

Expanding Your Application's Audience With Jarvis

Brian Becker
APL Tools Architect
Dyalog, LTD

Agenda

- Introductions
- Goals
- HttpCommand
- Break
- Jarvis
- Break
- WebSockets

Goals

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

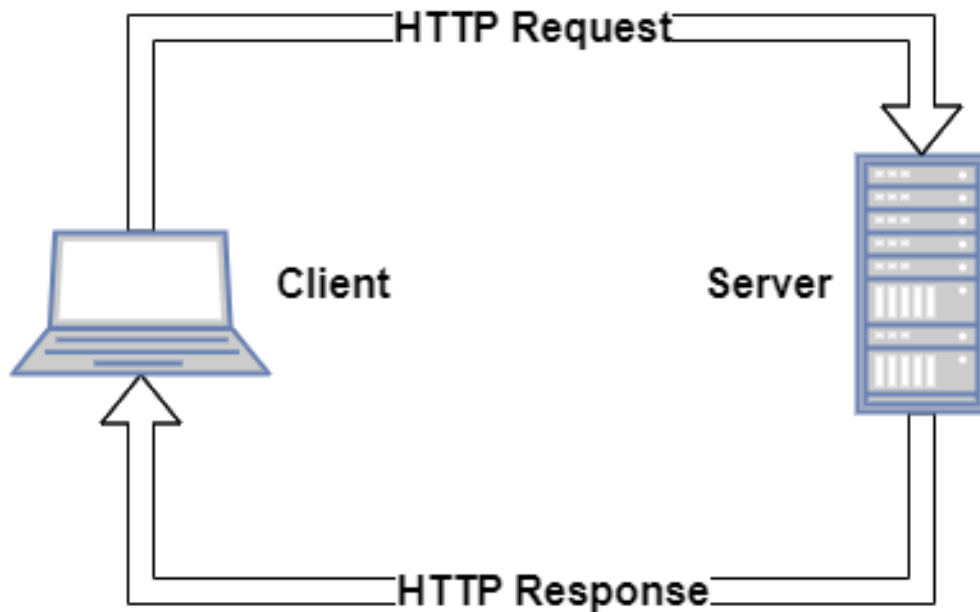
Client Examples:

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

Server Examples:

IIS, Apache, Nginx,
Jarvis,
DUI/Mi Server

HTTP Communications 101



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

Server Examples:
IIS, Apache, Nginx,
Jarvis,
DUI/Mi Server

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 ...
```

`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 3p␣nl -ι9)  
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 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'`

`req.Params←(name:'test-repo' ♦ description:'test repository')`

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

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
```

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

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/bpbecker/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":"plusdottimes/
  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
```

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` A contains the response headers

`resp.Data` A 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] A code to run on success
[30] :Else
[31] A 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

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 2p'function' 'INTRADAY' 'symbol' 'IBM' 'interval' '5min'
```

OR

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

OR

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

Translating API Examples into H t t pCommand

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

```
curl -L \  
  -X POST \  
  -H "Accept: application/vnd.github+json" \  
  -H "Authorization: Bearer [--Your Token--]" \  
  -H "X-GitHub-API-Version: 2022-11-28" \  
  https://api.github.com/user/repos \  
  -d '{"name":"test-repo","description":"test repository"}'
```

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

Translating API Examples into HttpCommand

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

```
curl -L \
  -X POST \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer [--Your Token--]" \
  -H "X-GitHub-API-Version: 2022-11-28" \
  https://api.github.com/user/repos \
  -d '{"name":"test-repo","description":"test repository"}'
```



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

Translating API Examples into HttpCommand

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

```
curl -L \
  -X POST \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer [--Your Token--]" \
  -H "X-GitHub-API-Version: 2022-11-28" \
  https://api.github.com/user/repos \
  -d '{"name":"test-repo","description":"test repository"}'
```



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

Translating API Examples into HttpCommand

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

```
curl -L \  
  -X POST \  
  -H "Accept: application/vnd.github+json" \  
  -H "Authorization: Bearer [--Your Token--]" \  
  -H "X-GitHub-API-Version: 2022-11-28" \  
  https://api.github.com/user/repos \  
  -d '{"name":"test-repo","description":"test repository"}'
```



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

Translating API Examples into HttpCommand

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

```
curl -L \  
  -X POST \  
  -H "Accept: application/vnd.github+json" \  
  -H "Authorization: Bearer [--Your Token--]" \  
  -H "X-GitHub-API-Version: 2022-11-28" \  
  https://api.github.com/user/repos \  
  -d '{"name":"test-repo","description":"test repository"}'
```

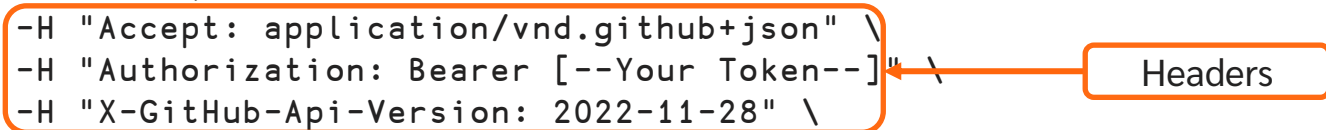
Params

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

Translating API Examples into H t t pCommand

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

```
curl -L \  
  -X POST \  
  -H "Accept: application/vnd.github+json" \  
  -H "Authorization: Bearer [--Your Token--]" \  
  -H "X-GitHub-API-Version: 2022-11-28" \  
  https://api.github.com/user/repos \  
  -d '{"name":"test-repo","description":"test repository"}'
```



The diagram highlights the header lines of the curl command: `-H "Accept: application/vnd.github+json" \`, `-H "Authorization: Bearer [--Your Token--]" \`, and `-H "X-GitHub-API-Version: 2022-11-28" \`. An orange arrow points from this group to a box labeled "Headers".

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

Exercises:

Using the GitHub API documentation see if you can use `HttpCommand` to determine:

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?

Using `dog.ceo`

1. How many dog breeds do they list?
2. Write a picture of your favorite dog breed to file

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

Load The Exercise Materials

- Unzip Jarvis.zip to a folder of your choosing
- We'll use /Jarvis/ for the examples in this workshop
(You should use whatever folder name you unzipped into)
- Using Dyalog v19.0 or later

```
]import # /Jarvis
```

Exercise: Make a JARVIS Web Service

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

```
)clear
sum←+∑
]load /Jarvis/Jarvis
j←Jarvis.New ''
j.Start
]load /Jarvis/HttpCommand
r←HttpCommand.GetJSON 'post' 'localhost:8080/sum' (10)
r
r.Data
]open http://localhost:8080
```

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

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 2p(2 2p14)'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 ''  a create a namespace  
myApp.Rotate←φ    a define an endpoint  
j.CodeLocation←#.myApp  a 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←'*. *' 'Δ*'
j.IncludeFns←'GetPortfolio' 'BuyStock'
```

Exercise: Limiting Your API

```
j.Stop
```

```
j.CodeLocation ← #
```

```
j.Start
```

Open your browser to localhost:8080

Look at the list of endpoints in the dropdown

Use IncludeFns and/or ExcludeFns to "hide" the functions in #.myApp

Debugging Jarvis

j.Debug←0 A Jarvis traps all errors (default setting)

j.Debug←1 A Stop on error

j.Debug←2 A Intentional stop before calling your code

j.Debug←4 A Intentional stop after receiving request

Codes are additive.

```
[1] ▽ r←req oops payload
    ▽
```

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.

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

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

1. Stop Jarvis, if it's running
2. Add sessions to the service
3. Write a new endpoint named Add that will add its argument to a running total kept in the session.
4. Use a 1 minute session timeout
5. Start Jarvis
6. Test Add

Exercise: Using Sessions

```
j.Stop
```

```
j.SessionTimeout←1  A 1 minute session timeout
```

```
j.SessionInitFn←'initSession'
```

```
j.SessionUseCookie←1
```

```
initSession←{ω.Session.Total←0}
```

```
add←{α.Session.Total → α.Session.Total+←+/εω}
```

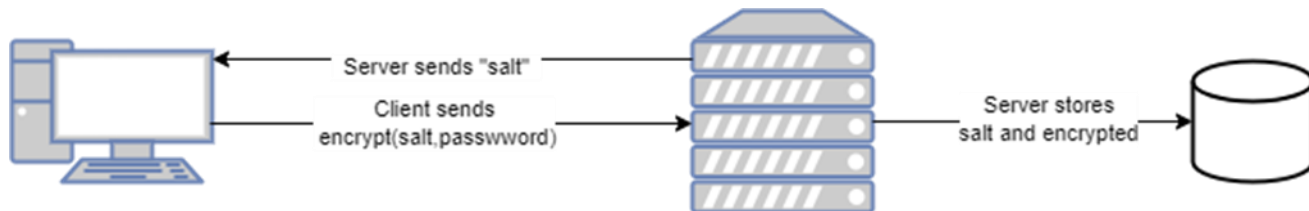
```
j.Start
```

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]  A 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 authentication setting)

When using HTTP Basic authentication, you must provide a Username and Password settings.

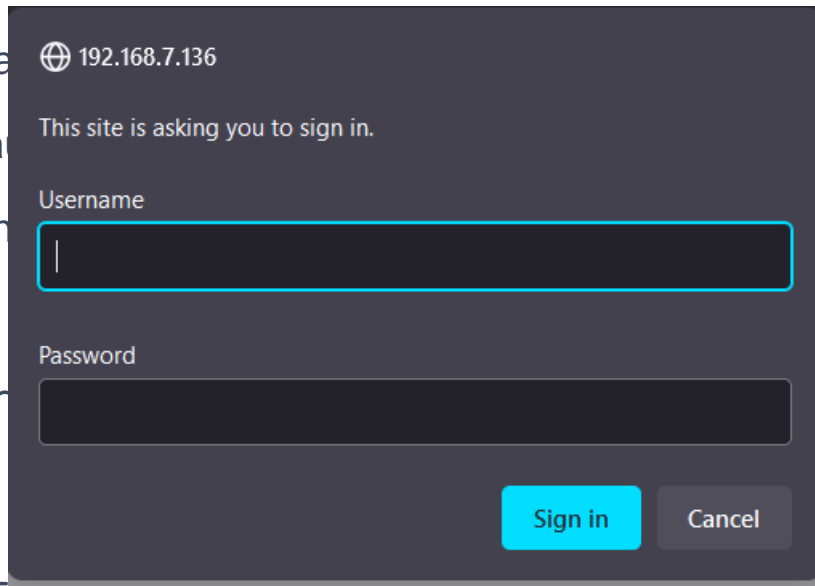
Browsers will send credentials in the request header.

```
▼ r←Login req
[1] A non-empty and
[2] r←(0∈preq.Username)
▼
```

```
j.Stop
```

```
j.AuthenticateFn← Login
```

```
j.Start
```

A screenshot of a web browser's HTTP Basic authentication dialog box. The dialog has a dark gray background. At the top, it shows a globe icon followed by the IP address '192.168.7.136'. Below this, it says 'This site is asking you to sign in.' There are two input fields: 'Username' and 'Password'. The 'Username' field is currently empty and has a red border. The 'Password' field is also empty. At the bottom right, there are two buttons: 'Sign in' (highlighted in red) and 'Cancel' (grayed out).

Exercise: Add Authentication

1. Write a simple authentication function
2. Use the UserID and Password request fields
3. The function should return a 0 if the authentication passes
4. Set `j.AuthenticateFn` to the name of your function
5. Start the server
6. Try the sample web page

Load and Examine the WebSocket Sample App

```
)clear
```

```
]import # /Jarvis
```

Last Exercise: Change the subscribe sample

- Change the subscribe function to instead return a random "dad joke" from <https://icanhasdadjoke.com>.

Jarvis Work In Progress

- ✧ Make Jarvis more "industrial"
 - ✧ Logging
 - ✧ HTTP logging
 - ✧ Error logging with email notification
 - ✧ Restartability
 - ✧ Remote monitoring / administration
 - ✧ Make cloud deployment easier (or at least document the process better)
- ✧ Finish the documentation!

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?