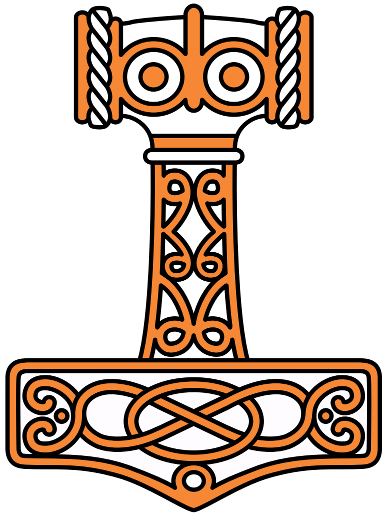
Olhão 2022

Building Web Services with Jarvis

(Workshop SA2)

Brian Becker



A Few Administrative Items

- The hotel has allotted one "snack" per attendee at the breaks. Please respect that.
- Please fill out the Workshop Feedback form:
  - Preferably **after** the workshop
  - If you are not comfortable giving the filled out form to me, there will be someone outside the room after the workshop to collect them.
  - If you want me to fill out the form for you, I will ☺

Introductions and Agenda

- A bit about me...
  - https://aplwiki.com/wiki/Brian_Becker
- And you?
- Three ~1-hour sections with two 15-minute breaks
  - Introduction to Web Services and Jarvis
    - Break
  - Jarvis Configuration and Web Service Design
    - Break
  - Sample "Phonebook" App

Objectives for this Workshop

- Be able to define a simple web service
- Understand most of the "important" Jarvis configuration settings
- Understand what's available in Jarvis to build more complex services
- Get your feedback
- Not an objective: teach you in depth Jarvis or HTTP

Miscellaneous Stuff…

- Ask questions!
  - But please be mindful of time and the specificity of the question.
- Offer suggestions
  - Features you'd like to see or think Jarvis should have
  - Techniques – is there a better way to do something?
- Internally, Jarvis uses `(⎕IO ⎕ML)←1` and today's exercises will as well
  - Your application code can use whatever best suits you
- We will be starting a lot of instances of Jarvis today. Best practice is to close the instance before opening another to avoid "port in use" conflicts.

Quick Survey

- How many of you have:
  - Used a web service either directly or indirectly?
  - Written a web service?
  - Used Jarvis?
  - Understand HTTP – cookies, headers, methods, etc?

On Your Mark…

When you see [SA2] in text and examples, it refers to the folder where you installed the SA2 workshop materials.

- ✓ SA2 materials downloaded?
- ✓ Jarvis downloaded?
- ✓ Local port available?

Get Set...

```
⍝ Start Dyalog APL
      )clear
      sum←{+/⍵}
      rotate←⌽
      ]load [SA2]/Jarvis
```

Go!

```
⍝ you can specify a port other than 8080 if necessary
     j←1⊃Jarvis.Run 8080 #
     ]open http://localhost:8080
     ]load HttpCommand
     (HttpCommand.GetJSON 'post' 'localhost:8080/sum' (⍳5)).Data
```

What did we just do?

- We defined and started a web service
  - Defined "endpoints" for the service
  - Started the service
  - Used a browser to open a page that contained a JavaScript client to communicate with the service
  - Used HttpCommand as a client

Web Service or Web Server

- Web Service
  - Uses HTTP
  - Machine-to-machine
  - Variety of clients
    - Python, C#, APL, JavaScript
  - Specific API

- Web Server
  - Uses HTTP
  - Human interface
  - Client is typically a browser using HTML/CSS/JavaScript

Introducing Jarvis

- JSON and REST Service
- Supports two "paradigms" - JSON and REST
- A service can run only one paradigm
- Jarvis' ancestry
  - Originally written as JSONServer in December 2017 for a client over a weekend
  - Core HTTP server has been in use for many years
  - REST capability was added at a client's request and renamed Jarvis

Jarvis Design Philosophy

- Assume as little as possible about how the user will use it
  - Be flexible - gives the user the flexibility to use Jarvis as he deems best, not how I dictate.
    - `CodeLocation` can be a ref, a name of a ref, or a folder specification
    - Configuration parameters can be specified in a configuration file, a namespace passed to the constructor, or set individually.
- Provide sensible default behavior to hide some of the nuances of HTTP and web services, but also provide low-level access for the users who need it.
- Use "hooks" for the user to inject code into the flow at obvious points.
  - Startup, at the start of each request, session initialization, authentication, …
  - If you feel the need to modify the Jarvis code itself – we probably need to add another hook.
- Need-driven design – if you need it, we'll try to put it in
  - CORS support and the REST paradigm are two examples

REST

- The HTTP method, URI, and payload specify what to do.
- Standard HTTP methods for operations

  GET – retrieve a resource
  POST – create a resource
  PUT – replace a resource
  PATCH – update a resource
  DELETE – delete a resource

- URI Endpoints are "resources"
- Payloads are often JSON or XML

The GitHub REST API is a good example
https://docs.github.com/en/rest/repos/repos

GitHub API (abbreviated) Examples

- Get the commits for a repository
  ```
  GET /repos/Dyalog/Jarvis/commits
  ```
- Create an organization repository
  ```
  POST /orgs/Dyalog/repos
     {"name":"NewRepo"}
  ```
- Update a repository
  ```
  PATCH /repos/Dyalog/Jarvis
     {"name":"NewName"}
  ```

GitHub Web Service REST Example

```
      ]load HttpCommand
#.HttpCommand

      ⊢ r ← HttpCommand.Get 'https://api.github.com/orgs/dyalog-training/repos'
[rc: 0 | msg: | HTTP Status: 200 "OK" | ⍴Data: 43023]

      100↑r.Data
[{"id":537497880,"node_id":"R_kgDOIAmRGA","name":"2022-SA1","full_name":"dyalog-training/2022-SA1","

      ⊢ r ← HttpCommand.GetJSON 'get' 'https://api.github.com/orgs/dyalog-training/repos'
[rc: 0 | msg: | HTTP Status: 200 "OK" | ⍴Data: 8]

      r.Data.name
 2022-SA1  2022-SA2  2022-TP2  2022-SA3  2022-SP1  2022-SP2  2022-TP3  .github

      ↑r.Data.(name updated_at)
 2022-SA1  2022-10-05T08:28:28Z
 2022-SA2  2022-10-05T21:24:30Z
 2022-TP2  2022-09-21T11:29:37Z
 2022-SA3  2022-09-24T06:56:29Z
 2022-SP1  2022-09-28T13:04:05Z
 2022-SP2  2022-10-06T14:00:03Z
 2022-TP3  2022-09-29T18:23:40Z
 .github   2022-10-06T13:35:40Z
```

REST Paradigm

- Write a function for each HTTP method that your service will support
  - `response ← GET request`
    `request` is the request object
    `response` is the response payload
  - The function will parse the path and endpoint to identify the resource

```
GET /customers                ⍝ get all customers
GET /customers/10             ⍝ get customer 10 information
GET /customers/10/invoices ⍝ get customer 10's invoices
```

- There are other principles that help determine a service's "RESTfulness" including:
  - Statelessness
  - Caching of responses

Jarvis does not address these

JSON Paradigm

- Endpoints are result-returning monadic or dyadic APL functions
  - Right argument is the request payload
  - Optional left argument is the request object itself
- All requests use HTTP POST method
- Request and response payloads are JSON
  - Jarvis handles all conversion between JSON and APL formats

REST or JSON?

REST

- Good for "database" applications
  CRUD – create, read, update, delete
- API requires thought/discipline
  For instance, how to implement a query?
  `get /customers/country/Denmark`
  `get /customers?country=Denmark`
- Need to understand HTTP requests
  - HTTP Method, Path, Query Parameters, Headers, Payload, Status Codes

JSON

- Good for functional endpoints
- API is more flexible
- API is easier to implement
- Probably suits the "APL mindset" better
- Understanding HTTP requests is useful but generally necessary

JSON Paradigm

To send a request to a Jarvis service running the JSON paradigm, the client performs the following:

- Specify the host and endpoint
  - `http://localhost:8080/sum`
- Specify the payload/data/body in JSON format
  - `[2,4,6]`
- Specify the content-type as 'application/json'
- Specify the HTTP method as POST

Anatomy of a JSON HTTP Request

HTTP Method

Endpoint

Host Address

Headers

```
POST /sum HTTP/1.1
Host: localhost:8080
User-Agent: Dyalog-HttpCommand/5.1.3
Accept: */*
Content-Type: application/json
Content-Length: 7
```

Payload

```
[2,4,6]
```

Client Examples

JavaScript

```
var xhr = new XMLHttpRequest();
xhr.open("POST", http://localhost:8080/sum);
xhr.setRequestHeader("content-type", "application/json");
xhr.send("[1,2,3,4]");
xhr.response;
```

PowerShell

```
$url = http://localhost:8080/sum
$hdrs = @{'content-type' = 'application/json'}
$body = '[1,3,5,7,9,11]'
Invoke-WebRequest -Method Post -URI $url -Headers $hdrs -Body $body
```

Python

```
import requests
import json
url = 'http://localhost:22333/sum'
hdrs = {"content-type":"application/json"}
array = [2,4,6,8]
resp = requests.post(url, data=json.dumps(array), headers=hdrs)
print(resp.json())
```

curl

```
curl -d "[1,2,3,4,5]" -H "content-type:application/json" http://localhost:8080/sum
```

APL

```
HttpCommand.GetJSON 'post' 'localhost:8080/sum' (⍳5)
```

Some Web Service Design Questions

- Stateful or Stateless?
  - Does your service need to maintain "state" between requests?
  - If so, where to maintain that state? On the client or in the server?
- Security?
  - HTTPS
  - Authentication/Authorization
- Scalability?
  - Come to the Deploying Services workshop ☺

JSON Briefly

- Lightweight, language-neutral, data-interchange format
- https://www.json.org/json-en.html
  - Scroll down to Languages section

JSON and APL

- JSON is a natural and complementary fit with APL
- ⎕JSON converts between JSON and APL representations
  - APL arrays with rank >1 can be split to make vectors of vectors (of vectors...)

	JSON	APL
Number	`42`	`42`
String	`"hello"`	`'hello'`

Array	`[ 2, "hello" ]`
	`[[1,2,3],["hi","there"]]`
		`2 'hello'`
		`(1 2 3)('hi' 'there')`

Object	`{"number": 2,`
	`"greeting":"hello"}`
		`obj←⎕NS ''`
		`obj.(number greeting)←2 'hello'`

Jarvis Configuration Settings

- Can be specified
  - in a JarvisConfig JSON file
  - in environment variables (must use Jarvis workspace) or in the constructor argument to the Jarvis class
  - directly in the Jarvis instance

Settings take precedence in the order above

We'll refer to the collection of settings as "JarvisConfig"

Running Jarvis

- Jarvis.dws
  - At least one Jarvis config setting must be set as an environment variable
- Jarvis.dyalog
  - Create an instance
  - Set configuration
  - And go!
- dyalog/Jarvis Docker container
  - a public container found on DockerHub https://hub.docker.com/dyalog/jarvis

Useful Functions

- `j←⊃Jarvis.Run args` – create and start a Jarvis server
- `j←Jarvis.New args` – create a Jarvis server
- `j.Start` – start the Jarvis server
- `j.Stop` – stop the Jarvis server
- `j.Config` – show the Jarvis server's configuration

`Jarvis.Run` and `Jarvis.New`

- `r←Jarvis.Run args` – creates and starts a Jarvis server
  - args is one of:
    - a character vector containing either the name of a JarvisConfig file or CodeLocation
    - a reference to a JarvisConfig namespace
    - [1] the port Jarvis is to list on
      [2] CodeLocation
      [3] (optional) the paradigm to use ('JSON' or 'REST'). Default is 'JSON'
      [4] (optional) the name of a JarvisConfig file or reference to a JarvisConfig namespace
  - r is
    [1] a reference to the Jarvis instance
    [2] a return code (0 means "OK" and Jarvis was started, non-zero means error)
    [3] a (hopefully useful) message if the return code is non-zero
  - If you forget to capture the result of `Jarvis.Run`, you can use `j←⊃⊃⎕INSTANCES Jarvis`
- `Jarvis.New` takes the same arguments as `Jarvis.Run` but just returns a reference to the instance

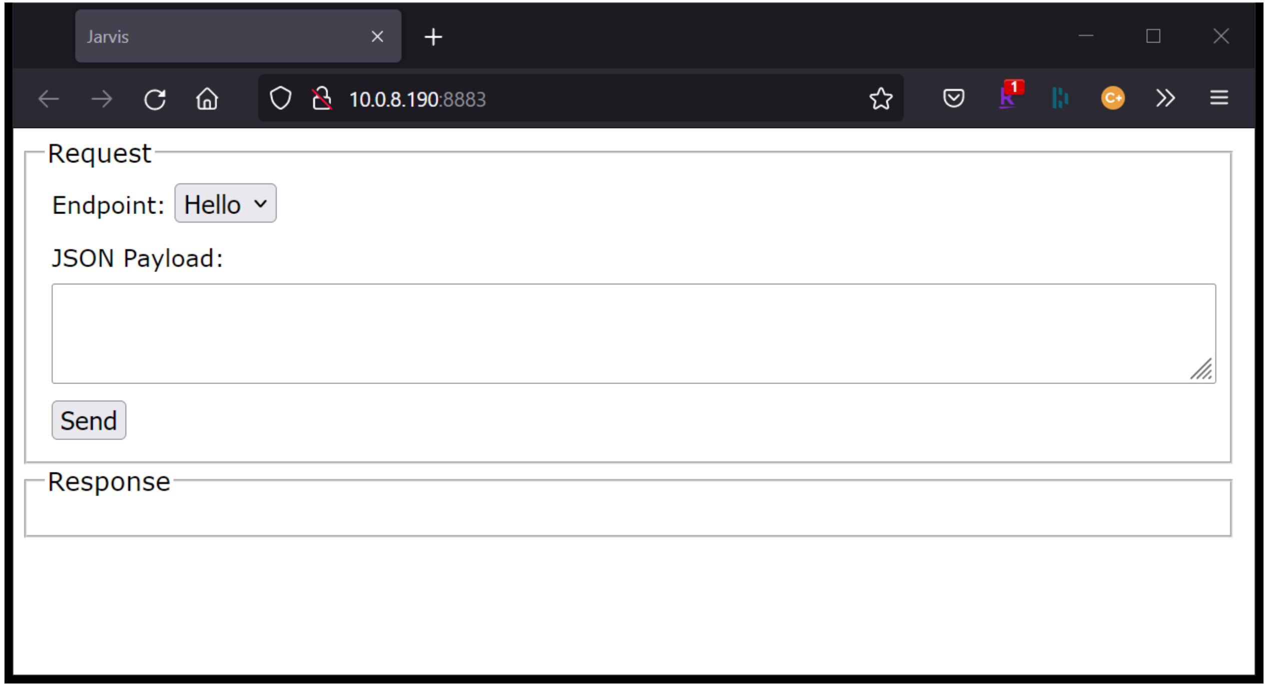
CodeLocation

- is where Jarvis looks for your code
  - Namespace reference or name: `#.myAPI` or `'#.myAPI'`
    `Jarvis.Run 8080 #`
  - Folder name: either fully qualified or relative to:
    - Workspace if not CLEAR WS
    - Folder of JarvisConfig file if it exists
    - Jarvis' source folder (assuming you loaded Jarvis from file)

HTML Interface

- Jarvis is not a web server but it can serve static HTML content and has a built-in, simple, HTML interface.
  - This interface was developed for demonstration and testing purposes.
  - It is useful for for showing what endpoints are exposed.
- The `HTMLInterface` configuration setting controls the HTMLInterface:
  - 0 means disable any HTML interface
  - 1 (the default) means enable the built-in HTML interface
  - The name of a folder or file containing the content for an HTML interface
    This is how TryAPL.org works.

HTML Interface



Exposing and Hiding Endpoints

- By default, all functions in `CodeLocation` (and below) are exposed as endpoints.

  -
    ```
    j←Jarvis.Run 8081 '[SA1]/SampleCodeLocation'
    ]open https://localhost:8081
    ```
- Use `IncludeFns` and `ExcludeFns` which are vector(s) of:
  - Function names: `'sum' 'rotate'`
  - Strings with wildcards: `'hidden.*'`
  - regex: `"^[A-Z].*"`
  - Any combination of the above

`IncludeFns` is run before `ExcludeFns`

Tying some of the pieces together…

```
settings←⎕NS ''
settings.Port←8882
settings.CodeLocation←'[SA2]/SampleCodeLocation'
settings.ExcludeFns←'hidden.*' 'utils.HideMe'

j←Jarvis.New settings
j.Start
j.Stop

)ed file://[SA2]/SampleCodeLocation/JarvisConfig.json
j←Jarvis.New '[SA2]/SampleCodeLocation/JarvisConfig.json'
j.Start
j.Stop
```

Ready for the next level?

Up to now we've used simple monadic functions as our endpoints.

- If you have a dyadic (or ambivalent) function, a reference to the HTTP `Request` object is passed as the left argument.
- This provides access to metadata for the request that can be used to further validate the request.
- It also makes it easier for us to "be a good citizen" and conform to some common practices for web services.

`Request` Object

- An instance is created for each HTTP request received by Jarvis.
- The two main uses for the request are:
  - querying request parameters sent by the client
    - headers, cookies, peer certificate, among others
  - managing response content to be send back by Jarvis
    - HTTP status code and message, and the payload
- Simple web services may never need to use `Request`

Useful `Request` Functions

- `{status}←{message} Fail HTTPStatus`
  `{status}←{message} SetStatus HTTPStatus`
  Sets the HTTP response status code and status message
  If message is not supplied, use the standard message (if there is one) for the code

- `value←GetCookie name`
  Return the value of the cookie named name or '' if no cookie with that name exists.

- `value←GetHeader name`
  Return the value of the HTTP header named name or '' if no header with that name exists.

- `name SetCookie cookie`
  Set a **response** cookie. cookie is the cookie value with optional additional cookie settings appended and separated by ';'

- `name SetHeader header`
  Set a **response** header

Some `Request` Object Fields

- `Response` – reference to a namespace containing `Status`, `StatusText`, and `Payload`
- `Server` – reference to the Jarvis server instance
- `Session` – reference to the session namespace, if using sessions
- `EndPoint` – the endpoint for the request
- `Password` – if using HTTP Basic authentication, the supplied password
- `UserID` – if using HTTP Basic authentication, the supplied user ID.

HTTP Response Statuses

- HTTP statuses reflect the success or failure of the server to satisfy the request
- Jarvis will set appropriate HTTP status codes for conditions it detects.
  - Success
  - Endpoint not found
  - Unauthorized request
- You can use `req.SetStatus` inside your endpoints to set appropriate statuses.

- 2xx – success

  200 - Success
  201 - Created
  204 - No content

- 4xx – Client Error

  400 - Bad Request
  401 - Unauthorized
  403 - Forbidden
  404 - Not found
  405 - Method not allowed

- 5xx – Server Error

  500 - Internal server error

Hooks

- Jarvis has several "hooks" where you can inject your code.
  You set a hook by assigning the name of your function that implements the hook to one of the following:

  `AppInitFn` – called when Jarvis starts
  `AppCloseFn` – called when Jarvis stops
  `SessionInitFn` – called when a new session is created (sessioning must be enabled)
  `AuthenticateFn` – called on every request
  `ValidateRequestFn` – called when the request is received but before Jarvis starts processing the request

- All of the hooks take a Request object as their right argument and return 0 if there is no error.

- If you do not specify a hook, Jarvis uses `{0}` as its definition.

Debugging

- We know that our application code won't fail.
- And we're confident that Jarvis itself is without flaw.
- And users always send us the data we're expecting.
- But *just in case* that smallest of possibilities happens and things don't behave as we expect...
- Here are some tips to help you debug a Jarvis web service...

Debugging

- `Jarvis.Debug←0`
  No debugging, Jarvis traps all errors and reports them as 500
- `Jarvis.Debug←1`
  Jarvis suspends on any error
- `Jarvis.Debug←2`
  Jarvis suspends just prior to calling user endpoints or hooks
- `Jarvis.Debug←4`
  Jarvis suspends just after receiving the client request
- Values are additive: 5 = 1+4

Debugging

- When you have a reproducible error, but don't try to reproduce it from a client running in the same APL process as Jarvis. In other words, don't use `HttpCommand` to produce the error from the same session that Jarvis is running in.
- Then, in the Jarvis process, set `Jarvis.Debug←1`.
- Switch to the client process and issue the request that causes the error.
- Switch back to the Jarvis process (it should be suspended) and do your normal debugging.
- Set `Jarvis.Debug←0` and try to reproduce the error from the client
- To debug your endpoint or hook code, `Jarvis.Debug←2` and use the debugger to step through your code.

Other Debugging Aids/Hints

- Check the configuration using `j.Config`
- Use the built-in HTML interface to query and test endpoints. `j.HTMLInterface←1`
- If you need to change Jarvis settings, it's safest to stop the server, make the changes, and start the server again.

Maintaining State between Requests

- Client side
  - All necessary state is "bundled" by the client in the request, updated and bundled in the response by the server endpoint.
  - This is how TryAPL.org works.
  - Good for distributed/load balanced applications – it doesn't matter which server instance handles the request

Maintaining State between Requests

- Server side
  - When a session starts, Jarvis creates
    - a session namespace
    - a session ID that is either sent as a cookie or a header
    - the cookie or header must be sent with every subsequent request to maintain session continuity. Cookies are preferred as they are sent automatically by many clients.
  - In a distributed/load balanced applications – you may need to make the request "sticky" so subsequent requests are handled by the same server

Session Configuration Settings

```
SessionIdHeader←'Jarvis-SessionID'
⍝ Name of the header field or cookie for the session token

SessionUseCookie←0
⍝ 0 - use the header; 1 - use an HTTP cookie

SessionPollingTime←1
⍝ how frequently (in minutes) we should poll for timed out sessions

SessionTimeout←0
⍝ 0 = do not use sessions, ¯1 = no timeout , 0< session timeout time (in
minutes)

SessionCleanupTime←60
⍝ how frequently (in minutes) do we clean up timed out session info from
_sessionsInfo
```

Session Example

In [SA2]/SessionDemo:

```
jarvisconfig.json:
{  "SessionInitFn" : "InitializeSession",
   "SessionTimeout" : .25,
   "Port" : 8889,
   "SessionUseCookie" : 1  }

     ∇ InitializeSession req
[1]  ⍝ initializes the session
[2]    req.Session.Sum←0
     ∇

     ∇ r←req Add arg
[1]  ⍝ arg is an integer array
[2]    req.Session.Sum+←+/arg
[3]    r←req.Session.Sum
     ∇

     j←Jarvis.Run '[SA2]/SessionDemo/jarvisconfig.json'
```

Authentication/Authorization

- Jarvis supports HTTP Basic authentication
  - When used through a browser, the familiar credentials dialog will appear.
  - Credentials can also be provided in the URL or in an Authorization header.
  - NOTE: HTTP Basic authentication encodes but does not encrypt the user credentials. It should never be used over a unencrypted link.
- You can also "roll your own" by creating a login endpoint and having the user enter their credentials.
  - There are usage patterns that you can employ to securely send credentials over an unencrypted link, but it's much simpler to use HTTPS.

Cross-Origin Resource Sharing (CORS)

- Jarvis CORS support. Why might this matter to you?
  - If someone wants to call your web service from within a web page they've developed, CORS enables browsers to accept responses from your web service.
- CORS is a deeper subject than we have time for in this workshop, but Jarvis' CORS support will be fully documented in the forthcoming documentation.

Exercise Time

- Write a web service with 2 endpoints
  - One endpoint can be simple (monadic)
    - The request payload can be as simple or complicated as you like
  - The other endpoint should be dyadic
    - The request payload can be as simple or complicated as you like
    - In addition to the response payload that's calculated from the request payload, include something about the request itself in the response
- If you're really brave, try adding hooks

Sample Phonebook Application

- Users table
  - contains user credentials (login and password) for "admins"
  - admins can edit Users table and Phonebook table
- Phonebook table
  - contains first name, last name, extension, and password
  - "owner" of an extension can edit their extension

Sample Phonebook Application

- Users endpoints
  - AddUser
  - DeleteUser
  - UpdateUser
  - GetUsers
  - GetUserByLogin
- Phonebook endpoints
  - AddPhonebookEntry
  - DeletePhonebookEntry
  - UpdatePhonebookEntry
  - GetPhonebookByExtension
  - SearchPhonebook

Sample Phonebook Application

- All endpoints take a namespace argument
  - {"lastName":"Kromberg", "firstName":"Morten", ...}
- All endpoints return a namespace containing
  - rc – return code, 0 means "no error"
  - msg – informational message
  - payload – any data returned by the endpoint

Sample Phonebook Application

- Three versions of the same application:
  - v1 – implements all the basic functionality for every endpoint but does not validate the request payloads nor implement any authentication/authorization.
  - v2 – implements authentication/authorization
  - v3 – implements request payload checking

What lies ahead…

- New functionality will be driven by user needs
- Release process will be more formal
  - Semantic versioning
  - GitHub Releases
  - Available as a Tatin package
- Documentation is being written https://dyalog.github.io/Jarvis/
- Training materials, more samples, webcasts are planned.