Elsinore 2023

Web Services Workshop

15 October 2023

Brian Becker, Rich Park, Josh David

Agenda

- Introductions
- Goals
- Using Web Services - HttpCommand
- Break
- Building Web Services Part 1 - Jarvis
- Break
- Building Web Services Part 2 - WebSockets

Goals

- Learn enough about `HttpCommand` to call web services
- Learn enough about `Jarvis` to implement a simple JSON-based web service
- Learn enough about `WebSocketServer` to build a simple "Publish/Subscribe" interface for the client side of our web service

Disclaimers

- Sample application
  - Client side uses HTML, CSS, and JavaScript – we will not cover these in detail
  - Server side implements a very simple “portfolio” application
- HTTP vs HTTPS
  - Use HTTPS in any production environment that uses authentication or confidential data

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

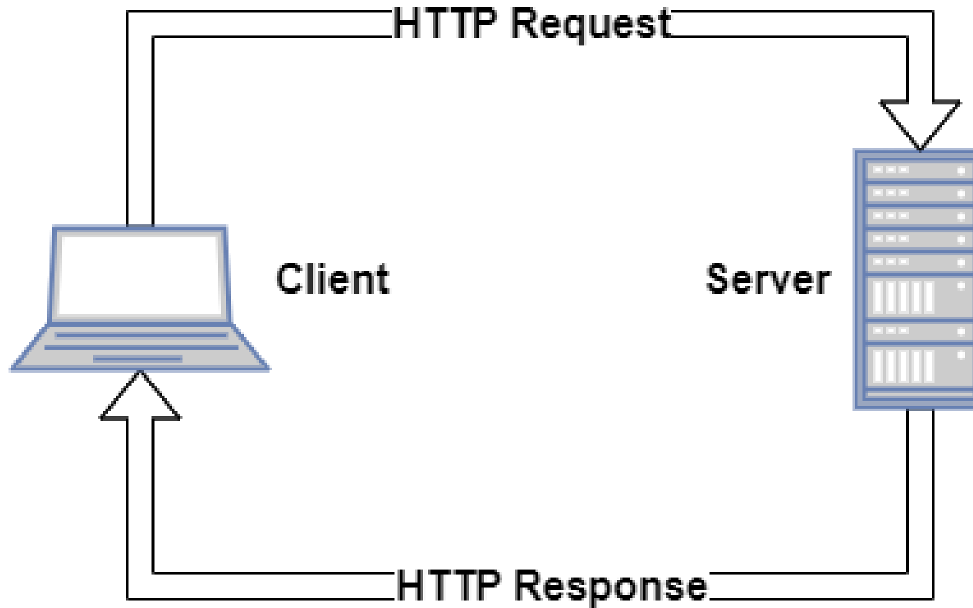
Client Examples:

A web browser, `HttpCommand`, cURL, JavaScript, Python

Server Examples:

IIS, Apache, Nginx, `Jarvis`, `DUI/MiServer`

HTTP Communications 101



Client Examples:

A web browser, `HttpCommand`, cURL, JavaScript, Python

Server Examples:

IIS, Apache, Nginx, `Jarvis`, `DUI/MiServer`

`HttpCommand`

`HttpCommand` is a utility that is well-suited to enable 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

Exercise 1: Obtaining `HttpCommand`

`HttpCommand` is bundled with Dyalog APL and can be loaded using `]load`

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

`HttpCommand.Upgrade` can obtain the latest released version, if one is available.
DO NOT use `HttpCommand.Upgrade` in production code as you won't know in advance if the new version has a major version change that potentially introduces a breaking change.

```
      HttpCommand.Upgrade
0  Upgraded to HttpCommand 5.3.6 2023-08-31 from HttpCommand ...
```

`HttpCommand` is documented online; `HttpCommand.Documentation` will display a link to the online documentation.

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

Your first HttpCommand

```
      ⊢ resp ← HttpCommand.Get 'dyalog.com'
[rc: 0 | msg:  | HTTP Status: 200 "OK" | ≢Data: 21783]

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

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

`resp` is a namespace that contains the response payload, if any, and metadata about the response.

`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

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

- request setting 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'
```

Anatomy of an HTTP Request

```
Method Endpoint HttpVersion
Headers

Body
```

Common HTTP Methods:

GET – read a resource

POST – update a resource

PUT – replace a resource

DELETE – delete a resource

PATCH – update a resource

Anatomy of an HTTP Request

```
Method Endpoint HttpVersion
Headers

Body

POST /user/repos HTTP/1.1
Host: api.github.com
User-Agent: Dyalog-HttpCommand/5.4.0
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"}
```

Common HTTP Methods:

GET – read a resource

POST – update a resource

PUT – replace a resource

DELETE – delete a resource

PATCH – update a resource

Anatomy of an HTTP Response

```
HttpVersion HttpStatus HttpMessage
Headers

Body
```

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
```

Using `HttpCommand`

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

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`.

Once you have created a request using `New`, you can specify any 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 ⍝ will return all settings for this request

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

Working with Headers

`HttpCommand` will generate several headers, unless you specify them yourself.

```
'header-name' req.SetHeader 'value' ⍝ unconditionally set a header
'header-name' req.AddHeader 'value' ⍝ set a header, if not already set
req.RemoveHeader 'header-name' ⍝ remove a header
req.Headers ⍝ contains the headers that you have set
'accept-encoding' req.SetHeader '' ⍝ suppress an HttpCommand default header
```

You can use `AuthType` and `Auth` to specify the Authorization header (or set the header directly)

You can use `ContentType` to specify the Content-Type header (or set the header directly)

req.TranslateData←1

Many web services return XML or JSON payloads.

Use `TranslateData←1` to automatically translate these `⎕XML` or `⎕JSON` as appropriate

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

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

      50↑resp.Data
[{"id":688060385,"node_id":"R_kgDOKQL34Q","name":"Public","full_name":"plusdotti

      req.TranslateData←1

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

      ↑resp.Data.(full_name created_at)
 plusdottimes/Public     2023-09-06T15:08:19Z
 plusdottimes/test-repo  2023-09-08T18:36:09Z
```

Using `HttpCommand`

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

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]
```

Using `HttpCommand`

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

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
```

Recap

1. Create an instance

```
[23]  req←HttpCommand.New 'get' 'someurl.com'
```

2. Configure your request

```
[24]  req.TranslateData←1
      [25]  'content-encoding' req.SetHeader ''
      [26]  req.MaxPayloadSize←200000
```

3. Send the request

```
      [27]  resp←req.Run
```

4. Inspect the response

```
[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 "google maps api"
- Authentication - some services may require an API key for usage tracking, billing, and to mitigate misuse.
  - GitHub authentication
- Cost - some services are free, others have a variety of billing models
  - Google Maps pricing

Translating API Examples into `HttpCommand`

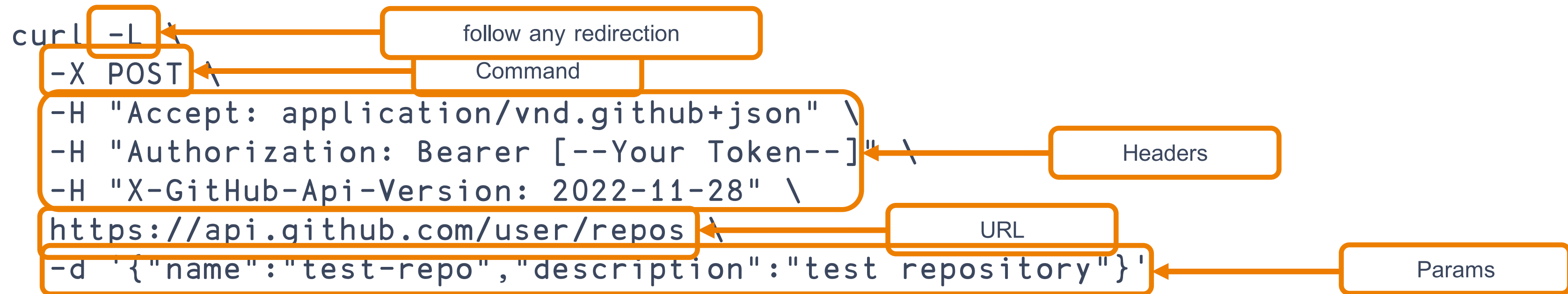
GET request parameters are in the query string of the URL

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

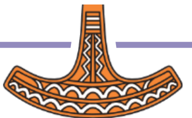
```
      req←HttpCommand.New 'get' 'https://www.alphavantage.co/query'

      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



Source: https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#create-a-repository-for-the-authenticated-user



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!
- 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 "fine-grained" personal access token has been created

  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
  - over 50 granular permissions that can be set to "no access", "read", or "read and write"
  - can specify specific repositories
  - have an expiration date

Exercise Setup

We need to get `GitHubAPIToken` for authenticated access to GitHub.

For the adventurous:

Connect to wireless network "WebService" with password: DyalogAPL

```
resp←HttpCommand.GetJSON 'post' '192.168.234.10?/get' 'GitHubAPIToken'
resp.IsOK
⎕FX resp.Data
```

For the not-so-adventurous:

Take one of the USB drives and:

```
]link.import # /SP3/HttpCommand
```

Exercise: Create a GitHub Repository

Because we'll be issuing several requests to the GitHub API, we can set up a request object that we can reuse by changing its settings. This will save us from having to re-specify a number of settings that will be common to all the requests we send.

```
h←HttpCommand.New ''
h.BaseURL←'https://api.github.com'
'X-GitHub-Api-Version' h.SetHeader '2022-11-28'
h.(AuthType Auth)←'bearer' GitHubAPIKey
h.TranslateData←1
```

Exercise: Create a GitHub Repository

Now that we have a request generically configured, we can specify the particular settings for to create a repository.

```
h.Command←'post'

h.URL←'user/repos'

p←⎕NS ''
p.(name description)←'your-repo-name' 'some description'

h.Params←p

h.Show

r←h.Run
```

Exercise: Update a GitHub Repository

```
h.Command←'patch'

h.URL←'repos/plusdottimes/your-repo-name'

p←⎕NS ''
p.(description visibility)←'new description' 'private'

h.Params←p

h.Show

r←h.Run
```

Exercise: Delete a GitHub Repository

```
h.Command←'delete'
h.URL←'repos/plusdottimes/your-repo-name'
h.Params←''
h.Show
r←h.Run
```

Exercises: (if we have time)

1. How many public repositories does the Dyalog organization have?

   Hint: it's not 30 – look at the per_page parameter

2. How many releases does Dyalog/Jarvis have?

3. Create a new repository and then create an issue for that repository.

JSON AND REST SERVICE

J A R VICE

JARVIS

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

`Jarvis` is a framework that makes it easy for an APLer to deploy applications as web services. How easy? Try this…

```
)clear
sum←+/
]load /SP3/Jarvis
j←Jarvis.New ''
j.Run
]load /SP3/HttpCommand
(HttpCommand.GetJSON 'post' 'localhost:8088/sum' (⍳10)).Data
]open http://localhost:8088
```

What just happened?

- We defined and started a web service
  - Defined an "endpoint" (the `sum` function)
  - Created (using `Jarvis.New)` and 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]      ...
```

Enough about REST… the rest of the workshop will focus on JSON

JSON in 3 Minutes

JSON – JavaScript Object Notation

String: "this is a string"

Number: 42

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

Object: {"name": "value"}

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

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

CodeLocation

`CodeLocation` is where Jarvis will look for your Endpoint code.

`CodeLocation` defaults to #

`CodeLocation` 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.

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

JarvisConfig File

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

JSON

```
{
  "Port": 22361,
  "CodeLocation": "./myApp"
}
```

JSON5

```
{
  Port: 22361,
  CodeLocation: "./myApp", // JSON5 allows comments
}
```

Filtering Endpoints

By default, `Jarvis` will see all result-returning, monadic, dyadic, and ambivalent functions in `CodeLocation` and all descendent namespaces as possible 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←'*.*' '∆*'
j.IncludeFns←'GetPortfolio' 'BuyStock'
```

Debugging Jarvis

```
      j.Debug←0  ⍝ Jarvis traps all errors (default setting)
      j.Debug←1  ⍝ Stop on error
      j.Debug←2  ⍝ Intentional stop before calling your code
      j.Debug←4  ⍝ Intentional stop after receiving request
Codes are additive.
    ∇ r←req oops payload
[1] ∘∘∘
    ∇
```

Optional Left Argument - Request

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.

```
AcceptEncodings  Body            Boundary      Charset
Complete         ContentType     ContentTypes  Cookies
Endpoint         ErrorInfoLevel  HTTPVersion   Headers
HttpStatus       Input           Method        Password
Payload          PeerAddr        PeerCert      QueryParams
Response         Server          Session       UserID
```

This means that some elements may not have meaning in one paradigm or the other.

For instance, in the JSON paradigm the `Method` is always 'POST'

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

Maintaining State With Sessions

If you need to maintain state between requests, Jarvis supports sessions using the following settings:

`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

`SessionCleanupTime` - how frequently (in minutes) do we clean up timed out session info

Exercise: Using Sessions

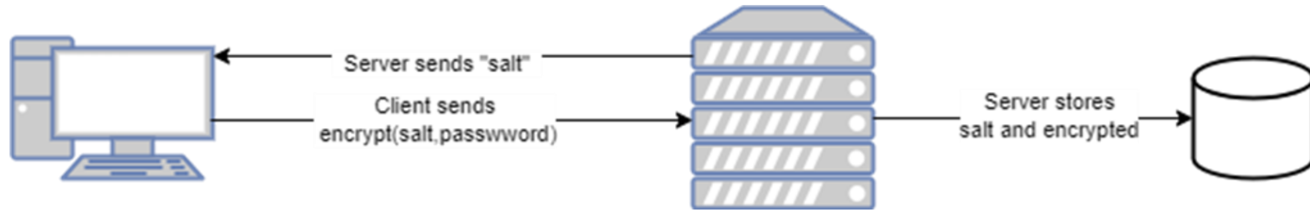
```
j.Stop
j.SessionTimeout←1 ⍝ 1 minute session timeout
j.SessionInitFn←'initSession'
j.SessionUseCookie←1
initSession←{⍵.Session.total←0}
add←{⍺.Session.Total ⊣ ⍺.Session.Total+←+/∊⍵}
j.Start
```

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.



Authenticating

`Jarvis` can use HTTP Basic authentication (using the `HTTPAuthentication` setting)

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

Browsers will send credentials with every subsequent request.

```
     ∇ r←Login req
[1]   ⍝ non-empty and UserID≡Password
[2]     r←(0∊⍴req.UserID)∨req.UserID≢req.Password
     ∇

      j.Stop

      j.AuthenticateFn←'Login'

      j.Start
```

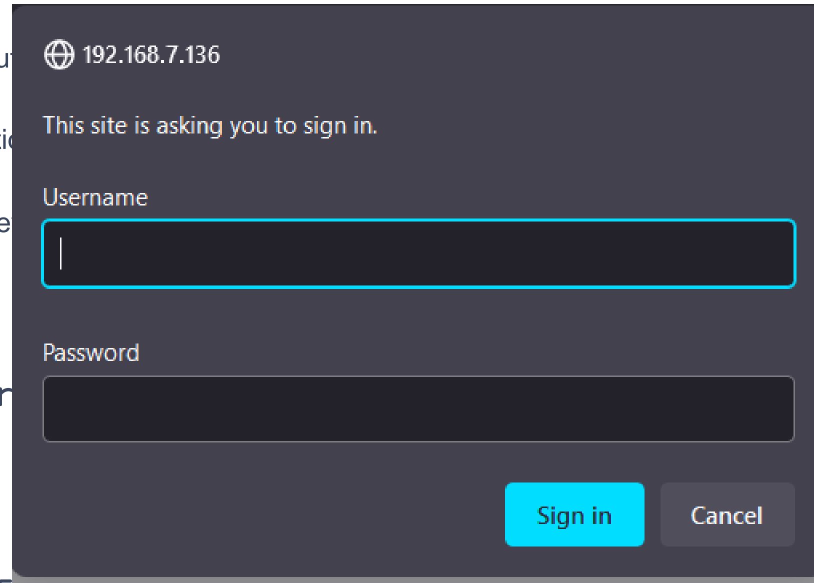
Authenticating

Jarvis can use HTTP Basic au   ng)

When using HTTP Basic authenticati

Browsers will send credentials with e

```
      ∇ r←Login req
[1]   ⍝ non-empty and
[2]     r←(0∊⍴req.User
      ∇

       j.Stop

       j.AuthenticateFn←'Login'

       j.Start
```



Jarvis Portfolio Service

This is a small, simple Jarvis service found in /SP3/Jarvis

It has a simple "database" defined in database.json5 that defines the users for the application (Huey, Dewey, and Louie) and the stocks (IBM, NVDA, and AAPL) that will be monitored.

It has 2 endpoints:

- Login – called after authentication
- Portfolio – calculates the user's portfolio value

It uses HTTP Basic authentication

It runs a simulation thread that triggers random stock price changes.

Jarvis Portfolio Service

Things to examine:

- `JarvisConfig.json5`
- `authenticate`
- `index.html index.js`
- `Portfolio`
- `Login`

Running the Jarvis Service

```
]load /SP3/Jarvis/Jarvis

]load /SP3/HttpCommand/HttpCommand

j←Jarvis.New '/SP3/Jarvis/JarvisConfig.json5'

j.Start

]open http://localhost:22335

h←HttpCommand.New 'post'
h.URL←'http://Huey:Huey@localhost:22335/Portfolio'
h.TranslateData←1
r←h.Run
```

Suppose…

- You have a web application with a HTML/CSS/JavaScript client.
- If you use standard HTTP requests, the only way to get updated information from the server is to ask for it.
- Wouldn't it be nice if the server could "push" updated information in real time without the client having to ask for it.
- WebSockets can accomplish precisely that (and more)

WebSockets

As we discussed earlier, HTTP requests originate from the client and wait for a response from the server.

A WebSocket is an upgraded HTTP connection that allows either the client or the server to send data to the end of the connection, without expecting a response.

WebSocket Uses

- PubSub (Publish/Subscribe) – clients can "subscribe" to a "channel". Whenever something "happens" on the channel, information is sent to all subscribers.
  This can be very useful when implementing real-time dashboards.
- RPC (Remote Procedure Call) – Suppose you have an endpoint for your web service that may run for a lengthy period of time. Rather than have the client wait for a response (and possibly time out), you can use a WebSocket to push the response whenever the endpoint finishes its task. This of this like an asynchronous Jarvis.

WSServer (WebSocket Server)

- This is relatively new work and will likely change in implementation, but not necessarily in how you, the application developer, will interact with it.
- I'd like to make it as easy to use as Jarvis.
- I'd like a better name for it.
- If we have time, I'd like to share some of my design ideas with you and get some feedback.
- Let's play with it and then see where that leads…

Portfolio Service a la WebSockets

```
)clear
]load /SP3/WSServer/*.dyalog
w←WSServer.New '/SP3/WSServer/WSSConfig.json5'
w.Start
]open file://c:/SP3/WSServer/index.html
```

WebSocket Portfolio Service

Things to examine:

- `database.json5`
- `WSSConfig.json5`
- `index.html index.js`
- `Portfolio.aplf`
- `Login.aplf`
- `Ticker.aplf`

Design Questions

- Currently WSServer is a 2-tiered architecture
  - A core (WSServer) that handles WebSocket connections, closures, etc.
  - A "paradigm" that implements either PubSub or RPC (or some other functionality)
  - I originally thought that PubSub and RPC were somewhat mutually exclusive, but I'm reconsidering that.
  - Look at `WSSConfig.json5`

Design Questions

- Jarvis + WSServer
  - I'm looking into adding WebSocket support within Jarvis. Then your web service may need to open only a single port. However, it may complicate Jarvis more than I'd like.
  - Perhaps they can run in concert with one another where Jarvis handles the incoming requests and WSServer serves only to push data out.

Design Questions

- WebSocket Protocol
  - The JavaScript WebSocket API hides a lot of the underpinnings of the WebSocket protocol.
  - Tools like Conga, JavaScript's XMLHttpRequest can make use of features not available through JavaScript.
  - Should we support the full protocol or will JavaScript be sufficient?