.NET.URI

The URI class encapsulates the target address, including the scheme (enumerated in `OpenEdge.Net.UriSchemeEnum`), host, port, path, query and friends. There's a great description of the pieces that comprise a URL at http://www.skorks.com/2010/05/what-every-developer-should-know-about-urls/ .

Typically, a new URI object is created via constructor, passing in at least the scheme and host. The other elements of the URI can be added via properties (most) or methods (queries).

```
using OpenEdge.Net.URI.

define variable oURI as URI no-undo.

oURI = new URI(UriSchemeEnum:http, 'oemobiledemo.progress.com').
oURI:Path = 'VehicleOrderService/rest/VehicleOrder/Cart'.
oURI:AddQuery('filter', '').
```

A URI can additionally be derived from a string via the static `Parse()` method. An encoded version of the URL is returned via the `Encode()` method.

```
using OpenEdge.Net.URI.

define variable oURI as URI no-undo.
oURI = OpenEdge.Net.URI:Parse('http://www.progress.com').
```

OPENEDGE.NET.HTTP.HTTPREQUEST

A request has 2 or 3 mandatory elements: a method ("verb" like GET; represented by the `OpenEdge.Net.HTTP.MethodEnum` enum) and a URI (the address for the request).

Some requests also require a payload, called the entity or body or message. To add an entity to a request, set the `Entity` property to an object instance. All entity data must be encapsulated in a `Progress.Lang.Object` (ie not a temp-table handle).

Headers can be added to the request via the `AddHeader()` method.

While the request type (`OpenEdge.Net.Http.HttpRequest`) can be manually built, using the Request Builder helper class to construct a request is indented to be simpler and more readable. Request objects are reusable across requests.

Building a request

The `OpenEdge.Net.Http.RequestBuilder` helper class provides a fluent (or chained) API for creating HttpRequest objects.

The initial methods to call are one of the static 'verb' methods:

- `RequestBuilder:Get(<uri>|<entity>)`
- `RequestBuilder:Put(<uri>, <entity>)`
- `RequestBuilder:Post(<uri>, <entity>)`
- `RequestBuilder:Delete(<uri>|<entity>)`
- `RequestBuilder:Head(<uri>)`
- `RequestBuilder:Options(<uri>|<entity>)`
- `RequestBuilder:Trace(<uri>)`

Where `<uri>` is either an instance of a URI object, or a character string, and `<entity>` is an instance of an object.

From this start, a number of additional methods can be chained

- `AddHeader()`: Allows custom headers to be added
- `Accept*`: Methods to add media types to the `Accept` header
- `HttpVersion`: Change the HTTP version from its 1.1 default
- `Add<data>`: Add Form, JSON or other data to the request
- `UsingBasicAuthentication`, `UsingDigestAuthentication` and the more general `UsingCredentials`: Add credentials to the request. Note that this does not resolve the challenge (see https://tools.ietf.org/html/rfc2617 for details of the challenge). Credentials will be retrieved only when the request is being executed.
- `AddCallback` : Adds a listener to events from the Request. This is used for things like asking the application for credentials for authentication purposes. Initially only a single callback is supported: the `IAuthFilterEventHandler` which can be used in place of the `Using*Authentication` or `UsingCredentials` operations.

Once the request is built, it must be returned via calling the `Request` property; it can then be passed into the HttpClient for execution. Various examples of using the RequestBuilder follow.

REQUESTBUILDER EXAMPLE 1 (BASIC GET)

```
using OpenEdge.Net.URI.
using OpenEdge.Net.UriSchemeEnum.
using OpenEdge.Net.HTTP.RequestBuilder.
using OpenEdge.Net.HTTP.HttpRequest.

define variable oURI as URI no-undo.
define variable oRequest as HttpRequest no-undo.

oURI = new URI(UriSchemeEnum:http, 'www.progress.com').

oRequest = RequestBuilder:Get(oURI)
                :Request.
```

REQUESTBUILDER EXAMPLE 2(SIMPLE PUT)

```
using OpenEdge.Net.URI.
using OpenEdge.Net.UriSchemeEnum.
using OpenEdge.Net.HTTP.RequestBuilder.
using OpenEdge.Net.HTTP.HttpRequest.
using Progress.Json.ObjectModel.JsonObject.

define variable oURI as URI no-undo.
define variable oRequest as HttpRequest no-undo.
define variable oJson as JsonObject no-undo.

oURI = new URI(UriSchemeEnum:http, 'oemobiledemo.progress.com').
oURI:Path = 'VehicleOrderService/rest/VehicleOrder/Cart'.

oJson = new JsonObject().
/* assume that there's some real data in this object */
oJson:Add('dsShoppingCart', new JsonObject()).

oRequest = RequestBuilder:Put(oURI, oJson)
                :AcceptJson()    /* we want to get JSON back */
                :Request.
```

REQUESTBUILDER EXAMPLE 3 (ADDING BASIC AUTH)

```
oURI = URI:Parse('http://localhost:9090/oem/resources').

oCredentials = new Credentials('application', 'admin', '4admin').
oRequest = RequestBuilder:Get(oURI)
                :AcceptJson()
                :UsingBasicAuthentication(oCredentials)
                :Request.
```

REQUESTBUILDER EXAMPLE 4 (ADDING GENERAL CREDENTIALS)

```
oURI = URI:Parse('http://localhost:9090/oem/resources').

oCredentials = new Credentials('application', 'admin', '4admin').
oRequest = RequestBuilder:Get(oURI)
                :AcceptJson()
                :UsingCredentials(oCredentials)
                :Request.
```

REQUESTBUILDER EXAMPLE 5 (ADDING AUTH FILTER CALLBACK)

```
oRequest = RequestBuilder:Get('http://localhost:9090/oem/resources')
                :AcceptJson()
                :AddCallback(get-class(IAuthFilterEventHandler),
                             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:UserName = 'admin'.
        poEventArgs:Credentials:Password = '4admin'.

    end method.
end class.
```

OPENEDGE.NET.HTTP.HTTPCLIENT

The client performs the request via its `Execute()` method, which simply takes an HttpRequest and returns an HttpResponse. The response's `StatusCode` is evaluated and the request may be modified and resubmitted.

EXECUTING A REQUEST

```
using OpenEdge.Net.HTTP.HttpRequest.
using OpenEdge.Net.HTTP.HttpResponse.
using OpenEdge.Net.HTTP.HttpClient.

define variable oRequest as HttpRequest no-undo.
define variable oResponse as HttpResponse no-undo.

oResponse = HttpClient:Instance():Execute(oRequest).
```

Cookies

The default HttpClient is a stateless client (ie no cookies). `OpenEdge.Net.Http.StatefulHttpClient` extends the standard HttpClient and writes cookies into the HttpRequest and reads them from the HttpResponse after request execution. Cookies are stored in an instance of an `OpenEdge.Net.HTTP.ICookieJar` class (the standard implementation is

`OpenEdge.Net.HTTP.CookieJar` which writes persistent cookies to disk in JSON form.

The `OpenEdge.Net.HTTP.Cookie` type encapsulates a single cookie. There's also a `OpenEdge.Net.HTTP.CookieCollection` class provided for working with collections of Cookies.

HTTP Authentication

The HttpClient supports HTTP Basic and Digest authentication, either pre-emptively (ie set by the developer in advance) or on request (via a **401/Unauthorized** status and the `WWW-Authenticate` header).

Options

The HttpClient has an `Options` property (of `OpenEdge.Net.HTTP.ClientOptions` type), which allows the setting of options for the HttpClient, such as timeouts and retry counts.

OPENEDGE.NET.HTTP.HTTPRESPONSE

Once a request successfully executes (note that success means a successful round-trip, not necessarily that the request returns what was desired), an `HttpResponse` object is returned to the caller. The two most important pieces of data this object returns are the `StatusCode` and the `Entity` properties. `StatusCode` indicates the result of the request. These values are enumerated by `OpenEdge.Net.HTTP.StatusCodeEnum`, which is based on http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.1.1

STATUS CODES

```
using OpenEdge.Net.HTTP.HttpRequest.
using OpenEdge.Net.HTTP.HttpResponse.
using OpenEdge.Net.HTTP.HttpClient.

define variable oRequest as HttpRequest no-undo.
define variable oResponse as HttpResponse no-undo.

oResponse = HttpClient:Instance():Execute(oRequest).
message
    oResponse:StatusCode:Value  skip
    oResponse:StatusCode:Name   skip
view-as alert-box.
```

The `Entity` property has a defined type of `Progress.Lang.Object` although the type of the object contained in it is usually something more specialized: the response will convert the (text) received by the HttpClient into a `JsonConstruct`, `OpenEdge.Core.String` or some other type, depending on the `ContentType` of the Response.

The example code below retrieves the Entity and writes it to disk in the correct format based on the type of the returned Entity. We can also use the Response's `ContentType` property to perform similar things.

EXTRACTING TYPED DATA FROM A RESPONSE

```
using Progress.Json.ObjectModel.JsonObject.
using Progress.Json.ObjectModel.ObjectModelParser.
using Progress.Lang.Object.
using OpenEdge.Core.WidgetHandle.
using OpenEdge.Core.String.
using OpenEdge.Net.HTTP.HttpRequest.
using OpenEdge.Net.HTTP.HttpResponse.
using OpenEdge.Net.HTTP.HttpClient.

define variable oRequest as HttpRequest no-undo.
define variable oResponse as HttpResponse no-undo.
define variable oEntity as Object no-undo.
define variable lcHTML as longchar no-undo.

oResponse = HttpClient:Instance():Execute(oRequest).
if type-of(oEntity, JsonObject) then
    cast(oEntity, JsonObject):WriteFile('temp/entity.json', true).
else
if type-of(oEntity, WidgetHandle) then
do:
    hXmlDoc = cast(oEntity, WidgetHandle):Value.
    hXmlDoc:save('file', 'temp/entity.xml').
end.
else
do:
    if type-of(oEntity, String) then
       lcHTML = cast(oEntity, String):Value.
    else
        lcHTML = oEntity:ToString().

    /* Change extension per the Response's ContentType */
    case oResponse:ContentType:
        when 'application/json' then
            copy-lob lcHTML to file 'temp/entity.json'.
        when 'text/html' then
            copy-lob lcHTML to file 'temp/entity.html'.
        otherwise
            copy-lob lcHTML to file 'temp/entity.txt'.
    end case.
end.
```

SECURITY CONSIDERATIONS

HTTPS/SSL & CLIENT-SIDE CERTIFICATE

The HttpClient can make requests to HTTPS URLs as long as the correct client-side certificates are installed in the OpenEdge certificate store. The HttpClient will **not** automatically install client certificates.

The OpenEdge Getting Started: Installation and Configuration guide has a section on **Managing certificate stores for OpenEdge clients and servers** which describes the process to follow to install certificates.

Probably the easiest way to retrieve the relevant certificates is to navigate to the URL you're interested in using your browser. Modern browsers indicate that they're connected via SSL with a padlock or some other visual affordance. This is typically clickable with a means to inspect and export the certificates for that site. Make sure you export all of the certificates for the site (there's usually a tree or chain of them) and import them into the OpenEdge certificate store.

CREDENTIALS AND AUTHENTICATION

Credentials are not persistently stored in the HttpClient library. Credentials can be added to a Request or URI via an `OpenEdge.Net.HTTP.Credentials` object, which is a simple set of properties for user domains, names and passwords. These credentials are used to provide authentication to a Request.

SUPPORTED OPENEDGE VERSIONS

This library was developed against OE 11.4.0 and ships with the product, in the `OpenEdge.Net.pl` procedure library. It ships in source (`$DLC/src`) and r-code (`$DLC/gui` and `$DLC/tty`) versions. It has been lightly tested with 11.3.0 and 10.2B, both in modified (lightly and heavily, resp.) form.

Notes:

- The built-in Progress.Json types do not exist in 10.2B and any JSON processing needs to be manually handled (`READ-` and `WRITE-JSON` from temp-tables and ProDataSets is available in 10.2B). Code to deal with JSON content types must check the response's `ContentType` property and not the type of the Entity property.
- There are assorted bugs in pre-11.4.0 versions of the OOABL that may cause the fluent API to fail (premature garbage collection issues, mainly). In this case, an intermediary variable will be needed. This is relevant particularly when working with the RequestBuilder .
- The version of this library that ships with 11.4.0 does not use the Auth filter chain for authentication; the implementation that is uses is not working.

INTERNALS

This section describes some of the internals of the HTTP Client, and is intended largely as informational.

OPENEDGE.NET.HTTP.LIB

The `OpenEdge.Net.Http` types represent the public API and act as a wrapper around an instance of the `OpenEdge.Net.HTTP.IHttpClientLibrary` interface. This is an interface representing the library performing actual HTTP requests and processing the responses. This library is not 'customer facing': the classes implementing this interface will be highly-specific to the underlying DLL/implementations.

The library currently shipped uses ABL sockets for its HTTP client. The socket library implementation used is an OpenSource library available at https://bitbucket.org/jmls/dotrsocket/ . Alternate implementations could include an external DLL like libcurl or the .NET classes. The ABL sockets version was chosen for its platform independence.

FILTERS

Much incremental functionality is added to the base library by means of filter classes. These filters allow additional functionality to be added to the Http Client and Request and/or Response objects without requiring changes to those types.

Filters are added during object initialisation, and are hard-coded (this is an area for improvement/refactoring).

Filters are chained and by default are executed in order. Individual filters can prevent execution of downstream/further filters. Filters are a specialisation of a singly-`LinkedList` and each filter has its own list. These are in the `OpenEdge.Net.HTTP.Filter` namespace, with further specialisation.

- **Auth**: Provides authentication information to the server for a request, based on the Authentication method and other data. Auth filters work on challenge texts, which are either the content of the `WWW-Authenticate` header, or potentially some HTML for form-based authentication. Used by requests.
- **Body**: Converts a request or response's payload between a `MEMPTR` (text) version as used by the library, and a more strongly-typed version (for example JSON is converted to/from an instance of the `Progress.Json.ObjectModel.JsonConstruct` type (ie `JsonObject` or `JsonArray`). Used by both requests and responses.
- **Entity**: Sets the appropriate `ContentType` property on a request based on the payload's type. Used by requests.
- **Header**: Handles headers and their values. For instance, the `Set-Cookie` header is converted into a 'real' cookie. Used by both requests and responses.

- **Status**: Performs actions based on a response's `StatusCode` property. Used to deal with redirects or authorisation requests. Used by responses.

GENERAL UTILITY CLASSES

A number of general-purpose utility classes were added to the OpenEdge Core library to support the HttpClient; some are detailed here.

OpenEdge.Core.Memptr

The data that the Client libraries return is typically in `MEMPTR` form (from a socket or DLL call). The `Memptr` and `ByteBucket` classes provide utilities for working with `MEMPTR` data.

The `Memptr` class provides a wrapper around an ABL `MEMPTR` data-type, primarily to allow for the passing of `MEMPTR`'s by reference (instead of the typical by-value passing the ABL does).

OpenEdge.Core.ByteBucket

ABL `MEMPTR`s are incapable of being re-sized once initialised (i.e. sized the first time). Client libraries often return data in chunk or streaming form, which means that it is impossible to know the size of the incoming data before receiving it (note this relates to the raw bytes being returned, not the HTTP content). This means that multiple `MEMPTR`s are needed, and need to be managed.

The `OpenEdge.Core.ByteBucket` class internally contains an array of `OpenEdge.Core.Memptr` objects, which it manages (sizing of the array, reading/writing of data into the array contents etc). The `ByteBucket` class allows a developer to work with a single object for reading and writing byte-based data.

The `ByteBucket` also reuses the memory referenced by its managed `MEMPTR`s.

The primary interaction with the `ByteBucket` is via

- `PutBytes()`: Adds bytes from an existing `MEMPTR` into the bucket. This is a deep copy, and the caller is (still) responsible for cleaning up the incoming `MEMPTR`.
- `PutString()`: Adds character data into the bucket.
- `GetBytes()`: Returns the partial or entire contents of the bucket as a single `Memptr` instance.