Integrating IBM i Applications with REST APIs Using Open Source

A Practical Guide for IBM i Developers

Image generated using ChatGPT

Richard Moulton

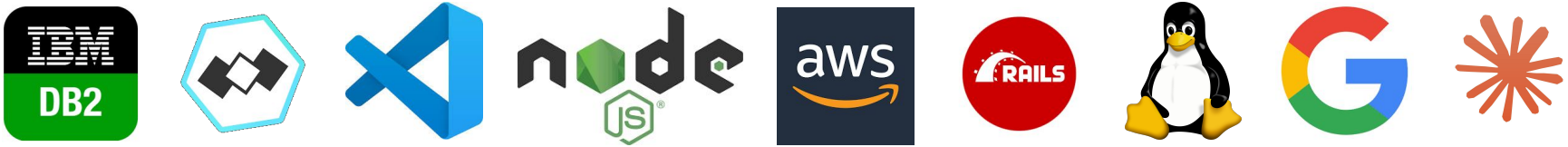
RM Software Services Ltd
richard@rmsoftwareservices.co.uk
https://rmsoftwareservices.co.uk/

RM SOFTWARE SERVICES

v1.0

Richard Moulton

- IBM i developer for 37 years
- Started working as a trainee RPG developer in 1989
- Now working as a freelance IBM i developer since 2019
- I provide development services to a handful of IBM i clients
- I specialise in IBM i RPGLE and of course DB2 for i
- Working more and more with Open Source Software
    - I've contributed to quite a few projects!
    - I've even created a few projects!



I work mostly from my home office in Weare Giffard, North Devon, UK

[illegible] miles from Milton Keynes (by road)

What are we going to discuss?

Setting The Scene
- Why use REST APIs?
- REST & HTTP Primer
- Options on IBM i

The Main Topic
- Introducing HTTPAPI
- HTTPAPI In Action

My Main Focus

Extra Topics
- Some Gotchas
- Beyond The Basics
- Useful Resources
- Q&A

My Aim With This Session

To show you just how easy it is to call REST APIs from your IBM i applications.

What I won't be covering:

I'm going to assume a basic knowledge of the following:
- Web Services
- JSON (JavaScript Object Notation)
- DCM & Certificate Stores

Bonus Slides
- Extras: Further Examples
- Extras: Certificate Stores
- Extras: Certificate Chains
- Extras: MITM Attack

Why use REST APIs?

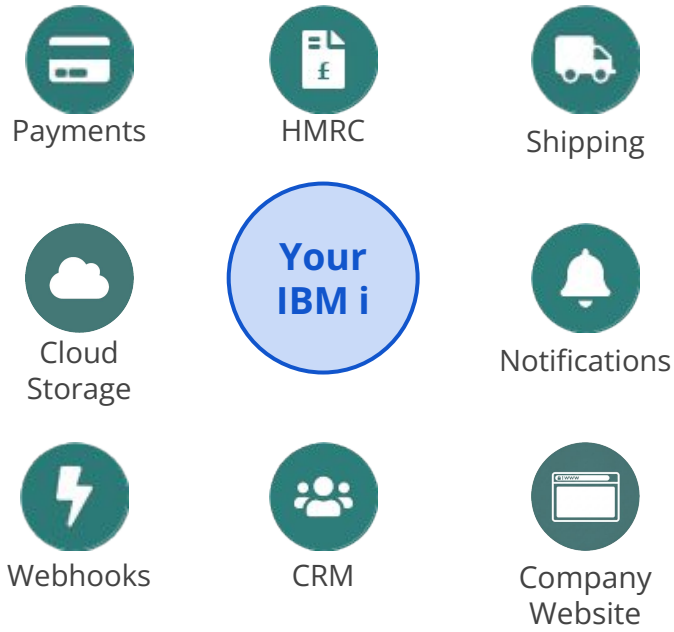
The world has gone REST

Integration used to mean FTP, EDI, CSV files, etc.

Today, REST is the default integration language.

- Most modern cloud applications expose a REST API
- Vendors are sunsetting SOAP
- New developers have grown up on it
- JSON payloads are lightweight, human-readable, and supported everywhere
- HMRC - MTD for VAT is REST only, more coming
- Internal services can talk JSON over HTTP
- Webhooks can fire on customer events

REST matters and can't be ignored!



REST & HTTP Primer

What is REST?

Think of it as a service program - one that lives on someone else's machine.

REST = **RE**presentational **S**tate **T**ransfer - an architectural style for letting programs talk to other programs over HTTP.

What you do in RPG / ILE	REST equivalent
CHAIN a customer record	**GET** /customers/12345
WRITE a new order	**POST** /orders
UPDATE an inventory row	**PUT** /inventory/SKU-123
DELETE an order	**DELETE** /orders/12345
Call a service program procedure	Call a REST endpoint
File feedback / SQLCODE	HTTP status code (200, 404, 500, etc)

REST Request-Response Parts … mapped to an RPG mindset

Every HTTP exchange is one client request, one server response.

Methods

GET, POST, PUT, etc
The action to perform

For a DB operation
CHAIN or READ, WRITE, UPDATE, DELETE

URLs

Identify the resource
/customers/42/orders

Think of this as the table or API name plus the key - the orders belonging to customer 42

Headers

Metadata + content type
Authorization, Accept, …

Think of these as program (or procedure) control parameters, both input and output

Status Codes

2xx good, 4xx your fault, 5xx their fault

Equivalent to an API return code, the outcome (success or failure)

Bodies: JSON in, JSON out (mostly). XML lives on, but rarely in n

For a DB operation this could be the record format or a data structure

REST & JSON

If you're new to JSON then read this!
Scott Klement's Working with JSON in RPG

Pretty much every new REST API expects, and responds with, JSON.

```
{
  "rate_response" : {
    "rates": [
      {
        "carrier_friendly_name" : "FedEx",
        "service_type" : "FedEx First Overnight®" ,
        "delivery_days" : 1,
        "shipping_amount" : {
          "currency": "usd",
          "amount": 141.55
        }
      },
      {
        "carrier_friendly_name" : "FedEx",
        "service_type" : "FedEx Priority Overnight®" ,
        "delivery_days" : 1,
        "shipping_amount" : {
          "currency": "usd",
          "amount": 110.55
        }
      },
      …
    ]
  }
}
```

It's pretty easy to read and understand and maps well to an RPG data structure.

```
dcl-ds rateResponse qualified inz;
  dcl-ds rate response;
    numRates int(10);
    rates likeds(shipRate) dim(50);
  end-ds;
end-ds;

dcl-ds shipRate qualified template;
  carrier friendly name varchar(50);
  service type          varchar(100);
  delivery days         int(5);
  dcl-ds shipping amount;
    currency varchar(10);
    amount   packed(11:2);
  end-ds;
end-ds;
```

REST API Authentication

The schemes you'll meet most often.

API Key	**Static secret sent in a header or query string**	**Simple**
Basic Authentication	**Base64 (user:password) in Authorization header**	**Simple**
Bearer / JWT	**Short or long lived token in Authorization header**	**Moderate**
OAuth 2.0	**Exchange credentials > token > use token. Refresh.**	**Complex**

Key authentication example in Demo 3

JWT authentication example in Demo 2

The Guiding Principles, in IBM i terms

A service interface must satisfy these constraints to be called RESTful.

Uniform Interface

Every resource is accessed in the same way (GET, POST, PUT, etc) - just as SQL gives one uniform language to access any table.

Client - Server

Caller and API don’t know each other’s internals. Your RPG client doesn’t care whether the API is Node, Java or another RPG program.

Stateless

Each call stands alone. Think submitted batch job, not an interactive 5250 session holding state between requests.

Cacheable

Responses can say "reuse this for N seconds" - like avoiding re-reading a reference file you just looked up.

Layered System

A gateway, load balancer, or auth proxy may sit between you and the API. You don't know, and don't need to.

Code on Demand

Optional. The server may send executable code (e.g. JavaScript) for the client to run. Rarely used in practice.

HTTP Status Codes

The common ones you'll see all of the time.

Your fault!
There was something wrong with your request.

2xx - Success

- **200 OK** - standard success
- **201 Created** - resource created (typical POST response)
- **204 No Content** - success, nothing to return (common on DELETE/PUT)

4xx - Client Errors

- **400 Bad Request** - malformed/invalid request
- **401 Unauthorized** - not authenticated
- **404 Not Found** - resource doesn't exist
- **422 Unprocessable Entity** - validation failed (popular in REST APIs)

1xx - Informational

The server has received your request and is telling the client to continue or that something is in progress.

3xx - Redirection

If you need to deal with redirection.

All good!
Your request was processed.

Their fault!
Your request was valid but something went wrong.

5xx - Server Errors

- **500 Internal Server Error** - generic server failure
- **504 Gateway Timeout** - upstream didn't respond in time

API Testing Tools

When integrating with a new REST API I always start with an API Testing tool and my tool of choice is **Postman**, the desktop application version, it's also available as a VS Code extension.

There are loads of similar tools:

- Insomnia
- Bruno
- Hoppscotch
- Thunder Client (VS Code extension)
- curl (command line)
- etc

Options on IBM i

HTTP Client Options

My main focus today

Native RPG/ILE Open Source

- HTTPAPI (Scott Klement)
- ILEvator (Sitemule)

PASE Open Source Tooling

- Node.js, Python, PHP, Java, Ruby
- curl

IBM Supplied

- SQL HTTP functions in QSYS2
- SQL HTTP functions in SYSTOOLS
- IWS Client for ILE (stub generation)

Commercial

- MDRest4i (Midrange Dynamics)
- RPG API Express (Krengeltech / Kato)

Introducing HTTPAPI

Why HTTPAPI for the RPG developer?

Pure RPG, ILE-native

Bind it, call it. No PASE, no QSHELL, no leaving RPG.

Battle-tested

Almost two decades in production on real IBM i shops.

Modern protocols

HTTPS, TLS 1.2/1.3, chunked transfer, redirects, proxies.

YAJL works alongside

JSON generation and parsing built for the same world.

Free and actively maintained

Open source. Active development for 20+ years - 1.54 is the current release

The HTTPAPI Library

The IBM i HTTP client

- Author: Scott Klement - long-time IBM Champion
- First released around 2001, still maintained
- ILE service program (HTTPAPIR4)
- Open source under a permissive license
- Pair with YAJL for JSON
- Built-in XML support
- Latest version = 1.54 (29/03/2026)

Scott states it requires V6R1 or later!

I know it installs fine on V7R4, V7R5 and V7R6.

Where to get it

http://www.scottklement.com/httpapi

Install methods

- Save file + RSTLIB from download
- Build from source (supplied scripts)
- Newer: git clone + gmake (GitHub repo)

In the box

- HTTPAPIR4 service program
- Expat service program (XML parsing)
- Headers/copybooks + dozens of examples

The YAJL Library - The HTTPAPI Companion Library!

Yet Another JSON Library (YAJL)

- Original author: Lloyd Hilaiel (of Mozilla)
- IBM i port by Scott Klement & David Russo (ILE C)
- First released around 2014, still maintained
- ILE service programs (YAJL + YAJLR4)
- Open source under a permissive license
- Integrates with RPG's DATA-INTO and DATA-GEN
- Latest update on Scott's site: 30/09/2024
- Works entirely in UTF-8 Unicode

Scott provides two versions of YAJL one for V7R1+ and the other for V6R1+.

Where to get it

http://www.scottklement.com/yajl

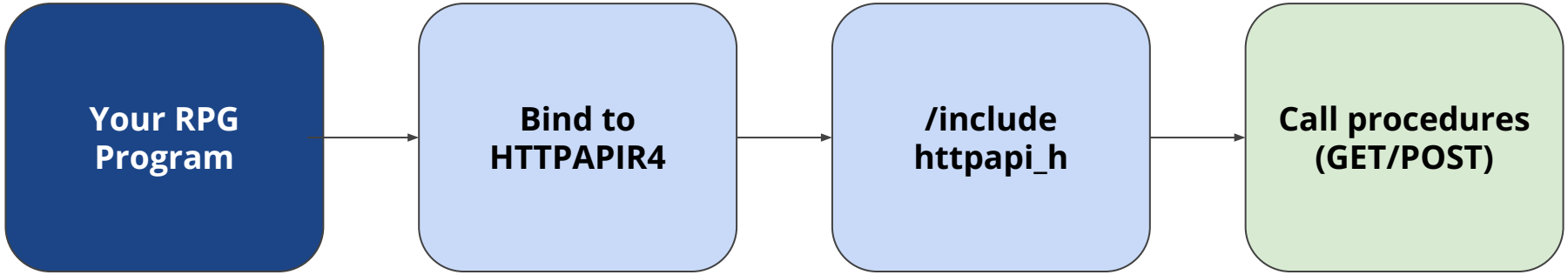
Install methods

- Save file + RSTOBJ from download
- Separate ZIP available for V6R1 systems

In the box

- YAJLR4 service program
- YAJL (ILE C) service program
- DATA-INTO parser & DATA-GEN generator
- YAJLGEN utility - generates an RPG DS from sample JSON
- Example programs here and here

How HTTPAPI fits in your program



HTTPAPI handles SSL/TLS, redirects, chunked transfers, headers, encoding.

You hand it a URL and a body; it hands back a string, an IFS file, or fires a callback procedure as data arrives.

HTTPAPI procedures you'll use 80% of the time

Procedure	Signature (abbreviated)	What's it for
`http_string`	`verb, url [, body, contentType]`	One-liner. Returns response as a string.
`http_stmf`	`verb, url [, resFile, reqFile, … ]`	Request and response via IFS. ★
`http_get`	`url, ifsFile`	Download to an IFS stream file.
`http_req`	`verb, url, …`	Full control: callbacks, custom transport.
`http_setAuth`	`type, user, password / token`	Basic, NTLM, Negotiate, Bearer.
`http_setHeader`	`name, value`	Add or override request headers.
`http_error`	-	Last error code (and helper text).

JSON with YAJL

Two modes you'll use:

Generating JSON

- yajl_genOpen() - start the generator
- yajl_beginObj() / yajl_endObj()
- yajl_beginArray() / yajl_endArray()
- yajl_addChar() / yajl_addNum() / yajl_addBool()
- yajl_copyBufStr() / yajl_saveBuf() - grab the result
- yajl_genClose() - release resources

Parsing JSON (tree mode)

- yajl_string_load_tree() - parse from memory
- yajl_object_find() - descend an object
- yajl_get_string() / yajl_get_number()
- yajl_array_size() / yajl_array_elem()
- yajl_tree_free() - release the tree
- Event-based parser available for huge payloads

HTTPAPI In Action

The Demos: Setting The Scene

All of the source I'll be showing is available in a companion GitHub repository, help yourself: https://github.com/richardm90/rm-rest-apis

- This has been tested on V7R5 but should work fine on V7R4 & V7R6.
- The installation instructions assume some familiarity with open source software on IBM i and connecting to an IBM i via SSH!
- The code can be deployed to pub400.com! However it's not perfect, no DCM access!
- Includes an action to push source to source files.

Example code is not production ready.

No passwords, keys or secrets in source.

Certificate Stores

For the following examples I have two certificate stores:

- *SYSTEM - this **does not** include the root CA certificates for these examples.
- RMCERTS - this does include the root CA certificates for these examples.

The Demos: Setting The Scene

I have included three examples:

1. Retrieve exchange rate information
2. Send an SMS message
3. Retrieve carrier shipping rates

Demo 1 - Get Exchange Rate

The first example will make a simple GET request to a currency exchange service asking the service to return a list of rates for the GBP currency.

I'll be using a free service from exchangerate-api.com.

The API docs can be found here: exchangerate-api.com/docs/standard-requests. This shows the format of the GET request and details of the response body.

It's very easy to use and requires no authentication!

Demo 1 - Get Exchange Rate

API Docs: https://www.exchangerate-api.com/docs/free

```
**free
ctl-opt dftactgrp(*no) actgrp(*new)
        bnddir('HTTPAPI');

/include QCPYSRC,HTTPAPI_H

dcl-s response  varchar(65535);
dcl-s url       varchar(256);

url = 'https://open.er-api.com/v6/latest/GBP' ;

monitor;
  // one line: verb, URL, returns body as a string
  response = http_string('GET':url);
  snd-msg ('OK, length = ' + %char(%len(response)));
  snd-msg ('Response: ' + %trim(response));
on-error;
  snd-msg ('HTTP Error: '   + %trim(http_error()));
endmon;

*inlr = *on;
```

Bind to HTTPAPIR4

/include httpapi_h

No authentication required on this REST API

Call procedures (GET/POST)

GETRATE1.pgm.rpgle

Demo 1 - The JSON Response Body


```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://www.exchangerate-api.com/docs/free",
   "terms of use": "https://www.exchangerate-api.com/terms",
   "time last update unix": 1778803351,
   "time last update utc": "Fri, 15 May 2026 00:02:31 +0000",
   "time next update unix": 1778890781,
   "time next update utc": "Sat, 16 May 2026 00:19:41 +0000",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
       "GBP": 1,
       …
       "EUR": 1.150985,
       …
       "USD": 1.345302,
       …
       "ZMW": 25.592189,
       "ZWG": 34.928,
       "ZWL": 34.928
   }
}
```

Demo 1 - Get Exchange Rate: Enhanced

Source	What does it demonstrate?
`GETRATE1.pgm.rpgle`	The simple example we've just seen, as simple as it gets.
`GETRATE2.pgm.rpgle`	Enable HTTPS strict mode - validates the server identity. Fails, not a trusted CA.
`GETRATE3.pgm.rpgle`	Specify the correct certificate store, no longer uses *SYSTEM. This now works.
`GETRATE4.pgm.rpgle`	Enable debugging, creates a log file in the IFS.
`GETRATE5.pgm.rpgle`	Parse the response and retrieve the GBP to EUR exchange rate.
`GETRATE6.pgm.rpgle`	Same as GETRATE5.pgm.rpgle but using event based parsing. Advanced,not for today!
`GETRATE7.pgm.rpgle`	Same as GETRATE5.pgm.rpgle but using DATA-INTO. More on this later!

HTTPAPI & Certificate Stores

HTTPAPI needs a certificate store, if you don’t specify one then it assumes *SYSTEM.

Certificate stores are created in DCM, this includes the *SYSTEM certificate store.

Application users must have read access to the certificate store file and also execute access to the directories - allowing them to traverse the file path.

The path to the *SYSTEM certificate store in the IFS:

```
/QIBM/UserData/ICSS/Cert/Server/DEFAULT.KDB
```

This doesn’t exist by default!

Gotcha #1

Checkout the Extras: Certificate Stores set of slides.

These slides describe how to create a certificate store and set the appropriate permissions.

More about this later!

What is a certificate store?

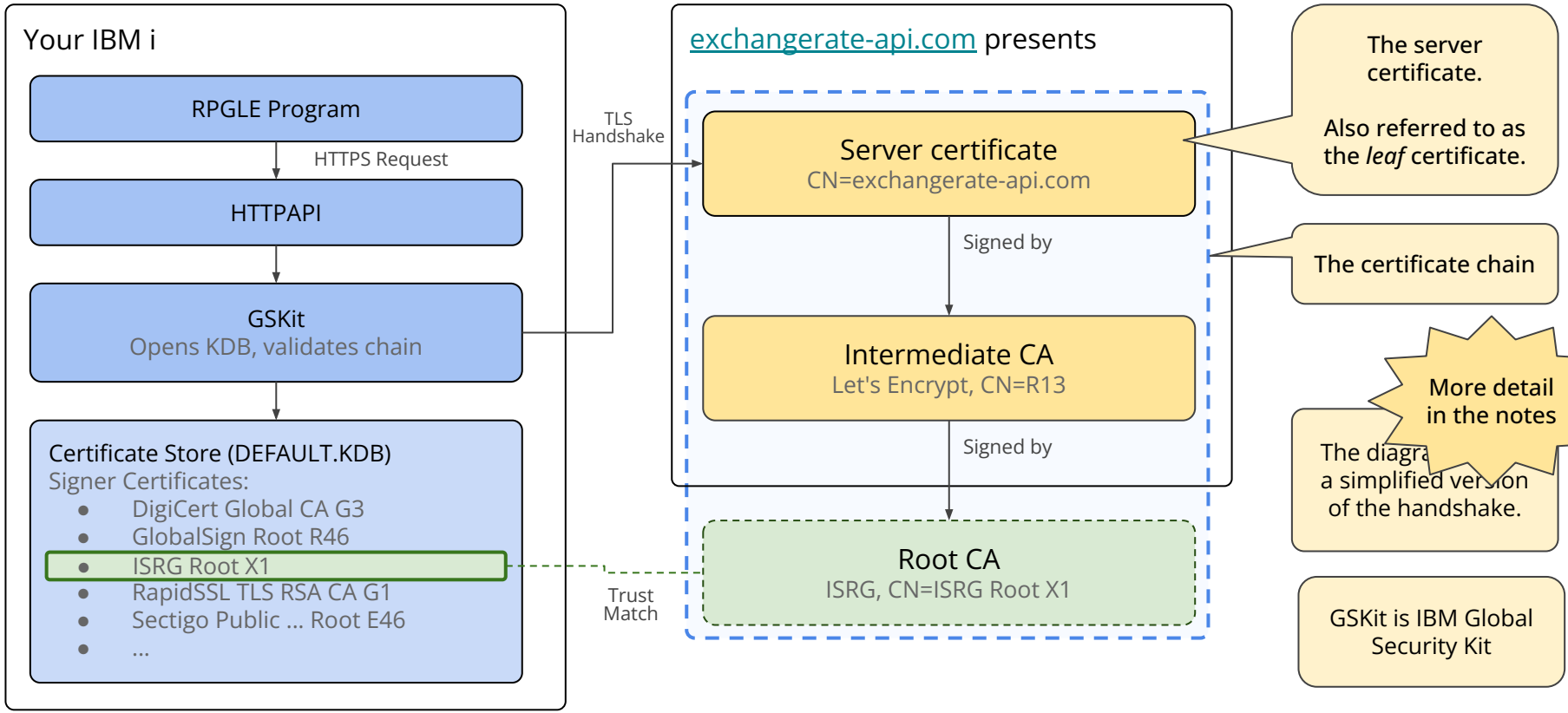
A certificate store holds the trusted certificates your IBM i uses for TLS.

HTTPAPI **can use it** to verify the server certificate, which proves you're talking to who you think you're talking to.

HTTPAPI & Certificate Store: TLS Handsha

Gotcha #2

How HTTPAPI verifies a server's identity



Demo 1 - Get Exchange Rate & Parse The Response

```
**free
ctl-opt dftactgrp(*no) actgrp(*new)
        bnddir('HTTPAPI':'YAJL');

/include QCPYSRC,HTTPAPI_H
/include QCPYSRC,YAJL_H
…
dcl-s currRate  packed(15:6);
dcl-s currAsAt  varchar(32);

url = 'https://open.er-api.com/v6/latest/GBP' ;
…
monitor;
 // one line: verb, URL, returns body as a string
 response = http string('GET':url);
 snd-msg ('OK, length = ' + %char(%len(response)));

 // Parse the response and extract the EUR rate
 if parse response(response:'EUR':currRate:currAsAt);
   snd-msg ('1 GBP = ' + %char(currRate) + ' EUR');
   snd-msg ('As at: ' + currAsAt + ' (UTC)');
 endif;
…
```

Bind to YAJLR4 too

/include yajl_h

Couple of new fields

Call a new procedure to parse

```
OK, length = 3031
1 GBP = 1.150985 EUR
As at: Fri, 15 May 2026 00:02:31 +0000 (UTC)
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()

I had to compress some of the information in the JSON message to fit it on the slide.

```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://…/docs/free",
   "terms of use": "https://…/terms",
   "time last update unix": 1778803351,
   "time last update utc": "Fri, 15 May 2026…",
   "time next update unix": 1778890781,
   "time next update utc": "Sat, 16 May 2026…",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
       "GBP": 1,
       …
       "EUR": 1.150985,
       …
       "USD": 1.345302,
       …
       "ZMW": 25.592189,
       "ZWG": 34.928,
       "ZWL": 34.928
   }
}
```

```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
);




// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');


eurNode = yajl_object_find(ratesNode :currCode);

// Extract the numeric value
currRate = yajl_get_number(eurNode);




// Always free the tree when finished
yajl_tree_free(docNode);
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()

```
{
    "result": "success",
    "provider": "https://www.exchangerate-api.com",
    "documentation": "https://…/docs/free",
    "terms of use": "https://…/terms",
    "time last update unix": 1778803351,
    "time last update utc": "Fri, 15 May 2026…",
    "time next update unix": 1778890781,
    "time next update utc": "Sat, 16 May 2026…",
    "time eol unix": 0,
    "base code": "GBP",
    "rates": {
        "GBP": 1,
        …
        "EUR": 1.150985,
        …
        "USD": 1.345302,
        …
        "ZMW": 25.592189,
        "ZWG": 34.928,
        "ZWL": 34.928
    }
}
```

```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
);




// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');


eurNode = yajl_object_find(ratesNode :currCode);

// Extract the numeric value
currRate = yajl_get_number(eurNode);




// Always free the tree when finished
yajl_tree_free(docNode);
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()

```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://…/docs/free",
   "terms of use": "https://…/terms",
   "time last update unix": 1778803351,
   "time last update utc": "Fri, 15 May 2026…",
   "time next update unix": 1778890781,
   "time next update utc": "Sat, 16 May 2026…",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
       "GBP": 1,
       …
       "EUR": 1.150985,
       …
       "USD": 1.345302,
       …
       "ZMW": 25.592189,
       "ZWG": 34.928,
       "ZWL": 34.928
   }
}
```

```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
);




// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');


eurNode = yajl_object_find(ratesNode :currCode);

// Extract the numeric value
currRate = yajl_get_number(eurNode);




// Always free the tree when finished
yajl_tree_free(docNode);
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()

```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://…/docs/free",
   "terms of use": "https://…/terms",
   "time last update unix": 1778803201,
   "time last update utc": "Fri, 15 May 2026…",
   "time next update unix": 1778889601,
   "time next update utc": "Sat, 16 May 2026…",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
       "GBP": 1,
       …
       "EUR": 1.150985,
       …
       "USD": 1.345302,
       …
       "ZMW": 25.592189,
       "ZWG": 34.928,
       "ZWL": 34.928
   }
}
```

```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
)



// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');


eurNode = yajl_object_find(ratesNode :currCode);

// Extract the numeric value
currRate = yajl_get_number(eurNode);



// Always free the tree when finished
yajl_tree_free(docNode);
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()

```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://…/docs/free",
   "terms of use": "https://…/terms",
   "time last update unix": 1778803351,
   "time last update utc": "Fri, 15 May 2026…",
   "time next update unix": 1778890781,
   "time next update utc": "Sat, 16 May 2026…",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
      "GBP": 1,
      …
      "EUR": 1.150985,
      …
      "USD": 1.345302,
      …
      "ZMW": 25.592189,
      "ZWG": 34.928,
      "ZWL": 34.928
   }
}
```

```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
);



// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');


eurNode = yajl_object_find(ratesNode :currCode);

// Extract the numeric value
currRate = yajl_get_number(eurNode);



// Always free the tree when finished
yajl_tree_free(docNode);
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()

```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://…/docs/free",
   "terms of use": "https://…/terms",
   "time last update unix": 1778803351,
   "time last update utc": "Fri, 15 May 2026…",
   "time next update unix": 1778890781,
   "time next update utc": "Sat, 16 May 2026…",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
       "GBP": 1,
       …
       "EUR": 1.150985,
       …
       "USD": 1.345302,
       …
       "ZMW": 25.592189,
       "ZWG": 34.928,
       "ZWL": 34.928
   }
}
```

```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
);




// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');

eurNode = yajl_object_find(ratesNode :currCode);

// Extract the numeric value
currRate = yajl_get_number(eurNode);



// Always free the tree when finished
yajl_tree_free(docNode);
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()


```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://…/docs/free",
   "terms of use": "https://…/terms",
   "time last update unix": 1778803351,
   "time last update utc": "Fri, 15 May 2026…",
   "time next update unix": 1778890781,
   "time next update utc": "Sat, 16 May 2026…",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
       "GBP": 1,
       …
       "EUR": 1.150985,
       …
       "USD": 1.345302,
       …
       "ZMW": 25.592189,
       "ZWG": 34.928,
       "ZWL": 34.928
   }
}
```


```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
);




// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');


eurNode = yajl_object_find(ratesNode :eurCode);

// Extract the numeric value
eurRate = yajl_get_number(eurNode);



// Always free the tree when finished
yajl_tree_free(docNode);
```

GETRATE5.pgm.rpgle

Demo 1 - parse_response()


```
{
   "result": "success",
   "provider": "https://www.exchangerate-api.com",
   "documentation": "https://…/docs/free",
   "terms of use": "https://…/terms",
   "time last update unix": 1778803351,
   "time last update utc": "Fri, 15 May 2026…",
   "time next update unix": 1778890781,
   "time next update utc": "Sat, 16 May 2026…",
   "time eol unix": 0,
   "base code": "GBP",
   "rates": {
       "GBP": 1,
       …
       "EUR": 1.150985,
       …
       "USD": 1.345302,
       …
       "ZMW": 25.592189,
       "ZWG": 34.928,
       "ZWL": 34.928
   }
}
```


```
// Parse the JSON in the response buffer into a YAJL tree
docNode = yajl_string_load_tree(response :errMsg);

// Optional: confirm the API reported success
result = yajl_get_string( yajl_object_find(docNode :'result') );


// Return the "as at" datetime string (optional)
currAsAt = yajl_get_string( yajl_object_find(docNode :'time_last_update_utc' )
);




// Drill down: docNode -> "rates" -> "EUR"
ratesNode = yajl_object_find(docNode :'rates');


eurNode = yajl_object_find(ratesNode :currCode);

// Extract the numeric value
currRate = yajl_get_number(eurNode);



// Always free the tree when finished
yajl_tree_free(docNode);
```

Now let's have a look at the full source.

GETRATE5.pgm.rpgle

Demo 1 - Get Exchange Rate: SQL & COBOL Equivalents

Source	What does it demonstrate?
`GETRATEC4.pgm.cblle`	Same as GETRATE4.pgm.rpgle but this time in ILE COBOL.
`GETRATES1.pgm.sqlrpgle`	Same as GETRATE1.pgm.rpgle but this time using QSYS2 SQL HTTP functions.
`GETRATES5.pgm.sqlrpgle`	Same as GETRATE5.pgm.rpgle using QSYS2 SQL HTTP functions.
`GETRATE7.pgm.rpgle`	Same as GETRATE5.pgm.rpgle but using DATA-INTO, comparable to GETRATES5.pgm.sqlrpgle

Demo 2 - Send SMS Text

The next example introduces a POST request to send an SMS text.

I'll be using a pay-as-you-go UK based service from thesmsworks.co.uk. It was simple to set up, receives good feedback and relatively cheap!

The API docs can be found here: api.thesmsworks.co.uk/docs/api-reference.html.

I'll be demonstrating the Send an SMS Message API.

This example introduces several new techniques.

- API authentication using a long-lived JWT
- Adding headers to the request
- Including body data in the request
- Performing a POST request
- Parsing the response with `data-into`

Good docs tell you the format of the message and what responses you need to handle.

Demo 2 - Send SMS Text

Step 1: Build the JSON request body

This is the JSON request body we need to build

```
{
 "sender": "RMSS",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl genOpen(*off);
yajl beginObj();
yajl addChar('sender'      : sender);
yajl addChar('destination' : destination);
yajl addChar('content'     : messageText);
yajl endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Declare some work fields and populate

I'm using hard-coded values and program described fields but these could easily be parameters passed into a standard utility program that sends an SMS text.

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 1: Build the JSON request body

This is the JSON request body we need to build

```
{
 "sender": "Richard Moulton",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl genOpen(*off);
yajl begInObj();
yajl addChar('sender'      : sender);
yajl addChar('destination' : destination);
yajl addChar('content'     : messageText);
yajl endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Initialise a new YAJL JSON generator

YAJL uses an in-memory buffer to which you add your JSON.

The `yajl_genOpen()` procedure includes two parameters, the first specifies whether the JSON is generated in a human readable format, we don't need that so pass `*off`.

The second parameter will escape forward slashes, which shouldn't be required and I've never used.

Returns `yajl_gen_status_ok` on success or `yajl_gen_open_fail` on error.

YAJL supports one active generator per job.

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

This is the JSON request body we need to build

```
{
 "sender": "Richard Moulton",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl_genOpen(*off);
yajl beginObj();
yajl addChar('sender'      : sender);
yajl addChar('destination' : destination);
yajl addChar('content'     : messageText);
yajl endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Step 1: Build the JSON request body

Open a JSON object in the generator buffer

The `yajl_beginObj()` has one optional parameter that allows you to specify the name of the object you've opening.

For example if you're opening an object called rates then `yajl_beginObj('rates')` would output `"address":{` to the buffer. In our case the root object doesn't have a name and so only the opening brace is output.

Returns `yajl_gen_status_ok` on success or a YAJL error code (e.g. `yajl_gen_in_error_state`).

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 1: Build the JSON request body

This is the JSON request body we need to build

```
{
 "sender": "Richard Moulton",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl genOpen(*off);
yajl beginObj();
yajl addChar('sender'      : sender);
yajl addChar('destination' : destination);
yajl addChar('content'     : messageText);
yajl endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Add an EBCDIC character string to the generator buffer

The `yajl_addChar()` has three parameters.

It's most frequently used as shown with a key-value pair.

The third parameter allows you to specify the value is null. If you used `yajl_addChar('sender':sender':*on)` then this would add `"sender":null` to the buffer.

You can also pass a one param e.g. `yajl_addChar('red')`, this could be used to add a value to an array.

Returns `yajl_gen_status_ok` on success, or an error code (e.g. `yajl_gen_keys_must_be_strings`).

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 1: Build the JSON request body

This is the JSON request body we need to build

```
{
 "sender": "Richard Moulton",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl genOpen(*off);
yajl beginObj();
yajl addChar('sender'      : sender);
yajl addChar('destination' : destination);
yajl addChar('content'     : messageText);
yajl endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Add an EBCDIC character string to the generator buffer

The same as the previous slide but notice `yajl_addChar()` adds EBCDIC data to the buffer.

If you have a UTF-16, or a UTF-8, string you can use the `yajl_addUtf16()` procedure.

If you provide a UTF-8 string, RPG will automatically handle the conversion to UTF-16.

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 1: Build the JSON request body

This is the JSON request body we need to build

```
{
 "sender": "Richard Moulton",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl_genOpen(*off);
yajl_beginObj();
yajl_addChar('sender'      : sender);
yajl_addChar('destination' : destination);
yajl_addChar('content'     : messageText);
yajl_endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Close a JSON object in the generator buffer

The `yajl_endObj()` has no parameters and closes the JSON object that was most recently opened.

Returns `yajl_gen_status_ok` on success, or an error code.

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 1: Build the JSON request body

This is the JSON request body we need to build

```
{
 "sender": "Richard Moulton",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl genOpen(*off);
yajl beginObj();
yajl addChar('sender'      : sender);
yajl addChar('destination' : destination);
yajl addChar('content'     : messageText);
yajl endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Return the contents of the generator buffer as a string

The `yajl_copyBufStr()` has no parameters and returns the JSON as a string, which will be empty on failure.

The YAJL buffer is UTF-8, if your string field is not then RPG will automatically handle the CCSID conversion.

The return value is limited to 2 million characters.

You need to execute this before `yajl_genClose()`.

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 1: Build the JSON request body

This is the JSON request body we need to build

```
{
 "sender": "Richard Moulton",
 "destination": "447588787623",
 "content": "Hello from RPG via The SMS Works"
}
```

```
dcl-s requestBody  varchar(2000);
dcl-s sender       varchar(15);
dcl-s destination  varchar(20);
dcl-s messageText  varchar(1000);

sender      = 'RMSS';
destination = '447588787623';
messageText = 'Hello from RPG via The SMS Works';

yajl_genOpen(*off);
yajl_beginObj();
yajl_addChar('sender'      : sender);
yajl_addChar('destination' : destination);
yajl_addChar('content'     : messageText);
yajl_endObj();
requestBody = yajl_copyBufStr();
yajl_genClose();
```

Close the YAJL JSON generator

No parameters and no return value.

Tears down the JSON generator that `yajl_genOpen()` created and frees the internal buffer that's been accumulating your JSON text.

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 2: Add authentication header

```
// Tell HTTPAPI to execute addHeaders() to add the
// authentication header
http_xproc(HTTP_POINT_ADDL_HEADER
           : %paddr(addHeaders)
           : %addr(smsJwt));
```

Register the procedure that will add the extra headers

This line is asking HTTPAPI to execute my `addHeaders()` procedure when it's ready to add the user defined headers to the HTTP request.

Additionally I'm asking HTTPAPI to pass `smsJwt` into my procedure, this third parameter is optional.

The first parameter identifies the exit point I'm registering, the `HTTP_POINT_ADDL_HEADER` exit point.

Only register the `HTTP_POINT_ADDL_HEADER` exit point once, HTTPAPI will only take the last one you register.

In my example I am only returning one but you can return multiple headers, which are key-value pairs.

Each header must terminate with a `CRLF`, which is a constant with a value of `x'0d25'`.

The maximum length of the returned headers value is 32767 and the second parameter is optional. You only need it if you registered the exit point with a data parameter.

My procedure to add the extra headers

```
dcl-proc addHeaders;
 dcl-pi *n;
   hdr varchar(32767);
   jwt char(512);
 end-pi;

  hdr = 'Authorization: ' + %trim(jwt) + CRLF;
end-proc;
```

Demo 2 - Send SMS Text

```
dcl-s url          varchar(256);
dcl-s requestBody  varchar(2000);
dcl-s responseBody varchar(8000);

url = 'https://api.thesmsworks.co.uk/v1/message/send';

responseBody = http_string('POST'
                           : url
                           : requestBody
                           : 'application/json');
```

As with the first example if the request was successful the response body string will have been returned - in the `responseBody` field.

If the request failed then an escape message will be sent.

Step 3: Perform the HTTP request

Perform the HTTP request

I'm using `http_string()` again but this time I'm performing a POST request.

The `requestBody` parameter contains the JSON message we saw earlier.

The fourth parameter is the content type and in this example this tells the server that the data we're sending is JSON. This parameter sets the `Content-Type` header in the request.

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 4: Parse the response body

This is the JSON response body we receive

```
{
   "messageid": "4789996520527950423340",
   "status": "SENT",
   "credits": 219,
   "creditsUsed": 1,
   "messageparts": 1,
   "metadata": {}
}
```

Parse the response data using the `data-into` opcode

Using the `data-into` opcode works really well for simple response bodies such as this one.

I still prefer to use the previous technique for parsing more complex JSON such as those with nested levels, arrays or if I need to convert data.

```
dcl-ds smsResponse qualified;
 messageid    varchar(50);
 status       varchar(20);
 credits      packed(7);
 creditsUsed  packed(7);
 messageparts packed(3);
end-ds;

data-into smsResponse
          %data(responseBody 'case=any allowmissing=yes allowextra=yes')
          %parser('YAJLINTO');
```

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 4: Parse the response body

This is the JSON response body we receive


```
{
   "messageid": "4789996520527950423340",
   "status": "SENT",
   "credits": 219,
   "creditsUsed": 1,
   "messageparts": 1,
   "metadata": {}
}
```


The parsed data will be placed in to *smsResponse*

In this examples *smsReponse* is a qualified data structure, which is how I would define it, but it could also be an unqualified data structure.

```
dcl-ds smsResponse qualified inz;
 messageid    varchar(50);
 status       varchar(20);
 credits      packed(7);
 creditsused  packed(7);
 messageparts packed(3);
end-ds;

data-into smsResponse
          %data(responseBody 'case=any allowmissing=yes allowextra=yes')
          %parser('YAJLINTO');
```

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 4: Parse the response body

This is the JSON response body we receive

This is the data to be parsed, the response body

Notice the data options I've specified here.

Without `allowextra=yes` the runtime would error as my data structure doesn't include `metadata`.

The `case` option specifies the expected case of the JSON document element names. I'm using `case=any` as the element names are mixed case.

```
{
   "messageid": "4789996520527950423340",
   "status": "SENT",
   "credits": 219,
   "creditsUsed": 1,
   "messageparts": 1,
   "metadata": {}
}
```

Link in the session notes for more details on the options

```
dcl-ds smsResponse qualified inz;
 messageid    varchar(50);
 status       varchar(20);
 credits      packed(7);
 creditsused  packed(7);
 messageparts packed(3);
end-ds;

data-into smsResponse
   %data(responseBody 'case=any allowmissing=yes allowextra=yes')
   %parser('JSONLEXIS');
```

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 4: Parse the response body

This is the JSON response body we receive


```
{
   "messageid": "4789996520527950423340",
   "status": "SENT",
   "credits": 219,
   "creditsUsed": 1,
   "messageparts": 1,
   "metadata": {}
}
```


The JSON data will be parsed using YAJLINTO

`YAJLINTO` is an excellent RPGLE program included as part of the YAJL installation.

This needs to exist in your library list or you can qualify it.

```
dcl-ds smsResponse qualified inz;
 messageid    varchar(50);
 status       varchar(20);
 credits      packed(7);
 creditsused  packed(7);
 messageparts packed(3);
end-ds;

data-into smsResponse
          %data(responseBody :case=any allowmissing=yes allowextra=yes')
          %parser('YAJLINTO');
```

SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text

Step 4: Parse the response body

What will the response look like if the request fails?

This is the response on success


```
{
   "messageid": "4789996520527950423340",
   "status": "SENT",
   "credits": 219,
   "creditsUsed": 1,
   "messageparts": 1,
   "metadata": {}
}
```


You also need to cope with a failure response

This is the response when the request failed. In this example the HTTP status in the response was 400 Bad Request.


```
{
   "status": "error",
   "errorCode": 4002,
   "message": "Message must be between 1 and 1280 characters in length.,"
   "permanent": true
}
```


SENDSMS1.pgm.rpgle

Demo 2 - Send SMS Text: Other HTTPAPI Options

Source	What does it demonstrate?
`SENDSMS1.pgm.rpgle`	The example we've just looked at
`SENDSMS2.pgm.rpgle`	Same as SENDSMS1.pgm.rpgle but this time using `http_stmf()` and stream files
`SENDSMS3.pgm.rpgle`	Same as SENDSMS1.pgm.rpgle but after the send message I get the message details
`SENDSMS4.pgm.rpgle`	Same as SENDSMS1.pgm.rpgle but using `http_setAuth()`

Demo 3 - Get Shipping Rates

The next example introduces API key authentication to obtain a shipping quote.

I'll be using a trial service from shipengine.com. It was also simple to set up and offers a sandbox when developing and testing integration. The trial sandbox is linked to US carriers, you need to subscribe to link to UK carriers.

The API docs can be found here: shipengine.github.io/shipengine-openapi.

I'll be demonstrating the Get Shipping Rates API but you could take this example and extend it to: Get Rate → Buy Label → Download Label

This example introduces several new techniques.

- API key authentication
- Nested JSON generation
- Nested JSON parsing

shipengine.com is becoming shipstation.com

Demo 3 - Get Shipping Rates

```
{
 "rate options": {
   "carrier_ids": ["se-385109","se-385110"]
 },
 "shipment": {
   "ship to": {
     "name": "Amanda Miller",
     "address line1": "525 S Winchester Blvd",
     "city locality": "San Jose",
     "state province": "CA",
     "postal code": "95128",
     "country_code": "US"
   },
   "ship from": {
     "name": "Acme Corp",
     "phone": "512-555-5555",
     "address line1": "4009 Marathon Blvd",
     "city locality": "Austin",
     "state province": "TX",
     "postal code": "78756",
     "country_code": "US"
   },
   "packages": [
     {
       "weight": {
         "value": 1.0,
         "unit": "pound"
       }
     }
   ]
 }
}
```

Step 1: Build the JSON request body

This is the JSON request body we need to build

The request body is more complex than we've seen in previous examples.

- Multiple nested objects
- Scalar array
- Object array

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates

Step 1: Build the JSON request body

Multiple nested objects

```
yajl_beginObj();

  yajl_beginObj('rate_options');
    …
  yajl_endObj();

  yajl_beginObj('shipment');

    yajl_beginObj('ship_to');
      …
    yajl_endObj();

    yajl_beginObj('ship_from');
    …
    yajl_endObj();
    …
  yajl_endObj();

yajl_endObj();
```

Demo 3 - Get Shipping Rates

Step 1: Build the JSON request body

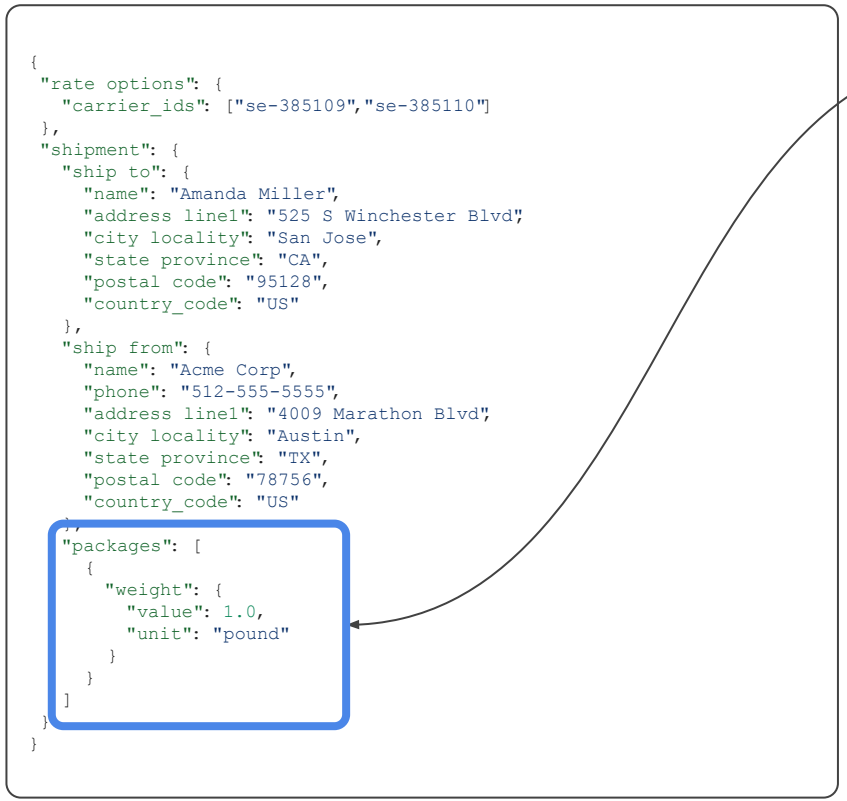
Scalar array

```
{
 "rate options": {
    "carrier_ids": ["se-385109","se-385110"]
 },
 "shipment": {
   "ship to": {
     "name": "Amanda Miller",
     "address line1": "525 S Winchester Blvd",
     "city locality": "San Jose",
     "state province": "CA",
     "postal code": "95128",
     "country_code": "US"
   },
   "ship from": {
     "name": "Acme Corp",
     "phone": "512-555-5555",
     "address line1": "4009 Marathon Blvd",
     "city locality": "Austin",
     "state province": "TX",
     "postal code": "78756",
     "country_code": "US"
   },
   "packages": [
     {
       "weight": {
         "value": 1.0,
         "unit": "pound"
       }
     }
   ]
 }
}
```

```
...
yajl beginObj('rate options');
  yajl beginArray('carrier ids');
    yajl addChar(%trim(carrierId1));
    yajl addChar(%trim(carrierId2));
  yajl endArray();
yajl_endObj();
...
```

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates



```
{
 "rate options": {
   "carrier_ids": ["se-385109","se-385110"]
 },
 "shipment": {
   "ship to": {
     "name": "Amanda Miller",
     "address line1": "525 S Winchester Blvd",
     "city locality": "San Jose",
     "state province": "CA",
     "postal code": "95128",
     "country_code": "US"
   },
   "ship from": {
     "name": "Acme Corp",
     "phone": "512-555-5555",
     "address line1": "4009 Marathon Blvd",
     "city locality": "Austin",
     "state province": "TX",
     "postal code": "78756",
     "country_code": "US"
   },
   "packages": [
     {
       "weight": {
         "value": 1.0,
         "unit": "pound"
       }
     }
   ]
 }
}
```

Step 1: Build the JSON request body

Object array

```
…
yajl beginArray('packages');
  yajl beginObj();
    yajl beginObj('weight');
      yajl addNum('value' : '1.0');
      yajl addChar('unit' : 'pound');
    yajl endObj();
  yajl endObj();
yajl_endArray();
…
```

Here I am generating an array of packages.

Each package within the array is an object (curly braces).

In this example I am only providing one hard-coded package but there could be many. This package object only contains a weight object but there are other non-required items such as dimensions that are not specified.

Demo 3 - Get Shipping Rates

Step 2: Add authentication header

```
// Tell HTTPAPI to execute addHeaders() to add the
// authentication header
http_xproc(HTTP_POINT_ADDL_HEADER
          : %paddr(addHeaders)
          : %addr(seApiKey));
```

Register the procedure that will add the extra headers

Essentially the same as Demo 2.

The only difference is that we're now dealing with an API key rather than a JWT.

My procedure to add the extra headers

```
dcl-proc addHeaders;
 dcl-pi *n;
   hdr    varchar(32767);
   apiKey char(64);
 end-pi;

 hdr = 'API-Key: ' + %trim(apiKey) + CRLF;
end-proc;
```

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates

Step 3: Perform the HTTP request

```
dcl-s url          varchar(256);
dcl-s requestBody  varchar(4000);
dcl-s responseBody varchar(65535);

url = 'https://api.shipengine.com/v1/rates';

responseBody = http_string('POST'
                           : url
                           : requestBody
                           : 'application/json');
```

Perform the HTTP request

Again, essentially the same as Demo 2.

Slight tweaks to the field lengths to cope with a larger request and response body.

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates


```
{
  "rate_response": {
    "rates": [
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx First Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 141.55
        }
      },
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx Priority Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 110.55
        }
      },
      …
    ]
  }
}
```


Step 4: Parse the response body

This is the JSON response body we need to parse

The response body for this example is large, nearly 800 lines of formatted JSON.

I'm only showing a small segment of the full message - just the first few service rates and only the specific properties I'm interested in parsing.

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates


```
{
  "rate_response": {
    "rates": [
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx First Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 141.55
        }
      },
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx Priority Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 110.55
        }
      },
      …
    ]
  }
}
```


Step 4: Parse the response body

These are the data structures that map to the JSON

```
dcl-ds rateResponse qualified inz;
 dcl-ds rate response;
   numRates int(10);
   rates likeds(shipRate) dim(50);
 end-ds;
end-ds;

dcl-ds shipRate qualified template;
 carrier friendly name varchar(50);
 service type          varchar(100);
 delivery days         int(5);
 dcl-ds shipping amount;
   currency varchar(10);
   amount   packed(11:2);
 end-ds;
end-ds;
```

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates

Step 4: Parse the response body

These are the data structures that map to the JSON

```
{
  "rate_response": {
    "rates": [
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx First Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 141.55
        }
      },
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx Priority Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 110.55
        }
      },
      …
    ]
  }
}
```

```
dcl-ds rateResponse qualified inz;
 dcl-ds rate response;
   numRates int(10);
   rates likeds(shipRate) dim(50);
 end-ds;
end-ds;

dcl-ds shipRate qualified template;
 carrier friendly name varchar(50);
 service type          varchar(100);
 delivery days         int(5);
 dcl-ds shipping amount;
   currency varchar(10);
   amount   packed(11:2);
 end-ds;
end-ds;
```

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates

Step 4: Parse the response body

The `rates` array data structure maps to the object array

```
{
  "rate_response": {
    "rates": [
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx First Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 141.55
        }
      },
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx Priority Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 110.55
        }
      },
      …
    ]
  }
}
```

```
dcl-ds rateResponse qualified inz;
 dcl-ds rate response;
  numRates int(10);
  rates likeds(shipRate) dim(50);
 end-ds;
end-ds;

dcl-ds shipRate qualified template;
 carrier friendly name varchar(50);
 service type          varchar(100);
 delivery days         int(5);
 dcl-ds shipping amount;
   currency varchar(10);
   amount   packed(11:2);
 end-ds;
end-ds;
```

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates

```
{
  "rate_response": {
    "rates": [
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx First Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 141.55
        }
      },
      {
        "carrier_friendly_name": "FedEx",
        "service_type": "FedEx Priority Overnight®",
        "delivery_days": 1,
        "shipping_amount": {
          "currency": "usd",
          "amount": 110.55
        }
      },
      …
    ]
  }
}
```

Step 4: Parse the response body

The `shipRate` data structure maps nicely to our data

```
dcl-ds rateResponse qualified inz;
 dcl-ds rate response;
   numRates int(10);
   rates likeds(shipRate) dim(50);
 end-ds;
end-ds;

dcl-ds shipRate qualified template;
carrier friendly name varchar(50);
service type          varchar(100);
delivery days         int(5);
dcl-ds shipping amount;
  currency varchar(10);
  amount   packed(11:2);
end-ds;
end-ds;
```

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates

We're using the `data-into` opcode again to parse the JSON response with one new option `countprefix`.

This specifies the prefix used by data-into to record the number of array elements that were populated.

In this example my `rates` data structure is populated and the number of array elements populated is set in `numRates`.

You don't to have specify `numRates`, the parser would still work, you just wouldn't be sure how many elements were parsed.

Step 4: Parse the response body

These are the data structures that map to the JSON

```
dcl-ds rateResponse qualified inz;
 dcl-ds rate response;
   numRates int(10);
   rates likeds(shipRate) dim(50);
 end-ds;
end-ds;

dcl-ds shipRate qualified template;
 carrier friendly name varchar(50);
 service type          varchar(100);
 delivery days         int(5);
 dcl-ds shipping amount;
   currency varchar(10);
   amount   packed(11:2);
 end-ds;
end-ds;
```

```
data-into rateResponse
          %data(responseBody:'case=any allowmissing=yes allowextra=yes countprefix=num')
          %parser('YAJLINTO');
```

GETSHIP1.pgm.rpgle

Demo 3 - Get Shipping Rates

```
4 > call getship1
    Demo directory: /home/RICHARDM/
    Returned 8 rate(s):
    1. FedEx - FedEx First Overnight® (usd 141.55) - 2 day(s)
    2. FedEx - FedEx Priority Overnight® (usd 110.55) - 2 day(s)
    3. FedEx - FedEx Standard Overnight® (usd 95.41) - 2 day(s)
    4. FedEx - FedEx 2Day® A.M. (usd 52.05) - 3 day(s)
    5. FedEx - FedEx 2Day® (usd 46.04) - 3 day(s)
    6. FedEx - FedEx Express Saver® (usd 37.51) - 4 day(s)
    7. FedEx - FedEx Ground® (usd 14.75) - 3 day(s)
    8. FedEx - FedEx Home Delivery® (usd 14.75) - 3 day(s)
                                                              Bottom
Type command, press Enter.
===>
```

Now let's have a look at the full source.

GETSHIP1.pgm.rpgle

Some Gotchas

Gotcha #1 - HTTPAPI

You run your application and when it attempts to make a HTTP request you get an error on the HTTPAPI procedure call and see the following errors in the joblog.

```
(GSKit) Access to the key database is not allowed.
gsk_env_init: (GSKit) Access to the key database is not allowed.
```

One of the first gotchas I hit when installing HTTPAPI for the first time.

HTTPAPI needs a certificate store, if you don't specify one then it assumes *SYSTEM.

This error means that the certificate store hasn't been created or your user doesn't have the necessary permission to read the certificate store file.

Checkout the Extras: Certificate Stores set of slides.

These slides describe how to create a certificate store and set the appropriate permissions.

Gotcha #2 (Part 1) - HTTPAPI

A two part gotcha!

If you don't use `https_strict(*on)` you open yourself up to MITM attacks!

Part 1 - Default Handling

From Scott's README:

> *"By default, HTTPAPI does not attempt to verify the identity of the HTTP server you are communicating with. This is turned off because many organizations use SSL with 'self-signed' certificates, and this makes things much simpler for them.*
>
> *However, this essentially disables the identity checking that is meant to be done with SSL. Therefore, you may wish to perform stricter checking within HTTPAPI. You can do this by calling the https_strict() function prior to using SSL in your application:*
>
> *- https_strict(*ON);"*

It's easy to turn on the strict server identity logic, simply include the following line before using HTTPS.

```
https_strict(*on);
```

By default, HTTPAPI will encrypt your traffic (when using HTTPS) so you're safe against a passive eavesdropper but the identity of the server won't be verified so you're not safe from an active man-in-the-middle attacker presenting a forged certificate.

Gotcha #2 (Part 2) - HTTPAPI

A two part gotcha!

Part 2 - DCM Requires CA and Root Certificates

You've turned on HTTPS strict server identity logic you run your application and when it attempts to make a HTTPS request you get an error on the HTTPAPI procedure call and see the following errors in the joblog.

```
(GSKit) Certificate is not signed by a trusted certificate authority.
SSL Handshake: (GSKit) Certificate is not signed by a trusted certificate author
```

This most likely means that your certificate store doesn't hold the root CA certificate to complete the certificate chain.

In DCM there is an option to "Populate with CAs" this can be used to import some of the most popular root CA certificates. If the missing one isn't listed then you can import it manually.

Warning - if you're manually importing then make sure you trust the source!

This gotcha is true for HTTPAPI, ILEVator and SQL QSYS2 HTTP functions - they all use GSKit!

Checkout the TLS Handshake slide notes for more details.

Checkout the Import CA Certificate to DCM slide.

Gotcha #3 - YAJL

YAJL provides two procedures to add a numeric field to a JSON generator buffer.

`yajl_addNum(node, str)` - this expects a string presentation of a number

`yajl_addNumf(node, num)` - this expects a numeric value

Be careful when using `yajl_addNum()` as can create an invalid JSON numeric value. The JSON number grammar doesn't allow a number to begin with a decimal point.

Valid JSON: `{ "value": 0.5 }` Invalid JSON: `{ "value": .5 }`

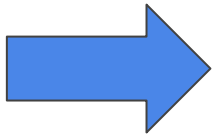
It's pretty easy to solve!

If you're using the `yajl_addNum()` procedure then ensure you include the following control specification.

```
// YAJL - numbers in JSON must start with a digit
ctl-opt decedit('0.');
```

With this in place `%char(shipAmount)` would generate `"shippingAmount":0.00`.

```
dcl-s shipAmount packed(11:2) inz(0);
dcl-s invAmount  packed(11:2) inz(1234.56);

yajl genOpen(*off);
yajl beginObj();
  yajl addNum ('shippingAmount':%char(shipAmount));
  yajl addNumf('invoiceAmount' :invAmount);
yajl endObj();
json = yajl copyBufStr();
yajl_genClose();
```



```
{
 "shippingAmount":.00,
 "invoiceAmount": 1234.56
}
```

Gotcha #4 - JSON, Dates & Times

What timezone?

When communicating with REST APIs you need to be aware of the timezone.

It's common practice for REST APIs to expect dates and times to be provided in UTC and also provide them in UTC. The REST API docs should specify.

What format?

JSON dates and times are in string format, there isn't a specific JSON date or time type.

It's common practice for REST APIs to expect dates and times to be ISO format. The REST API docs should specify.

- Dates in "YYYY-MM-DD" format e.g. "2026-05-22"
- Times in "HH:MM:SS.mmm" format e.g. "16:54:12", "16:54:12.123"
- Timestamps in "YYYY-MM-DDTHH:MM:DDZ" format e.g. "2026-05-23T16:54:12Z"

Gotcha #5 - HTTPAPI Settings Persist

The HTTPAPIR4 service program is compiled with ACTGRP(*CALLER).

The settings you enable will persist across requests within the same activation group.

Examples of this include:

- `http_setOption()` - ccsid, debug, timeout, etc
- `http_debug()` - turn debug log on/off
- `http_setAuth()` - authentication settings
- `http_use_cookies()`
- `http_setProxy()`
- `https_strict()`
- `https_init()`
- `https_certStore()` - this procedure has a `*clear` option

You could use a copybook to define your settings or create a procedure that initializes all of the settings you use in your application.

Beyond The Basics

HTTPAPI & YAJL Installation

You don’t need the entire HTTPAPI or YAJL libraries in the library list. You can install a small subset of copybooks and objects to compile and runtime.

Compilation

- HTTPAPI
  - CONFIG_H copybook
  - HTTPAPI_H copybook
  - HTTPAPI *BNDDIR
  - HTTPAPIR4 *SRVPGM
- YAJL
  - YAJL_H copybook
  - YAJL *BNDDIR
  - YAJLR4 *SRVPGM

Runtime

- HTTPAPI
  - EXPAT *SRVPGM
  - HTTPAPIR4 *SRVPGM
- YAJL
  - YAJL *SRVPGM
  - YAJLR4 *SRVPGM

Uploading Files: HTTPAPI's Multipart Encoder

When JSON isn't enough!

- Some APIs don't want a JSON body - they want a file upload: `multipart/form-data`
- The RPGLE equivalent of curl's -F file=@something
- Body is split into boundary-delimited **parts** - each with its own headers and content type, so you can mix files and text fields in one request
- Real-world targets: Jira attachments, document management, AI services taking an image/audio file
- **The gap it fills:** the QSYS2 SQL HTTP functions have no real story for file uploads - this is a genuine HTTPAPI advantage

See HTTPAPI's EXAMPLE7 for full working source

Encoder lifecycle

```
// open() RETURNS the content-type
// (incl. boundary) in ContentType
enc = http_mfd_encoder_open( tempFile  : ContentType );

// Add ordinary fields
http_mfd_encoder_addvar_s( enc  : 'operation'    : 'VERIFY'
);
http_mfd_encoder_addvar_s( enc  : 'data_format'  : 'PDF' );

// Add the file (name = the form field)
http mfd encoder addstmf( enc  : 'handout'
                        : '/tmp/handout.pdf'
                        : 'application/octet-stream'  );

// Finish the body
http_mfd_encoder_close( enc );

// Post it - pass ContentType from open()
rc = http url post stmf( url  : tempFile
                       : '/tmp/result.txt'
                       : HTTP TIMEOUT
                      :  HTTP USERAGENT
                       : ContentType );
```

Handle UTF-8

Parsing UTF-8 character data:
see example UTF8DATA3.pgm.sqlrpgle.

Generating UTF-8 character data:
see example UTF8DATA4.pgm.sqlrpgle.

Encoding boundaries

- HTTPAPI moves bytes.
- JSON over HTTP is UTF-8 (CCSID 1208).
- YAJL is a UTF-8 parser/generator at heart, so the data it touches lives most naturally as 1208.
- By default, RPG will treat character data as EBCDIC in your job CCSID.
- The character data will be converted when assigned across CCSIDs.

Keep the JSON document in UTF-8.

Let the EBCDIC conversion happen on individual values and be deliberate about it.

```
// Define our UTF-8 string field
// - CCSID 1208 = UTF-8
dcl-s country_name varchar(40) ccsid(1208);

// Retrieve country name in job CCSID
country_name = yajl_get_string(nameNode);

// Retrieve country name in UTF-8
// - YAJLR4 returns Unicode character data in
//   UTF-16, RPG translates to UTF-8
country_name = yajl_get_string_utf16(nameNode);
```

Note that the `country_name` fields is defined here as `varchar(40)` but this means 40 bytes not 40 characters. Each UTF-8 can be between 1 and 4 bytes long so this field could fit anything from 10 to 40 UTF-8 characters.

Handle UTF-8 & YAJL

Encoding boundaries

- HTTPAPI moves bytes.
- JSON over HTTP is UTF-8 (CCSID 1208).
- YAJL is a UTF-8 parser/generator at heart, so the data it touches lives most naturally as 1208.
- By default, RPG will treat character data as EBCDIC in your job CCSID.
- The character data will be converted when assigned across CCSIDs.

Keep the JSON document in UTF-8.

Let the EBCDIC conversion happen on individual values and be deliberate about it.

Parsing JSON with UTF-8 data:
see example UTF8DATA3.pgm.sqlrpgle.

```
// Define our UTF-8 string field
// - CCSID 1208 = UTF-8
dcl-s country_name varchar(40) ccsid(1208);

// Retrieve country name in job CCSID
country_name = yajl_get_string(nameNode);

// Retrieve country name in UTF-8
// - YAJLR4 returns Unicode character data in
//   UTF-16, RPG translates to UTF-8
country_name = yajl_get_string_utf16(nameNode);
```

Generating JSON with UTF-8 data:
see example UTF8DATA4.pgm.sqlrpgle.

```
// Define our UTF-8 string field
// - CCSID 1208 = UTF-8
dcl-s country_name varchar(40) ccsid(1208);

// Add country name in job CCSID
yajl_addChar('countryName':country_name);

// Add country name in UTF-8
yajl_addUtf16('countryName':country_name);
```

UTF-8 & HTTPAPI

In the `COMMON_C.rpgle` copybook I set two HTTPAPI options.

```
// Set the CCSID for the data sent over the network
// - JSON is always UTF-8
http_setOption('network-ccsid':'1208');

// Set the CCSID for files created in the IFS
// - JSON is always UTF-8
http_setOption('file-ccsid':'1208');
```

If you don't add this then the default CCSID used by HTTPAPI is 819 (Latin-1).

Without the `COMMON_C` CCSID settings you would see character translation issues, for example: `FedEx - FedEx Priority OvernightÂ®` rather than `FedEx - FedEx Priority Overnight®`.

HTTPAPI Options

You can set various options to control behaviour of HTTPAPI using `http_setOption(opt:value)`. Some of the common ones you' ll use:

Source	What does it demonstrate?
`accept`	Media types that you are willing to accept in response to an HTTP request e.g. “application/json”.
`content-type`	When uploading a stream in a POST or PUT request, specifies the data type you're sending e.g. “application/json”.
`debug-level`	Level of detail written to the debug/trace file.
`file-ccsid`	When a new file is created in the IFS, HTTPAPI will assign this CCSID.
`network-ccsid`	CCSID to to use for the data sent over the network to remote sites.
`timeout`	How many seconds pass without any network activity, the request is aborted.

http_setAuth()

The examples I've shown used the `http_xproc()` procedure to add an `Authorization` header to the request. You can also use the http_setAuth() procedure to achieve the same thing.

Rather than this:

```
http_xproc(HTTP_POINT_ADDL_HEADER:%paddr(addHeaders):%addr(smsJwt));
```

You can use this:

```
http_setAuth(HTTP_AUTH_USRDFN:'JWT':%trim(smsJwt));
```

The first parameter to http_setAuth() describes the type of authentication.

- HTTP_AUTH_BASIC
- HTTP_AUTH_BEARER
- HTTP_AUTH_MD5_DIGEST
- HTTP_AUTH_NONE
- HTTP_AUTH_NTLM
- HTTP_AUTH_USRDFN

```
http_setAuth(HTTP_AUTH_BASIC:user:password);
http_setAuth(HTTP_AUTH_BEARER:'':token);
http_setAuth(HTTP_AUTH_MD5_DIGEST:user:password);
http_setAuth(HTTP_AUTH_NONE:'':'');
http_setAuth(HTTP_AUTH_NTLM:user:password);
http_setAuth(HTTP_AUTH_USRDFN:'JWT':%trim(smsJwt));
```

Redirects (3xx responses)

A redirect is the server saying "what you want lives somewhere else" via a HTTP status code 301, etc, plus a `Location` header pointing at the new URL.

HTTPAPI does not follow redirects automatically, because auto-following a redirect can silently change your HTTP method, drop your auth headers, or send you to a different host.

The API docs should tell if you need to deal with redirects.

The simple approach is to check the return code. After a call, inspect `http_error()` status code, if you get a 3xx, either use the `http_redir_loc()` procedure or read the `Location` response header and re-issue the request against that URL. You could even wrap this in a small loop with a max-redirect counter (5 is a sane limit) to avoid redirect loops.

Proxies

If your IBM i sits behind a corporate forward proxy for outbound internet, your direct HTTPS calls will just hang or fail to connect until you route them through your proxy.

HTTPAPI supports this via the `http_setproxy()` procedure, where you supply the proxy host and port, and credentials if it requires authentication.

For an HTTPS target through a proxy, the mechanism is HTTP CONNECT tunneling - the proxy establishes a raw tunnel and the TLS handshake happens end-to-end through it, so your certificate validation still works against the real server, not the proxy.

The thing to clarify with your network team before you write a line of code: is there a proxy at all on the path to your target, does it require authentication, and does it do TLS interception (some corporate proxies terminate TLS and re-sign with an internal CA, which means you'd need that CA cert in your DCM truststore). Knowing those three answers up front saves a lot of "why does curl work from my laptop but not from the IBM i" debugging.

Cookies

A cookie is a small piece of state the server asks the client to store and send back on later requests to the same host. HTTP itself is stateless, so cookies are a standard mechanism for carrying state (sessions, preferences, tokens) across requests.

REST favours stateless requests - each call should carry what the server needs to process it.

Token-based schemes (e.g. an Authorization: Bearer header) are common precisely because they keep each request self-describing.

Cookies can still appear in REST APIs - most often for session affinity, CSRF tokens, or because a gateway / auth layer sets them.

Cookies

How HTTPAPI handles cookies

Cookie handling is automatic and on by default. When a host sends a cookie, HTTPAPI remembers it and resends it on subsequent requests to that same host (until it expires).

`http_use_cookies()`

Turns HTTPAPI's cookie parsing and caching on or off.

```
http_use_cookies(*on);    // read + send (the default)
http_use_cookies(*off);   // ignore
```

When a host sends a cookie, HTTPAPI automatically resends it on the next request to that same host (until it expires), and by default it only remembers cookies until the activation group ends.

`http_cookie_file()`

Sets an IFS file so cookies persist beyond the activation group.

```
http_cookie_file( '/home/richard/cookies.txt' : *on );
```

HTTPAPI vs QSYS2 SQL HTTP functions

HTTPAPI + YAJL

Reach for HTTPAPI when: dynamic token lifecycles (OAuth refresh), file uploads, granular error/debug control, deeply nested JSON

Pros

- Streaming - no 2 GB ceiling
- Detailed debugging / trace logs
- Granular error reporting (HTTPAPI error number and HTTP status)
- Built-in multipart/form-data, configurable cookies, redirects, proxies
- Full range of auth schemes, including user-defined headers via exit point
- YAJL gives fine-grained control navigating complex/nested JSON
- YAJL DATA-INTO & DATA-GEN handlers

Cons

- Extra deployment required
- Can be more code
- Two new libraries to learn
- ILE (RPG, etc) specific
- **Server identity validation is off by default**

QSYS2 SQL HTTP + JSON functions

Pros

- Nothing to install - JVM-free, lower overhead than older SYSTOOLS
- Can be less code
- Same DCM-managed certificates
- It's SQL - no new library to learn
- Built-in auth, proxy, redirect & SSL options
- More portable to Node, Python, etc - SQL syntax

Reach for QSYS2 when: static auth (basic / fixed bearer / API key), simple request, straightforward JSON

Cons

- 2 GB CLOB limit
- No real handling for multipart/form-data file uploads
- Thinner debugging
- Static auth only - dynamic/multi-step flows (e.g. OAuth token refresh) are on you
- SQL/JSON-path syntax is awkward for dynamic navigation

Useful Resources

Useful Resources

Scott Klement Related

- Scott's Forums - Including HTTPAPI and YAJL
- Scott Klement: Options for Consuming REST APIs from RPG
- Scott Klement: SSL Sockets From RPG? Of Course You Can! - Some great examples on secure sockets in RPG

REST and HTTP Related

- REST API Tutorial: What is REST?
- Mozilla: HTTP response status codes

QSYS2 SQL HTTP Functions

- TechChannel: Using the New SQL HTTP Functions (Part 1 of 2)
- TechChannel: Using the New SQL HTTP Functions (Part 2)

Certificates

- Seiden Group: How to Set Up the IBM i Open Source Environment
- Seiden Group: How to Validate Self-Signed SSL / TLS Certificates from IBM i

Q&A

Extras: Further Examples

Extras: Further Examples

Source	What does it demonstrate?
`HTTPERROR1.pgm.rpgle`	Show how to capture HTTPAPI error information.
`JSONNUMS1.pgm.rpgle`	Illustrates the Gotcha #3 YAJL JSON numbers issue.
`UTF8DATA1.pgm.rpgle`	Retrieve JSON UTF-8 data using HTTPAPI - into a IFS file.
`UTF8DATA2.pgm.rpgle`	Parse JSON UTF-8 data using an RPG data structure created by YAJLGEN.
`UTF8DATA3.pgm.rpgle`	Parse JSON UTF-8 data using and output to a table.
`UTF8DATA4.pgm.rpgle`	Generate JSON UTF-8 data from a table.
`UTF8DATAS3.pgm.rpgle`	Equivalent to UTF8DATA3.pgm.rpgle but with SQL parsing.

HTTPAPI: http_error()

```
...
dcl-s response  varchar(65535);
dcl-s url       varchar(256);
dcl-s errMsg    char(80);
dcl-s errNo     int(10);
dcl-s httpCode  int(10);

// Force an error by using an invalid URL
url = 'https://open.er-api.com/v6';

monitor;
 // one line: verb, URL, returns body as a string
 response = http string('GET':url);
 snd-msg ('OK, length = ' + %char(%len(response)));
 snd-msg ('Response: ' + %trim(response));
on-error;
 errMsg = http error(errNo:httpCode);
 snd-msg ('HTTP Error: '   + %trim(errMsg));
 snd-msg ('Error Number: ' + %char(errNo));
 snd-msg ('HTTP Status: '  + %char(httpCode));
endmon;

*inlr = *on;
```

An adapted version of GETRATE1.pgm.rpgle to illustrate the error information you can get back from HTTPAPI.

errNo: HTTPAPI's own internal error number. The same number you see in the debug logs. The HTTPAPI_H copybook includes a constant list of the values e.g. 13 = HTTP_RESP.

httpCode: The HTTP response code e.g. 404, 500, etc. Only meaningful when the error was an HTTP-level error.

```
HTTP Error: HTTP/1.1 404 Not Found
Error Number: 13
HTTP Status: 404
```

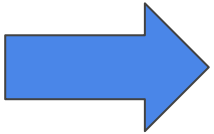
HTTPERROR1.pgm.rpgle

YAJL & JSON Data Types

```
// Add an EBCDIC string
// "name": "Richard Moulton"
yajl_addChar('name':custName);

// Add a numeric value (from a string)
// "invoiceNumber": 123456
yajl_addNum('invoiceNumber':%editc(invNum:'Z'));

// Add a date, as an EBCDIC string
// "invoiceDate": "2026-05-22"
yajl_addChar('invoiceDate':invDate);

// Add a numeric value (from a string)
// "exchangeRate": 1.150985
yajl_addNumf('exchangeRate':currRate);

// Add a boolean value
// "emailed": true
yajl_addBool('unread':emailed);

// Add a null value
// "shipDate": null
yajl_addNull('shipDate');
```

JSON doesn't have specified data and time data types.

For more details on JSON date and time handling refer to Gotcha #4.



```
{
 "name": "Richard Moulton",
 "invoiceNumber": 123456,
 "invoiceDate": "2026-05-22",
 "exchangeRate": 1.150985,
 "emailed": true,
 "shipDate": null
}
```

OAuth Flows

There are many OAuth flows but the two most common are **Authorization Code** and **Client Credentials**.

Authorization Code

The authorization code flow is the one with redirects.

It exists because a human needs to log in and consent in a browser.

The sequence is: your app sends the user's browser to the auth server, the user logs in, and the auth server redirects the browser back to your redirect_uri carrying an authorization code. This is what you use when you're acting on behalf of a user - "log in with Google," a SaaS app accessing someone's mailbox, that kind of thing.

Client Credentials

The one you're most likely to encounter.

The client credentials flow is the one without redirects.

There's no user, no browser, no consent screen - it's just your backend authenticating as itself to access resources.

The entire thing is a single POST: you send your client ID and secret to the token endpoint, you get back an access token, done. No 3xx anywhere.

For an IBM i shop doing back-office integration - pulling data from a vendor API, pushing orders to a logistics provider, syncing with a payment processor's server-side endpoints - you're almost always in client credentials territory. There's no human in the loop, so there's nothing for a redirect to accomplish. You'll authenticate with a POST, get a bearer token, and attach it as an Authorization: Bearer xxx header on subsequent calls.

That's the whole story, and HTTPAPI handles it as ordinary POSTs and headers.

Token Lifecycle

Token lifecycle is the part that'll actually demand thought, for short-lived client credential tokens. Bearer tokens can expire (often 30–60 minutes), so you'll want to cache the token somewhere - a data area, a user space, or just module-level storage depending on your job structure. Keep track of when the token expires, ~60 seconds before it is due to expire perform a refresh. You will also need to handle an expired token and detect a 401 to trigger a refresh rather than re-authenticating on every single call. That caching-and-refresh logic is where the real engineering is in server-side OAuth.

Extras: Certificate Stores

HTTPAPI & Certificate Stores

What is a certificate store?

A certificate store on IBM i is a GSKit format key database - a set of encrypted files holding the certificates and (optionally) their private keys.

DCM (Digital Certificate Manager) is the browser UI on top of it.

The *SYSTEM certificate store is used by operating system components (Telnet, FTP, DDM, host servers, the HTTP server, etc.) when they need a TLS certificate.

What is GSKit?

GSKit is IBM Global Security Kit - a cryptographic library and set of command-line tools that IBM uses across its product line to provide SSL/TLS and certificate/key management.

Think of it as IBM's in-house equivalent of OpenSSL.

HTTPAPI & Certificate Stores

To create a certificate store first login to DCM.

- http://myibmi.com:2006/dcm
- https://myibmi.com:2007/dcm

Select the “Create Certificate Store” menu option.

Select a store e.g. *SYSTEM for the certificate store. If you want to create your own certificate store then create Other.

You will be asked to enter and confirm a password for the certificate store - you’ll need to keep this safe.

If you selected the *SYSTEM store then you’ll have two new files in the IFS:

- `/QIBM/UserData/ICSS/Cert/Server/DEFAULT.KDB`
- `/QIBM/UserData/ICSS/Cert/Server/DEFAULT.RDB`

To use HTTPAPI, your application users need read access to this file

HTTPAPI & Certificate Stores

Checking permissions on your certificate store file:

`/QIBM/UserData/ICSS/Cert/Server/DEFAULT.KDB`

I'm using the *SYSTEM certificate store in this example

The following directories all ship with *PUBLIC *RX, which is what we want.

- `/QIBM`
- `/QIBM/UserData`
- `/QIBM/UserData/ICSS`
- `/QIBM/UserData/ICSS/Cert`

You should check the defaults haven't been changed on your system

The following directory ships with *PUBLIC *EXCLUDE.

- `/QIBM/UserData/ICSS/Cert/Server`

We'll need to change this to allow application users access.

HTTPAPI & Certificate Stores

Setting permissions on your certificate store file:

`/QIBM/UserData/ICSS/Cert/Server/DEFAULT.KDB`

To grant access to the certificate file, first allow access to the `Server` directory.

```
CHGAUT OBJ('/QIBM/UserData/ICSS/Cert/Server') USER(MYAPPGRP) DTAAUT(*RX)
```

Next allow access to the certificate store file itself.

```
CHGAUT OBJ('/QIBM/UserData/ICSS/Cert/Server/DEFAULT.KDB') USER(MYAPPGRP)
DTAAUT(*R)
```

I've used the *MYAPPGRP* group profile in this example but you could use a specific user profile or a group profile.

Certificate Stores & IBM i Services

You can use SQL to retrieve certificate store details.

```
SELECT CERTIFICATE_LABEL, VALIDITY_START, VALIDITY_END, TRUSTED
  FROM TABLE(
    QSYS2.CERTIFICATE_INFO(
      CERTIFICATE_STORE => '/QIBM/UserData/ICSS/Cert/Server/DEFAULT.KDB',
      CERTIFICATE_STORE_PASSWORD => '*NOPWD'
));
```

IBM i Documentation: CERTIFICATE_INFO table function

You can also specify your own certificate store.

```
SELECT CERTIFICATE_LABEL, VALIDITY_START, VALIDITY_END, TRUSTED
  FROM TABLE(
    QSYS2.CERTIFICATE_INFO(
      CERTIFICATE_STORE => '/home/RICHARDM/RMCERTS.KDB',
      CERTIFICATE_STORE_PASSWORD => '*NOPWD'
));
```

Extras: Certificate Chains

Certificate Chains: How To View

Also checkout the Import CA Certificate to DCM slide.

You can use the PASE `openssl` command to view the certificate chain for any given server. The `openssl` command can also be run on your PC, if installed.

```
# First ensure the PASE openssl package is installed, as an admin user
yum install openssl

# Show the chain summary without PEM blobs
openssl s_client -connect open.er-api.com:443 </dev/null 2>/dev/null
```

The upper section shows you the certificate chain.

```
Certificate chain
 0 s:CN = er-api.com
   i:C = US, O = Google Trust Services, CN = WE1
   a:PKEY: id-ecPublicKey, 256 (bit); sigalg: ecdsa-with-SHA256
   v:NotBefore: Mar 27 01:04:12 2026 GMT; NotAfter: Jun 25 02:03:57 2026 GMT
 1 s:C = US, O = Google Trust Services, CN = WE1
   i:C = US, O = Google Trust Services LLC, CN = GTS Root R4
   a:PKEY: id-ecPublicKey, 256 (bit); sigalg: ecdsa-with-SHA384
   v:NotBefore: Dec 13 09:00:00 2023 GMT; NotAfter: Feb 20 14:00:00 2029 GMT
 2 s:C = US, O = Google Trust Services LLC, CN = GTS Root R4
   i:C = BE, O = GlobalSign nv-sa, OU = Root CA, CN = GlobalSign Root CA
   a:PKEY: id-ecPublicKey, 384 (bit); sigalg: RSA-SHA256
   v:NotBefore: Nov 15 03:43:21 2023 GMT; NotAfter: Jan 28 00:00:42 2028 GMT
```

If you want to grab one of the certificates from the chain, to import into DCM then show the certificate PEM blobs.

```
# Show the chain summary with PEM blobs
openssl s_client -connect api.exchangerate-api.com:443 -showcerts </dev/null 2>/dev/null
```

You can then copy everything between (and including) the `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----` lines for the relevant certificate.

In this example if I wanted the Let's Encrypt R13 intermediate CA certificate I would paste this data into a text editor and save it as a file called `r13.pem`. However, for Let's Encrypt I would grab the certificates directly from their site: https://letsencrypt.org/certificates.

Certificate Chains: Import CA Certificate to DCM

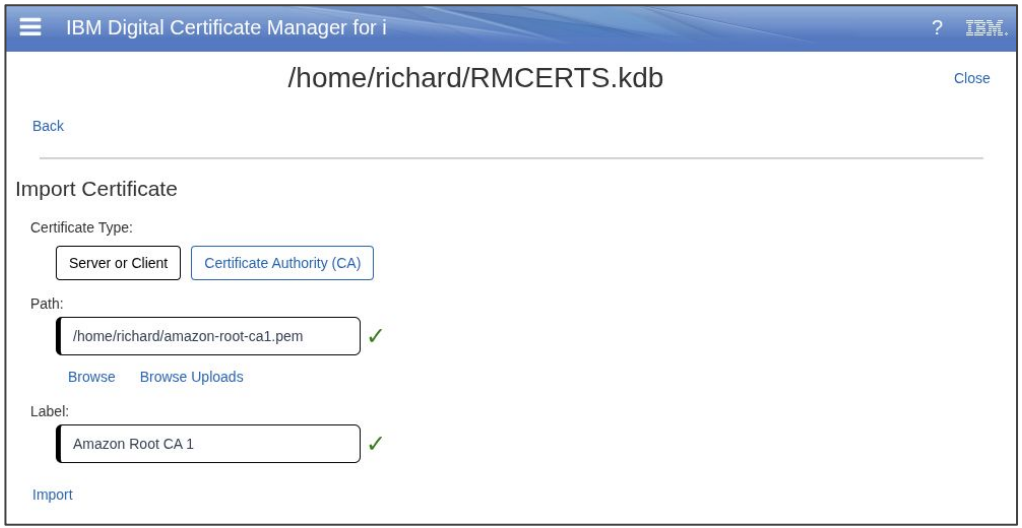
To install a CA certificate file into DCM you first need to copy it to the IFS.

Before importing it to DCM you have to set the CCSID to 1252, otherwise it can fail!

```
CHGATR OBJ('/home/richard/r13.pem') ATR(*CCSID) VALUE(1252)
```

This should be another gotcha!

I wasted a lot of time working this out.



Certificate Chains: Self-Signed Certificates

Self-signed certificates should be treated in the same way - in terms of trust.

If you trust that self-signed certificate then import it to your certificate store - your trust store.

If you have your own internal CA then it’s your root CA that you’d import.

When dealing with self-signed certificates you need to ensure the server name matches the SAN (Subject Alternative Name) on the certificate, which often trips people up when addressing servers by IP address or a short name.

Extras: MITM Attack

How a MITM Attack Works

Picture three parties: **your IBM i**, the **real server** (say, `api.partner.com`), and an **attacker** who has positioned themselves somewhere on the network path between you.

1. Your program initiates an HTTPS call to `api.partner.com`
2. The attacker intercepts the connection before it reaches the real server
3. The attacker presents *their own* certificate to your IBM i, pretending to be `api.partner.com`
4. Without `https_strict(*on)`, HTTPAPI doesn't validate the certificate at all. The attacker's certificate is accepted, and the TLS tunnel is built with the attacker - not the partner
    a. HTTPAPI doesn't validate the certificate at all, so the attacker's certificate is accepted
5. The attacker opens their own genuine TLS connection onward to the real `api.partner.com`
6. From now on, the attacker sits in the middle: decrypting your traffic, reading or modifying it, then re-encrypting it to the real server (and vice versa for responses)
7. Both ends see a "successful" HTTPS exchange - neither side knows there's a third party in the room
    a. Though the real server would see the attacker's IP address not the IBM i IP address

Sources: OWASP, MITRE ATT&CK (T1557), NIST SP 800-52 Rev. 2