DYALOG

Glasgow 2024

Web Services

Brian Becker

Morten Kromberg

Michael Baas

Welcome!

- Introductions
  - Name
  - Any experience?
  - Goals
- Schedule
  - 60 15 60 15 60
- Participate
  - Ask questions
  - Work together
  - Ask for help if you need it
  - Have fun
- `(⎕IO ⎕ML)←1`

Goals

Learn enough about `HttpCommand` to call web services

Learn enough about `Jarvis` to implement a simple JSON-based web service

HTTP Communications 101

HTTP is a request-response protocol

A client sends a request to a server

The server receives the request

The server runs an application to process the request

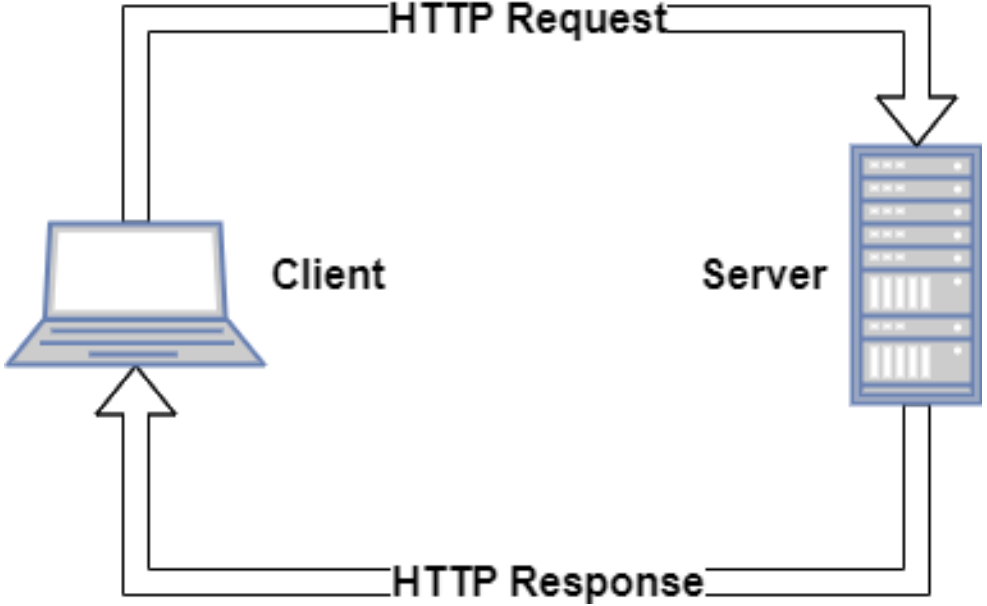
The server sends a response back to the client

The client receives the response

Client Examples:
A web browser, `HttpCommand`, cURL, JavaScript, Python

Server Examples:
IIS, Apache, Nginx, `Jarvis`, `DUI/MiServer`

HTTP Communications 101



HttpCommand

`HttpCommand` is a utility that enables the APLer to interact with web services because it:

- Allows you to specify an HTTP request in a manner that is conducive to an APLer
- Sends a properly formatted HTTP request to the server
- Receives the server's response
- Decomposes the response in a manner that is conducive to an APLer
- Minimizes the need for you to learn a lot about HTTP

Obtaining HttpCommand

```
      ]load HttpCommand
#.HttpCommand
```

Under Dyalog 19.0 and later

```
      ]link.import HttpCommand

       HttpCommand.Version
 HttpCommand  5.4.6  2024-02-28
```

Upgrading `HttpCommand`

`HttpCommand.Upgrade` will download the latest released version from GitHub if it's newer than your current version.

```
      HttpCommand.Upgrade
1  Upgraded to HttpCommand 5.8.0 2024-07-17 ...
```

DO NOT use **`HttpCommand.Upgrade`** in production code.
A major version bump might introduce a breaking change in your application.
Upgrade in a development environment, test it, and then save it.

HttpCommand documentation

HttpCommand is documented online.

```
      HttpCommand.Documentation
See https://dyalog.github.io/HttpCommand/
```

Your first `HttpCommand`

```
      ⊢ resp ← HttpCommand.Get 'dyalog.com'
[rc: 0 | msg:  | HTTP Status: 200 "OK" | ≠Data: 24371]
```

`Data` is the payload sent by the host

HTTP Status is the status, if any, set by the host
2XX means successful

`rc` is ≠0 if HttpCommand could not successfully send and receive
`msg` is a message explaining why

resp is a namespace

```
      resp.(7 3p⍬nl ¯⍳9)
BytesWritten  Command       Cookies
Data          Elapsed       GetHeader
Headers       Host          HttpMessage
HttpStatus    HttpVersion   IsOK
OutFile       Path          PeerCert
Port          Redirections  Secure
URL           msg           rc
```

resp

`resp.Data` contains the response payload

```
      50↑resp.Data
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Trans

      resp.GetHeader 'content-type'
text/html; charset=utf-8

      'hr' ⎕WC 'HTMLRenderer' ('HTML' resp.Data)
```

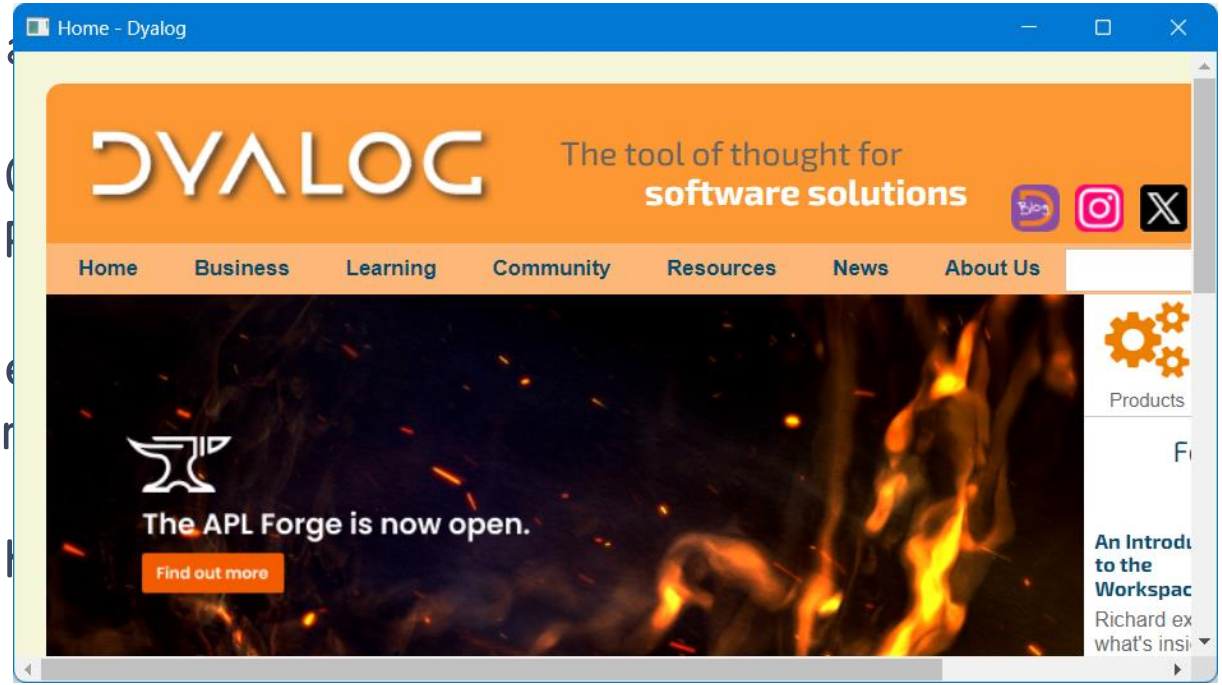
resp

```
resp.Da

      5(
<!DOCTYP                                                    Trans

      re
text/htm

      'H                                                      ta)
```



HttpCommand "Shortcut" Functions

"One time" functions:

- Get - Issue a GET request

```
resp← HttpCommand.Get URL Params Headers
```

- Do - Send any HTTP Command:

```
resp← HttpCommand.Do Command URL Params Headers
```

- GetJSON - Interact with JSON-based web services

```
resp← HttpCommand.GetJSON Command URL Params Headers
```

New - Create a new request instance:

```
req← HttpCommand.New Command URL Params Headers
```

"One time" vs "Create an Instance"

The "One time" `HttpCommand` functions (`Get`, `GetJSON`, and `Do`):

- create, configure and run a local `HttpCommand` instance. They send the request and return the response namespace. The instance, being local to the function, disappears when the function exits.
- No information is carried over from one invocation to the next

"One time" vs "Create an Instance"

When you create an `HttpCommand` instance using **`HttpCommand.New`**:

- request settings that you set persist in the instance - you don't need to respecify them each time
- HTTP cookies that are returned by the server are preserved and sent on subsequent requests
- the connection to the server remains open unless it's closed by the server

Anatomy of an HTTP Request

Create a new "POST" HTTP request to create a GitHub repository

```
req←HttpCommand.New 'post' 'https://api.github.com/user/repos'
```

Set the authentication for the request

```
req.(AuthType Auth)←'bearer' GitHubAPIToken
```

Create parameters for the request

```
req.Params←⎕NS ''
req.Params.(name description)←'test-repo' 'test repository'
```

Run the request

```
resp←Req.Run
```

Anatomy of an HTTP Request

```
Method Endpoint HttpVersion
Headers
Body
```

```
POST /user/repos HTTP/1.1
Host: api.github.com
User-Agent: Dyalog-HttpCommand/5.8.0
Accept: */*
Accept-Encoding: gzip, deflate
Authorization: Bearer [--Your Token--]
Content-Type: application/json;charset=utf-8
Content-Length: 52
{"description":"test repository","name":"test-repo"}
```

Anatomy of an HTTP Response

```
HttpVersion HttpStatus HttpMessage
Headers
Body
```

```
HTTP/1.1 201 Created
Server: GitHub.com
Date: Fri, 08 Sep 2023 18:36:10 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 5562
Location: https://api.github.com/repos/plusdottimes/test-repo
```

```
{"id":689076423,"node_id":"R_kgDOKRJ4xw","name":"test-
repo","full_name":"plusdottimes/test-repo" ...
```

Using `HttpCommand`

1. Create an instance
2. Configure your request
3. Send the request
4. Inspect the response

1. Create an instance

```
h←HttpCommand.New args
```

The following are all equivalent:

```
req←HttpCommand.New 'post' 'bloofo.com' (⍳10) ('content-type'
'application/json')

req←HttpCommand.New ''
req.(Command URL Params)←'post' 'bloofo.com' (⍳10)
req.Headers←'content-type' 'application/json'

ns←⎕NS ''
ns.(Command URL Params)←'post' 'bloofo.com' (⍳10)
ns.Headers←'content-type' 'application/json'
req←HttpCommand.New ns
```

2. Configure your request

`Command`, **`URL`**, **`Params`**, and **`Headers`** are the most-commonly specified settings.

This is why they are arguments to **`Get`**, **`Do`**, **`GetJSON`**, and **`New`**.
If you create a request using `New`, you can specify additional settings before sending the request.

```
req←HttpCommand.New 'get'
req.URL←'https://api.github.com/users/plusdottimes/repos'
req.OutFile←'/tmp/myfile.json'
req.MaxPayloadSize←250000

req.Config ⍝ returns all settings for this request

req.Show   ⍝ returns the request as it will be sent to the server
```

Working with Headers

`HttpCommand` will generate several headers, unless you specify them yourself.
Headers are stored in **`req.Headers`**

Unconditionally set a header

```
'header-name' req.SetHeader 'value'
```

Set a header if not already set

```
'header-name' req.AddHeader 'value'
```

Remove a header

```
req.RemoveHeader 'header-name'
```

Suppress an HttpCommand default header

```
'accept-encoding' req.SetHeader ''
```

3. Send the request

```
      req←HttpCommand.New 'get'
      req.URL←'https://api.github.com/users/plusdottimes/repos'
```

Use the Run method to send the request

```
      ⊢resp←req.Run
[rc: 0 | msg:  | HTTP Status: 200 "OK" | ≠Data: 10026]
```

4. Inspect the response

`resp.IsOK` checks that **`0=rc`** and **`2=⌊0.01×HttpStatus`**

```
      resp.IsOK
1
      resp.Headers  ⍝ contains the response headers
      resp.Data     ⍝ contains the response payload
```

rc and HttpStatus

Try:

```
      HttpCommand.Get ''
[rc: ¯1 | msg: No URL specified | HTTP Status:  "" | ≠Data: 0]

      HttpCommand.Get 'bloofo.com'
[rc: 1106 | msg: Conga client creation failed...

      HttpCommand.Get 'wikipedia.com/bloofo'
[rc: 0 | msg:  | HTTP Status: 404 "Not Found"...
```

Best Practices

```
resp←HttpCommand.Get 'someurl'
:If resp.IsOK
    ⍝ process response
:Else
    ⍝ deal with failed request
:EndIf
```

`req.TranslateData←1`

Many web services return XML or JSON payloads.

Use **`TranslateData←1`** to automatically translate these using `⎕XML` or `⎕JSON` based on the content-type returned by the host.

If `⎕XML` or `⎕JSON` fails, **`resp.rc`** is set to `¯2`, but the response payload is preserved, untranslated, in **`resp.Data`**

req.TranslateData←1

```
      url←'https://api.github.com/users/plusdottimes/repos'
      req←HttpCommand.New 'get' url

      ⊢resp←req.Run
[rc: 0 | msg:  | HTTP Status: 200 "OK" | ≠Data: 10659]

      40↑resp.Data
[{"id":854679147,"node_id":"R_kgDOMvFeaw
```

req.TranslateData←1

```
      req.TranslateData←1

      ⊢resp←req.Run
[rc: 0 | msg:  | HTTP Status: 200 "OK" | ≠Data: 2]

      ↑r.Data.(full_name created_at)
 plusdottimes/MyPrivateRepo  2024-09-09T15:38:10Z
 plusdottimes/MyPublicRepo   2024-09-09T15:47:32Z
```

Recap

1. Create an instance
2. Configure your request
3. Send the request
4. Inspect the response

```
[23]  req←HttpCommand.New 'get' 'someurl.com'
[24]  req.TranslateData←1
[25]  'content-encoding' req.SetHeader ''
[26]  req.MaxPayloadSize←200000
[27]  resp←req.Run
[28] :If resp.IsOK
[29]    ⍝ code to run on success
[30] :Else
[31]    ⍝ code to run on failure
[32] :EndIf
```

Web Service APIs

Find the API description for the service

For example, search for "github api" or "openai api"

Authentication - some services may require an API key

For billing, usage tracking, security

Cost - some services are free, others have various billing models

Examples: OpenAI, Google Maps

Translating API Examples into `HttpCommand`

GET request parameters are in the query string of the URL

https://www.alphavantage.co/query?function=INTRADAY&symbol=IBM&interval=5min

```
req←HttpCommand.New 'get' 'https://www.alphavantage.co/query'
req.Params←'function' 'INTRADAY' 'symbol' 'IBM' 'interval' '5min'
```

OR

```
req.Params←('function' 'INTRADAY') ('symbol' 'IBM') ('interval' '5min')
```

OR

```
req.Params←3 2⍴'function' 'INTRADAY' 'symbol' 'IBM' 'interval' '5min'
```

OR

```
req.Params←⎕NS ''
req.Params.(function symbol interval)←'INTRADAY' 'IBM' '5min'
```

Translating API Examples into `HttpCommand`

POST, PUT, DELETE request parameters are in the body of the request

Build the equivalent request with `HttpCommand`

Create a new "POST" HTTP request to create a GitHub repository

```
req←HttpCommand.New 'post' 'https://api.github.com/user/repos'
```

Set the authentication for the request

```
req.(AuthType Auth)←'bearer' GitHubAPIToken
```

Set the API version header

```
'X-GitHub-Api-Version' req.SetHeader '2022-11-28'
```

Create parameters for the request

```
req.Params←⎕NS ''
req.Params.(name description)←'test-repo' 'test repository'
```

Generic Steps to Using an API

Once you've identified a web service, generally you will need to:

- Create a UserID
- Give some form of payment information for services that charge for use
- Generate an API key and define the scope of use for that API key
  - Keep your API key secure!
    DO NOT push them to a public repository!
- Use your API key in requests that need authorization

The GitHub API

We're going to the GitHub API in the coming exercises:

- GitHub UserID plusdottimes has been created for this workshop
- A personal access token has been created for your use
  This will allow us to read and write repositories in this account
- For security purposes, this UserID will be deleted following this workshop

GitHub Personal Access Tokens

GitHub has two types of Personal Access Tokens

Classic

- have access to all repositories and organizations that the user can access
- allowed to live forever

Fine-grained

- granular permissions with settings "no access", "read", or "read and write"
- can specify specific repositories
- have an expiration date

Exercise Time!

Web Service vs. 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

JARVIS Exercise 1

Create a Jarvis web service in 5 minutes or so…

What just happened?

We defined and started a web service

- Defined an "endpoint" (the `sum` function)
- Created the server using **`j←Jarvis.New ''`**
- Started the server using **`j.Run`**
- Used **`HttpCommand`** as a client
- Used a browser to open `Jarvis'` built-in HTML page that contains a JavaScript client to communicate with the web service

What happened under the covers?

JavaScript running in the browser created an XMLHttpRequest and sent the contents of the input window as its payload

Jarvis received the request and converted the payload to APL

Jarvis called the endpoint, passing the APL payload as its right argument

`sum` did its thing and returned an APL array as its result

Jarvis translated the result into JSON and sent it back to the client as the response payload

JavaScript in the client updated the output area on the page with the response payload

`Jarvis'` Two Paradigms

JSON

Endpoints are result-returning monadic or dyadic APL functions

All requests use HTTP POST

Request and response payloads are JSON

- Jarvis handles all conversion between JSON and APL

Use this when your endpoints are "functional"

REST

Write a function for each HTTP method your service will support (GET, POST, PUT, etc)

Each function will:

- Take the HTTP request as its right argument
- Parse the requested resource and query parameters/payload
- Take some appropriate action

Consider this when you are managing resources

GET requests are easier for the client

Jarvis' Two Paradigms - JSON

Client Request:
POST /GetPortfolio

{myid: 12345}

Server Code:

```
    ∇r←GetPortfolio payload
[1]  r←CalcPortfolio payload.myid
    ∇
```

Jarvis' Two Paradigms - REST

Client Request:
GET /Portfolio?myid=12345

Server Code:

```
     ∇r←GET req
[1] :Select req.EndPoint
[2]   :Case '/portfolio'
[3]      myid←2⊃⎕VFI req.QueryParameters req.GetHeader 'myid'
[4]      r←CalcPortfolio myid
[5]   :Case '/somethingelse'
[6]      ⍝ something else code
[7]   :Case '/yetanotherthing'
[8]      ...
```

JSON in Brief

JSON – JavaScript Object Notation

String: "this is a string"

Number: 42

Array: [1,2,"hello world"]

Object: {"name": "value"}

JSON in Brief

```
      ns←⎕NS ''
      ns.(name age)←'Dyalog' 40
      array←2 2⍴(2 2⍴⍳4)'Jarvis'('APL' 23)ns

      ⎕JSON⍠('HighRank' 'Split')⊢array
[[[[1,2],[3,4]],"Jarvis"],[["APL",23],{"age":40,"name":"Dyalog"}]]
```

CodeLocation

`CodeLocation`

- is where Jarvis will look for your Endpoint code.
- defaults to #
- can be the name of or reference to an existing namespace

```
j.Stop
'myApp' #.⎕NS '' ⍝ create a namespace
myApp.Rotate←⍵   ⍝ define an endpoint
j.CodeLocation←#.myApp ⍝ or '#.myApp'
j.Start
```

CodeLocation

CodeLocation can also be the name of a folder from where Jarvis will load your code into a namespace named **#.CodeLocation**.

This is similar to what `Link` does

If the folder is a relative file name, it will be relative to the path of:

- your workspace if you are running in a saved workspace
- your JarvisConfig file (we'll get to what this is in a couple slides)
- the `Jarvis` source file

CodeLocation

`Jarvis` will look in only **`CodeLocation`** and below for your code.

`Jarvis` will consider any result-returning functions that are monadic, dyadic, or ambivalent as endpoints for your service.

This includes functions in subordinate namespaces and public methods in classes. For example: **`#.CodeLocation.Utils.center`** is exposed as /Utils/center.

Filtering Endpoints

You can use **`IncludeFns`** and **`ExcludeFns`** to restrict what functions seen as endpoints.

Both can contain individual function names, simple wildcarded expressions, or regex (or any combination thereof).

```
j.ExcludeFns←'*.*' 'Δ*'
j.IncludeFns←'GetPortfolio' 'BuyStock'
```

In the case where there's a conflict between **`IncludeFns`** and **`ExcludeFns`**, **`ExcludeFns`** wins.

JarvisConfig File

You can specify all your Jarvis settings in a JSON or JSON5 file.

```
{
   "Port": 22321,
   "CodeLocation": "./myApp"
}
```

Debugging Jarvis

```
j.Debug←0  ⍝ Jarvis traps all errors
j.Debug←1  ⍝ Stop on error
j.Debug←2  ⍝ Stop before calling your code
j.Debug←4  ⍝ Stop after receiving request
j.Debug←8  ⍝ Log Conga events to the session
```

Codes are additive.

When Debug is 0, Jarvis has a "safety net" **`:Trap 0`** that will catch any untrapped errors in your endpoint code and report an HTTP 500 Internal Server Error status.

Debugging Jarvis

`ErrorInfoLevel` controls how much information is sent in the HTTP status message when an error occurs.

```
j.ErrorInfoLevel←0  ⍝ no additional information
j.ErrorInfoLevel←1  ⍝ APL error name
j.ErrorInfoLevel←2  ⍝ + fn[line] where error occurred
```

Optional Left Argument – Request Object

If your endpoint function is dyadic or ambivalent, Jarvis will pass the request object as the left argument.

The request object is the same for both JSON and REST paradigms.

The request object contains all the information from the client request as well as some useful functions to manipulate that information.

Debugging - Error Trapping Endpoints

How should you handle errors that occur in your endpoint code?

Options include:

- Let Jarvis' safety net handle it
- Trap or preemptively check for the error and
  - use `req.Fail` to convey the error
  - send information back in the response payload

Exercise Time – Jarvis Exercise 2

User "Hooks"

There are several points (hooks) in `Jarvis'` flow where you can inject custom behavior.

You specify these by setting a hook setting to the name of a function to execute.

`AppCloseFn` - called when `Jarvis` shuts down

`AppInitFn` - called when Jarvis starts

`AuthenticateFn` - called on every request to authenticate the request

`SessionInitFn` - called when a new session is initialized

`ValidateRequestFn` - called on every request to perform any other validation you need

HTTP Basic Authentication

`Jarvis` can use HTTP Basic authentication by default.
This can be disabled by setting **`HTTPAuthentication`** to `''`

Credentials are base64-encoded (not encrypted) and sent in the `Authorization` HTTP header

When using HTTP Basic authentication, Jarvis will set the request **`UserID`** and **`Password`** settings.

Once authenticated, browsers will send credentials with every subsequent request.

Other Forms of Authentication

You do not have to use HTTP Basic Authentication.

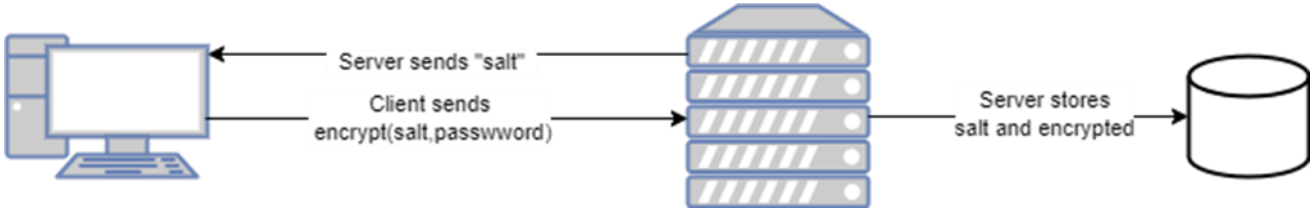
You can implement your own scheme – for instance passing credentials in the request payload, or using a token-based scheme with the Authorization header

Authenticating

`AuthenticateFn` specifies the name of a function to perform authentication.

`AuthenticateFn` should return a 0 if the authentication succeeds or is not necessary.

If you use HTTPS, you can safely transmit credentials in plaintext. Otherwise, you should be running on a network you trust or using salt and encryption to encrypt credentials.



Exercise Time – Jarvis Exercise 3

Maintaining State With Sessions

If you need to maintain state on the server between requests, Jarvis supports sessions by including a reference to a session namespace in the request object – **`req.Session`**

This means endpoints that need to reference the session MUST take a left argument (the request object).

If you specify **`SessionInitFn`**, it's the name of the function to perform session initialization.

The function specified by **`SessionInitFn`** takes the request object as its right argument.

```
j.SessionInitFn←'InitSession'

InitSession←{⍵.Session.RunningTotal←0}
```

Maintaining State With Sessions

Jarvis uses the following settings to control sessions:

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

`SessionIdHeader` – the name of the header field for the session token

`SessionUseCookie` - 0 = just use the header; 1 = use an HTTP cookie

`SessionPollingTime` - how frequently (in minutes) we should poll for timed out sessions

Exercise Time – Jarvis Exercise 4

HTMLInterface

`HTMLInterface` controls the JSON mode's HTML interface

`HTMLInterface` is turned on by default in JSON mode

```
HTMLInterface←0 ⍝ disable the HTML interface
HTMLInterface←1 ⍝ enable the built-in HTML page
HTMLInterface←'folder' ⍝ looks for folder/index.html
HTMLInterface←'file.html' ⍝ specific file
HTMLInterface←'' 'function' ⍝ function returns HTML code
```

AllowGETs

Okay, we lied... you can use HTTP GET if you set **`AllowGETs←1`**

The JSON payload is sent in the request query string

```
j.AllowGETs←1
HttpCommand.Get 'http://localhost:8080/sum?[5,6,7]'
```

AllowFormData

Okay, we lied again... the payload doesn't have to be JSON.

If a web page has a form enctype set to multipart/form-data

And `AllowFormData←1`, Jarvis will process the request.

This is useful for uploading files to Jarvis.

HttpCommand, Jarvis, and Conga

Conga is Dyalog's TCP/IP utility framework.

`HttpCommand` and `Jarvis` both use Conga.
So do other Dyalog utilities like `isolate`. So might your application itself.

If you have more than one component that uses Conga, the best practice is to copy the `Conga` namespace, preferably into # and then point each Conga-using utility to that copy.

Both `HttpCommand` and `Jarvis` have a `CongaRef` setting that can be set to point to the `Conga` namespace.

```
(HttpCommand Jarvis).CongaRef←#.Conga
```

Exercise Time – Jarvis/HttpCommand