EWD.js

Reference Guide

Table of Contents

Introduction 1

Background 1

Installation & Configuration 3

Pre-requisites and Background 3

Node.js Interfacing To Mumps Databases 4

Caché & GlobalsDB 4

GT.M 4

Architecture 5

Node.js Interfacing to MongoDB 5

First Configuration Steps 5

ewd.js 6

ewd.js architecture 6

Installing ewd.js 7

Windows 7

Mac OS X or Linux 7

All Platforms 7

Setting up the EWD.js Environment 8

Using The EWD.js Directory Structure 9

Installing and Configuring the Database Interfaces 9

GlobalsDB 9

Caché 10

GT.M 10

Running ewd.js 11

Starting ewd.js 11

Defining EWD.js Startup parameters 12

Running EWD.js 13

Stopping ewd.js 14

The ewdMonitor Application 16

The ewdMonitor Application 16

Creating an EWD.js Application 18

Anatomy of an EWD.js Application 18

The HTML Container Page 19

The app.js File 19

Back-end Node.js Module 21

Form Handling 22

User Authentication Control 23

The ewd Object 24

Starting EWD.js Applications in a Browser 25

Debugging EWD.js Applications using Node Inspector 26

Pulling It All Together: Building an EWD.js Application 26

ewdMonitor Application: 26

VistADemo Application: 27

Micro-Service Support in EWD.js 28

What are Micro-Services? 28

Back-end Micro-Services 28

Front-end Micro-Services 30

Demonstration Example 31

Conclusion 34

Externally-generated Messages 35

Background 35

The External Message Input Interface 35

Defining and Routing an Externally-Generated Message 36

Messages destined for all users 36

Messages destined for all users of a specified EWD.js Application 36

Messages destined for all users matching specified EWD.js Session contents 36

Handling Externally-Generated Messages 37

Sending in Messages from GT.M and Caché Processes 37

Sending Externally-generated Messages from Other environments 38

JavaScript Access to Mumps Data 39

Background to the Mumps Database 39

EWD.js's Projection of Mumps Arrays 39

Mapping Mumps Persistent Arrays To JavaScript Objects 39

The GlobalNode Object 40

GlobalNode Properties and Methods 42

Examples 44

_count() 44

_delete() 44

_exists 44

_first 44

_forEach() 45

_forPrefix() 45

_forRange() 45

_getDocument() 46

_hasProperties 47

_hasValue 47

_increment() 47

_last 47

_next() 48

_parent 48

_previous() 48

_setDocument() 48

_value 49

Other functions provided by the ewd.mumps Object in EWD.js 49

ewd.mumps.function() 49

ewd.mumps.deleteGlobal() 50

ewd.mumps.getGlobalDirectory() 50

ewd.mumps.version() 50

Indexing Mumps Data 51

About Mumps Indices 51

Maintaining Indices in EWD.js 53

The globalIndexer Module 53

Web Service Interface 55

EWD.js-based Web Services 55

Web Service Authentication 55

Creating an EWD.js Web Service 56

Invoking an EWD.js Web Service 56

The Node.js EWD.js Web Service Client 57

Registering an EWD.js Web Service User 58

Programmatic Registrations 58

Converting Mumps Code into a Web Service 59

Inner Wrapper function 59

Outer Wrapper Function 59

Invoking the Outer Wrapper from your Node.js Module 60

Invoking as a Web Service 60

Invoking the Web Service from Node.js 60

Invoking the Web Service from other languages or environments 61

Turning off Web Service Security 61

Updating EWD.js 63

Updating EWD.js 63

Appendix 1 64

Creating a GlobalsDB-based EWD.js/Ubuntu System From Scratch 64

Background 64

Load and Run Installer 64

Start Up ewd.js 65

Run the ewdMonitor application 65

Appendix 2 66

Creating a GT.M-based EWD.js/Ubuntu 14.04 System From Scratch 66

Background 66

Load and Run Installer 66

Start Up ewd.js 67

Run the ewdMonitor application 67

Appendix 3 68

Installing EWD.js on a dEWDrop v5 Server 68

Background 68

Installing a dEWDrop VM 68

Step 1: 68

Step 2: 68

Step 3: 68

Step 4: 68

Step 5: 69

Step 6: 69

Step 7: 69

Step 8: 69

Step 9: 70

Updating a dEWDrop VM 70

Updating Your Existing EWD.js Applications 70

Start Up ewd.js 71

Run the ewdMonitor application 71

Appendix 4 72

A Beginner's Guide to EWD.js 72

A Simple Hello World Application 72

Your EWD.js Home Directory 72

Start the ewd.js Module 73

The HTML Page 73

The app.js File 75

Sending our First WebSocket Message 75

The helloworld Back-end Module 77

Adding a Type-specific Message Handler 77

Debugging Errors in your Module 78

The Type-specific Message Handler in Action 78

Storing our record into the Mumps database 79

Using the ewdMonitor Application to inspect the Mumps Database 79

Handling the Response Message in the Browser 81

A Second Button to Retrieve the Saved Message 83

Add a Back-end Message Handler for the Second Message 84

Try Running the New Version 84

Add a Message Handler to the Browser 84

Silent Handlers and Sending Multiple Messages from the Back-end 85

Using EWD.js with the Bootstrap Framework 87

The Bootstrap 3 Template Files 87

Helloworld, Bootstrap-style 87

Sending a Message from the Bootstrap 3 Page 92

Retrieving Data using Bootstrap 3 95

Adding Navigation Tabs 97

Conclusions 99

Appendix 5 100

Installing EWD.js on the Raspberry Pi 100

Background 100

First Steps with the Raspberry Pi 100

Installing Node.js 101

Installing ewd.js 101

Starting EWD.js on your Raspberry Pi 102

Installing MongDB 104

Installing the Node.js interface for MongoDB 104

Running EWD.js with MongoDB exclusively 105

Running EWD.js as a hybrid environment 105

Appendix 6 107

Configuring EWD.js for use with MongoDB 107

Modes of Operation 107

Node.js Synchronous APIs for MongoDB? 107

Using MongoDB exclusively with EWD.js 108

Creating a Hybrid Mumps/MongoDB EWD.js System 109

A Summary of the Synchronous MongoDB APIs 110

open(connectionObject) 110

insert(collectionName, object) 111

update(collectionName, matchObject, replacementObject) 111

retrieve(collectionName, matchObject) 111

remove(collectionName, matchObject) 111

createIndex(collectionName, indexObject, [params]) 111

command([params]) 112

version() 112

close() 112

Introduction

Background

EWD.js is an Open Source, Node.js / JavaScript-based framework for building high-performance browser-based applications that integrate with MongoDB and/or Mumps databases (eg Caché, GlobalsDB and GT.M). The EWD.js run-time environment operates as a Node.js application server/ container: in effect a JavaScript equivalent of Apache Tomcat.

EWD.js takes a completely new approach to browser-based applications, and uses WebSockets as the means of communication between the browser and the Node.js middle-tier. EWD.js requires the *ewd.js* module for Node.js. *ewd.js* provides a range of capabilities:

- it acts as a web server for serving up static content
- it provides the WebSockets back-end tier
- it manages and maintains a pool of Node.js-based child processes, each of which is responsible for integrating with the MongoDB and/or Mumps database that you've chosen to use
- it creates, manages and maintains user sessions
- it projects a Mumps database as a Native JSON Database, such that Mumps data storage can be treated as persistent JSON storage. You can also use EWD.js with MongoDB, either exclusively or together with a Mumps database. In the latter configuration, you can let the Mumps database provide very high-performance session management/persistence, whilst using MongoDB for all your other database requirements.
- it handles all the security needed to protect your database from unauthorised use
- it provides an application server environment in which multiple EWD.js applications can be run simultaneously, and in which applications can be edited and changed without the need for the application server to be restarted

EWD.js applications are written entirely in JavaScript, for both their front-end and back-end logic. No knowledge of the Mumps language is required. However, if you are using Caché or GT.M, you can access and invoke legacy Mumps functions from within your back-end JavaScript logic if necessary.

For background to the thinking and philosophy behind EWD.js and the unique capabilities of the Mumps database, see the many blog articles at http://robtweed.wordpress.com/

This document explains how to install and use EWD.js.

If you want to quickly create an EWD.js system to try out, read Appendices 1 to 3.

If you're new to EWD.js you should also spend some time reading and following the step-by-step tutorial in Appendix 4: it takes you through the process of building a simple "Hello World" EWD.js example. It should help you get to grips with the concepts and mechanics behind EWD.js, and appreciate the ease which a Mumps database stores and retrieves JSON documents. If you prefer to use MongoDB, you should consult Appendix 6.

Installation & Configuration

Pre-requisites and Background

An EWD.js environment consists of the following components:

- Node.js
- MongoDB and/or a Mumps database, eg:
  - GT.M
  - GlobalsDB
  - Caché
- The *ewd.js* module for Node.js

EWD.js can be installed on Windows, Linux or Mac OS X. Caché or GlobalsDB can be used as the database on any of these operating systems. GT.M can only be used on Linux systems. MongoDB is available for all three operating systems.

If you wish to use a Mumps database and need to choose which one to use, bear the following in mind:

- GlobalsDB is a free, but closed source product. However, it has no limitations in terms of its use or re-distribution. It is essentially the core Mumps database engine from Caché. Versions are available for Windows, Mac OS X and Linux. No technical support or maintenance is available for GlobalsDB from InterSystems: it is provided for use on an "as-is" basis. GlobalsDB is the quickest and simplest of the Mumps databases to install, and is probably the best option to use if you are new to EWD.js. Read Appendix 1 which describes how to create a fully-working GlobalsDB-based version of EWD.js in just a few minutes.
- GT.M is a free, Open Source, industrial-strength product. It is limited to Linux systems. Read Appendix 2 which describes how to create a fully-working GT.M-based version of EWD.js in just a few minutes. If you're interested in using EWD.js with the Open Source Electronic Healthcare Record (EHR) called VistA, then you may want to download the dEWDrop Virtual Machine (http://www.fourthwatchsoftware.com/) This will provide you with a pre-built, pre-configured GT.M system that includes a fully-working version of VistA. Follow the simple instructions in Appendix 3 to get EWD.js up and running on a dEWDrop VM.
- Caché is a proprietary and commercially-licensed industrial-strength product, with versions available for Windows, Mac OS X and Linux. EWD.js only requires its Mumps database engine, but if you're already a Caché user, you can execute any existing code or classes from within the JavaScript back-end code of your EWD.js applications. EWD.js therefore provides a great (and much simpler and lightweight) alternative to the CSP and Zen web frameworks that come with Caché, and uses your available Caché licenses much more efficiently.

For more details about these databases, see:

- GlobalsDB: http://www.globalsdb.org/
- GT.M: http://www.fisglobal.com/products-technologyplatforms-gtm

- Caché: http://www.intersystems.com/cache/index.html

All three databases are extremely fast, and both Caché and GT.M can scale to extremely large enterprise levels. The Node.js interface for GlobalsDB and Caché currently has about twice the speed of performance as the NodeM interface for GT.M.

Note that when you use EWD.js, all three Mumps databases appear to be identical in terms of how you access and manipulate data from within your application logic. The only difference you'll need to be aware of is a small difference in the configuration settings within the startup file you'll use for the *ewd.js* module (see later). Otherwise, EWD.js applications that you develop for, say, Caché, should normally work without any coding changes on a GT.M system.

If you want to use EWD.js with MongoDB, you have two choices:

- You can use MongoDB exclusively as the sole database. If you decide to use this approach, then EWD.js will manage and maintain user sessions using a MongoDB emulation of Mumps globals - all the JSON-based APIs for Mumps databases that are described in this document will work identically with MongoDB. However, you should be aware that the performance of this emulation is currently significantly slower that if you use a Mumps database. Provided you only use a very limited amount of EWD.js session storage, performance should be sufficient for most small to medium-sized applications. Of course, EWD.js allows you to use and access MongoDB in its standard way, as persistent collections of JSON objects, with the same level of performance that you'd normally expect from MongoDB.

- Alternatively you can use a hybrid approach, using a Mumps database to provide very high-performance session management, and allowing you to use MongoDB for any other database activities. In this mode you will use MongoDB in its normal way, as persistent collections of JSON objects, with the same level of performance that you'd normally expect from MongoDB. In this hybrid operation, you could also access legacy Mumps data at the same time as maintaining data in MongoDB: useful for modernising and extending legacy healthcare applications.

Depending on your choice of browser-side JavaScript framework, you'll of course need to also install it (eg ExtJS, jQuery, Dojo etc).

Node.js Interfacing To Mumps Databases

Node.js interfaces are available for all three Mumps databases.

Caché & GlobalsDB

InterSystems provides their own interface file for Caché and GlobalsDB. The same file is actually used for both products, and they can be freely interchanged. You'll find that versions of the interface file for the most recent versions of Node.js are included with the latest version of GlobalsDB. If you want to use Caché with Node.js version 0.10.x, for example, then download and install a copy of GlobalsDB (it's a very small download and very simple installation process) on a spare machine and copy the file you need to your Caché system.

Interface files are included with Caché 2012.x and later, but, due to the InterSystems release cycles, these files tend to be out of date. Additionally, the Node.js interface files can be used with almost all versions of Caché, including those pre-dating 2012.x, so you should be able to use EWD.js with most version of Caché.

You'll find the interface files in the bin directory of a Caché and GlobalsDB installation: they are named *cache[nnn].node* where *nnn* indicates the Node.js version. For example, *cache0100.node* is the file for Node.js version 0.10.x. This file has to be copied to the appropriate location (see later) and renamed to *cache.node*.

GT.M

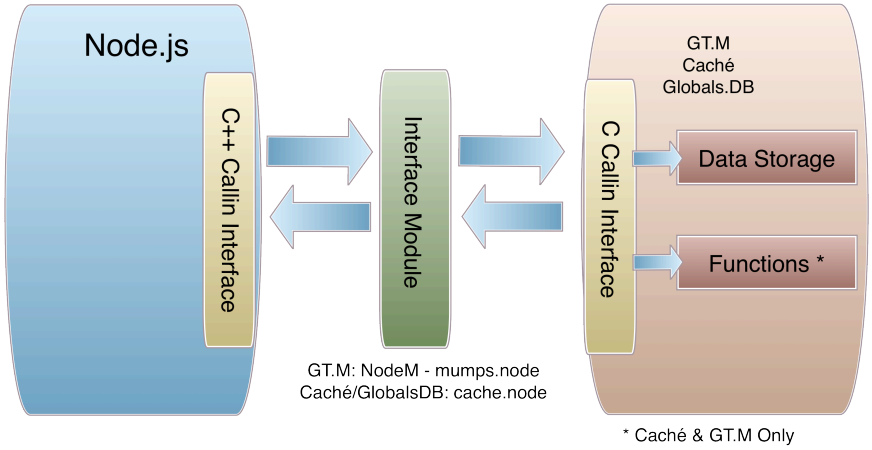
An Open Source Node.js interface to GT.M, named *NodeM*, has been created by David Wicksell. You'll find it at https://github.com/dlwicksell/nodem. This is fully-compatible with the APIs provided by the InterSystems interface for Node.js.

The dEWDrop VM already includes NodeM. Otherwise, follow the instructions for installing it on the Github page.

The NodeM interface file is named *mumps.node*.

Architecture

The architecture for the Node.js/Mumps database interfacing is as shown below:



Although fully asynchronous versions of the APIs exist in both versions of the interface, EWD.js uses the synchronous versions, as these are much simpler to use by developers and are significantly faster than their asynchronous counterparts. This might seem to fly in the face of accepted wisdom for Node.js/database access, but, as you'll see later, the architecture of the *ewd.js* module (on which EWD.js depends) ensures that this synchronous access isn't actually the problem that it might at first seem.

Node.js Interfacing to MongoDB

The request queue/pre-forked child-process pool architecture of the *ewd.js* module that underpins EWD.js (see the section below) is deliberately designed to allow database application developers to use synchronous logic, whilst still achieving the many benefits of Node.js. For that reason, EWD.js uses a synchronous interface module for MongoDB that has been specially developed by M/Gateway Developments Ltd. It provides a custom wrapper around the standard MongoDB APIs, so all the usual MongoDB functionality is available to you, and it connects to MongoDB in the standard way: via TCP to a specified address and port (localhost and port 27017 by default as usual).

The key difference is that you can write all your database handling logic using standard, intuitive synchronous logic, rather than the usual asynchronous logic that is the norm when working with Node.js.

First Configuration Steps

See Appendices 1 to 3 at the end of this document for instructions on automatically building a number of reference configurations - these are recommended for most users, particularly those new to EWD.js and Mumps databases.

If you need to install and configure EWD.js manually (eg for a custom environment), the first steps in creating an EWD.js environment are:

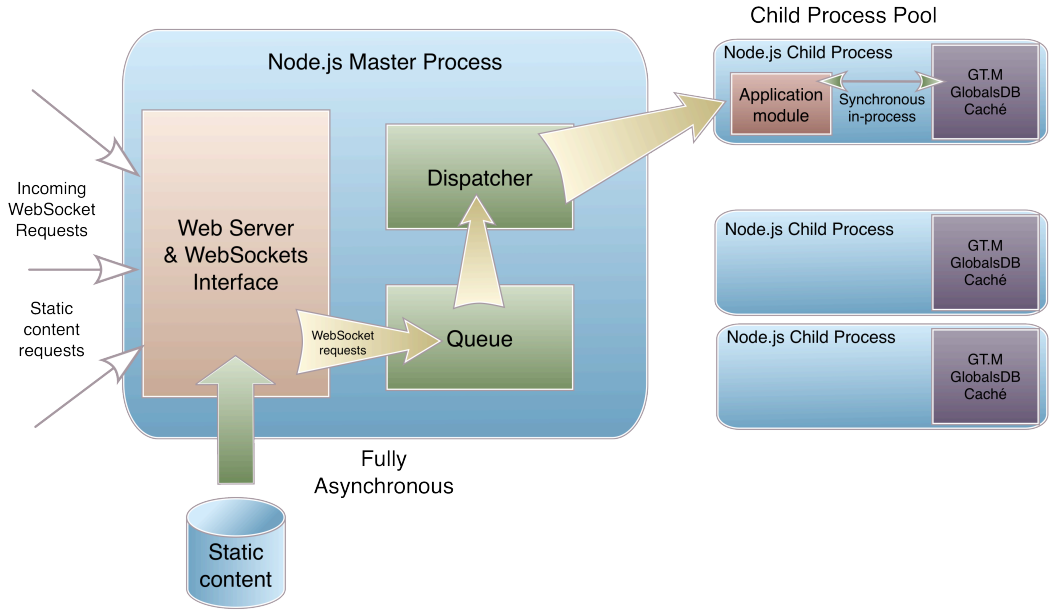
- Install Node.js: See http://nodejs.org/ EWD.js can be used with most versions of Node.js, but the latest version is recommended (0.10.x at the time of writing).
- Install MongoDB and/or your chosen Mumps database: see the relevant web-site for details.
- Install/configure the appropriate Node.js interface for your Mumps database.

ewd.js

The *ewd.js* module for Node.js provides the run-time environment for EWD.js. It is published and maintained as an Apache 2 licensed Open Source project at https://github.com/robtweed/ewd.js

ewd.js architecture

The architecture for the *ewd.js* module is summarised in the diagram below.



The master Node.js process acts as a web-server and websocket server. When used with EWD.js, web-server requests are limited to requests for static content. Your *ewd.js* startup/configuration file will specify a *poolsize*: the number of Node.js child processes that will be started up. Each child process will automatically create a connection to MongoDB and/or a Mumps database process. Both the InterSystems interface and NodeM create in-process connections to the database. Note that if you are using Caché, each child process will consume a Caché license. If you are using MongoDB, the connection uses the standard TCP-based connection, but the APIs are synchronous.

Incoming websocket messages from your application are placed on a queue which is processed automatically: if a Node.js child process is free (ie not already processing a websocket message), a queued websocket message is sent to it for processing. *ewd.js* ensures that a child process only processes a single websocket message at a time by automatically removing it from the available pool as soon as it is given a message to process. As soon as it has finished processing, the child process is immediately and automatically returned to the available child process pool.

Processing of WebSocket messages is carried out by your own application logic, written by you, in JavaScript, as a Node.js module. Your module has full access to MongoDB and/or your Mumps database, and, if you are using GT.M or Caché, can also invoke legacy Mumps functions.

By decoupling access to the Mumps database from the front-end master Node.js process, *ewd.js* achieves two things:

- it allows the architecture to scale appropriately, and the child processes can make use of the CPUs of a multiple-core processor.
- it allows use of the synchronous APIs within the Node.js/Mumps or Node.js/MongoDB interface, making coding easier and more intuitive for the application developer, and benefitting from the fact that the synchronous APIs perform significantly faster than the asynchronous ones. Complex database manipulation logic becomes very straightforward,

without any need for deeply-nested callbacks. You can, nevertheless, use standard asynchronous coding for any other logic within your back-end module - EWD.js really does give you the best of all worlds.

EWD.js is specifically designed to provide you with a framework for developing applications that make use of the *ewd.js* module..

Installing ewd.js

Installing EWD.js is very straightforward and largely automated for most configurations. See Appendices 1 to 3 at the end of this document.

If you need to create your own custom installation that differs from the ones covered by the Appendices, then this section explains the "moving parts" and how they need to be installed.

You should use the Node Package Manager (*npm*) which is installed as part of Node.js.

Note: if you haven't used Node.js before, it's a product that you mainly manage and control via command-line prompts. So you'll need to open up either a Linux or OS X terminal window, or a Windows Command Prompt window to carry out the steps described below.

Before you install *ewd.js*, you should make sure you're in the directory path where you'll want EWD.js to run. This will depend on your OS, Mumps database and your personal choice. It is recommended that you create a subdirectory named *ewdjs* under the directory path of your choosing, navigate to that directory and then invoke *npm* to install the *ewd.js* module. We'll refer to the *ewdjs* directory as your *EWD.js Home Directory*. For example:

Windows

Create the *ewdjs* directory in, for example, your C drive and install from within it, eg:

```
cd c:\ewdjs
npm install ewdjs
```

Your EWD.js Home Directory will be *c:\ewdjs*

Mac OS X or Linux

Create the ewdjs directory under your home directory and install from within it, eg:

```
cd ~/ewdjs
npm install ewdjs
```

Your EWD.js Home Directory will be *~/ewdjs*

All Platforms

During the installation process, you'll be asked to confirm the path in which EWD.js has been installed. Just accept the default that it suggests by pressing the Enter key, ie:

```
Install EWD.js to directory path (/home/ubuntu/ewdjs):
```

The essential sub-components of EWD.js will be installed relative to this path.

You'll then be asked if you want to install the extra, optional sub-components for EWD.js:

```
EWD.js has been installed and configured successfully

Do you want to install the additional resources from the /extras directory?
If you're new to EWD.js or want to create a test environment, enter Y
If you're an experienced user or this is a production environment, enter N
Enter Y/N:
```

If you're new to EWD.js, typing *Y* (followed by the *Enter* key) is recommended.

In all cases, after *npm* has finished installing the *ewdjs* module, you'll find that a sub-directory named *node_modules* has been created under your EWD.js Home Directory. This contains the *ewd.js* module and supporting files and modules needed by EWD.js

Setting up the EWD.js Environment

If you've used one of the automated install scripts for EWD.js (as described in Appendices 1 to 3), then the EWD.js environment will have been set up for you, in which case you can ignore this section. If you want to know more about how the EWD.js environment is created and/or need to create a custom installation, this section will provide the information you require.

As described in the previous section above, when you installed EWD.js, the install script asked you to enter / confirm the path that provided the baseline for moving essential and optional extra files. You'll now find that the following directory path structure has been created under your EWD.js Home Directory:

```
- node_modules
    o ewdjs (directory containing the ewd.js module)
    o a number of Node.js module files

- www
    o ewdjs (a directory containing the components of EWD.js that run in the
      browser)

    o ewd (a directory containing a number of pre-build EWD.js applications.
      All browser-side markup, JavaScript and CSS for your applications will
      reside under this directory)

    o respond (contains files used for IE support of EWD.js / Bootstrap appli-
      cations)

    o services (may contain front-end micro-service files)

- ssl (This directory is where you should copy your SSL key and certificate file
  if required. If you installed the optional extras, a self-signed certificate is
  installed for you to use in testing)
    o ssl.key
    o ssl.crt

If you installed the optional extras you'll find a set of sample startup files in the
home directory, eg:

- ewdStart-cache-win.js
- ewdStart-globals.js
- ewdStart-globals-win.js
- ewdStart-gtm.js
- ewdStart-mongo.js
- ewdStart-pi.js
- test-gtm.js
```

Using The EWD.js Directory Structure

The directory structure that EWD.js has created (see section above) should be used as follows:

~/ewdjs (ie your EWD.js Home Directory): Your EWD.js startup files should reside here. If you need to install any additional Node.js modules, you should run *npm* from within this directory.

~/ewdjs/node_modules: This is where all back-end modules for your EWD.js applications should reside. You'll already have some examples present such as ewdMonitor.js which provide the back-end logic for the EWD.js Manager/Monitor application. Any other Node.js modules that are required by your applications should also be installed into this directory (see note above).

~/ewdjs/ssl: Use this directory for your SSL key and certificate files (if required). EWD.js installs a self-signed certificate that you can use for testing EWD.js with SSL. Substitute these files with proper ones for production use.

~/ewdjs/www: This directory acts as the web server root path for EWD.js. All files, paths and sub-paths within this directory will be accessible from browsers and remote clients. You should install any JavaScript libraries that you want to use in your applications within this directory, eg *~/ewdjs/www/bootstrap-3.0.0*

~/ewdjs/www/ewdjs*:* This directory contains the JavaScript files that create the EWD.js run-time environment in the browser. Do not modify any of these files that are included in the EWD.js installation.

~/ewdjs/www/ewd: All your EWD.js application sub-directories reside in this directory. During installation a number of pre-built ones will already have been created for you (eg *ewdMonitor, VistADemo*). You can add as many further application sub-directories as you wish. Each application sub-directory contains:

- a main "container" HTML file, normally named *index.html*
- optionally, one or more "fragment" HTML files
- a JavaScript file that defines the application's browser-side dynamic functionality. This file is normally named *app.js*
- optionally any other JavaScript files and/or CSS files required by the application.

~/ewdjs/www/respond: This directory is created for you during installation and contains JavaScript files that are used for Bootstrap support on Internet Explorer browsers.

~/ewdjs/www/services: This directory is created for you during installation and contains an example front-end Micro-Service JavaScript module.

Installing and Configuring the Database Interfaces

GlobalsDB

When you install GlobalsDB, you'll find the interface module files in its installation's */bin* directory - it is named *cache0100.node*. You should copy this file to the *~/ewdjs/node_modules* directory and, **most importantly** rename it to *cache.node*.

Caché

Depending on the version of Caché that you're using, you may or may not be able to find a *cache0100.node* file in your */Cache/bin* directory. If it is present, you should copy it to the *~/ewdjs/node_modules* directory and, **most importantly** rename it to *cache.node*.

If it isn't present, then the easiest thing to do is install a copy of GlobalsDB somewhere unimportant, eg on a virtual machine - just make sure it's for the same platform as your Caché system. Find the *cache0100.node* file in the */globalsdb/bin* directory. You should copy this file to the *~/ewdjs/node_modules* directory and, **most importantly** rename it to *cache.node*. The *cache0100.node* file from GlobalsDB should work with most early versions of Caché.

GT.M

If you haven't used the automated installer for GT.M, you should install the NodeM module:

```
cd ~/ewdjs
npm install nodem
```

You will need to configure NodeM. If you've used the automated installers for GT.M, then this will have been done for you. Otherwise you can take a look in the second install script to see what's needed: https://github.com/robtweed/ewd.js/blob/master/gtm/install2.sh

In summary. the steps are as follows:

- In the directory *~/ewdjs/node_modules/nodem/lib* you'll find a number of files: these are interface modules for different versions of Node.js and for 32-bit and 64-bit Linux. If you're using 32-bit Linux, you need to select the file *mumps10.node.i686*. If you're using 64-bit Linux, you need to select the file *mumps10.node.x8664*. Rename the selected file to *mumps.node*. Make sure *mumps.node* still resides in the *~/ewdjs/node_modules/nodem/lib* path.

- In the directory */usr/local/lib* et up a symbolic link to the file *libgtmshr.so* that you will find in your GT.M installation directory, eg:

```
sudo ln -s /usr/lib/fis-gtm/V6.0-003_x86_64/libgtmshr.so /usr/local/lib/libgtmshr.so
sudo ldconfig
```

- Ensure that the *GTMCI* environment variable is created and pointing to the path *~/ewdjs/**node_modules/nodem/resources/calltab.ci***

- Ensure that the *gtmroutines* environment variable is extended to include the path *~/ewdjs/node_modules/nodem/src*

Note: for the latter two steps, you can do this at runttime by using the module file *dewdrop-config.js* if you wish. See later.

Running ewd.js

Starting ewd.js

You can now start up the *ewd.js* Node.js environment. You do this by using the *ewdStart*.js* file that is appropriate to the Mumps database and OS you're using, eg:

- GlobalsDB on Linux or Mac OS X: *ewdStart-globals.js*

```
cd ~/ewdjs
node ewdStart-globals
```

- GlobalsDB on Windows: *ewdStart-globals-win.js*

```
cd c:\ewdjs
node ewdStart-globals-win
```

- GT.M, installed as per Appendix 2: *ewdStart-gtm.js*

```
cd ~/ewdjs
node ewdStart-gtm gtm-config
```

- GT.M running in a dEWDrop VM as per Appendix 3: *ewdStart-gtm.js*

```
cd /home/vista/ewdjs
node ewdStart-gtm dewdrop-config
```

- Other startup files have been created for you to use or customise appropriately:
  - ewdStart-cache-win.js: EWD.js + Windows + Caché for Windows
  - ewdStart-pi: EWD.js running in a Raspberry Pi (see Appendix 5)
  - ewdStart-mongo.js: EWD + Windows + MongoDB (using MongoDB emulation of Mumps globals)
- If you'd like to use MongoDB in a hybrid environment, see Appendix 6. Essentially the EWD.js environment is focused on the Mumps database/OS you're using, and MongoDB can then be added for all or selected EWD.js applications you create.
- If you're using a different configuration from the one I've described, you'll need to create a copy of the appropriate startup JavaScript file and edit it appropriately. If you're using Caché or GlobalsDB, use *ewdStart-globals.js* or *ewdStart-globals-win.js* as your starting point. If you're using GT.M, use *ewdStart-gtm.js* as your starting point.

Defining EWD.js Startup parameters

If you look at any of the EWD.js startup files described above, you'll see that they all follow a common pattern:

- The *ewd.js* module is loaded using *require()*
- A startup parameter object is defined
- The ewd.js *start()* method is invoked, passing the startup parameter object as an argument.

When EWD.js is started (by invoking its *start()* method) it first creates a default set of startup parameters. Any values defined in your startup parameter object are then used to override the default values. In most situations you can accept the majority of EWD.js's default parameter values, and you just need to specify a small number of values that are specific to your environment. You can override any of the default values which are defined as follows:

```
    var defaults = {
      database: {
        type: 'gtm',
      },
      httpPort: 8080,
      https: {
        enabled: false,
        keyPath: cwd + '/ssl/ssl.key',
        certificatePath: cwd + '/ssl/ssl.crt',
      },
      webSockets: {
        socketIoPath: 'socket.io',
        externalListenerPort: 10000
      },
      logFile: 'ewdLog.txt',
      logTo: 'console',
      modulePath: cwd + '/node_modules',
      monitorInterval: 30000,
      poolSize: 2,
      traceLevel: 1,
      webServerRootPath: cwd + '/www',
      webservice: {
        json: {
          path: '/json'
        },
        authentication: true
      },
      management: {
        path: '/ewdjsMgr',
        password: 'makeSureYouChangeThis!'
      }
    };
```

So, for example, to start up EWD.js using SSL and listening on port 8081, you'd have to set the following override startup parameter values:

```
var params = {
  httpPort: 8081,
  https: {
    ssl: true
  }
};
```

All other startup values would remain as their default values.

The meaning of the available startup parameters is as follows.

- **database.type**: gtm | cache | mongodb (Note: the value *cache* is used for both Caché and GlobalsDB)
- **httpPort:** the port on which the EWD.js web server will listen
- **https.enabled**: true | false
- **https.keyPath**: the path for your SSL key file if *https.enabled* is *true*
- **https.certificatePath**: the path for your SSL certificate file if *https.enabled* is *true*
- **poolSize**: the number of Child Processes that will be pre-forked when EWD.js starts up. The EWDMonitor application can be used to adjust this while EWD.js is running.
- **webServerRootPath**: the physical path that is mapped by EWD.js to the Web Server root. Note that all files and subdirectories of this path are accessible from browsers and HTTP(S) clients. It is recommended that you use the default setting in most circumstances
- **webSockets.externalListenerPort**: the TCP port on which EWD.js will listen for incoming external messages. Set to *false* to disable this port and its functionality
- **logTo**: the destination for EWD.js's logging information (console | file)
- **logFile**: if *logTo* is set to file, this defines the file name to which logs are written by EWD.js
- **traceLevel**: the level of detail of EWD.js's logging. 0 = none, 1 = low, 2 = medium, 3 = verbose
- **modulePath**: the path in which EWD.js module files reside. This is used to construct the *require()* path for your back-end application modules. It is strongly recommended that you use the default setting under most circumstances
- **monitorInterval:** used by the EWDMonitor application to specify the number of milliseconds between each information update
- **webService.json.path**: the URL path prefix that is used by EWD.js to recognise incoming Web Service requests
- **management.path**: the URL path prefix that is used by EWD.js to recognise incoming HTTP-based EWD.js management requests
- **management.password**: access to the EWDMonitor application and to HTTP-based management requests requires authentication using this password. ***Make sure that production systems and/or EWD.js systems that are publicly accessible have a unique password defined***. Only use the default password for initial testing.

Running EWD.js

When you start up the *ewd.js* module, don't be surprised by the amount of information it writes out: this is because the default logging level (see the traceLevel parameter in the previous section) has been set to the maximum value of 3. You can reset this any integer value between 0 (no logging) and 3 (maximum) either in your startup parameters object or using the EWDMonitor application while EWD.js is running,

EWD.js will try to recognise any problems due to mis-configuration and will display an appropriate error message if possible and will stop. If this happens, look for any diagnostic or advisory messages in the EWD.js log, modify the configuration appropriately and try restarting EWD.js.

Usual reasons for EWD.js not starting include:

- using the wrong NodeM or *cache.node* interface file for your architecture
- insufficient privileges to run EWD.js (usually restricted to GlobalsDB-based systems created using Mike Clayton's installer).
- path mismatches in the startup parameters object

Stopping ewd.js

If you're running *ewd.js* in a terminal window, you can simply type *CRTL & C* to safely stop the master and child processes.

You can also shut down the *ewd.js* process using the following three methods:

- Start and log on to the *ewdMonitor* application (see next chapter) and click the *Stop Node.js Process* button that you'll see above the *Master Process* grid.
- If you are running EWD.js as a service, and/or you want to shut down the EWD.js processes from another process:

  a) Identify the pid of master process. The easiest way is to type:

```
ps -ax | grep node
```

  Look for the one that refers to your startup file. eg in the example below, it's the first one (pid 12259), referring to *ewdStart-globals*:

```
ubuntu@ip-172-30-1-88:~/ewdjs$ ps -ax | grep node

12259 pts/0    Sl+    0:21 node ewdStart-globals

12261 pts/0    Sl+    0:04 /home/ubuntu/.nvm/v0.10.35/bin/node
/home/ubuntu/ewdjs/node_modules/ewdjs/lib/ewdChildProcess.js
19064 pts/1    S+     0:00 grep --color=auto node
21294 ?        Sl     0:03 /home/ubuntu/.nvm/v0.10.35/bin/node
/home/ubuntu/ewdjs/node_modules/ewdjs/lib/ewdChildProcess.js
23238 ?        Sl     0:04 /home/ubuntu/.nvm/v0.10.35/bin/node
/home/ubuntu/ewdjs/node_modules/ewdjs/lib/ewdChildProcess.js
```

  b) Now simply invoke a kill command for that pid, eg:

```
kill 12259
```

  Doing so sends a SIGTERM signal to the master EWD.js process, which initiates the graceful shutdown procedure. Note that invoking a kill for any of the child processes will be ignored by that child process.

- Send an HTTP(S) request to the *ewd.js* process, of the following structure:

  http[s]://[ip address]:[port]/ewdjsMgr?password=[management password]&exit=true

  Replace the items in square brackets with values appropriate to your *ewd.js* instance. The management password is the one defined in the *ewd.js* startup file (eg *ewdStart*.js)* (By default this is set to *keepThisSecret!,* but for obvious reasons, it is strongly recommended that you change the password to something else). For example:

  https://192.168.1.101:8088/ewdjsMgr?password=keepThisSecret!&exit=true

All these methods have the same effect: the *ewd.js* master process instructs each of its connected child processes to cleanly close the Mumps database. As soon as all the child processes have done so, the master process exits which, in turn, causes the child processes to also exit.

The ewdMonitor Application

The ewdMonitor Application

Included in the *ewd.js* installation kit is a ready-made EWD.js application named *ewdMonitor.* This application serves two purposes:

- it provides a good example of an advanced Bootstrap 3-based EWD.js application
- it provides you with an application through which you can monitor and manage the *ewd.js* module, and with which you can inspect various aspects of your EWD.js environment and Mumps database.

You start the application in a browser by using the URL:

```
http://127.0.0.1:8080/ewd/ewdMonitor/index.html
```

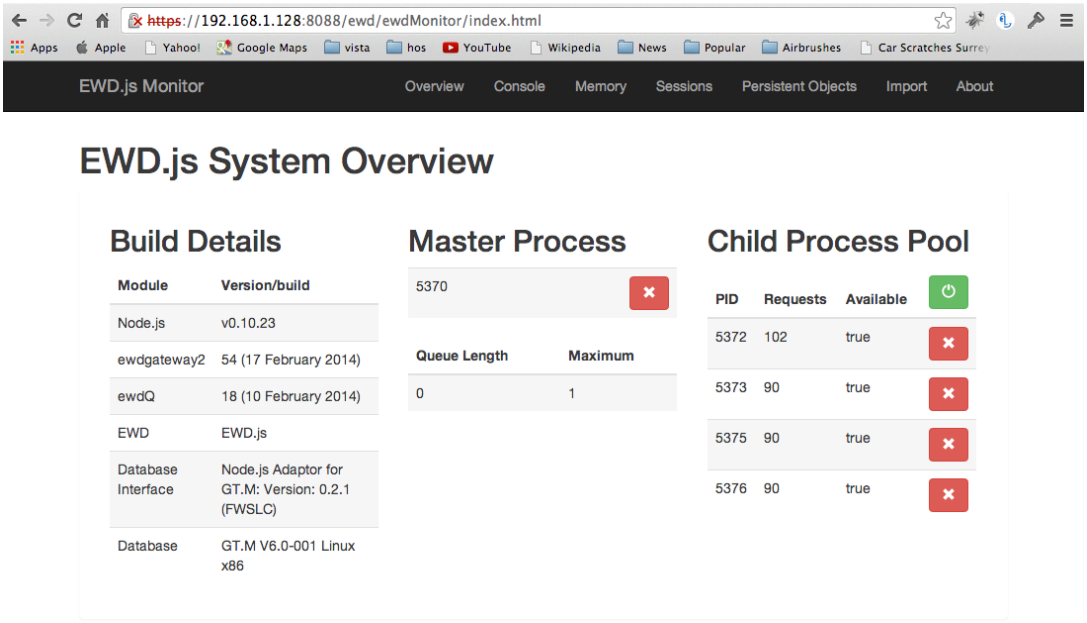
Change the IP address or host name and port appropriately.

If you've modified the EWD.js startup parameters to specify SSL, you'll need to change the URL above to begin with *https://* and you'll probably get a warning about the self-certificated SSL keys used by your *ewd.js* web server: tell it that it's OK to continue.

You'll be asked for a password: use the one specified in the *ewd.js* startup file, ie as specified in this section:

```
management: {
  password: 'keepThisSecret!'
}
```

Enter this password and the application should burst into life and should look something like this:



In the top panel you'll see basic information about the Node.js, ewd.js, and database environments, including statistics about the master Node.js process and the number of requests handled by each of the child processes.

One important feature is the *Stop Node.js Process* button in the Master Process panel. You should always try to use this to shutdown the *ewd.js* process and its associated connections to the Mumps database processes.

Other functionality provided by this application include:

- hovering over the process Ids brings up a panel that displays the current memory utilisation of the process. For Child Processes, you'll also see the list of application modules that are currently loaded
- a live console log: useful if you're running *ewd.js* as a service
- a live chart, showing memory usage by the *ewd.js* master and child processes: useful to check for memory leaks
- a table showing currently active EWD.js sessions. You can view each session's contents and/or terminate sessions
- a tree menu that allows you to view, explore and optionally delete the contents of your Mumps database storage
- an import option for importing data into your Mumps database
- options for changing the logging/tracing level and switching between logging to the live console and a text file
- a form for maintaining access rights and security keys for EWD.js-based Web Services.

Spend some time exploring this application: you should find that it's a useful window into the *ewd.js* and EWD.js environment.

Creating an EWD.js Application

Anatomy of an EWD.js Application

EWD.js applications use WebSocket messages as the sole means of communication between the browser and the application's back-end logic. Essentially the only moving parts of an EWD.js application are a pair of JavaScript files, one in the browser and the other a Node.js module. They send websocket messages to each other and process the corresponding messages they receive. The back-end Node.js module has access to MongoDB and/or your selected Mumps database, the latter being abstracted as a collection of persistent JavaScript objects.

An EWD.js Application consists of a number of key parts:

- A static HTML Page that provides the basic container and main UI
- Optionally, one or more fragment files: files of static HTML markup that can be injected into the main container page
- A static JavaScript file, normally named *app.js,* that defines:
  - outgoing websocket messages, triggered by events in the UI
  - the handlers for incoming websocket messages from the EWD.js application's back-end
- a back-end Node.js module that defines the handlers for incoming websocket message from browsers using the EWD.js application. This name of this module file is usually the same as the EWD.js application name.

EWD.js expects these to be named and placed appropriately in the directory paths you created during the installation steps. For example, if we were creating an EWD.js application named *myDemo*, you would name and place the above components as follows (relative to your Home Directory, eg ~/ewdjs):

```
-    node_modules
       o    myDemo.js [ back-end Node.js module]
-    www
       o    ewd
              ▪    myDemo [ sub-directory, with same name as application ]
                     •    index.html [ main HTML page / container ]
                     •    app.js     [ front-end JavaScript logic ]
                     •    xxx.html   [ fragment files             ]
```

To fully understand how EWD.js works and how to create EWD.js applications, you are encouraged to read and follow the simple Hello World application tutorial in Appendix 4.

The HTML Container Page

Every EWD.js application needs a main HTML container page. This file must reside in a a subdirectory of the *www/ewd* directory that was originally created during the installation steps under your EWD.js Home Directory. The directory name must be the same as the EWD.js application name, eg, in the example above, *myDemo*.

The HTML page can have any name you like, but the normal convention is to name it *index.html*.

For Bootstrap 3 applications, you should use the template page (*index.html)* that you'll find in the *bootstrap3* application folder, or at https://github.com/robtweed/ewd.js/blob/master/www/ewd/bootstrap3/index.html

See Appendix 4 for more details.

The app.js File

An EWD.js application's dynamic behaviour is created by JSON content being delivered into the browser via websocket messages, whereupon your browser-side message handlers use that JSON content and modify the UI appropriately.

This is the role and purpose of the *app.js* file. It normally resides in the same directory as the *index.html* file.

The browser-side websocket controller JavaScript file can actually have any name you like, but the normal convention is to name it *app.js*.

A template app.js file for use with Bootstrap 3 is provided in the bootstrap3 application folder, or at https://github.com/robtweed/ewd.js/blob/master/www/ewd/bootstrap3/app.js

Its constituent parts are as follows. It is recommended you adhere to the structure shown below. For further information, see Appendix 4.

```
EWD.sockets.log = true;   // *** set this to false after testing / development

EWD.application = {
  name: 'bootstrap3', // **** change to your application name
  timeout: 3600,
  login: true,  // set to false if you don't want a Login panel popping up at the start
  labels: {
    // text for various headings etc
    'ewd-title': 'Demo',                                  // *** Change as needed
    'ewd-navbar-title-phone': 'Demo App',                 // *** Change as needed
    'ewd-navbar-title-other': 'Demonstration Application'    // *** Change as needed
  },
  navFragments: {
    // definitions of Navigation tab operation
    // nav names should match fragment names, eg main & main.html
    main: {
      cache: true
    },
    about: {
      cache: true
    }
  },
  onStartup: function() {
    // stuff that should happen when the EWD.js is ready to start
    // Enable tooltips
    //$('[data-toggle="tooltip"]').tooltip()

    //$('#InfoPanelCloseBtn').click(function(e) {
    //  $('#InfoPanel').modal('hide');
    //});
    // load initial set of fragment files
    EWD.getFragment('login.html', 'loginPanel');
    EWD.getFragment('navlist.html', 'navList');
    EWD.getFragment('infoPanel.html', 'infoPanel');
    EWD.getFragment('confirm.html', 'confirmPanel');
    EWD.getFragment('main.html', 'main_Container');
  },
  onPageSwap: {
    // add handlers that fire after pages are swapped via top nav menu
    /* eg:
    about: function() {
      console.log('"about" menu was selected');
    }
    */
  },
  onFragment: {
    // add handlers that fire after fragment files are loaded into browser, for example:

    'navlist.html': function(messageObj) {
      EWD.bootstrap3.nav.enable();
    },

    'login.html': function(messageObj) {
      $('#loginBtn').show();
      $('#loginPanel').on('show.bs.modal', function() {
        setTimeout(function() {
          document.getElementById('username').focus();
        },1000);
      });

      $('#loginPanelBody').keydown(function(event){
        if (event.keyCode === 13) {
          document.getElementById('loginBtn').click();
        }
      });
    }

  },
  onMessage: {

    // handlers that fire after JSON WebSocket messages are received from back-end, eg to handle a message with type == loggedIn

    loggedIn: function(messageObj) {
      toastr.options.target = 'body';
      $('#main_Container').show();
      $('#mainPageTitle').text('Welcome to VistA, ' + messageObj.message.name);
    }
  }

};
```

EWD.js websocket messages have a mandatory type. EWD.js provides a number of pre-defined reserved type names, and these also have predetermined properties associated with them. In the main, however, it is up to the developer to determine the type names and content structure of the messages. The payloads of WebSocket messages are described as JSON content.

To send an EWD.js websocket message from the browser, use the EWD.sockets.sendMessage API with the syntax:

```
EWD.sockets.sendMessage({
  type: messageType,
  params: {
    //JSON payload: as simple or as complex as you like
  }
});

eg:

EWD.sockets.sendMessage({
  type: 'myMessage',
  params: {
    name: 'Rob',
    gender: 'male'
    address: {
      town: 'Reigate',
      country: 'UK'
    }
  }
});
```

Note that the payload must be placed in the *params* property. Only content inside the *params* property is conveyed to the child process where your back-end code will run.

Back-end Node.js Module

The final piece in an EWD.js application is the back-end Node.js module.

This file must reside in the *node_modules* directory that was originally created during the installation steps under your EWD.js Home Directory (eg *~/ewdjs/node_modules*). The module's filename must be the same as the EWD.js application name, eg, an application named *ewdMonitor* will have a back-end module file named *ewdMonitor.js*.

The constituent parts of a back-end EWD.js module are as follows. It is recommended you adhere to the structure shown below:

```
// Everything that is to be externally-accessible must be inside a module.exports object

module.exports = {

  // incoming socket messages from a user's browser are handled by the onMessage() functions

  onMessage: {

    // write a handler for each incoming message type,
    // eg for message with type === getPatientsByPrefix:

    getPatientsByPrefix: function(params, ewd) {

      // optional: for convenience I usually break out the constituent parts of the ewd object

      var sessid = ewd.session.$('ewd_sessid')._value;  // the user's EWD.js session id

      console.log('getPatientsByPrefix: ' + JSON.stringify(params));
      var matches = [];
      if (params.prefix === '') return matches;
      var index = new ewd.mumps.GlobalNode('CLPPatIndex', ['lastName']);
      index._forPrefix(params.prefix, function(name, subNode) {
        subNode._forEach(function(id, subNode2) {
          matches.push({name: subNode2._value, id: id});
        });
      });
      return matches; // returns a response websocket message to the user's browser.
                      // The returned message has the same type, 'getPatientsByPrefix' in
                      // this example.  The JSON payload for a returned message is in the
                      // "message" property, so the browser's handler for this message response
                      // will extract the matches in the example above by accessing:
                      //  messageObj.message
    }

  }

};
```

Form Handling

EWD.js has a special built-in browser-side function (*EWD.sockets.submitForm*) for processing forms. If you are using ExtJS, then EWD.js provides some additional automation, but it can be used with any other framework including your own.

If you are using this form-handling function with ExtJS, it automatically collects the values of all the form fields in an ExtJS form component (ie xtype: 'form') and sends them as the payload of a message whose type should be prefixed *EWD.form*. The syntax is as follows:

```
EWD.sockets.submitForm({
  id: 'myForm',                     // the id of the ExtJS form component
  alertTitle: 'An error occurred',  // optional heading for the Ext.msg.Alert panel for displaying
                                    //   error messages resulting from back-end form validation
  messageType: 'EWD.form.myForm'    // the message type that will be used for sending this form's
                                    //   content to the back-end
                                    // The form field values will be automatically packaged into
                                    // the message's 'params' payload object.  The form field
                                    //  'name' property will be used in the payload also

});
```

If you are using other frameworks, you need to marshall the form values that need submitting by defining the *fields* object, for example, if you were using Bootstrap, a form submit button handler might look like this. Note the way that a *toastr* widget can be specified to handle display of any error messages:

```
$('body').on( 'click', '#loginBtn', function(event) {
  event.preventDefault(); // prevent default bootstrap behavior
  EWD.sockets.submitForm({
    fields: {
      username: $('#username').val(),
      password: $('#password').val()
    },
    messageType: 'EWD.form.login',
    alertTitle: 'Login Error',
    toastr: {
      target: 'loginPanel'
    }
  });
});
```

The back-end Node.js module will have a corresponding handler for the specified message type, eg to handle an *EWD.form.login* message for a form with fields having name properties with the values *username* and *password* , we might do the following:

```
  'EWD.form.login': function(params, ewd) {
    if (params.username === '') return 'You must enter a username';
    if (params.password === '') return 'You must enter a password';
    var auth = new ewd.mumps.GlobalNode('CLPPassword', [params.username]);
    if (!auth._hasValue) return 'No such user';
    if (auth._value !== params.password) return 'Invalid login attempt';
    ewd.session.setAuthenticated();
    return '';
  }
```

You can see that the *username* and *password* field values are passed to your handler function as the *params* argument: this contains the *params* payload from the incoming message that was sent from the browser.

Form processing in EWD.js is very simple: for each error condition you determine, simply return with a string that you wish to use the text of an error message to appear in the browser. If an *EWD.form.xxx* message handler returns a non-empty string and the application running in the browser has been built using the EWD.js / Bootstrap 3 framwork, the error message will automatically appear on the user's browser in a *toastr* widget. If you are using hand-crafted HTML and JavaScript in the browser, the error will appear as a JavaScript alert window.

However, if a form handler function returns a null string, EWD.js returns a message of the same type (eg *EWD.form.login* in the example above) with the payload *ok: true*.

Therefore, the browser-side *app.js* file should include a handler for an incoming message of type EWD.form.login in order to modify the UI in response to a successful login - for example by removing the login form, eg:

```
'EWD.form.login': function(messageObj) {
  if (messageObj.ok) $('#loginPanel').hide();
}
```

User Authentication Control

In many EWD.js applications you will want to establish the authentication credentials of the user before allowing them to continue. In such applications, it is important that you don't leave a back-door that allows an un-authenticated user to try invoking websocket messages (eg from Chrome's Developer Tools console). In order to do this, make sure you do the following:

1) In your back-end module's logic that processes the initial login form, use the special function *ewd.session.setAuthenticated()* to mark the user as having been successfully authenticated. By default, users are flagged as not authenticated.

2) Make sure that all your back-end message-handling functions, apart from the login authentication function, check that the user has been authenticated, eg

```
  getGlobals: function(params, ewd) {
    if (ewd.session.isAuthenticated) {
      // processing takes place here for authenticated users only
    }
  },
```

Note: *ewd.session.isAuthenticated* is a property that exists only at the back-end, and the back-end can only be accessed via a WebSocket message whose processing is entirely controlled by the developer's logic.

The ewd Object

You'll have seen that your back-end message handler functions within the *onMessage* object has two arguments: *params* and *ewd*. The second argument, *ewd*, is an object that contains several sub-component objects that are useful in your back-end logic, in particular:

- **ewd.session**: a pointer to the user's EWD.js session which is stored and maintained in the Mumps or MongoDB database. EWD.js automatically garbage-collects any timed out sessions. EWD.js sessions have a default timeout of 1 hour.

- **ewd.webSocketMessage**: a pointer to the complete incoming WebSocket message object. Normally you'll use the params argument, since that's where the incoming message's payload is normally placed. However, the entire message object is made available for you for cases where you might require it;

- **ewd.sendWebSocketMsg()**: a function for sending WebSocket messages from the back-end Node.js module to the user's browser. The syntax for this function is:

```
ewd.sendWebSocketMsg({
  type: messageType,      // the message type for the messaging being sent
  message: payload        // the message JSON payload
});

eg:

ewd.sendWebSocketMsg({
  type: 'myMessage',
  message: {
    name: 'Rob',
    gender: 'male'
    address: {
      town: 'Reigate',
      country: 'UK'
    }
  }
});
```

- **ewd.mumps**: a pointer that gives you access to the Mumps database and to legacy Mumps functions (the latter is available on Caché and GT.M only). This is described in detail in the next chapter.

Here's an example from the demo application of how Mumps data can be manipulated from within your Node.js module:

```
'EWD.form.selectPatient': function(params, ewd) {
  if (!params.patientId) return 'You must select a patient';

  // set a pointer to a Mumps Global Node object, representing the persistent
  // object:  CLPPats.patientId, eg CLPPats[123456]

  var patient = new ewd.mumps.GlobalNode('CLPPats', [params.patientId]);

  // does the patient have any properties? If not then it can't currently exist

  if (!patient._hasProperties) return 'Invalid selection';

  ewd.sendWebSocketMsg({
    type: 'patientDocument',

    // use the _getDocument() method to copy all the data for the persistent
    // object's sub-properties into a corresponding local JSON object

    message: patient._getDocument()
  });
  return '';
}
```

- **ewd.util**: a collection of pre-built functions that can be useful in your applications, including:
  - ewd.util.getSessid(token): returns the unique EWD.js session Id associated with a security token
  - ewd.util.isTokenExpired(token): returns true if the session has timed out
  - ewd.util.sendMessageToAppUsers(paramsObject): sends a websocket message to all users of a specified application. The paramsObject properties are as follows:
    - type: the message type (string)
    - content: message payload (JSON object)
  - ewd.util.requireAndWatch(path): alternative to require(). This loads the specified module and also sets a watch on it. If the module is edited, it is automatically reloaded. Useful during application/module development.

Starting EWD.js Applications in a Browser

EWD.js applications are started using a URL of the form:

http[s]://[ip address/domain name]:[port]/ewd/[application name]/index.html

- Specify http:// or https:// depending on whether or not your *ewd.js* startup file enables HTTPS or not
- Adjust the ip address or domain name according to the server on which you're running *ewd.js*
- Specify the port that corresponds to the port defined in your *ewd.js* startup file
- Specify the application name that corresponds, case-sensitively, to the name used as the directory path for your *index.html and app.js* files.

Debugging EWD.js Applications using Node Inspector

Node Inspector (https://github.com/node-inspector/node-inspector) has become a standard utility for debugging Node.js applications. EWD.js provides integration with it, so you can use it for stepping through and examining how your EWD.js application is running (or failing!).

Just follow these steps:

- You must be using EWD.js Build 67 or later. Upgrade your EWD.js installation if necessary
- Install Node Inspector:
  - *cd ~/ewdjs* (or wherever your EWD.js Home directory resides)
  - *npm install -g node-inspector*
- Start Node Inspector in a terminal/SSH window
  - *cd ~/ewdjs* (or wherever your EWD.js Home directory resides)
  - *node-inspector --web-port 8081*
- Start EWD.js in the normal way (if it isn't already running)
- Using Chrome, bring up the *ewdMonitor* application and login. In the main Overview panel you'll see a button in the *Child Process Pool* section that will start up a new Child Process in debug mode. Click it and then close the other Child Processes so the only one running is your new debug-enabled one. This ensures that all activity will be handled by this process
- Click on the blue Debug button for the Child Process. A new browser tab will open and the Node Inspector debugger interface will appear - it takes a few seconds to fully load. If you look carefully, it's almost identical to the familiar Chrome JavaScript Console / Developer Tools UI. You can now add breakpoints and examine your running module(s)!
- When you're done, start new, normal Child Processes (ie not in debug mode) and close the one you enabled in debug mode. *CTRL&C* to stop the Node Inspector process in its terminal window

Note: If you want to change the web-port used by Node Inspector, click on the *Internals* tab in the *ewdMonitor* application and click the new button you'll find in there

If you've not used Node Inspector before, it's worth reading some of the many tutorials and articles on its use. For example:

https://www.youtube.com/watch?v=03qGA-GJXjI

Pulling It All Together: Building an EWD.js Application

Appendix 4 takes you through the process of building an EWD.js application, first a very basic HTML and handcrafted JavaScript application, and then the same demo application using the EWD.js / Bootstrap 3 framework.

To see examples of EWD.js applications, take a look at the *ewdMonitor* application that is included in the EWD.js installation.

To see an example of building an EWD.js application integrating with the Open Source VistA Electronic Healthcare Record (EHR), see the *VistADemo* application which is also included in the EWD.js installation.

The source code for these applications can be found within the *ewd.js* Github repository at https://github.com/robtweed/ewd.js, eg:

ewdMonitor Application:

Front-end: https://github.com/robtweed/ewd.js/tree/master/www/ewd/ewdMonitor

Back-end: https://github.com/robtweed/ewd.js/blob/master/modules/ewdMonitor.js

VistADemo Application:

Front-end: https://github.com/robtweed/ewd.js/tree/master/www/ewd/VistADemo

Back-end: https://github.com/robtweed/ewd.js/blob/master/modules/VistADemo.js

Back-end Mumps functions for accessing VistA (courtesy of Chris Casey): https://github.com/robtweed/ewd.js/blob/master/OSEHRA/ZZCPCR00.m

Micro-Service Support in EWD.js

What are Micro-Services?

Micro-Services are the latest big buzz in the web and browser-based application development world. As with many buzzwords in IT, the term is somewhat vague and difficult to specifically pin-point, but it's worth reading the definitive paper by Martin Fowler on the subject:

http://martinfowler.com/articles/microservices.html

In a nutshell, Micro-Services are all about moving away from the development of huge, monolithic web/browser applications, and instead creating applications that are an assembly of many, small, re-usable components. It's all about creating an environment where various, disparate teams in an organisation can focus on just their part of an enterprise, and build the micro-service(s) that others can use in their applications.

The classic example is Amazon's web site which looks like one large monolithic application to the user, but which is actually an assembly of micro-services for which each department or section within Amazon is individually responsible. The web page(s) that make up the Amazon site that we, the user, sees, is just a mechanism for pulling together those micro-services in a coherent way.

EWD.js Build 85 (and later) now includes a Micro-Service architecture to allow you to work similarly. It actually allows you to create and work with two kinds of Micro-Services:

- back-end Services
- front-end Services

An EWD.js application may use either or both types.

Back-end Micro-Services

These are the simplest and easiest to understand and use. The idea is that instead of an EWD.js application having a single back-end module that contains all of its message handlers and associated back-end business logic, the EWD.js application can also make use of one or more back-end service modules, each of which focus on a particular functional or business area. For example, your application may require an assembly of back-end services for:

- accounts
- personnel
- stock control
- billing
- customer support
- etc

A back-end Micro-Service is simply a set of message handlers, but it can be written in isolation and used by any EWD.js application that needs it.

For example, here's a simple back-end example Service:

```
module.exports = {
 onMessage: {
   backEndServiceMessage: function(params, ewd) {
     return {status: 'Success!'};
   }
 }
};
```

It is saved in the *~/ewdjs/node_modules* directory along with all other back-end modules. Suppose we save it as *exampleService.js* (You can name it anything you wish, but it's a good idea to use a meaningful name that describes the functional area it covers, eg *accounts.js, stockControl.js* etc).

To use this Micro-Service in your application, you simply add a *services* function to its back-end module:

```
module.exports = {
  onMessage: {
     // your application's own specific message handlers
  },

  services: function() {
    // return an array of micro-services that this application
    // is allowed to use
   return ['exampleService'];
  }
};
```

This tells EWD.js the list of back-end Micro-Services that our application is allowed to use. Each name in the array must match the name of a back-end Micro-Service module (without the *.js* file extension). This *services()* function is important because we need to prevent a malicious user attempting to send rogue messages that attempt to use Micro-Services that are not normally available to the application!

Now our application's front-end *app.js* file can send messages to the *exampleService* Micro-Service module, simply by adding a new property named *service* to the EWD.sockets.sendMessage() function's input object:

```
     EWD.sockets.sendMessage({
       type:'backEndServiceMessage',
        service: 'exampleService',
       params: {
         // message payload
       },
       done: function(messageObj) {
         // handle the response from the handler in
         // the exampleService Micro-Service
       }
     });
```

If our EWD.js application is named *myApp*, then instead of this *backEndServiceMessage* message being handled by its back-end module *myApp.js*, it is handled by the corresponding handler in the back-end Micro-Service module named *exampleService.js* instead.

Of course, if you don't specify a *service* property in an EWD.sockets.sendMessage() function call, it will behave as normal and try to use a message handler in the application's own back-end Module (in this case *myApp.js*)

Front-end Micro-Services

EWD.js also allows you to define a set of actions and message handlers that can be used by the front-end of any EWD.js applications. Just as with back-end Micro-Services, the idea is that the author of such a front-end Micro-Service does not need to know anything about the EWD.js applications it will be used in, so they are completely re-usable.

A front-end Micro-Service may also optionally include a fragment file that can be loaded into an EWD.js application's index.html, but fragments can alternatively be tailor-made by the application author(s) to ensure that they are correctly structured and styled to fit with the design of the overall page.

A front-end Micro-Service may be designed to focus on handling a particular piece of business functionality (eg how to handle stock-control listings), or they may focus on one or more UI widgets (eg a generic calendar or grid control). They are a way of creating re-usable front-end behaviour and functionality.

Front-end Micro-Services are somewhat more complex than Back-end Micro-Services, but the trick to using them is to make use of their standard pattern. Use the examples below as templates that you subsequently modify and extend.

When you install EWD.js build 85 (or later), you'll find a new directory under ~/ewdjs/www:

~/ewdjs/www/services

This is where you should file all your front-end Micro-Service modules and their associated fragment files. You'll find an example one in there: *demoPatientProfile.js* and its associated fragment file *demoPatientProfile.html*

The general pattern of a front-end Micro-Service module file is as follows:

```
define([],function() {

  // put any private functions in here

  return {

    init: function(serviceName) {
      // anything here is run when the service is loaded by EWD.require()
    },

    fragmentName: 'main.html', // optional, overrides automatic fragment path

    onMessage: {
      // message handlers for front-end service messages
    },

    onFragment: {
      // optional additional fragment load handlers
    }
  }
});
```

A front-end Micro-service is loaded into an application's browser by using the built-in function: *EWD.require().* This makes use of the *require.js* JavaScript module loader (http://requirejs.org), which must therefore be loaded into your application's *index.html* page. A typical use of *EWD.require()* is shown below. This would be invoked from within your EWD.js application's app.js file:

```
EWD.require({
  serviceName: 'patientProfile',
  targetSelector: '#main_Container',
  done: function() {
    console.log('done!');
  }
});
```

- ***serviceName*** specifies the name of the front-end Micro-service you want to load. EWD.js will look for a file of the same name in your *~ewdjs/www/services* directory with a file extension of *.js* – so in the example above it will load */services/patientProfile.js*

- ***targetSelector*** specifies the target HTML element(s) into which the service's fragment will be loaded. If a simple string value is used, EWD.js will load the fragment into the element that has an Id of that value. Alternatively you can use a jQuery selector, in which case the fragment will be loaded into element(s) that match the specified jQuery selector. So, in the example above, the service's fragment will be loaded into the element with an *id* of *mainContainer*

- ***fragmentName*** may or may not be specified. If not specified (as in the example above), EWD.js will attempt to load a file of the same name as the *serviceName* with an extension of *.html* from your */services* directory – so in the example above it will load */services/patientProfile.html* If *fragmentName* is specified, EWD.js will attempt to load a file of the name you specify, with an extension of *.html,* from the application's own directory (ie */www/ewd/{applicationName}*. You can optionally suppress the loading of the service's fragment by specifying *fragmentName: false*

When a front-end Micro-service is loaded using EWD.require(), four things happen in strict (synchronous) sequence:

- the fragment is loaded
- the corresponding *onFragment()* handler, if defined in the application's *app.js* file, fires
- the service's *init()* function fires
- the d*one()* function in the *EWD.require()*, if defined, fires

Typically you'll use each handler as follows:

- in most circumstances the application developer won't need to provide an *onFragment()* handler. However, if the developer understands the service in detail, he/she can provide an *onFragment()* handler to customise the fragment, modify its markup, extend its handlers etc.

- the service's *init()* function is where the service author invokes code that defines the behaviour of the service in terms of handlers, events etc. Note that the *init()* function has access to any private functions (if any) that are defined at the top of the service module. The *init()* function may also be used to extend EWD.application with extra methods that are then available to the *app,js* of any EWD.js application that loads the service.

- the *EWD.require()*'s *done()* function is where the application author (ie the consumer of the service) can optionally invoke code specific to the application's use of the service. By the time this code runs, the fragment is fully loaded and the service is initialised. We'd anticipate that the *done()* function can be ignored for most services, and should only be used by advanced users who understand in detail the service they are loading.

If the application author decides to use his/her own fragment instead of one provided by the service (if any), then it is the application author's responsibility to provide the *onFragment()* handler for it. Once again, we'd only anticipate advanced users who fully understand the workings of a service to specify their own fragment. Note that doing so will not otherwise affect the event sequence described above.

Demonstration Example

If you install or upgrade to EWD.js Build 85 (or later), you'll find a demonstration example in:

~/ewdjs/www/ewd/demoMicroServices

You can run this by using the usual URL:

http://{ipAddress}:{httpPort}/ewd/demoMicroServices/index.html

You should see a page coming up with some patient profile details. What's actually happened is that the profile content page has come from a front-end Micro-Service, populated with data via a back-end Micro-Service.

Take a look in the file *~/ewdjs/www/ewd/demoMicroServices/app.js*. In its *onStartup()* handler, you'll see where it loads the front-end Micro-Service named *demoPatientProfile*:

```
    EWD.require({
      serviceName:'demoPatientProfile',
      targetSelector:'#main_Container',
      done: function() {
        console.log('app.js: demoPatientProfile service loaded successfully');
      }
    });
```

You'll find this Micro-Service in *~/ewdjs/www/services*:

- its fragment is named *demoPatientProfile.html*
- its logic is in *demoPatientProfile.js*

Because *fragmentName* hasn't been specified in the *EWD.require()*, the fragment is automatically loaded and the *targetSelector* property in the *EWD.require()* tells EWD.js to inject it into the element whose *Id* is *main_Container*.

Now take a look inside *demoPatientProfile.js* and find its *init()* function.

```
    init: function(serviceName) {
      console.log('demoPatientProfile service: init() firing');
      // lets call upon the profile back end service now
      EWD.sockets.sendMessage({
        type:'getUserData',
        service:'demoPatientProfile',
        frontEndService: serviceName
      });
      console.log('demoPatientProfile service: message sent to demoPatientProfile back-end service');
    },
```

You'll see that it sends a message to the back-end with a type of *getUserData*. However, notice the *service* property of *demoPatientProfile*. This means it's accessing a back-end Micro-Service named *demoPatientProfile*.

Take a look at the back-end module for our *demoMicroServices* application:*~/ewdjs/node_modules/demoMicroServices.js*:

```
module.exports = {
  onMessage: {
  },
  // authorise this app to access the demoPatientProfile back-end Micro-Service
  services: function() {
   return ['demoPatientProfile'];
  }
};
```

Because the front-end Micro-Service used by our *demoMicroServices* application is using the back-end Micro-Service named *demoPatientProfile*, our application must first be authorised to use it: that's done in the *services()* function shown above.

Now take a look at the back-end MicroService in *~/ewdjs/node_modules/demoPatientProfile.js*:

```
module.exports = {
  onMessage: {

    // returns some patient info
    // this likely would have been fetched from our database
    getUserData: function(params,ewd) {
      var patientData = {
        'First Name': 'Patient',
        'Last Name': 'Zero',
        'Date of Birth': '11/11/1950',
        'Address': '10 The Lane',
        'City': 'London',
        'State': 'UK'
      }
      return patientData;
    }
  }
};
```

This looks just like any standard EWD.js back-end module. Its response is returned to the front end, but instead of being handled by the application's app.js, it's handled inside the *demoPatientProfile* service. To understand why, look again at how it sent that message to the back-end Micro-service:

```
    init: function(serviceName) {
      console.log('demoPatientProfile service: init() firing');
      // lets call upon the profile back end service now
      EWD.sockets.sendMessage({
        type:'getUserData',
        service:'demoPatientProfile',
        frontEndService: serviceName
      });
      console.log('demoPatientProfile service: message sent to demoPatientProfile back-end service');
    },
```

Notice the frontEndService property set to the value of *serviceName* – this tells EWD.js that the handler for this message's response is within this front-end Service. Further down you'll find that handler:

```
    onMessage: {
      // handle the profile data our init() method has fetched
      getUserData: function(messageObj) {
        console.log('demoPatientProfile service: response received from dempPatientProfile back-end service');
        var patientData = messageObj.message;
        populateProfileData(patientData);
        $('.js-profile').show();
      }
    }
```

This populates the fragment and displays it. That's the service completed, and the *done()* function in the *EWD.require()* that loaded the Micro-Service will now fire:

```
  EWD.require({
    serviceName:'demoPatientProfile',
    targetSelector:'#main_Container',
    done: function() {
      console.log('app.js: demoPatientProfile service loaded successfully');
    }
  });
```

Use of the *done()* function is optional: we're using it here just to display a log message in the console to confirm the end of the sequence of events. We could have left it out completely, and in most circumstances you'll probably do just that.

That's it! Our application has successfully loaded and used a front-end and back-end Micro-Service. The interesting thing is that both the front-end and back-end Micro-Services were written without any knowledge of how they would be used within the application, and any application developer can simply load this front-end Micro-Service simply by using *EWD.require()* and treating it like a "black-box".

Try running the application again, this time with the browser's JavaScript console open. You'll see a whole series of log messages being written out: they've been deliberately added to the demo application and front-end Micro-Service. You can find them within the various JavaScript files and confirm for yourself how the sequence of events takes place.

Conclusion

So that's EWD.js Micro-Services. They are a very powerful concept, and allow true team development across the Enterprise and code re-use.

Once you become familiar with EWD.js, their use is highly recommended.

Externally-generated Messages

Background

So far we've looked at WebSocket messages as a means of integrating a browser UI with a back-end Mumps database. However, EWD.js also allows external processes to send WebSocket messages to one or more users of EWD.js applications. These messages can be either a simple signal or a means of delivering as complex a JSON payload as you wish to one or more users.

The processes that generate these external messages can be other Mumps or Cache processes, or, in fact, any process on the same machine, or, depending on how you configure the security of your systems, other system on the same network.

The External Message Input Interface

The *ewd.js* module includes a TCP socket server interface that is automatically activated and configured, by default, to listen on port 10000.

You can change this port by modifying the startup parameters object in your startup file (ie your *ewdStart*.js* file). You do this by adding the *webSockets.externalListenerPort* definition, for example:

```
var ewd = require('ewdjs');

params = {
  cwd: '/opt/ewdlite/',
  httpPort: 8080,
  traceLevel: 3,
  database: {
    type: 'globals',
    path:"/opt/globalsdb/mgr"
  },
  management: {
    password: 'keepThisSecret!'
  },
  webSockets: {
    externalListenerPort: 12001
  }
};

ewd.start(params);
```

An external process simply needs to open the listener port (ie 10000 unless you've reconfigured it to use a different port), write a JSON-formatted string and then close the listener port. EWD.js will do the rest.

Note: To disable the External listener port, simply set the start-up file's parameter value to *false*, ie:

```
  webSockets: {
    externalListenerPort: false
  }
```

Defining and Routing an Externally-Generated Message

You can send messages to:

- all currently-active EWD.js users
- all current users of a specified EWD.js application
- all users whose EWD.js Session match a list of Session names and values

You must specify a message *type* (just as you would for web-socket messages within an EWD.js application) and a message payload. For security reasons, all externally-injected messages must include a password: this must match the one specified in the *ewdStart*.js* file.

The JSON string you write to the TCP listener port determines your message's required destination. The JSON structure and properties differs slightly for each destination category:

Messages destined for all users

```
{
 "recipients": "all",
 "password": "keepThisSecret!",    // mandatory: this must match the password in the ewdStart*.js file

 "type": "myExternalMessage",      // mandatory: you provide a message type.  The value is for you to determine
 "message" {
    // your JSON message payload, the structure of which is for you to decide
 }
}
```

All currently-active EWD.js users will have the above message sent to their browser.

Messages destined for all users of a specified EWD.js Application

```
{
 "recipients": "byApplication",
 "application": "myApp",           // mandatory: specify the name of the EWD.js application
 "password": "keepThisSecret!",    // mandatory: this must match the password in the ewdStart*.js file

 "type": "myExternalMessage",      // mandatory: you provide a message type.  The value is for you to determine
 "message" {
    // your JSON message payload, the structure of which is for you to decide
 }
}
```

All currently-active users of an EWD.js application named *myApp* will have the above message sent to their browser.

Messages destined for all users matching specified EWD.js Session contents

```
{
 "recipients": "bySession",
 "session": [                       // specify an array of Session name/value pairs, eg:
   {
    "name": "username",
    "value": "rob"
   }
 ],
 "password": "keepThisSecret!",    // mandatory: this must match the password in the ewdStart*.js file

 "type": "myExternalMessage",      // mandatory: you provide a message type.  The value is for you to determine
 "message" {
    // your JSON message payload, the structure of which is for you to decide
 }
}
```

All currently-active EWD.js users whose *username* is *rob* will have the above message sent to their browser. More specifically and accurately, the message is sent to all users whose EWD.js Session contains a variable named *username* whose value is *rob*.

You can specify as many Session name/value pairs as you like within the array. The message will only be sent if ***all*** name/value pairs match in a user's Session.

Handling Externally-Generated Messages

Externally-generated messages are sent to the relevant users' browser.

In order for externally-injected messages to be processed by EWD.js applications, you must include an appropriate handler for the incoming message type in the ***browser-side*** JavaScript for each application (eg in the *app.js* file). If no handler exists for the incoming message type, it will be ignored.

Handlers for externally-generated messages are no different from those for normal EWD.js WebSocket messages, eg:

```
onMessage: {

  myExternalMessage: function(messageObj) {
    console.log('External message received: ' + JSON.stringify(messageObj.message));
  },

  // ...etc

};
```

If you need to do something at the back-end in order to handle an incoming externally-generated message, simply send a WebSocket message to the back-end from within your handler along with some or all of the externally-generated message's payload, eg:

```
onMessage: {

  myExternalMessage: function(messageObj) {
    EWD.sockets.sendMessage({
      type: 'processXternalMsg',
      params: {
        msg: messageObj.message
      }
    });
  }

  // ...etc

};
```

Sending in Messages from GT.M and Caché Processes

If you want to send messages from external GT.M or Caché processes, you can make use of the pre-built method (*sendExternalMessage()*) that is provided by the Mumps routine named *ewdjsUtils* that is included in the EWD.js repository. See:

https://github.com/robtweed/ewd.js/blob/master/mumps/ewdjsUtils.m

if you are using GT.M, copy this routine file to an appropriate directory that is mapped for compiling and executing routine files. If you are using Caché, copy and paste the code into Caché Studio and compile/save it using the name *ewdjsUtils*.

You'll find an example of how to use the *sendExternalMessage()* method within the routine file, ie:

```
externalMessageTest(type,port,password)
 n array
 i $g(password)="" s password="keepThisSecret!"
 i $g(port)="" s port=10000
 i type=1 d
 . s array("type")="fromGTM1"
 . s array("password")=password
 . s array("recipients")="all"
 . s array("message","x")=123
 . s array("message","y","z")="hello world"
 i type=2 d
 . s array("type")="fromGTM2"
 . s array("password")=password
 . s array("recipients")="all"
 . s array("message","x")=123
 . s array("message","y","z")="hello world"
 i type=3 d
 . s array("type")="fromGTM3"
 . s array("password")=password
 . s array("recipients")="bySessionValue"
 . s array("session",1,"name")="username"
 . s array("session",1,"value")="zzg38984"
 . s array("session",2,"name")="ewd_appName"
 . s array("session",2,"value")="portal"
 . s array("message","x")=123
 . s array("message","y","z")="hello world"
 d sendExternalMessage^ewdjsUtils(.array,port)
 QUIT
```

externalMessageTest() exercises all three types of external message. Instead of creating a JSON-formatted string, the Mumps code above allows you to build the JSON structure as an equivalent local Mumps array. The code for opening, closing and writing to EWD.js's TCP port on GT.M and Caché systems is included in the routine file.

So, to test the sending of a message from an external GT.M or Caché process, you can invoke the following, which assumes your *ewd.js* process is using the default TCP port, 10000:

```
do externalMessageTest^ewdjsUtils(1)
do externalMessageTest^ewdjsUtils(2)
do externalMessageTest^ewdjsUtils(3)
```

Of course, you'll need to write in-browser handlers for the three message types if you want them to do anything.

Modify the code in the *externalMessageTest()* procedure to create external messages of your own.

Sending Externally-generated Messages from Other environments

Of course, you aren't restricted to GT.M or Caché processes. Any process that can open EWD.js's TCP socket listener's port can write a JSON-formatted message string to it and therefore send messages to relevant EWD.js users. The implementation details will vary depending on the language you use within such processes.

JavaScript Access to Mumps Data

Background to the Mumps Database

A Mumps database stores data in a schema-free hierarchical format. In Mumps technology parlance, the individual unit of storage is known as a Global (an abbreviation of Globally-Scoped Variables). Given the usual modern meaning of the term Global, it is perhaps better to think of the unit of storage as a persistent associative array. Some examples would be:

- myTable("101-22-2238","Chicago",2)="Some information"
- account("New York", "026002561", 35120218433001)=123456.45

Each persistent associative array has a name (eg myTable, account in the examples above). There then follows a number of subscripts whose values can be numeric or text strings. You can have any number of subscripts.

Each "node" (a node is defined by an array name and a specific set of subscripts) stores a data value which is a text string (empty strings are allowed). You can create or destroy nodes whenever you like. They are entirely dynamic and require no pre-declaration or schema.

A Mumps database has no built-in schema or data dictionary. It is up to the developer to design the higher-level abstraction and meaning of a database that is physically stored as a set of persistent arrays.

A Mumps database also has no built-in indexing. Indices are the key to being able to effectively and efficiently search, query and traverse data in a Mumps database, but it is up to the developer to design, create and maintain the indices that are associated with the main data arrays. Indices are, themselves, stored in Mumps persistent arrays.

You can read more background to the Mumps database technology and its importance as a powerful NoSQL database engine in a paper titled *A Universal NoSQL Engine, Using a Tried and Tested Technology*: http://www.mgateway.com/docs/universalNoSQL.pdf.

EWD.js's Projection of Mumps Arrays

Included in the *ewd.js* distribution is a Javascript file named *ewdGlobals.js*. *ewdGlobals.js* is used by EWD.js to create an abstraction layer on top of the low-level APIs provided by the Node.js interfaces to GlobalsDB, GT.M and Caché, *ewdGlobals.js* projects the collection of persistent associative arrays in a Mumps database as a collection of persistent JavaScript objects.

Mapping Mumps Persistent Arrays To JavaScript Objects

The theory behind the projection used by *ewdGlobals.js* is really quite straightforward, and best explained and illustrated by way of an example.

Suppose we have a set of patient records that we wish to represent as objects. We could define a top-level object named *patient* that represents a particular physical patient. Usually we'd have some kind of patient identifier key that distinguishes our particular patient: for purposes of this example, let's say that patient identifier is a simple integer value: *123456*.

JavaScript's object notation would allow us to represent the patient's information using properties which could, in turn, be nested using sub-properties, for example:

```
patient.name = "John Smith"
patient.dateOfBirth = "03/01/1975"
patient.address.town = "New York"
patient.address.zipcode = 10027
.. etc
```

This patient object could be represented as follows in a Mumps database as the following persistent array:

```
patient(123456, "name") = "John Smith"
patient(123456, "dateOfBirth") = "03/01/1975"
patient(123456, "address", "town") = "New York"
patient(123456, "address", "zipcode") = 10027
.. etc
```

So you can see that there's a direct one-to-one correspondence that can be made between an object's properties and the subscripts used in a Mumps persistent array. The converse is also true: any existing Mumps persistent array could be represented by a corresponding JavaScript object's hierarchy of properties.

When data is stored into a Mumps database, we tend to refer to each unit of storage as a *Global Node*. A *Global Node* is the combination of a persistent array's name (in this case *patient*) and a number of specific subscripts. A *Global Node* may effectively have any number of subscripts, including zero. Data, in the form of numeric or alphanumeric values, are stored at each leaf *Global Node*.

Each level of subscripting represents an individual *Global Node*. So, taking our *zipcode* example above, we can represent the following *Global Nodes*:

```
^patient
^patient(123456)
^patient(123456, "address")
^patient(123456, "address", "zipcode") = 10027
```

Note that data has only been stored in the lowest-level (or leaf) *Global Node* shown above. All the other *Global Nodes* exist but are just intermediate nodes: they have lower-level subscripts, but don't have any data.

There is nothing in the Mumps database that will tell us that subscripts of "address" and "zipcode" have been used in this particular persistent array other than by introspection of the actual *Global Nodes*: ie there is no built-in data dictionary or schema that we can reference. Conversely, if we want to add more data to this persistent array, we can just add it, arbitrarily using whatever subscripts we wish. So we could add a *County* record:

```
^patient(123456, "address", "county") = "Albany"
```

Or we could add the patient's weight:

```
^patient(123456, "measurement", "weight") = "175"
```

Note that I could have used any subscripting I liked: there was nothing that forced me to use these particular subscripts (though in an application you'd want to make sure all records consistently used the same subscripting scheme).

The GlobalNode Object

The *ewdGlobals.js* projection provided by EWD.js allows you to instantiate a *GlobalNode* object. For example:

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
```

A *GlobalNode* Object represents a physical *Global Node* in a Mumps database and has available to it a set of properties and methods that allow it to be manipulated and examined using JavaScript. An important feature of a *GlobalNode* Object is that it *may or may not actually physically exist when first instantiated*, but this may or may not change later during its existence within a user's EWD.js session.

A key property of a *GlobalNode* Object is *_value*. This is a read/write property that allows you to inspect or set the value of the physical *Global Node* in the Mumps database, eg:

```
var zipNode = new ewd.mumps.GlobalNode('patient', [123456, "address", "zipcode"]);
var zipCode = zipNode._value;  // 10027
console.log("Patient's zipcode = " + zipCode);
```

When you access the *_value* property, you're accessing the physical *Global Node*'s value on disk in the Mumps database, but of course we're doing so via a JavaScript object, in this case named *zipNode*. Note that in the example above, the first line that sets up the pointer to the *Global Node* does not actually access the Mumps database: indeed the physical Mumps *Global Node* may not even exist in the database when then pointer is created. It's only when a *GlobalNode* Object's method is used that requires physical access to the database that a physical linkage between the *GlobalNode* Object and the physical Mumps *Global Node* is made.

Of course, ideally we'd like to be able to do the following:

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var name = patient.name._value;
```

But there's a problem: the dynamic schema-free nature of Mumps persistent arrays means that there is no way in advance of knowing that the physical Mumps *Global Node* representing the *patient GlobalNode* object actually has a subscript of *name* and therefore no way to know in advance that it's possible instantiate a corresponding property of *patient* called *name*. In theory, *ewdGlobals.js* could instantiate as a property every subscript that physically exists under a specified Mumps *Global Node*. However a *Global Node* might have thousands or tens of thousands of subscripts: it could take a significant amount of time and processing power to find and instantiate every subscript as a property and it would consume a lot of memory within Javascript in doing so. Furthermore, in a typical EWD.js application, you only need access to a small number of *GlobalNode* properties at any one time, so it would be very wasteful to have them all instantiated and then only use one or two.

In order to deal with this, a *GlobalNode* Object has a special method available to it called *_getProperty()*, normally abbreviated to *$()* (taking a leaf out of jQuery's book!). The *$()* method does two things:

- instantiates the specified subscript name as a property of the parent *GlobalNode* object
- returns another *GlobalNode* object that represents the lower-subscripted physical *Global Node*.

For example:

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var nameObj = patient.$('name');
var name = nameObj._value;
```

What this code does is to first create a *GlobalNode* object that points to the physical Mumps *Global Node*:

patient(123456)

The second line does two things:

- extends the *patient* object with a property named *name*;
- returns a new *GlobalNode* object that points to the physical *Global Node*:

    ^patient(123456, "name")

These two effects of using the *$()* method are both very interesting and powerful. First, now that we've used it once as a pointer to the *name* subscript, *name* has now been properly instantiated as an actual property of patient, so we can subsequently refer directly to the *name* property instead of using the *$()* method again. So, extending the example above, we can now change the patient's name by referring directly to the patient's name property:

```
patient.name._value = "James Smith";
```

Secondly, because the *$()* method returns a *GlobalNode* object (which may or may not exist or have a data value), we can chain them as deep as we like. So we can get the *patient*'s town like this:

```
var town = patient.$('address').$('town')._value;
```

Each of the chained *$()* method calls has returned a new *GlobalNode* object, representing that level of subscripting, so we now have the following physical Mumps *Global Nodes* defined as objects:

patient - representing the physical Mumps Global Node: patient(123456)

patient.address - representing patient(123456,"address")

patient.address.town - representing patient(123456,"address","town")

So we can now use those properties instead of using *$()* again for them. For example, to get the zip code, we just need to use the *$()* method for the zipcode property (since we've not accessed it yet):

```
var zip = patient.address.$('zipcode')._value;
```

But if we want to report the patient's town, we already have all the properties instantiated, so we don't need to use the *$()* method at all, eg:

```
console.log("Patient is from " + patient.address.town._value);
```

As you can see, therefore, the projection provided by the *ewdGlobals.js* file within EWD.js provides a means of handling data within a Mumps database as if it was a collection of persistent JavaScript objects. The fact that you are manipulating data stored on disk is largely hidden from you.

GlobalNode Properties and Methods

A *GlobalNode* Object has a number of methods and properties that you can use to inspect and manipulate the physical Mumps *Global Node* that it represents:

Method / Property	Description
$()	Returns a *GlobalNode* Object that represents a subscripted sub-node of the Mumps *Global Node* represented by the current *GlobalNode* Object. The $() method specifies the name of the subscript to be instantiated as a new *GlobalNode* Object
_count()	Returns the number of subscripts that exist under the Mumps *Global Node* represented by the current *GlobalNode* Object
_delete()	Physically deletes any data for the current *GlobalNode* Object and physically deletes any Mumps *Global Nodes* with lower-level subscripting beneath the specified *GlobalNode*. Note that any Javascript *GlobalNode* objects that you may have instantiated for lower-level subscripts continue to exist, but their properties that relate to their physical values will have changed (eg their *_value*).
_exists	true if the *GlobalNode* Object physically exists as a Mumps *Global Node*. Note that it may exist as either an intermediary or leaf *Global Node*
_first	Returns the name of the first subscript under the Mumps *Global Node* represented by the current *GlobalNode* Object
_forEach()	Iterator function, firing a callback function for every subscript that exists under the Mumps *Global Node* represented by the *GlobalNode* Object.
_forPrefix()	Iterator function, firing a callback function for every subscript that exists under the Mumps *Global Node* represented by the *GlobalNode* Object, where the subscript name starts with the specified prefix.
_forRange()	Iterator function, firing a callback function for every subscript that exists under the Mumps *Global Node* represented by the *GlobalNode* Object, where the subscript name falls within a specified alphanumeric range.
_getDocument()	Retrieves the sub-tree of Mumps *Global Nodes* that physically exists under the Mumps *Global Node* represented by the current *GlobalNode* Object, and returns it as a JSON document.
_hasProperties	true if the Mumps *Global Node* represented by the *GlobalNode* Object has one or more existing subscripts (and therefore potential properties of the *GlobalNode* Object) underneath it.
_hasValue	true if the *GlobalNode* Object physically exists as a Mumps *Global Node* and has a value saved against it.
_increment()	Performs an atomic increment of the current *GlobalNode* Object's *_value* property.
_last	Returns the name of the last subscript under the Mumps *Global Node* represented by the current *GlobalNode* Object
_next()	Returns the name of the next subscript following the one specified, under the Mumps *Global Node* represented by the current *GlobalNode* Object
_parent	Returns the current *GlobalNode* Object's parent Node as a *GlobalNode* Object
_previous()	Returns the name of the next subscript preceding the one specified, under the Mumps *Global Node* represented by the current *GlobalNode* Object
_setDocument()	Saves the specified JSON document as a sub-tree of Mumps *Global Nodes* under the Mumps *Global Node* represented by the current *GlobalNode* Object
_value	Read/write property, used to get or set the physical value of the Mumps *Global Node* represented by the current *GlobalNode* Object.

Examples

Suppose we have the following data stored in a Mumps persistent array:

```
patient(123456,"birthdate")=-851884200
patient(123456,"conditions",0,"causeOfDeath")="pneumonia"
patient(123456,"conditions",0,"codes","ICD-10-CM",0)="I21.01"
patient(123456,"conditions",0,"codes","ICD-9-CM",0)="410.00"
patient(123456,"conditions",0,"description")="Diagnosis, Active: Hospital Measures"
patient(123456,"conditions",0,"end_time")=1273104000
```

_count()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var count = patient._count();                    // 2: birthdate and conditions
var count2 = patient.$('conditions').$(0)._count();  // 4: causeOfDeath, codes, description, end_time
```

_delete()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
patient.$('conditions')._delete(); // will delete all nodes under and including the conditions subscript

// all that would be left in the patient persistent array would be:

// patient(123456,"birthdate")=-851884200
```

_exists

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var exists1 = patient._exists;                 // true
var exists2 = patient.$('conditions')._exists; // true
var exists3 = patient.$('name')._exists;       // false

var dummy = new ewd.mumps.GlobalNode('dummy', ['a', 'b']);
var exists1 = dummy._exists;                  // false
```

_first

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var first1 = patient._first;                     // birthdate
var first2 = patient.$('conditions').$(0)._first; // causeOfDeath
```

_forEach()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
patient._forEach(function(subscript, subNode) {
  // subscript = next subscript found under the patient GlobalNode
  // subNode = GlobalNode object representing the sub-node with the returned subscript
  var value = 'intermediate node';
  if (subNode._hasValue) value = subNode._value;
  console.log(subscript + ': ' + value);
  }
});

would display:

 birthdate: -851884200
 conditions: intermediate node
```

You can reverse the direction of the iterations by adding {direction: 'reverse'} as a first argument to the forEach() function.

_forPrefix()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
patient.$('conditions').$(0)._forPrefix('c', function(subscript, subNode) {
  // subscript = next subscript found under the patient GlobalNode
  // subNode = GlobalNode object representing the sub-node with the returned subscript
  var value = 'intermediate node';
  if (subNode._hasValue) value = subNode._value;
  console.log(subscript + ': ' + value);
  }
});

would display:

 causeOfDeath: pneumonia
 codes: intermediate node
```

You can reverse the direction of the iterations by replacing the first argument with an object:

{prefix: 'c', direction: 'reverse'}

_forRange()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
patient.$('conditions').$(0)._forRange('co', 'de', function(subscript, subNode) {
  // subscript = next subscript found under the patient GlobalNode
  // subNode = GlobalNode object representing the sub-node with the returned subscript
  var value = 'intermediate node';
  if (subNode._hasValue) value = subNode._value;
  console.log(subscript + ': ' + value);
  }
});

would display:

 codes: intermediate node
 description: Diagnosis, Active: Hospital Measures
```

You can reverse the direction of the iterations by replacing the first two arguments with an object:

{from: 'co', to: 'de', direction: 'reverse'}

_getDocument()

```
 var patient = new ewd.mumps.GlobalNode('patient', [123456]);
 var doc = patient._getDocument();
 console.log(JSON.stringify(doc, null, 3);

would display:

{
  birthdate: -851884200,
  conditions: [
    {
      causeOfDeath: "pneumonia",
      codes: {
        ICD-9-CM: [
          "410.00"
        ],
        ICD-10-CM: [
          "I21.01"
        ]
      },
      description: "Diagnosis, Active: Hospital Measures",
      end_time: 1273104000
    }
  ]
};

====================

var patient = new ewd.mumps.GlobalNode('patient', [123456]);
 var doc = patient.$('conditions').$(0)._getDocument();
 console.log(JSON.stringify(doc, null, 3);

would display:

 {
      causeOfDeath: "pneumonia",
      codes: {
        ICD-9-CM: [
          "410.00"
        ],
        ICD-10-CM: [
          "I21.01"
        ]
      },
      description: "Diagnosis, Active: Hospital Measures",
      end_time: 1273104000
    }
```

Note: *_getDocument()* has two optional arguments:

- **base**: This defaults to zero, and defines the first subscript in the Mumps global node to which the corresponding first element in the JavaScript array should be mapped. If you are using *_getDocument()* to fetch from a legacy Mumps Global, you'll probably want to set base to 1, since, by convention, most legacy Mumps arrays are 1-based;
- **useArrays**: This defaults to *true*, which means that it will check to see whether a numeric subscript can map to a JavaScript array. In order to do this, it first iterates through all values of this subscript until it finds either a break in the numeric sequence or a non-numeric value, in which case it will opt instead to map to an object. This inevitably comes at some cost in terms of performance. If you are happy for *_getDocument()* to always create objects for each subscript level in the Global Node's hierarchy (ie even if a subscript could have mapped to an array), then you can set *useArrays* to *false*. Doing so will very significantly improve the performance of *_getDocument()*. Note that if *useArrays* is false, the *base* argument is ignored.

```
Example:

If the persistent Mumps data contained the following:

patient(123456,"conditions",0,"codes","ICD-9-CM",1)="410.00"
patient(123456,"conditions",0,"codes","ICD-9-CM",2)="410.01"
patient(123456,"conditions",0,"codes","ICD-9-CM",3)="410.02"


Then:

  var patient = new ewd.mumps.GlobalNode('patient', [123456, 'conditions', 0, 'ICD-9-CM']);
  var doc = patient._getDocument(1);
  console.log(JSON.stringify(doc));

 would display:

["410.00", "410.01", "410.02"]


And:

 var patient = new ewd.mumps.GlobalNode('patient', [123456, 'conditions', 0, 'ICD-9-CM']);
  var doc = patient._getDocument(0, false);
  console.log(JSON.stringify(doc));

 would display:

  {"1": "410.00", "2": "410.01", "3": "410.02"}
```

_hasProperties

```
  var patient = new ewd.mumps.GlobalNode('patient', [123456]);
  var hp1 = patient._hasProperties;                  // true
  var hp2 = patient.$('birthdate')._hasProperties;   // false (no sub-nodes under this node)
  var hp3 = patient.$('conditions')._hasProperties;  // true
```

_hasValue

```
  var patient = new ewd.mumps.GlobalNode('patient', [123456]);
  var hv1 = patient._hasValue;                       // false
  var hv2 = patient.$('birthdate')._hasValue;        // true
  var hv3 = patient.$('conditions')._hasValue;       // false
```

_increment()

```
  var patient = new ewd.mumps.GlobalNode('patient', [123456]);
  var count = patient._$('counter')._increment(); // 1
  count = patient.counter._increment();           // 2
  var counterValue = patient.counter._value;      // 2
```

_last

```
  var patient = new ewd.mumps.GlobalNode('patient', [123456]);
  var last1 = patient._last;                      // conditions
  var last2 = patient.$('conditions').$(0)._last;   // end_time
```

_next()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var next = patient._next('');                    // birthdate
next = patient._next(next);                      // conditions
next = patient._next(next);                      // empty string: ''
next = patient.$('conditions').$(0)._next('');  // causeOfDeath
```

_parent

```
var conditions = new ewd.mumps.GlobalNode('patient', [123456, 'conditions']);
var patient = conditions._parent;

// patient is the same as if we'd used:

var patient = new ewd.mumps.GlobalNode('patient', [123456]);
```

_previous()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var prev = patient._previous('');                   // conditions
prev = patient._previous(prev);                     // birthdate
prev = patient._previous(prev);                     // empty string: ''
prev = patient.$('conditions').$(0)._previous('');  // end_time
```

_setDocument()

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
var doc = {
  "name": "John Doe",
  "city": "New York",
  "treatments" : {
    "surgeries" : [ "apendicectomy", "biopsy" ],
    "radiation" : [ "gamma", "x-rays" ],
    "physiotherapy" : [ "knee", "shoulder" ]
   },
};

patient._setDocument(doc, true);

 would result in the Mumps persistent array now looking like this:


patient(123456,"birthdate")=-851884200
patient(123456,"city")="New York"
patient(123456,"conditions",0,"causeOfDeath")="pneumonia"
patient(123456,"conditions",0,"codes","ICD-10-CM",0)="I21.01"
patient(123456,"conditions",0,"codes","ICD-9-CM",0)="410.00"
patient(123456,"conditions",0,"description")="Diagnosis, Active: Hospital Measures"
patient(123456,"conditions",0,"end_time")=1273104000
patient(123456,"name")="John Doe"
patient(123456,"treatments","surgeries",0)="appendicectomy"
patient(123456,"treatments","surgeries",1)="biopsy"
patient(123456,"treatments","radiation",0)="gamma"
patient(123456,"treatments","radiation",1)="x-rays"
patient(123456,"treatments","physiotherapy",0)="knee"
patient(123456,"treatments","physiotherapy",1)="shoulder"

Note that the new data from the JSON document is merged with any existing data
```

Note: *_setDocument()* has two additional optional arguments:

- **fast**: This second argument defaults to *false*, which means that it will create GlobalNode Objects for each physical Global Node it creates as it traverses the JavaScript object. This inevitably comes with a performance cost and memory

overhead which can quickly become significant if the JavaScript object that you are saving is very big. ***Under most circumstances, it's a good idea to set this second argument to true,*** which stops the creation of intermediate GlobalNode objects**.**

- **base**: This third argument defaults to zero, and defines the first subscript in the Mumps global node to which the corresponding first element in the JavaScript array should be mapped. If you are using *_setDocument()* to write back array data to a legacy Mumps Global, you'll probably want to set base to 1, since, by convention, most legacy Mumps arrays are 1-based;

_value

```
var patient = new ewd.mumps.GlobalNode('patient', [123456]);
// getting values:

var birthdate = patient.$('birthdate')._value;                 // -851884200
var val1 = patient.$('conditions')._value;                     // null string (because intermediate node)
var val3 = patient.conditions.$(0).$('causeOfDeath')._value;   // pneumonia

// setting values

patient.$('address').$('zipcode')._value = 1365231;

// would add the following node into the Mumps persistent array:

// patient(123456,"address","zipcode")=1365231
```

Other functions provided by the *ewd.mumps* Object in EWD.js

In addition to the *GlobalNode()* constructor that has been described in detail above, the *ewd.mumps* object provides several additional functions that may be used within your EWD.js applications.

ewd.mumps.function()

This is available for use on GT.M and Caché databases, but not GlobalsDB. This is because GlobalsDB is just a Mumps database engine and does not include a Mumps language processor.

The *ewd.mumps.function()* API allows you to invoke legacy code that is written in the Mumps language. Note that this API only allows you to invoke what are known in Mumps language parlance as extrinsic functions. If you want to invoke procedures or Caché classes, you'll need to wrap them inside a Mumps extrinsic function wrapper.

The syntax for using the ewd.mumps.function() API is as follows:

```
var response = ewd.mumps.function('[label]^[routineName]', [arg1] ..[,argn]);

  where:
    label       = extrinsic function label
    routineName = name of Mumps routine containing the function
    arg1..argn  = arguments
    response    = string value containing the returnValue of the function

eg:

var result = ewd.mumps.function('getPatientVitals^MyEHR', params.patientId, params.date);

 This is the equivalent of the Mumps code:

    set result=$$getPatientVitals^MyEHR(patientId,date)
```

Note that the maximum returnValue string length for this API when using Caché is just 4k. if the function is going to return a lot of data, eg a large JSON document, then it's a good idea to save the data to the user's EWD.js Session and then recover it from the session within your EWD.js JavaScript module.

To do this, you'll need to wrap the legacy code in a function to which you pass the EWD.js Session Id, eg:

```
onSocketMessage: function(ewd) {

  var wsMsg = ewd.webSocketMessage;
  var type = wsMsg.type;
  var params = wsMsg.params;
  var sessid = ewd.session.$('ewd_sessid')._value;

  if (type === 'getLegacyData') {
    var result = ewd.mumps.function('getLegacyData^myOldStuff', params.patientId, sessid);
    // legacy function saves the output to an EWD.js Session Array named 'legacyData'
    //
    //    eg merge ^%zewdSession("session",sessid,"legacyData") = legacyData
    //
    // once the function has completed, pick up the data
    var document = ewd.session.$('legacyData')._getDocument();
    // legacy data is now held in a JSON document
    ewd.session.legacyData._delete(); // clear out the temporary data (would be garbage-collected later anyway)
  }

}
```

ewd.mumps.deleteGlobal()

This should be used with care. It will delete an entire named Mumps persistent array. Note that Mumps databases provide no *undelete* capability, so this function can be extremely dangerous: in one command you can instantly and permanently delete an entire database!

Usage:

ewd.mumps.deleteGlobal('myGlobal');

will immediately and permanently delete a Mumps persistent array named *myGlobal*

ewd.mumps.getGlobalDirectory()

Returns an array containing the names of all the persistent arrays within your Mumps database

ewd.mumps.version()

Returns a string that identifies both the version of the Node.js/Mumps interface and the version and type of Mumps database being used.

Indexing Mumps Data

About Mumps Indices

Indexing Mumps persistent arrays is the key to creating high-performance, flexible and powerful databases. Traditionally, indices were the responsibility of the developer to maintain, often buried inside functions or APIs that handled the updating of data in a Mumps database.

An index in a Mumps database is just another persistent array. Sometimes, the same named array is used, but a different set of subscripts are used for the index. At other times, indices can be held in differently-named persistent arrays. The choice is somewhat arbitrary, but can be a combination of personal style and/or data management issues. For example, if the main data and indices are held in a single named persistent array, then the array can become very large and will require backing up in one great big process. If the indices are kept in one or more separately-named arrays, then the main data array can be smaller, easier to back up. Keeping indices in separately-named arrays can also be beneficial if you're using mechanisms such as journalling or mapping arrays.

Let's take a simple example: consider a patient database where we store a patient's last name (amongst many other data about a patient).

```
patient(123456,"lastName")="Smith"
patient(123457,"lastName")="Jones"
patient(123458,"lastName")="Thornton"
 ...etc
```

If we want to create a lookup facility where the user can search for patients by last name and select one from a matching list, what we don't want to have to do is traverse the entire patient array, id by id, looking, exhaustively for all lastNames of *Smith*. Instead we can create and maintain a parallel persistent array that stores a lastName index. It's up to us what we name this array, and what subscripts we use, but here's one way it could be done:

```
patientByLastName("Jones",123457)=""
 ....
patientByLastName("Smith",101430)=""
patientByLastName("Smith",123456)=""
patientByLastName("Smith",128694)=""
patientByLastName("Smith",153094)=""
patientByLastName("Smith",204123)=""
patientByLastName("Smith",740847)=""
 ...
patientByLastName("Thornton",123458)=""
 ...etc
```

With such an index array, to find all the patients with a last name of Smith, all we need to do is use the _forEach() method to iterate through the Smith records to get the patient Ids for all the matching patients. The patient Id then provides us with the pointer we need to access the patient record that the user has selected, eg:

```
var smiths = new ewd.mumps.GlobalNode('patientByLastName', ['Smith']);
smiths._forEach(function(patientId) {
  // do whatever is needed to display the patient Ids we find,
  // eg build an array for use in a menu component
});


// once a patientId is selected, we can set a pointer to the main patient array:

var patient = new ewd.mumps.GlobalNode('patient', [patientId]);
```

If we wanted to find all names starting with a particular prefix, eg *Smi*, we could use the *_forPrefix()* method instead, or we could use the *_forRange()* method to find all names between *Sma* and *Sme*.

An improvement to the index might be to convert the last name to lower-case before using it in the index, eg:

```
patientByLastName("jones",123457)=""
 ....
patientByLastName("smith",101430)=""
patientByLastName("smith",123456)=""
patientByLastName("smith",128694)=""
patientByLastName("smith",153094)=""
patientByLastName("smith",204123)=""
patientByLastName("smith",740847)=""
 ...
patientByLastName("thornton",123458)=""
 ...etc
```

This would ensure you don't miss patients who have, for example, hyphenated last names.

Of course, an index by last name is just one of many you'll probably want to maintain for a patient database. For example, we might want to have an index by birthdate, and one by gender. We could create specifically-named arrays for these, but this could become unwieldy and difficult to maintain and remember if we have lots of them. As an alternative, we might want to maintain all indices in one single persistent array. We could do this easily by adding a first subscript that denotes the index type, for example:

```
patientIndex("gender","f",101430)=""
patientIndex("gender","f",123457)=""
patientIndex("gender","f",204123)=""
 .....
patientIndex("gender","m",123456)=""
patientIndex("gender","m",128694)=""
patientIndex("gender","m",153094)=""

 ...etc


patientIndex("lastName","jones",123457)=""
 ....
patientIndex("lastName","smith",101430)=""
patientIndex("lastName","smith",123456)=""
patientIndex("lastName","smith",128694)=""
patientIndex("lastName","smith",153094)=""
patientIndex("lastName","smith",204123)=""
patientIndex("lastName","smith",740847)=""
 ...
patientIndex("lastName","thornton",123458)=""
 ...etc
```

This is clearly extensible also: we can add new index types by simply using a new first subscript.

Sometimes, you'll want to search across two or more parameters, eg last name and gender. You can, of course, design your logic to use combinations of simple indices, but you can, of course, create and maintain multi-parameter indices, eg:

```
patientIndex("genderAndName","f","Smith",101430)=""
patientIndex("genderAndName","m","Thornton",123458)=""

 ...etc
```

Effective index design for Mumps databases is something that comes with experience. You can hopefully see that it is infinitely flexible, and your index design is entirely dependent on your skill and understanding of how your applications need to be built around the data that drives it.

Tip: only hold data or values in an index that originate in the main data array or can be derived from data in the main data array. If you do this, you can recreate or rebuild your indices from just the main data array.

Maintaining Indices in EWD.js

Mumps indices don't create themselves, unfortunately. It's down to you, the developer, to create and maintain the indices you need every time you create, change or delete Mumps *Global Nodes*. For a traditional Mumps developer, this represented quite a chore, unless a set of APIs had been designed to handle database updates. Failing to create all the necessary indices consistently throughout an application could lead to missing index records and therefore erroneous searches.

EWD.js makes things a lot slicker and efficient by using the ability of Node.js to emit events that can be handled by the developer. Therefore, the *ewdGlobals.js* module emits events whenever you use the *_value*, *_delete()* and *_increment()* APIs: ie the methods that change data values in your Mumps database.

Events emitted are:

- **beforesave:** emitted immediately before a value is set into a Mumps *Global Node*. The event passes you a pointer to the *GlobalNode* Object that is about to be changed
- **aftersave:** emitted immediately after a value is set into a Mumps *Global Node*. The event passes you a pointer to the corresponding *GlobalNode* Object. Two additional properties - *oldValue* and *newValue* - are made available in the *GlobalNode* Object. *oldValue* will be a null string if no value previously existed before the save event. Because the *oldValue* is made available via the *aftersave* event, in most circumstances you'll be able to just use the *aftersave* event to update your indices.
- **beforedelete:** emitted immediately before a Mumps *Global Node* is deleted. The event passes you a pointer to the *GlobalNode* Object that is about to be deleted.
- **afterdelete:** emitted immediately after a Mumps *Global Node* is deleted. The event passes you a pointer to the *GlobalNode* Object that was deleted. An additional property - *oldValue* is made available in the node object. In many circumstances, you'll be able to just use the *afterdelete* event. However, remember that the *_delete()* method may have been applied to an intermediate-level Global Node, in which case the deleted values at lower-levels of subscripting may need to be taken into account if those values were used in indices. In this case, the *beforedelete* event will allow you to traverse the sub-nodes that are about to be deleted and modify any indices appropriately.

The globalIndexer Module

Included in the *ewd.js* installation kit is a module named *globalIndexer.js* which you'll find in the *node_modules* directory under your EWD.js Home Directory. This module is automatically loaded by EWD.js to provide event handlers for the above events whenever they occur.

You'll find that stubs have been created in *globalIndexer.js* that you should extend and maintain as appropriate to the Mumps arrays you want to maintain and the corresponding indices you'll require for them. You'll see a simple example that has been defined for indexing the Mumps persistent array used in the *demo* application.

By using the *globalIndexer* module, you'll be able to specify the indexing of all your main Mumps data arrays in one place. Your applications simply need to focus on the changes that need to be made to your main Mumps data array(s). The events emitted by *ewdGlobals.js* and handled by your logic within the *globalIndexer* module means that indexing can be handled automatically and consistently, without cluttering up your main application logic.

Note that *ewd.js*, when running, detects any changes you make to the *globalIndexer* module and automatically reloads it into any child processes that are already using it. You shouldn't need to stop and restart the *ewd.js* module whenever you modify the *globalIndexer* module.

Web Service Interface

EWD.js-based Web Services

Whilst EWD.js is primarily a framework for creating and running WebSocket-based applications that run in a browser, it also provides a secure mechanism for exposing back-end JavaScript business functions as web services. EWD.js web services return a JSON response as an *application/json* HTTP response.

The EWD.js application developer simply needs to write a function within a back-end module that accesses the Mumps database and/or uses legacy Mumps functions, in exactly the same way as he/she would if writing standard EWD.js back-end websocket message-handling functions. The function should return a JSON object that either contains an error string or a JSON document. EWD.js looks after the HTTP request parsing and handling, packaging up the developer's JSON response as an HTTP response, and authenticating incoming web service HTTP requests.

Web Service Authentication

Security is provided via an HMAC-SHA authentication mechanism that is based on the security model used by Amazon Web Services in, for example, their SimpleDB database: See http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/HMACAuth.htm for details of the mechanics on which EWD.js's security is based.

If a user is to be granted access to applications on an EWD.js system, they are registered using the *ewdMonitor* application (click the Security tab). Each user must be given a unique *accessId*. Against that *accessId* are recorded:

- a secret key, which should be a long alphanumeric string that is difficult to guess
- a list of EWD.js applications that the user is allowed to access

The user must be sent their *accessId* and secret key: this should be done in a secure way. It is important that the user does not share or misplace these credentials.

The secret key is used by the user's client software to digitally sign his/her HTTP requests. The user's *accessId* and signature are added to the HTTP request as queryString name/value pairs.

When the EWD.js system receives a web service request:

- the *accessId* is first checked to see whether it is recognised or not
- next, a check is made to ensure that the *accessId* has the right to access the specified application
- EWD.js then checks the incoming signature value against the value that it calculates from the incoming request, using the secret key it holds for the *accessId* that it received in the HTTP request.
- If the signatures match, the user is deemed to be valid and the request is processed.

If an error is detected in any of these steps, an error response is sent to the user and no further processing takes place.

Creating an EWD.js Web Service

The business logic of an EWD.js Web Service is written in JavaScript as a function within a Node.js module. The name of the module is used as the application name. If you wish, and if it makes sense to do so, EWD.js Web Service functions can be included within a module that is also used for a browser-based application. You'll see an example of this in the *demo.js* module that is included in the *ewd.js i*nstallation kit:

```
module.exports = {
  webServiceExample: function(ewd) {
    var session = new ewd.mumps.GlobalNode('%zewdSession', ["session", ewd.query.sessid]);
    if (!session._exists) return {error: 'EWD.js Session ' + ewd.query.sessid + ' does not exist'};
    return session._getDocument();
  }
};
```

This is a simple web service that will return the contents of a specified EWD.js Session as a JSON document. If the Session Id is invalid, an error JSON document is sent instead.

In the example above, the EWD Web Service *Application Name* is *demo*: the same as the module name (ie without the .js file extension). The module should reside in the *node_modules* directory under your EWD.js Home Directory.

The method to be invoked is known as the Web *Service Name*, in this case *webServiceExample.* It has a single argument which is an object that is automatically passed to it by EWD.js at run-time. By convention we name that object *ewd*. The ewd object for an EWD.js Web Service function contains two sub-objects:

- **mumps**: providing access to the Mumps global storage and legacy Mumps functions. This is used identically to websocket message-handling functions.
- **query**: containing the incoming name/value pairs that were in the incoming HTTP request's queryString. It's up to the developer to determine the names and expected values of the name/value pairs required by his/her function that must be provided by incoming HTTP requests.

As soon as the function is written and the module saved, it is available as a web service in EWD.js.

That's all there is to it as far as the developer is concerned.

Invoking an EWD.js Web Service

An EWD.js web service is invoked using an HTTP request of the following structure:

http(s)://[hostname]:[port]/json/[application name]/[service name]?name1=value1&name2=value2....etc

The following must be included in the queryString as name/value pairs:

- name/value pairs required by the specific service function that is being requested, plus:
- **accessId**: the registered accessId for the user sending the HTTP request
- **timestamp:** the current date/time in JavaScript *toUTCString()* format (http://www.w3schools.com/jsref/jsref_toutcstring.asp)
- **signature:** the HMAC-SHA256 digest value calculated from the normalised name/value pair string (see the earlier link to the Amazon Web Services SimpleDB authentication for details of the normalisation algorithm that is also used in EWD.js).

eg:

```
https://192.168.1.89:8080/json/demo/webServiceExample?
  id=1233&
  accessId=rob12kjh1i23&
  timestamp=Wed, 19 Jun 2013 14:14:35 GMT&
  signature=P0blakNehj2TkuadxbKRsIgJCGlhY1EvntJdSce5XvQ=
```

Of course the name/value pairs must be URI encoded before they are sent, so the request will actually be as follows:

```
https://192.168.1.89:8080/json/demo/webServiceExample?
  id=1233&
  accessId=rob12kjh1i23&
  timestamp=Wed%2C%2019%20Jun%202013%2014%3A14%3A35%20GMT&
  signature=P0blakNehj2TkuadxbKRsIgJCGlhY1EvntJdSce5XvQ=
```

The Node.js EWD.js Web Service Client

In order to make it easy to use EWD.js Web Services, we've created an Open Source Node.js module that will automate the process of sending correctly signed requests and handling their responses.

The module is published at: https://github.com/robtweed/ewdliteclient

Installation is very simple: use npm:

```
npm install ewdliteclient
```

Included in the module is an example showing how to invoke the *webServiceExample* service that is defined in the *demo.js* module. You'll need to edit the parameters to the correct host IP address / domain name, port etc to match your EWD.js system. You'll also need to change the accessId to one that you have registered on your EWD.js system (see next section).

To try out the EWD Web Service Client in the Node.js REPL:

```
node
>  var client = require('ewdliteclient')
   undefined
>  client.example(1234) // runs the example - getting EWD.js Session Id 1234
>  results
 ...should list the Session data (if it exists) as a JSON document if it correctly
    accessed and ran the EWD.js web service
```

To use the EWD.js Web Service client in your own Node.js application, you do the following:

```
var client = require('ewdliteclient');
var args = {
  host: '192.168.1.98',  // ip address / domain name of EWD.js server
  port: 8080,            // port on which EWD.js server is listening
  ssl: true,             // true if EWD.js server is set up for HTTPS / SSL access
  appName: 'demo',       // the application name of the EWD.js Web Service you want to use
  serviceName: 'webServiceExample',  // the EWD.js service name (function) you wish to invoke
                                     // for the specified application
  params: {
    // query string name/value pairs
    accessId: 'rob',  // required by EWD.js's security
    sessid: 1233          // Session Id (required by the application/service you're invoking)
  },
  secretKey: 'a1234567'  // the secretKey for the specified accessId
                         // this must be registered on the EWD.js system
};

client.run(args, function(error, data) {
  // do whatever you need to do with the returned JSON data, eg:

  if (error) {
    console.log('An error occurred: ' + JSON.stringify(error));
  }
  else {
    console.log('Data returned by web service: ' + JSON.stringify(data));
  }
});
```

Equivalent clients can be developed for other languages. Use the Node.js client and the Amazon SimpleDB documentation as a guide to implementing the correct signing algorithm.

Registering an EWD.js Web Service User

You can use the EWDMonitor Application (see the earlier chapter on using this application). Just click the Security Tab: you can add new users and maintain or delete existing ones from within its UI which should be self-explanatory.

You start the EWDMonitor application in a browser by using the URL:

```
http://127.0.0.1:8080/ewd/ewdMonitor/index.html
```

Change the IP address or host name and port appropriately according to your EWD.js configuration.

Programmatic Registrations

It is possible to create a registered user programmatically. For an example, see https://github.com/robtweed/ewd.js/blob/master/OSEHRA/registerWSClient.js as a starting point. You'll need to modify paths etc according to your EWD.js configuration. You'll also need to modify the *EWDLiteServiceAccessId* object that is defined within the *zewd.setDocument()* method within the registerWSClient.js file.

Your version should be saved in and run from your EWD.js Home Directory, eg:

```
cd ~/ewdjs
node registerWSClient.js
```

Converting Mumps Code into a Web Service

EWD.js makes it very straightforward to expose new or legacy Mumps code as a JSON/HTTP web service that can be invoked securely. It involves building two simple wrappers around your Mumps code:

Inner Wrapper function

This function should surround the Mumps code that you want to expose as a web service. Note that you can also use this technique to expose Caché class methods. The wrapper function has two arguments:

- **inputs**: a local array that contains all input name/value pairs required by your Mumps code
- **outputs**: a local array that contains the outputs from your Mumps code. This array can be as complex and as deeply-nested as you wish. EWD.js will automatically convert your outputs array contents to a corresponding JSON response object that will be returned from the Web Service.

The wrapper function should use *New* commands to ensure that variables created by your Mumps code does not leak out of the function. Additionally, all data required by your Mumps code ***must*** be supplied by the *inputs* array: you ***cannot*** rely on any globally-scoped data being available and leaking into the wrapper function.

The example below shows how the VistA procedure GET^VPRD can be wrappered for use with EWD.js:

```
vistaExtract(inputs,outputs) ;
 ;
 ;  ensure nothing leaks out from this function
 ;
 new dfn,DT,U,PORTAL
 ;
 ; create the inputs required by GET^VPRD
 ;  some are constants, some come from the inputs array
 ;
 set U="^"
 set DT=$p($$NOW^XLFDT,".")
 set PORTAL=1
 set dfn=$g(inputs("IEN"))
 ;
 ; now we can call the procedure
 ;
 do GET^VPRD(,dfn,"demograph;allerg;meds;immunization;problem;vital;visit;appointment",,,,)
 ;
 ; now transfer the results into the outputs array
 ;
 merge outputs=^TMP("VPR",$j)
 set outputs(0)="VISTA Extract for patient IEN "_dfn
 ;
 ; tidy up and finish
 ;
 kill ^TMP("VPR",$j)
 QUIT ""
 ;
```

Save this in a Cache routine named ^vistADemo or in a GT.M routine file named vistADemo.m

So now we have a clear, self-contained, re-usable function that we can use to invoke our legacy VistA procedure.

Outer Wrapper Function

The next step is to create a second, outer function wrapper. This provides a standard, normalised interface that EWD.js can invoke. It maps the inputs and outputs array into a temporary Global that EWD.js knows to use. This Global is named ^*%zewdTemp*. By adding a first subscript that defines the process Id, we can ensure that multiple EWD.js processes won't clobber data written to this temporary Global. The process Id will passed automatically to the outer wrapper function by EWD.js.

This is all we need to add to the Cache routine or GT.M routine file that we created earlier:

```
vistaExtractWrapper(pid)
 ;
 new inputs,ok,outputs
 ;
 ; map the inputs array from EWD.js's temporary Global
 ;
 merge inputs=^%zewdTemp(pid,"inputs")
 ;
 ; invoke the inner wrapper function
 ;
 set ok=$$vistaExtract(.inputs,.outputs)
 ;
 ; map the outputs to EWD.js's temporary Global
 ;
 kill ^%zewdTemp(pid,"outputs")
 merge ^%zewdTemp(pid,"outputs")=outputs
 ;
 ; all done!
 QUIT ok
```

Invoking the Outer Wrapper from your Node.js Module

EWD.js provides a built-in function with which EWD.js can invoke your wrappered Mumps code:

ewd.util.invokeWrapperFunction(MumpsFunctionRef, ewd);

Here's an example of a back-end Node.js module that can be used to invoke the VistA function we wrappered earlier:

```
module.exports = {

  getVistAData: function(ewd) {
    return ewd.util.invokeWrapperFunction('vistaExtractWrapper^vistADemo', ewd);
  }

}
```

Invoking as a Web Service

Your Mumps code can now be invoked in the same way as any EWD.js web service. For example, if the back-end Node.js module shown above is named *vistATest.js*, we could send an HTTP(S) request from a client system looking something like this (URL encoding is not used for clarity). The IP address, port and values for accessId, timestamp and signature would vary of course:

```
https://192.168.1.89:8080/json/vistATest/getVistAData?
  IEN=123456&
  accessId=rob12kjh1i23&
  timestamp=Wed, 19 Jun 2013 14:14:35 GMT&
  signature=P0blakNehj2TkuadxbKRsIgJCGIhY1EvntJdSce5XvQ=
```

This will invoke the *getVistAData* method within the *vistATest* module, and *IEN* will be passed in automatically as an input. All name/value pairs in the URL (apart from *accessid, timestamp* and *signature*) are automatically mapped to the *inputs* array of any Mumps wrapper function that you invoke via the *ewd.invokeWrapper()* function. So if your Mumps function requires more inputs, simply add them to the name/value pair list of the HTTP request. Note that input names are case-sensitive.

The response that will be received back from this web service will be delivered as a JSON payload, containing the contents of the Mumps function's output array, expressed as an equivalent JSON object.

Invoking the Web Service from Node.js

If you want to invoke an EWD.js Web Service interface to your Mumps code from within a Node.js module (eg from EWD.js running on a remote system), then you can use the *ewdliteclient* module that was described earlier in this chapter. This

module will automatically look after all the digital signature mechanics, allowing you to just focus on defining the calling interface.

For example, the web service described above can be invoked from a remote EWD.js (or other Node.js) system as follows:

```
var client = require('ewdliteclient');
var args = {
  host: '192.168.1.98',  // ip address / domain name of EWD.js server
  port: 8080,            // port on which the remote EWD.js server is listening
  ssl: true,             // true if remote EWD.js server is set up for HTTPS / SSL access
  appName: 'vistATest',       // the application name of the EWD.js Web Service you want to use
  serviceName: 'getVistAData',  // the EWD.js service name (function) you wish to invoke
                                // for the specified application
  params: {
    // query string name/value pairs
    accessId: 'rob',  // required by EWD.js's security
    IEN: 123456          // passed into the Mumps function's inputs array
  },
  secretKey: 'a1234567'  // the secretKey for the specified accessId
                         // this must be registered on the remote EWD.js system
};

client.run(args, function(error, data) {
  // do whatever you need to do with the returned JSON data, eg:

  if (error) {
    console.log('An error occurred: ' + JSON.stringify(error));
  }
  else {
    console.log('Data returned by Mumps code: ' + JSON.stringify(data));
  }
});
```

Invoking the Web Service from other languages or environments

All you need to do in order to invoke an EWD.js Web Service interface to your Mumps code from any other language or environment is to construct a properly-signed HTTP request of the type shown earlier. You'll need to implement the signature rules according to the rules described in the Amazon Web Services documentation:

See http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/HMACAuth.htm

You can use the JavaScript implementation in the *ewdliteclient* module as a guide:

See: https://github.com/robtweed/ewdliteclient/blob/master/lib/ewdliteclient.js

Your client interface should expect a JSON response.

Turning off Web Service Security

During development and testing of your web services, it can be convenient to disable the built-in digital signature-based security.

Additionally, if you are using the EWD REST Server for external access (https://github.com/robtweed/ewdrest) and if (***and only if***) the EWD.js server is protected/fire-walled from external access, there are performance advantages in turning off the security mechanism.

To turn off Web Service security, add the following parameter in your EWD.js startup file:

```
  webservice: {
    authenticate: false
  }
```

Make sure you understand the security risks of doing this! Turning off the built-in security allows anyone to access and invoke any web services that your have defined!

Updating EWD.js

Updating EWD.js

ewd.js is continually being enhanced, so it's worth checking on GitHub to see whether an update has been released. Announcements will be made on Twitter: follow *@rtweed* to receive them.

EWD.js is updated by updating the *ewd.js module.*

Node.js and *npm* make it trivially simple to update *ewd.js*. Just navigate to your EWD.js Home Directory and use *npm install*, eg:

```
cd ~/ewdjs
npm install ewdjs
```

Updates and enhancements to the *ewdMonitor* application will be included in the installation package. You'll be asked to confirm the path in which EWD.js was originally installed, eg:

```
Install EWD.js to directory path (/home/ubuntu/ewdjs):
```

If the path suggested is correct, just press the Enter key, otherwise type in the correct path. The latest versions of the essential sub-components of EWD.js will be installed relative to this path.

You'll then be asked if you want to install the extra, optional sub-components for EWD.js:

```
EWD.js has been installed and configured successfully

Do you want to install the additional resources from the /extras directory?
If you're new to EWD.js or want to create a test environment, enter Y
If you're an experienced user or this is a production environment, enter N
Enter Y/N:
```

Follow the instructions, and EWD.js will now be fully updated and ready to restart.

Note that EWD.js applications, startup files etc that were originally created by the EWD.js install process will be replaced by this update procedure. Your own applications and/or differently-named files will remain unchanged. If you have modified any of the files that were originally installed by EWD.js, make sure you rename, copy and/or back them up before running the update in order to prevent losing your changes.

Appendix 1

Creating a GlobalsDB-based EWD.js/Ubuntu System From Scratch

Background

EWD.js includes an automated installation script for building a complete, fully-working GlobalsDB-based EWD.js system from scratch in just a few minutes. The starting point is a 64-bit Ubuntu desktop or server system, either a physical machine, virtual machine or EC2 instance. The EWD.js installer will install and configure:

- NVM (The Node.js Version Manager, allowing quick and simple updating of Node.js in the future)
- Node.js
- GlobalsDB
- NodeM
- EWD.js

Load and Run Installer

Start up a terminal session on your Ubuntu machine and do the following:

```
cd ~
sudo apt-get -y install git
git clone https://github.com/robtweed/ewd-installers
source ewd-installers/globalsdb/install.sh
```

During the installation process, you'll be asked to confirm the path in which EWD.js has been installed. Just accept the default that it suggests by pressing the Enter key, ie:

```
Install EWD.js to directory path (/home/ubuntu/ewdjs):
```

The essential sub-components of EWD.js will be installed relative to this path.

You'll then be asked if you want to install the extra, optional sub-components for EWD.js:

```
EWD.js has been installed and configured successfully

Do you want to install the additional resources from the /extras directory?
If you're new to EWD.js or want to create a test environment, enter Y
If you're an experienced user or this is a production environment, enter N
Enter Y/N:
```

If you're new to EWD.js, typing Y (followed by the Enter key) is recommended.

EWD.js is now ready for use!

Start Up ewd.js

```
cd ~/ewdjs
node ewdStart-globals
```

Run the *ewdMonitor* application

In your browser, enter the URL (changing the IP address appropriately for that assigned to your Ubuntu machine):

http://192.168.1.101:8080/ewd/ewdMonitor/index.html

If you want to use SSL, modify the defaults object in the *ewdStart-globals.js* file as shown below:

```
var defaults = {
  cwd: process.env.HOME + '/ewdjs',
  path: process.env.HOME + '/globalsdb/mgr',
  port: 8080,
  poolsize: 2,
  tracelevel: 3,
  password: 'keepThisSecret!',
  ssl: true
};
```

If you then get a warning about the SSL certificates, just tell the browser it's OK: it's because *ewd.js* is using self-signed certificates.

The *ewdMonitor* application should now spring into life and ask you for a password. This is defined in the *ewdStart-globals.js* startup file and is pre-defined as *keepThisSecret! [It's a good idea to change this password as soon as possible, especially if your dEWDrop VM has public access. However, for now, leave it as the default.]*

Enter this password, click the *Login* button and you should be successfully up and running with EWD.js and the *ewdMonitor* application.

You're now ready to build your own EWD.js applications on your dEWDrop VM. Turn to Appendix 4.

Appendix 2

Creating a GT.M-based EWD.js/Ubuntu 14.04 System From Scratch

Background

Ubuntu 14.04 introduces the new apt-get installer for GT.M. The new version of EWD.js includes an installation script that make use of this, allowing the creation of a complete, fully-working system. The starting point is a Ubuntu 14.04 desktop or server system, either a physical machine, virtual machine or EC2 instance. The EWD.js installer will install and configure:

- NVM (The Node.js Version Manager, allowing quick and simple updating of Node.js in the future)
- Node.js
- GT.M
- NodeM
- EWD.js

Load and Run Installer

Start up a terminal session on your Ubuntu 14.04 machine and do the following:

```
cd ~
sudo apt-get -y install git
git clone https://github.com/robtweed/ewd-installers
source ewd-installers/gtm/install.sh
```

During the installation process, you'll be asked to confirm the path in which EWD.js has been installed. Just accept the default that it suggests by pressing the Enter key, ie:

```
Install EWD.js to directory path (/home/ubuntu/ewdjs):
```

The essential sub-components of EWD.js will be installed relative to this path.

You'll then be asked if you want to install the extra, optional sub-components for EWD.js:

```
EWD.js has been installed and configured successfully

Do you want to install the additional resources from the /extras directory?
If you're new to EWD.js or want to create a test environment, enter Y
If you're an experienced user or this is a production environment, enter N
Enter Y/N:
```

If you're new to EWD.js, typing Y (followed by the Enter key) is recommended.

EWD.js is now ready for use!

Start Up ewd.js

```
cd ~/ewdjs
node ewdStart-gtm gtm-config
```

Run the *ewdMonitor* application

In your browser, enter the URL (changing the IP address appropriately for that assigned to your Ubuntu machine):

http://192.168.1.101:8080/ewd/ewdMonitor/index.html

If you want to use SSL, modify the defaults object in the *ewdStart-gtm.js* file as shown below:

```
var defaults = {
  port: 8080,
  poolsize: 2,
  tracelevel: 3,
  password: 'keepThisSecret!',
  ssl: true,
  database: 'gtm'
};
```

If you then get a warning about the SSL certificates, just tell the browser it's OK: it's because *ewd.js* is using self-signed certificates.

The *ewdMonitor* application should now spring into life and ask you for a password. This is defined in the *ewdStart-gtm.js* startup file and is pre-defined as *keepThisSecret! [It's a good idea to change this password as soon as possible, especially if your dEWDrop VM has public access. However, for now, leave it as the default.]*

Enter this password, click the *Login* button and you should be successfully up and running with EWD.js and the *ewdMonitor* application.

You're now ready to build your own EWD.js applications on your dEWDrop VM. Turn to Appendix 4.

Appendix 3

Installing EWD.js on a dEWDrop v5 Server

Background

The pre-built, GT.M-based dEWDrop v5 Virtual Machine (VM) provides a ready-to-run, VistA-based environment that included the older "classic" version of EWD.

The following instructions should allow you to quickly create a new, updated fully-working EWD.js environment on a dEWDrop v5 VM.

Installing a dEWDrop VM

If you've already got a dEWDrop VM up and running, skip to the next section: *Updating a dEWDrop VM*.

However, if you haven't already installed a dEWDrop VM, follow these steps. Although you can use a variety of Virtual Machine hosting packages, I'll assume here that you'll be using the free VMWare Player:

Step 1:

Download a copy of the latest dEWDrop VM (version 5 at the time of writing) from

http://www.fourthwatchsoftware.com/dEWDrop/dEWDrop.7z

This file is about 1.6Gb in size.

Step 2:

Create a directory for the Virtual Machine you're going to unpack from the zip file, eg:

~/Virtual_Machines/dEWDrop5

or on Windows machines, something like:

c:\Virtual_Machines\dEWDrop5

Step 3:

Move the downloaded zip file into this directory

Step 4:

Exand the zip file. You'll need to use a 7-zip expander.

If your host machine is running Ubuntu Linux, you can install one using:

sudo apt-get install p7zip

If you're using Windows or Mac OS X, Stuffit Expander will do the job, or, for Windows users, check out:

http://www.7-zip.org/

Note: if you're using Windows as your host machine, you'll need to place your Virtual_Machines directory on a drive that is formatted as NTFS, because the main file (dEWDrop.vmdk) expands to about 8.7Gb.

If your host machine is running Ubuntu Linux, expand the dEWDrop VM files as follows:

```
cd ~/Virtual_Machines/dEWDrop5
7za e dEWDrop.7z
```

Step 5:

You'll need to have VMWare Player installed to proceed to the next steps. If you don't already have VMWare Player installed, you'll find the details on how to download and install it at:

http://www.vmware.com/products/player/

Step 6:

Start up VMWare Player and select the menu option to open a new File.

Navigate to your *Virtual_Machines/dEWDrop5* directory and select the file:

dEWDrop.vmx

(Alternatively use your host machine's file manager utility and double click on *dEWDrop.vmx*).

Note: VMWare Player may tell you that it can't find the Virtual Disk file. If so, navigate to and select the file dEWDrop.vmdk.

The first time you start the dEWDrop VM, VMWare will ask you whether you moved it or copied it. Select *"I Copied It"*

If it asks you whether you want to download VMWare Tools, you can tell it to not bother.

Step 7:

The dEWDrop VM should now boot up inside the VMWare Player console. When it asks you for *Username*, enter:

vista

and when it asks for *Password*, enter:

ewd

You should now be logged in to the dEWDrop VM, which is a Ubuntu 12.04 system.

Step 8:

Find out the IP Address that was assigned to the VM by your host's machine by typing the command:

ifconfig

Look for the section titled eth0 and you should see a line commencing: i*net addr*. This should tell you the IP address, eg 192.168.1.68

Step 9:

You should leave the console alone at this stage and fire up a terminal or puTTY session and make an SSH connection into the dEWDrop server. Use the IP address you discovered in *Step 8*. eg from a Linux/Unix terminal, connect using:

ssh vista@192.168.1.68

Then enter the password: *ewd*

You should now have a working dEWDrop VM up and running.

Updating a dEWDrop VM

Follow the steps below from within a terminal session on the dEWDrop VM:

```
cd ~
sudo apt-get -y install git
git clone https://github.com/robtweed/ewd-installers
source ewd-installers/dEWDrop/upgrade.sh
```

During the installation process, you'll be asked to confirm the path in which EWD.js has been installed. Just accept the default that it suggests by pressing the Enter key, ie:

```
Install EWD.js to directory path (/home/ubuntu/ewdjs):
```

The essential sub-components of EWD.js will be installed relative to this path.

You'll then be asked if you want to install the extra, optional sub-components for EWD.js:

```
EWD.js has been installed and configured successfully

Do you want to install the additional resources from the /extras directory?
If you're new to EWD.js or want to create a test environment, enter Y
If you're an experienced user or this is a production environment, enter N
Enter Y/N:
```

If you're new to EWD.js, typing Y (followed by the Enter key) is recommended.

The dEWDrop server is now completely updated and ready for use with EWD.js.

The new EWD.js working environment has been created in the directory ~/ewdjs and all EWD.js-related work should take place within and below this directory. Access between EWD.js and the existing GT.M database (complete with a fully-working VistA system) has been automatically set up and configured for you by the install scripts.

Updating Your Existing EWD.js Applications

If you had already created any of your own EWD.js applications using the older version of EWD.js (ie the version that used the *ewdgateway2* module), you should copy these applications into the new application directory: *~/ewdjs/www/ewd* and copy their back-end module files into *~/ewdjs/node_modules*.

You will also need to modify any existing index.html files: references to */ewdLite/EWD.js* and */ewdLite/ewdBootstrap3.js* must be changed to */ewdjs/EWD.js* and */ewdjs/ewdBootstrap3.js* respectively.

Start Up ewd.js

```
cd ~/ewdjs
node ewdStart-gtm dewdrop-config
```

Run the *ewdMonitor* application

In your browser, enter the URL (changing the IP address appropriately for that assigned to your dEWDrop VM):

http://192.168.1.101:8080/ewd/ewdMonitor/index.html

If you want to use SSL, modify the defaults object in the *ewdStart-gtm.js* file as shown below:

```
var defaults = {
  port: 8080,
  poolsize: 2,
  tracelevel: 3,
  password: 'keepThisSecret!',
  ssl: true,
  database: 'gtm'
};
```

If you then get a warning about the SSL certificates, just tell the browser it's OK: it's because *ewd.js* is using self-signed certificates.

The *ewdMonitor* application should now spring into life and ask you for a password. This is defined in the *ewdStart-gtm.js* startup file and is pre-defined as *keepThisSecret! [It's a good idea to change this password as soon as possible, especially if your dEWDrop VM has public access. However, for now, leave it as the default.]*

Enter this password, click the *Login* button and you should be successfully up and running with EWD.js and the *ewdMonitor* application.

You're now ready to build your own EWD.js applications on your dEWDrop VM. Turn to Appendix 4.

Appendix 4

A Beginner's Guide to EWD.js

A Simple Hello World Application

This Appendix provides a guide for the complete EWD.js novice, explaining how to build a very simple application that demonstrates the basic principles behind the framework. We'll build a simple Hello World application using nothing more that a straightforward HTML page and a couple of buttons: we won't use any fancy JavaScript frameworks. We'll create two basic functions in our Hello World application:

- First we'll create a button that sends a WebSocket message from the HTML page in the browser to a back-end module where we'll save the contents of the message into a Mumps persistent array;
- Secondly we'll create a button that sends a WebSocket message to the back-end where it instructs it to retrieve the first message's payload and return it to the browser.

This guide assumes you've already successfully installed and configured EWD.js on your system. If you haven't done this yet, follow the instructions in the main chapters of this document, or Appendices 1 to 3 for quick, automated installations.

So let's get started.

Your EWD.js Home Directory

Throughout this Appendix, I'll be referring to your EWD.js Home Directory. This is explained in the main body of this documentation, but in summary, your EWD.js Home Directory is the directory you were in when you installed *ewd.js*, ie when you ran:

```
npm install ewdjs
```

So it's the directory that contains the sub-directory named *node_modules*, inside of which is the *ewd.js* module.

From now on, we'll denote the EWD.js Home Directory by *~/ewdjs*

On a Windows machine this path may equate to:

c:\ewdjs

Start the *ewd.js* Module

Use the appropriate start file. For example:

On a GT.M system built using the installer described in Appendix 3:

```
cd ~/ewdjs
node ewdStart-gtm gtm-config
```

or on a dEWDrop VM:

```
cd ~/ewdjs
node ewdStart-gtm dewdrop-config
```

The HTML Page

Create a new sub-directory named *helloworld* under your *~/ewdjs/www/ewd* directory.

Now create a file named *index.html* within the *~/ewdjs/www/ewd/helloworld* directory containing the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
  <html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
   <head>
     <title>EWD.js Hello World</title>
     <!--[if (IE 6)|(IE 7)|(IE 8)]>
     <script type="text/javascript"
src="//ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script>
     <![endif]-->
   </head>
 <body>
   <h2>EWD.js Hello World Application</h2>
   <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
   <script src="/socket.io/socket.io.js"></script>
   <script src="/ewdjs/EWD.js"></script>
 </body>
</html>
```

Note that as of Build 54, EWD.js requires the use of JQuery: this has simplified support for older browsers such as older versions of Internet Explorer.

Fire up a browser, ideally Chrome. If you're using Chrome, open its Developer Tools console by doing the following:

- Click the top right menu icon (represented by three short horizontal lines)
- Select Tools from the drop-down menu
- Select JavaScript Console from the sub-menu
- Initially the Console panel is docked to the bottom of the Chrome window. Click the 2nd-right-most icon (it looks a little like a computer terminal with a shadow)

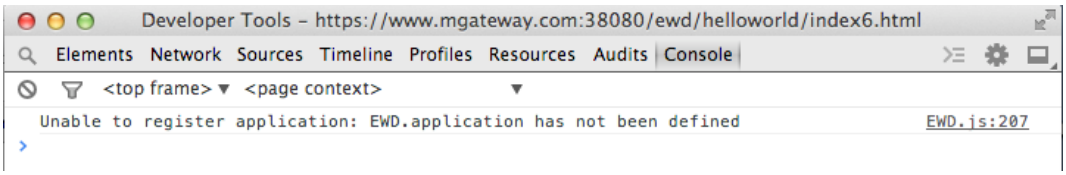
Now run your HTML page by using the URL (change the IP address appropriately for your system):

```
http://192.168.1.101:8080/ewd/helloworld/index1.html
```

The following should appear in your browser:

EWD.js Hello World Application

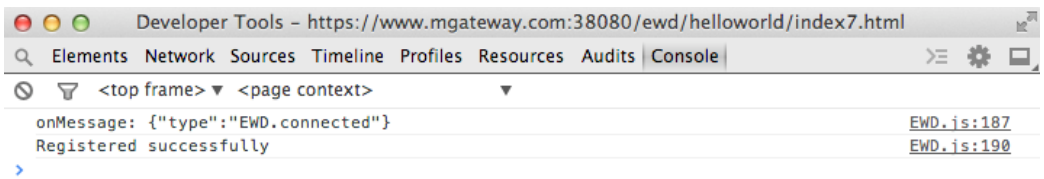
If you look in Chrome's JavaScript console, you should see the following:



That error message has been returned by the EWD.js framework. We can resolve it by editing the *index.html* page as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>EWD.js Hello World</title>
    <!--[if (IE 6)|(IE 7)|(IE 8)]>
    <script type="text/javascript"
src="//ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script>
    <![endif]-->
  </head>
  <body>
    <h2>EWD.js Hello World Application</h2>
    <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/ewdjs/EWD.js"></script>
    <script>
      EWD.application = {
        name: 'helloworld'
      };
      EWD.sockets.log = true;
    </script>
 </body>
</html>
```

This tells EWD.js that you're running an application named *helloworld*. The last line tells EWD.js to log its activity into the JavaScript console (otherwise EWD.js doesn't report anything apart from runtime errors). Refresh the browser and this time you should see a message similar to this in the JavaScript Console:



Our Hello World application is now working properly with EWD.js. Now we're ready to build out our application.

The app.js File

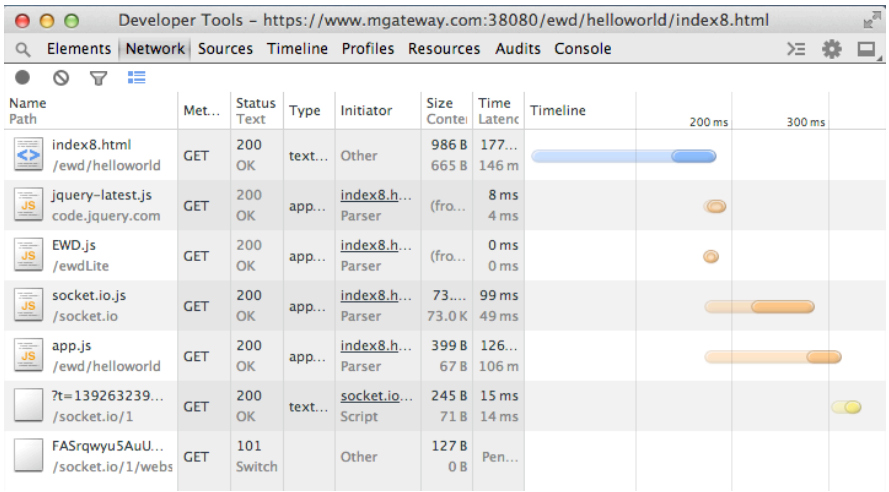
Before we proceed any further, let's adopt best practice and, instead of creating inline JavaScript within your *index.html* file, we'll move it all into a separate JavaScript file. By convention, we name this file *app.js*. So, create a new file named *app.js* in the same directory as your *index.html* file and move that Javascript into it. *app.js* should look like this:

```
EWD.application = {
  name: 'helloworld'
};
EWD.sockets.log = true;
```

Edit your index.html file so that it loads *app.js*:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>EWD.js Hello World</title>
    <!--[if (IE 6)|(IE 7)|(IE 8)]>
    <script type="text/javascript"
src="//ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script>
    <![endif]-->
  </head>
  <body>
    <h2>EWD.js Hello World Application</h2>
    <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/ewdjs/EWD.js"></script>
    <script src="app.js"></script>
 </body>
</html>
```

Try re-loading the URL in the browser: the application should run identically. We can prove that it's loading the app.js file correctly by clicking the Network tab at the top of the JavaScript console. You should see something like this in the console:



On the 5th line you can see that app.js was loaded.

Sending our First WebSocket Message

Add a button to the body of the page, assign an *onClick()* event handler to it:

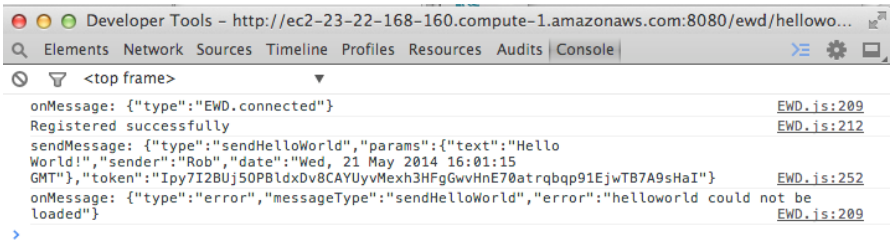
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>EWD.js Hello World</title>
    <!--[if (IE 6)|(IE 7)|(IE 8)]>
    <script type="text/javascript"
src="//ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script>
    <![endif]-->
  </head>
  <body>
    <h2>EWD.js Hello World Application</h2>
    <input type="button" value="Send Message" onClick="sendMessage()" />
    <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/ewdjs/EWD.js"></script>
    <script src="app.js"></script>
 </body>
</html>
```

Now define the handler function named *sendMessage()* in the *app.js* file. This handler function is going to send an EWD.js WebSocket message to the back-end. We've decided to define the message's type as *sendHelloWorld*. The message will contain a number of name/value pairs in its payload. It's up to us to decide the payload contents and structure. The payload goes into the property named *params*:

```
EWD.application = {
  name: 'helloworld'
};
var sendMessage = function() {
  EWD.sockets.sendMessage({
    type: "sendHelloWorld",
    params: {
      text: 'Hello World!',
      sender: 'Rob',
      date: new Date().toUTCString()
    }
  });
};
EWD.sockets.log = true;
```

When you reload the browser, you should now see a button that contains the text *Send Message*.

Click the button and you should see the following error message appear in the JavaScript Console.



Well, this error is actually good news! It tells us that our message was sent to the EWD.js back-end successfully. What the error is telling us is that EWD.js couldn't find a module named *helloworld.js* in your *~/ewdjs/node_modules* directory. It's actually looking for a module named *helloworld* because of that command we added to *app.js* earlier:

```
EWD.application = {
  name: 'helloworld'
};
```

So let's now create the back-end module for our *helloworld* application.

The *helloworld* Back-end Module

Create a file named *helloworld.js* in your *~/ewdjs/node_modules* directory containing the following:

```
module.exports = {
  onSocketMessage: function(ewd) {
    var wsMsg = ewd.webSocketMessage;
    ewd.log('*** Incoming Web Socket message received: ' + JSON.stringify(wsMsg, null, 2), 1);
  }
};
```

Initially all this is going to do is listen for all incoming WebSocket messages from any users of the *helloworld* application and log their contents to the *ewd.js* console. Remember: this JavaScript module runs at the back-end, not in the browser!

Reload the *index.html* page into the browser and click the button. Now look at the *ewd.js* module and you should see the following (you may need to scroll back through the messages quite a distance):

```
*** Incoming Web Socket message received: {
  "type": "sendHelloWorld",
  "params": {
    "text": "Hello World!",
    "sender": "Rob",
    "date": "Mon, 17 Feb 2014 11:09:41 GMT"
  }
}
child process 5372 returned response {"ok":5372,"type":"log","message":"*** Incoming Web Socket message received:
{\n  \"type\": \"sendHelloWorld\",\n  \"params\": {\n    \"text\": \"Hello World!\",\n    \"sender\": \"Rob\",\n
\"date\": \"Mon, 17 Feb 2014 11:09:41 GMT\"\n  }\n}"}
Child process 5372 returned to available pool
onBeforeRender completed at 328.398
child process 5372 returned response {"ok":5372,"type":"log","message":"onBeforeRender completed at 328.398"}
Child process 5372 returned to available pool
child process 5372 returned response {"ok":5372,"response":""}
Child process 5372 returned to available pool
running handler
wsResponseHandler has been fired!
```

There at the top is the message we sent!

Adding a Type-specific Message Handler

OK so now let's add a specific handler for messages of type *sendHelloWorld*. To do this, replace the onSocketMessage() function with an object named *onMessage*, into which we define specific message handlers. So, in order to handle our sendHelloWorld message, replace the contents of the *helloworld.js* file as follows:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      return {received: true};
    }
  }
};
```

Save this edited version of the file.

Debugging Errors in your Module

If you look carefully at the *ewd.js* console when you saved the edited version of the *helloworld.js* module, you should see the following appearing in the log (in fact it may appear more than once):

```
child process 23829 returned response {"ok":23829,"type":"log","message":"23829:
helloworld(/home/vista/www/node/node_modules/helloworld.js) reloaded successfully"}
Child process 23829 returned to available pool
```

ewd.js adds an event handler automatically whenever a back-end module is loaded into one of its child processes: whenever a change is made to the module, the child process will attempt to reload it. Note that if you have any syntax errors in your module, it will fail to be reloaded but unfortunately the child process is often unable to give you any meaningful diagnosis of why it failed to load.

Here's therefore a tip: if your module won't load, fire up a new terminal session and start the Node.js REPL (make sure your in your EWD.js Home Directory first):

```
cd ~/ewdjs // replace with your EWD.js Home Directory if different
node
>
```

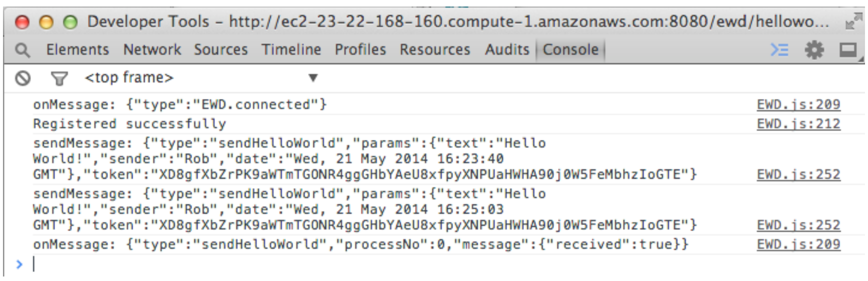
Now try to load the module in the REPL using the *require()* function. If there are any syntax errors, you'll be able to see what they are, eg:

```
vista@dEWDrop:~$ cd ~/ewdjs
vista@dEWDrop:~/ewdjs$ node
> var x = require('helloworld')

/home/vista/www/node/node_modules/helloworld.js:10
      ewd.log('*** Incoming Web Socket message received:  + JSON.stringify(par
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
...
```

You can also use Node Inspector to debug your application. For details, see the Section titled *Debugging EWD.js Applications using Node Inspector* in the Chapter *Creating an EWD.js Application* earlier in this document.

The Type-specific Message Handler in Action

Reload the *index.html* page in the browser and click the button again. This time you should see the following message logged in the JavaScript console:



We no longer get any errors. That last message has appeared as a result of the *return* we added to our back-end message handler:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      return {received: true};
    }
  }
};
```

When your back-end module *returns* with a JSON object as its *returnValue*, a WebSocket message of the same type as the incoming message is returned to the browser. The *returnValue* object is in the *message* property of the WebSocket message received by the browser, ie:

```
{"type":"sendHelloWorld","message":{"received":true}}
```

Note that although, in this example, we returned a very simple object, the JSON object we return from the back-end can be as complex and as deeply nested as we like.

So, we've been able to handle the incoming message correctly and return a response to the browser. Now let's look at how we can store that incoming message into the Mumps database.

Storing our record into the Mumps database

In this exercise, we're not going to do anything too complex. All we'll do is save the incoming message object directly into a Mumps persistent array by using the *_setDocument()* method. We're going to use an array name of *%AMessage* because this will appear near the top of the list of persistent object names when we go looking for it with the ewdMonitor application.

So, edit the *helloworld.js* module file as follows:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      savedMsg._setDocument(params);
      return {savedInto: 'AMessage'};
    }
  }
};
```

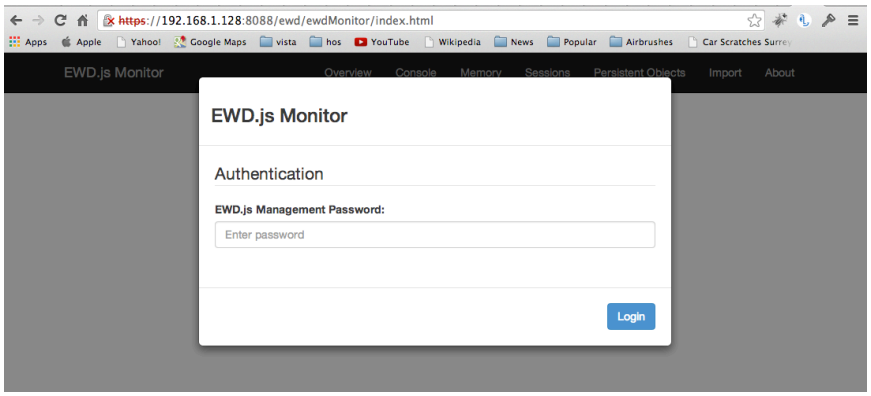
Reload the page and click the button. You should see the modified response message appearing in the JavaScript console. Now let's find out what happened to the message at the back end. For this, we'll use the *ewdMonitor* application.

Using the *ewdMonitor* Application to inspect the Mumps Database

Open up a new browser tab and start the ewdMonitor application by using the URL (modify the IP address as appropriate):

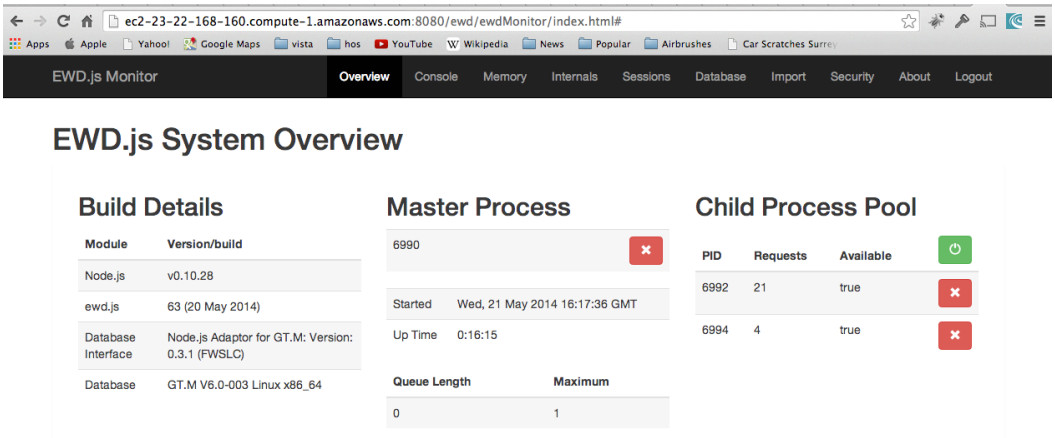
```
http://192.168.1.101:8080/ewd/ewdMonitor/index.html
```

The *ewdMonitor* application should start up and ask you for a password.

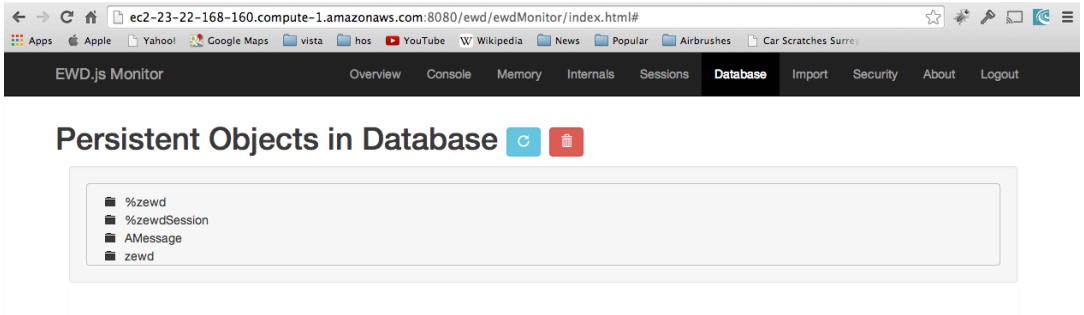


Unless you changed the password in the startup file for *ewd.js* (which you should do once you start using EWD.js in earnest), you should enter the default password which is *keepThisSecret!*

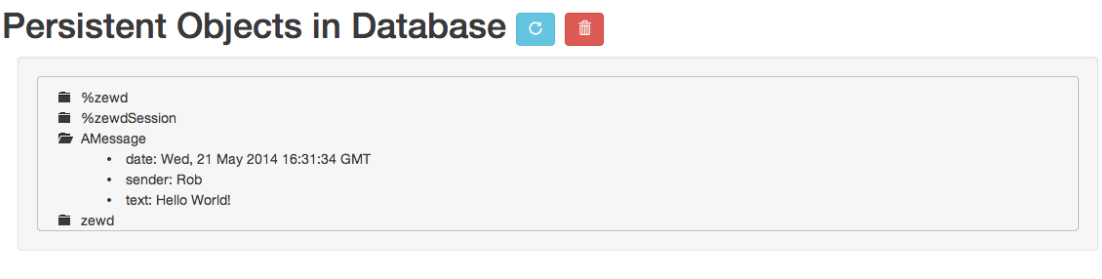
The *ewdMonitor* application should now burst into life:



Click the tab named *Database.* You should see a list of Mumps Globals appear - the list you see will depend on the type of system you're running and you may need to scroll the list:



In the list you'll find the persistent object we created: *AMessage*. Click on the folder symbol next to the name to expand it, and repeat to expand all its levels. You should see that it exactly mirrors the structure and content of the WebSocket message you sent from your browser:



The difference between this and the WebSocket message is that this version is stored permanently on disc (at least until we decide to delete or change it).

Notice that we didn't have to pre-declare anything to save this data to the Mumps database, and we didn't have to define a schema: we simply decided on a name for our persistent object and saved the JSON document straight into it using the *_setDocument()* method:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      savedMsg._setDocument(params);
      return {savedInto: 'AMessage'};
    }
  }
};
```

Storing JSON documents into a Mumps database is trivially simple!

Try clicking the button in the browser again, press the *Refresh* button in the *ewdMonitor* application's Persistent Objects panel and examine the *AMessage* object's contents again: you should see that the date has been overwritten with the new value of the latest message.

Handling the Response Message in the Browser

There's just one last step we need to do in order to create a complete round-trip for our example WebSocket message, and that is to properly handle the response that was returned to the browser. To do that, we need to add a WebSocket message handler to the *app.js* file, and a placeholder for displaying a message in the *index.html* file.

Add a *div* tag to your *index.html* file as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>EWD.js Hello World</title>
    <!--[if (IE 6)|(IE 7)|(IE 8)]>
    <script type="text/javascript"
src="//ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script>
    <![endif]-->
  </head>
  <body>
    <h2>EWD.js Hello World Application</h2>
    <input type="button" value="Send Message" onClick="sendMessage()" />
    <div id="response"></div>
    <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/ewdjs/EWD.js"></script>
    <script src="app.js"></script>
 </body>
</html>
```

Then add the message handler to *app.js*:

```
EWD.application = {
  name: 'helloworld',

  onMessage: {
    sendHelloWorld: function(messageObj) {
      var text = 'Your message was successfully saved into ' + messageObj.message.savedInto;
      document.getElementById('response').innerHTML = text;
      setTimeout(function() {
        document.getElementById('response').innerHTML = '';
      },2000);
    }
  }
};

EWD.sockets.log = true;

var sendMessage = function() {
  EWD.sockets.sendMessage({
    type: "sendHelloWorld",
    params: {
      text: 'Hello World!',
      sender: 'Rob',
      date: new Date().toUTCString()
    }
  });
};
```

The *EWD.application.onMessage* object is used to define handlers for incoming WebSocket messages from the EWD.js back-end. Each handler is a function whose name matches the type property of the incoming message: in this case '*sendHelloWorld*'. Each handler function has a single argument: the incoming message object. Usually your message handlers modify the UI in some way in response to the incoming message's JSON payload. In this example we're just showing a confirmation message which disappears again after 2 seconds.

Of course your handler can be as complex as you wish. Take a look at the behaviour of the *ewdMonitor* application to get an idea of what's possible: it's an EWD.js application responding to all sorts of incoming WebSocket messages.

A Second Button to Retrieve the Saved Message

Let's now add a second button to our HTML page, and get it to retrieve the saved message whenever it's clicked. This is really pretty simple. First edit the *index.html* file as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>EWD.js Hello World</title>
    <!--[if (IE 6)|(IE 7)|(IE 8)]>
    <script type="text/javascript"
src="//ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script>
    <![endif]-->
  </head>
  <body>
    <h2>EWD.js Hello World Application</h2>
    <input type="button" value="Send Message" onClick="sendMessage()" />
    <div id="response"></div>
    <input type="button" value="Retrieve Saved Message" onClick="getMessage()" />
    <div id="response2"></div>
    <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script src="/ewdjs/EWD.js"></script>
    <script src="app.js"></script>
 </body>
</html>
```

Add the *getMessage()* click handler function to *app.js*:

```
EWD.application = {
  name: 'helloworld',

  onMessage: {
    sendHelloWorld: function(messageObj) {
      var text = 'Your message was successfully saved into ' + messageObj.message.savedInto;
      document.getElementById('response').innerHTML = text;
      setTimeout(function() {
        document.getElementById('response').innerHTML = '';
      },2000);
    }
  }
};

EWD.sockets.log = true;

var sendMessage = function() {
  EWD.sockets.sendMessage({
    type: "sendHelloWorld",
    params: {
      text: 'Hello World!',
      sender: 'Rob',
      date: new Date().toUTCString()
    }
  });
};

var getMessage = function() {
  EWD.sockets.sendMessage({
    type: "getHelloWorld"
  });
};
```

You can probably see what this will do now: clicking the second button will send a WebSocket message of type *getHelloWorld* to the back-end. Note that for this message we aren't sending any payload: we're essentially using a message to send a signal to the back-end that we wish to fetch the stored message.

Add a Back-end Message Handler for the Second Message

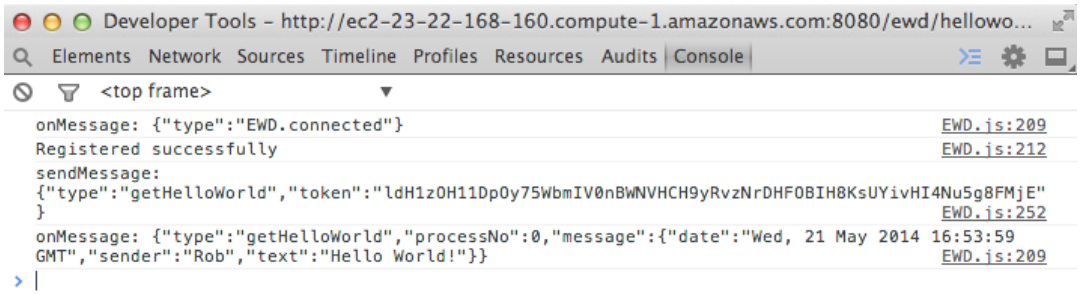
Clearly we also need to add the back-end event handler that will respond to this incoming message and retrieve the saved message. So edit the back-end module (ie the *helloworld.js)* file as follows:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      var wsMsg = params;
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      savedMsg._setDocument(params);
      return {savedInto: 'AMessage'};
    }, // don't forget this comma!
    getHelloWorld: function(params, ewd) {
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      return savedMsg._getDocument();
    }
  }
};
```

Try Running the New Version

Save this file and reload the *index.html* file into your browser. It should now appear like this:

Try clicking the *Retrieve Saved Message* button and look at the JavaScript Console. You should see this:



There's the original JSON message that we'd saved into the Mumps database. As you can see, the message received has a type of *getHelloWorld* and the contents of the saved JSON is in its *message* property.

Add a Message Handler to the Browser

So now all we have to do is add a handler to our *app.js* file to display the retrieved message details in the browser:

```
EWD.application = {
  name: 'helloworld',

  onMessage: {
    sendHelloWorld: function(messageObj) {
      var text = 'Your message was successfully saved into ' + messageObj.message.savedInto;
      document.getElementById('response').innerHTML = text;
      setTimeout(function() {
        document.getElementById('response').innerHTML = '';
      },2000);
    },
    getHelloWorld: function(messageObj) {
      var text = 'Saved message: ' + JSON.stringify(messageObj.message);;
      document.getElementById('response2').innerHTML = text;
    }
  }
};

EWD.sockets.log = true;

var sendMessage = function() {
  EWD.sockets.sendMessage({
    type: "sendHelloWorld",
    params: {
      text: 'Hello World!',
      sender: 'Rob',
      date: new Date().toUTCString()
    }
  });
};

var getMessage = function() {
  EWD.sockets.sendMessage({
    type: "getHelloWorld"
  });
};
```

Now when you click the Retrieve Saved Message you'll see the results in the browser:

In the example above, I'm just dumping out the raw JSON message. Try separating out its components and displaying them properly in the browser.

Also, try clicking the *Send Message* and then the *Retrieve Saved Message* button: each time you do this you'll get a new version of the message coming back. You'll be able to tell because the date value will be different.

Silent Handlers and Sending Multiple Messages from the Back-end

There's one last thing to try. A back-end message handler doesn't have to send a return message at all: for example, we could make the *sendHelloWorld* message handler a silent "fire and forget" handler, ie:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      var wsMsg = params;
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      savedMsg._setDocument(params);
      return;
    },
    getHelloWorld: function(params, ewd) {
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      return saveMsg._getDocument();
    }
  }
};
```

Furthermore, a back-end message handler can send as many messages as it likes back to the browser. So, for example, we could send the saved message contents in a separate message with its own type, and use the handler's returnValue to simply signal to the browser that the message has been successfully retrieved, eg:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      var wsMsg = params;
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      savedMsg._setDocument(params);
      return {savedInto: 'AMessage'};
    },
    getHelloWorld: function(params, ewd) {
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      var message = savedMsg._getDocument();
      ewd.sendWebSocketMsg({
        type: 'savedMessage',
        message: message
      });
      return {messageRetrieved: true};
    }
  }
};
```

Try re-loading the *index.html* page, click the *Send Message* button. This time you shouldn't get an acknowledgement appearing in the browser and you won't see any incoming message appearing in the Developer Tools Console.

Now click the *Retrieve Saved Message* button and take a look at the JavaScript console. You should now see the following:

As you can see, the browser has now been sent the two separate messages from our back-end handler, with types *savedMessage* and *getHelloWorld*.

You can hopefully see that this is a very powerful and flexible, yet simple-to-use framework: it's now up to you how to harness and exploit it in your applications.

You've also seen how EWD.js's messaging and back-end Mumps storage engine can be used with a simple HTML page. It can, of course, be used with any JavaScript framework: choose your favourite framework and start developing with EWD.js.

Using EWD.js with the Bootstrap Framework

If you use the JavaScript framework called Bootstrap (http://getbootstrap.com/), you'll find that EWD.js has particularly advanced support for it. The *ewdMonitor* application has been built using Bootstrap. In the next section, we'll examine how your application can be further refined by using the Bootstrap-related features of EWD.js.

The Bootstrap 3 Template Files

In your ~/ewdjs/www/ewd directory is an application named *bootstrap3*. This isn't actually a working application. Instead it's a set of example pages and fragment files that you can use as a starting point for any Bootstrap 3 EWD.js applications that you build.

Helloworld, Bootstrap-style

Create a new *index.html* page for your *helloworld* application - let's name it *indexbs.html*. We'll do this by copying the *index.html* file from the *bootstrap3* application directory. It should look like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
  <head>

    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta http-equiv="Cache-Control" content="no-cache, no-store, must-revalidate">
    <meta http-equiv="Pragma" content="no-cache">
    <meta http-equiv="Expires" content="0">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="apple-touch-fullscreen" content="yes">
    <meta name="viewport" content="user-scalable=no, width=device-width, initial-scale=1.0, maximum-scale=1.0">
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="author" content="Rob Tweed">

    <link href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.css" rel="stylesheet" />
    <link href="//cdnjs.cloudflare.com/ajax/libs/select2/3.4.5/select2.css" rel="stylesheet" />
    <link href="//cdnjs.cloudflare.com/ajax/libs/toastr.js/2.0.1/css/toastr.min.css" rel="stylesheet" />
    <link href="//code.jquery.com/ui/1.10.4/themes/smoothness/jquery-ui.css" rel="stylesheet" />
    <link href="//www.fuelcdn.com/fuelux/2.4.1/css/fuelux.css" rel="stylesheet" />
    <link href="//www.fuelcdn.com/fuelux/2.4.1/css/fuelux-responsive.css" rel="stylesheet" />

    <!-- Fav and touch icons -->
    <link rel="shortcut icon" href="//ematic-solutions.com/cdn/bootstrap/2.3.1/ico/favicon.png" />
    <link rel="apple-touch-icon-precomposed" sizes="144x144"
href="//ematic-solutions.com/cdn/bootstrap/2.3.1/ico/apple-touch-icon-144-precomposed.png" />
    <link rel="apple-touch-icon-precomposed" sizes="114x114"
href="//ematic-solutions.com/cdn/bootstrap/2.3.1/ico/apple-touch-icon-114-precomposed.png" />
    <link rel="apple-touch-icon-precomposed" sizes="72x72"
href="//ematic-solutions.com/cdn/bootstrap/2.3.1/ico/apple-touch-icon-72-precomposed.png" />
    <link rel="apple-touch-icon-precomposed"
href="//ematic-solutions.com/cdn/bootstrap/2.3.1/ico/apple-touch-icon-57-precomposed.png" />

    <script src="/socket.io/socket.io.js"></script>
    <!--[if (IE 6)|(IE 7)|(IE 8)]><script type="text/javascript"
src="//ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js"></script><![endif]-->

    <title id="ewd-title"></title>

    <style type="text/css">
      body {
        padding-top: 60px;
        padding-bottom: 40px;
      }
      .sidebar-nav {
        padding: 9px 0;
      }
      .focusedInput {
        border-color: rgba(82,168,236,.8);
        outline: 0;
        outline: thin dotted \9;
        -moz-box-shadow: 0 0 8px rgba(82,168,236,.6);
        box-shadow: 0 0 8px rgba(82,168,236,.6) !important;
      }
      .graph-Container {
        box-sizing: border-box;
        width: 850px;
        height: 460px;
        padding: 20px 15px 15px 15px;
        margin: 15px auto 30px auto;
        border: 1px solid #ddd;
        background: #fff;
        background: linear-gradient(#f6f6f6 0, #fff 50px);
        background: -o-linear-gradient(#f6f6f6 0, #fff 50px);
        background: -ms-linear-gradient(#f6f6f6 0, #fff 50px);
        background: -moz-linear-gradient(#f6f6f6 0, #fff 50px);
        background: -webkit-linear-gradient(#f6f6f6 0, #fff 50px);
        box-shadow: 0 3px 10px rgba(0,0,0,0.15);
        -o-box-shadow: 0 3px 10px rgba(0,0,0,0.1);
        -ms-box-shadow: 0 3px 10px rgba(0,0,0,0.1);
        -moz-box-shadow: 0 3px 10px rgba(0,0,0,0.1);
        -webkit-box-shadow: 0 3px 10px rgba(0,0,0,0.1);
      }
      .graph-Placeholder {
        width: 820px;
        height: 420px;
        font-size: 14px;
        line-height: 1.2em;
      }
      .ui-widget-content .ui-state-default {
        background: blue;
      }
      .fuelux .tree {
        overflow-x: scroll;
      }

    </style>

    <!-- HTML5 shim and Respond.js IE8 support of HTML5 elements and media queries -->
    <!--[if lt IE 9]>
      <script src="//oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>
      <script src="//oss.maxcdn.com/libs/respond.js/1.4.2/respond.min.js"></script>
    <![endif]-->

  </head>
```

```
   <body>

   <!-- Modal Login Form -->

   <div id="loginPanel" class="modal fade"></div>

   <!-- Main Page Definition -->

   <!-- NavBar header -->
   <nav class="navbar navbar-inverse navbar-fixed-top">
     <div class="container">
       <div class="navbar-header">
         <div class="navbar-brand visible-xs" id="ewd-navbar-title-phone"></div>
         <div class="navbar-brand hidden-xs" id="ewd-navbar-title-other"></div>
         <button class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
           <span class="icon-bar"></span>
           <span class="icon-bar"></span>
           <span class="icon-bar"></span>
         </button>
       </div>
       <div class="navbar-collapse collapse navbar-ex1-collapse">
         <ul class="nav navbar-nav pull-right" id="navList"></ul>
       </div>
     </div>
   </nav>

   <!-- Main body -->
   <div id="content">

     <!-- main CONTAINER -->
     <div id="main_Container" class="container in" style="display: none"></div>

     <!-- about CONTAINER -->
     <div id="about_Container" class="container collapse"></div>

   </div>

   <!-- Modal info panel -->
   <div id="infoPanel" class="modal fade"></div>

   <div id="confirmPanel" class="modal fade"></div>

   <div id="patientSelectionPanel" class="modal fade"></div>

   <!-- Placed at the end of the document so the pages load faster -->
   <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
   <script src="//www.fuelcdn.com/fuelux/2.4.1/loader.js" type="text/javascript"></script>
   <script type="text/javascript" src="//code.jquery.com/ui/1.10.4/jquery-ui.js"></script>
   <script type="text/javascript" src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></script>
   <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/select2/3.4.5/select2.js"></script>
   <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/toastr.js/2.0.1/js/toastr.min.js"></script>
   <!--[if lte IE 8]><script language="javascript" type="text/javascript"
src="//cdnjs.cloudflare.com/ajax/libs/flot/0.8.2/excanvas.min.js"></script><![endif]-->
   <script language="javascript" type="text/javascript"
src="//cdnjs.cloudflare.com/ajax/libs/flot/0.8.2/jquery.flot.min.js"></script>
   <script src="/ewdjs/EWD.js"></script>
   <script type="text/javascript" src="/ewdjs/ewdBootstrap3.js"></script>

   <script type="text/javascript" src="app.js"></script>

  </body>
</html>
```

You can see immediately that there's a lot of stuff that gets loaded in this file. Not all of it is needed for all applications, but for the purpose of this tutorial, it will do no harm leaving it all in place. Notice that all the JavaScript and CSS files are fetched from Content Delivery Network (CDN) repositories. As you become familiar with the various component frameworks and utilities that are used with Bootstrap 3, you can install local copies and modify the *<script>* and *<link>* tags appropriately.

Next, find the file named *main.html* in the *bootstrap3* application directory and copy it into your *helloworld* directory.

Most of the work you do in EWD.js / Bootstrap 3 applications takes place in the *app.js* file and in fragment files that get fetched and injected into the *index.html* file. You really shouldn't need to make many, if any, changes to *index.html*. For our demo *helloworld* application, we're just going to use a single fragment file that we're going to build around the *main.html* fragment file.

Next, copy the *app.js* file from the bootstrap3 application directory, replacing the previous version of *app.js* (perhaps make a copy of the original first). *app.js* should look like the following:

```
EWD.sockets.log = true;   // *** set this to false after testing / development

EWD.application = {
  name: 'bootstrap3', // **** change to your application name
  timeout: 3600,
  login: true,
  labels: {
    'ewd-title': 'Demo',                                    // *** Change as needed
    'ewd-navbar-title-phone': 'Demo App',                   // *** Change as needed
    'ewd-navbar-title-other': 'Demonstration Application'    // *** Change as needed
  },
  navFragments: {
    main: {
      cache: true
    },
    about: {
      cache: true
    }
  },

  onStartup: function() {

    // Enable tooltips
    //$('[data-toggle="tooltip"]').tooltip()

    //$('#InfoPanelCloseBtn').click(function(e) {
    //  $('#InfoPanel').modal('hide');
    //});

    EWD.getFragment('login.html', 'loginPanel');
    EWD.getFragment('navlist.html', 'navList');
    EWD.getFragment('infoPanel.html', 'infoPanel');
    EWD.getFragment('confirm.html', 'confirmPanel');
    EWD.getFragment('main.html', 'main_Container');

  },

  onPageSwap: {
    // add handlers that fire after pages are swapped via top nav menu
    /* eg:
    about: function() {
      console.log('"about" menu was selected');
    }
    */
  },

  onFragment: {
    // add handlers that fire after fragment contents are loaded into browser

    'navlist.html': function(messageObj) {
      EWD.bootstrap3.nav.enable();
    },

    'login.html': function(messageObj) {
      $('#loginBtn').show();
      $('#loginPanel').on('show.bs.modal', function() {
        setTimeout(function() {
          document.getElementById('username').focus();
        },1000);
      });

      $('#loginPanelBody').keydown(function(event){
        if (event.keyCode === 13) {
          document.getElementById('loginBtn').click();
        }
      });
    }

  },

  onMessage: {

    // add handlers that fire after JSON WebSocket messages are received from back-end

    loggedIn: function(messageObj) {
      toastr.options.target = 'body';
      $('#main_Container').show();
      $('#mainPageTitle').text('Welcome to VistA, ' + messageObj.message.name);
    }
  }

};
```

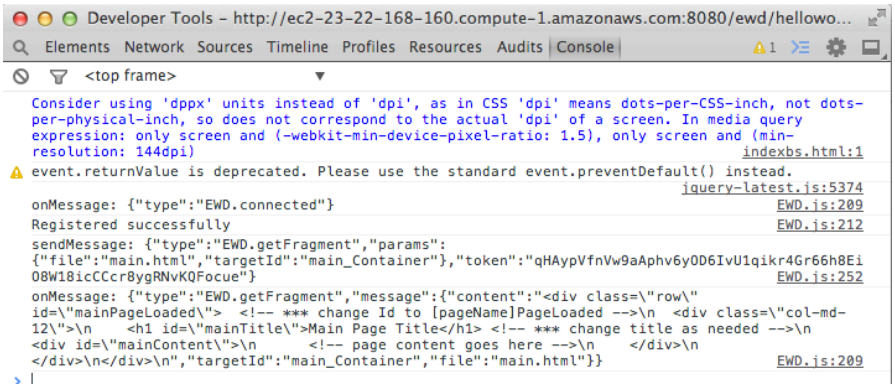
Edit app.js as follows (denoted by the lines shown in bold):

```
EWD.sockets.log = true;   // *** set this to false after testing / development

EWD.application = {
  name: 'helloworld',
  timeout: 3600,
  login: false,
  labels: {
    'ewd-title': 'Hello World',
    'ewd-navbar-title-phone': 'Hello World App',
    'ewd-navbar-title-other': 'Hello World Application'
  },
  navFragments: {
    main: {
      cache: true
    },
    about: {
      cache: true
    }
  },

  onStartup: function() {

    // remove everything in here apart from this line:

    EWD.getFragment('main.html', 'main_Container');

  },

  onPageSwap: {
    // add handlers that fire after pages are swapped via top nav menu
    /* eg:
    about: function() {
      console.log('"about" menu was selected');
    }
    */
  },

  onFragment: {
    // add handlers that fire after fragment contents are loaded into browser

    // remove everything from here

  },

  onMessage: {

    // add handlers that fire after JSON WebSocket messages are received from back-end

    // remove everything from here
  }

};
```

Don't worry about the back-end module just yet. Try loading *indexbs.html* into your browser. You should see the following:

Main Page Title

Take a look in the JavaScript console. You should see the following:



Notice the final message that shows the delivery of the contents of the main.html fragment file. Also notice that the application has registered itself automatically.

Before we begin adding to our application's functionality, take a look at the app.js file again. Notice the following sections:

- ***onStartup***: This function is triggered automatically when the index.html and all the associated JavaScript and CSS files have loaded and initialised, and not until the socket.io library has made a WebSocket connection to the back-end and EWD.js has registered the application. This function is where you should define your initial handlers for any buttons etc that are defined in the *index.html* file. In our application, it's where we fetch the *main.html* fragment by using the *EWD.getFragment()* function.

- ***onPageSwap***: This object contains any handler functions that you want to fire when navigation tabs in the Bootstrap nav bar are clicked. Initially we're not going to use a nav bar, so we can ignore this for now.

- ***onFragment***: This object contains any handler functions that you want to fire when a fragment file is loaded into the browser. This is the place to define any handlers for buttons, etc that are defined in the fragment file. This is a really nice and important feature in EWD.js: it allows you to add dynamic, run-time features to otherwise static fragment files.

- ***onMessage***: This object contains any handler functions for incoming WebSocket messages that have been sent from your application's EWD.js back-end module.

You can see that when we use Bootstrap 3, we have a lot more and slicker mechanisms to use for handling events than we saw in our initial, basic Hello World application.

Sending a Message from the Bootstrap 3 Page

Just as in our original Hello World application, let's first add to the page a button that will send a message to the back-end. We could do that by modifying the *main.html* file, but let's be a bit more adventurous and define the button in its own fragment file that will be fetched when *main.html* is loaded. This is a bit over-the-top, but will nicely demonstrate the use of the *onFragment* object in *app.js*.

So, first, create a new file named *button1.html* in the *helloworld* directory, containing the following:

```
<button class="btn btn-info" type="button" id="sendMsgBtn" data-toggle="tooltip" data-
placement="top" title="" data-original-title="Send Message">
  <span class="glyphicon glyphicon-send"></span>
</button>
```

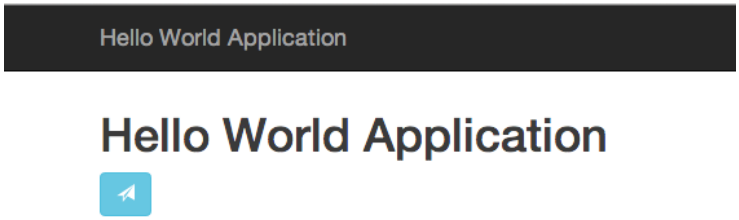
Now edit *app.js* as shown in bold:

```
EWD.sockets.log = true;   // *** set this to false after testing / development

EWD.application = {
  name: 'helloworld',
  timeout: 3600,
  login: false,
  labels: {
    'ewd-title': 'Hello World',
    'ewd-navbar-title-phone': 'Hello World App',
    'ewd-navbar-title-other': 'Hello World Application'
  },
  navFragments: {
    main: {
      cache: true
    },
    about: {
      cache: true
    }
  },

  onStartup: function() {

    EWD.getFragment('main.html', 'main_Container');

  },

  onPageSwap: {
    // add handlers that fire after pages are swapped via top nav menu
    /* eg:
    about: function() {
      console.log('"about" menu was selected');
    }
    */
  },

  onFragment: {
    // add handlers that fire after fragment contents are loaded into browser

    'main.html': function() {
      $('#mainTitle').text('Hello World Application');
      EWD.getFragment('button1.html', 'mainContent');
    }

  },

  onMessage: {
    // add handlers that fire after JSON WebSocket messages are received from back-end

  }

};
```

This function will fire after the main.html fragment has loaded and does two things:

- changes the text of the title of the *main.html* fragment. We could just hard-edited main.html, but this demonstrates how the contents of the hard-coded fragment files can be dynamically modified;
- loads our new *button1.html* fragment into the <*div*> tag within *main.html* that has the id: *mainContent*

Try reloading the indexbs.html file into the browser and you should now see:



There's our nice Bootstrap 3 button, but there's two things we need to now add:

- the button tag included a tooltip. That doesn't yet appear because we need to activate the tooltip
- when clicked, the button should send our sendHelloWorld message to the back-end.

We can do these two things by adding another *onFragment* handler function, this time being one that fires after the *button1.html* fragment has loaded. So edit the *onFragment* section of *app.js* as shown in bold below:

```
onFragment: {
  // add handlers that fire after fragment contents are loaded into browser

  // remove everything from here

  'main.html': function() {
    $('#mainTitle').text('Hello World Application');
    EWD.getFragment('button1.html', 'mainContent');
  },  // remember to add this comma!

  'button1.html': function() {
    $('[data-toggle="tooltip"]').tooltip();
    $('#sendMsgBtn').on('click', function(e) {
      EWD.sockets.sendMessage({
        type: "sendHelloWorld",
        params: {
          text: 'Hello World!',
          sender: 'Rob',
          date: new Date().toUTCString()
        }
      });
    });
  }

},
```

Now try reloading the *indexbs.html* page: you should now see the tooltip when you hover over the button. When you click the button, it will send our *sendHelloWorld* message, just like in our original basic demo application. The JavaScript console should show the following:

There's no response message coming back from the back-end, but if you remember, the last thing we did was to make it a silent fire-and-forget handler. Just to prove that it's working, let's put back the *helloworld.js* file in *node_modules* to how it was before:

```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      var wsMsg = params;
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      savedMsg._setDocument(params);
      return {savedInto: 'AMessage'};
    },
    getHelloWorld: function(params, ewd) {
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      var message = savedMsg._getDocument();
      ewd.sendWebSocketMsg({
        type: 'savedMessage',
        message: message
      });
      return {messageRetrieved: true};
    }
  }
};
```

And, just for good measure, add the following to the *onMessage* object within *app.js*:

```
  onMessage: {

    // add handlers that fire after JSON WebSocket messages are received from back-end

    sendHelloWorld: function(messageObj) {
      toastr.clear();
      toastr.info('Message saved into ' + messageObj.message.savedInto);
    }
  }
```

Try it out now - when you click the save button, you should see a Toaster message appear in the top right corner, confirming that the message has been saved. The *toastr* utility is one of the pre-loaded widgets in the EWD.js / Bootstrap 3 framework.

Retrieving Data using Bootstrap 3

Finally, we'll add a second button that will retrieve our previously-saved message. Let's make it appear only after the Send button has been clicked once. There's several ways we could do this, but let's do the following. First edit button1.html as shown in bold below:

```
<button class="btn btn-info" type="button" id="sendMsgBtn" data-toggle="tooltip" data-placement="top" title=""
data-original-title="Send Message">
  <span class="glyphicon glyphicon-send"></span>
</button>

<button class="btn btn-warning" type="button" id="getMsgBtn" data-toggle="tooltip" data-placement="top" title=""
data-original-title="Retrieve Message">
  <span class="glyphicon glyphicon-import"></span>
</button>
```

Now edit *app.js* as shown below in bold:

```
EWD.sockets.log = true;   // *** set this to false after testing / development

EWD.application = {
  name: 'helloworld',
  timeout: 3600,
  login: false,
  labels: {
    'ewd-title': 'Hello World',
    'ewd-navbar-title-phone': 'Hello World App',
    'ewd-navbar-title-other': 'Hello World Application'
  },
  navFragments: {
    main: {
      cache: true
    },
    about: {
      cache: true
    }
  },

  onStartup: function() {

    // remove everything in here apart from this line:
    EWD.getFragment('main.html', 'main_Container');

  },

  onPageSwap: {
    // add handlers that fire after pages are swapped via top nav menu
    /* eg:
    about: function() {
      console.log('"about" menu was selected');
    }
    */
  },

  onFragment: {
    // add handlers that fire after fragment contents are loaded into browser

    // remove everything from here

    'main.html': function() {
      $('#mainTitle').text('Hello World Application');
      EWD.getFragment('button1.html', 'mainContent');
    },

    'button1.html': function() {
      $('[data-toggle="tooltip"]').tooltip();
      $('#getMsgBtn').hide();
      $('#sendMsgBtn').on('click', function(e) {
        EWD.sockets.sendMessage({
          type: "sendHelloWorld",
          params: {
            text: 'Hello World!',
            sender: 'Rob',
            date: new Date().toUTCString()
          }
        });
      });
      $('#getMsgBtn').on('click', function(e) {
        EWD.sockets.sendMessage({
          type: "getHelloWorld"
        });
      });
    }

  },

  onMessage: {

    // add handlers that fire after JSON WebSocket messages are received from back-end

    sendHelloWorld: function(messageObj) {
      toastr.clear();
      toastr.info('Message saved into ' + messageObj.message.savedInto);
      $('#getMsgBtn').show();
    },  // don't forget this comma!
    savedMessage: function(messageObj) {
      toastr.clear();
      toastr.warning(JSON.stringify(messageObj.message));
    }
  }

};
```

Try it out - you should see the second button appear after the response is received to clicking the first button. Clicking the second button will retrieve the saved message and display it in the Toaster.

Adding Navigation Tabs

Let's demonstrate one further facility that is included with the Bootstrap 3 framework for EWD.js.

First, find the file named *navlist.html* in the *bootstrap3* application directory and copy it into the *helloworld* application directory. It comes with two tabs defined: Main and About. We already have the Main page in operation, but you'll now need to find *about.html* in the *bootstrap3* application directory and copy it into the *helloworld* application directory.

if you look in the *indexbs.html* file, you'll find a placeholder *Container* div for the about fragment. You'll also see in the *app.js* file, in the *navFragments* object, what's needed to make the Nav tabs work with our *Main* and *About* fragments. In both cases they'll be cached after the first fetch, so subsequent clicks of the Nav tabs will just reveal the existing content in the page, without having to fetch it again.

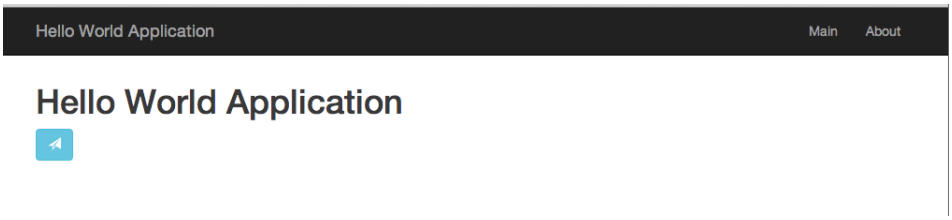
We just have to make two simple changes to *app.js* to bring the Nav tabs to life:

- fetch the navlist.html fragment when the application has started up (onStartup)
- activate the nav tabs after the navlist fragment has been loaded

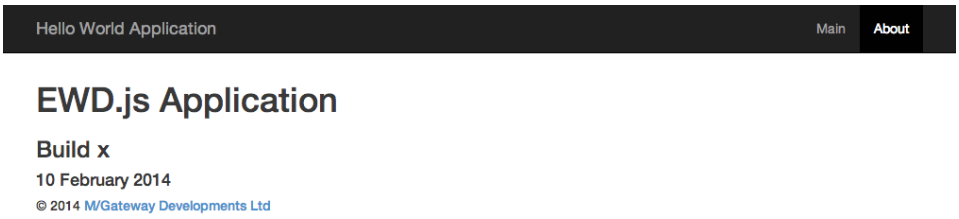
Simply make the changes shown below in bold:

```
EWD.sockets.log = true;   // *** set this to false after testing / development

EWD.application = {
  name: 'helloworld',
  timeout: 3600,
  login: false,
  labels: {
    'ewd-title': 'Hello World',
    'ewd-navbar-title-phone': 'Hello World App',
    'ewd-navbar-title-other': 'Hello World Application'
  },
  navFragments: {
    main: {
      cache: true
    },
    about: {
      cache: true
    }
  },

  onStartup: function() {

    // remove everything in here apart from this line:
    EWD.getFragment('main.html', 'main_Container');
    EWD.getFragment('navlist.html', 'navList');

  },

  onPageSwap: {
    // add handlers that fire after pages are swapped via top nav menu
    /* eg:
    about: function() {
      console.log('"about" menu was selected');
    }
    */
  },

  onFragment: {
    // add handlers that fire after fragment contents are loaded into browser

    'navlist.html': function(messageObj) {
      EWD.bootstrap3.nav.enable();
    },

    'main.html': function() {
      $('#mainTitle').text('Hello World Application');
      EWD.getFragment('button1.html', 'mainContent');
    },

    'button1.html': function() {
      $('[data-toggle="tooltip"]').tooltip();
      $('#getMsgBtn').hide();
      $('#sendMsgBtn').on('click', function(e) {
        EWD.sockets.sendMessage({
          type: "sendHelloWorld",
          params: {
            text: 'Hello World!',
            sender: 'Rob',
            date: new Date().toUTCString()
          }
        });
      });
      $('#getMsgBtn').on('click', function(e) {
        EWD.sockets.sendMessage({
          type: "getHelloWorld"
        });
      });
    }

  },

  onMessage: {

    // add handlers that fire after JSON WebSocket messages are received from back-end

    sendHelloWorld: function(messageObj) {
      toastr.clear();
      toastr.info('Message saved into ' + messageObj.message.savedInto);
      $('#getMsgBtn').show();
    },
    savedMessage: function(messageObj) {
      toastr.clear();
      toastr.warning(JSON.stringify(messageObj.message));
      $('#getMsgBtn').hide();
    }
  }

};
```

Try it out now - you should see the two Navigation tabs:



Clicking the *About* tab will display its contents which are initially as shown:



Simply edit *about.html* to change its contents to whatever you'd prefer to see!

You can add more Nav options as you wish. Simply copy and paste new ones into *navlist.html*. Make sure you have a placeholder Container in *indexbs.html* into which they will be loaded, and add them into the *onPageSwap* object in the *app.js* file. The caching is determined by a very simple rule: the first tag of a fragment should have an *id* of the form:

[navName]PageLoaded

For example, look at *main.html* and *about.html*: their first tags have ids of *mainPageLoaded* and *aboutPageLoaded* respectively.

Conclusions

That's the end of this tutorial. Hopefully you can now understand how EWD.js applications work and can be built. You also have most of the basic information you need in order to build responsive Bootstrap 3 applications. For more advanced techniques, examine the source code for the *ewdMonitor* application that is included with EWD.js.

We hope you enjoy developing applications using EWD.js!

Appendix 5

Installing EWD.js on the Raspberry Pi

Background

The Raspberry Pi is a very low-cost single-board ARM-based computer that has become extremely popular. There are two models, A and B which cost approximately $25 and $35 respectively. Either can be used with EWD.js

EWD.js requires a Mumps database in order to operate, but neither GT.M, Cache nor GlobalsDB have been ported to the ARM processor. However, EWD.js now includes a module named noDB.js which emulates a Mumps database via an identical set of APIs to those used for the real Mumps databases. noDB.js uses a JavaScript object to hold the data that would otherwise be stored in and manipulated within a Mumps database. This JavaScript object is regularly copied out to a text file named noDB.txt during the normal operation of EWD.js. When EWD.js is (re)started, the contents of the noDB.txt file are used to pre-populate the JavaScript object used by noDB.js. Hence, noDB.js is fully-persistent and behaves as if it is a single-user Mumps database.

The one limitation is that when using noDB.js, EWD.js must only be configured to spawn a single Child Process. Since we're going to be running EWD.js on a Raspberry Pi, this really isn't a significant limitation.

First Steps with the Raspberry Pi

When your Raspberry Pi arrives, you'll discover that by default it doesn't come with an operating system (OS). Furthermore instead of a hard-drive, it uses an SD card for its permanent storage. Get as fast an SD card as you can: Class 10 is currently the fastest. It should be 4Gb or larger. I got a 16Gb Class 10 SD for less than £10 and it works really well in the Raspberry Pi.

The first thing you need to do is, using another computer, format your SD card and copy an OS installer package called NOOBS onto the SD. You then put the SD into the Raspberry Pi's SD slot, plug in all the cables etc and turn it on. The NOOBS installer will fire up and you'll be offered a choice of OS's to install. You should choose Raspbian.

These steps are described fully here:

http://www.raspberrypi.org/wp-content/uploads/2012/04/quick-start-guide-v2_1.pdf

Be patient: it takes some time for Raspbian to install, but it all should just work for you.

To login to the Raspberry Pi:

username: pi
password: raspberry

Installing Node.js

When Raspbian boots for the first time, a set of configurations options will appear: you should choose the advanced option that allows you to enable the SSH server and client. This will allow you to access the Raspberry Pi remotely across your network.

Now you're ready to install Node.js. This is now an incredibly simple process. Just use the two commands described here:

http://www.raspberrypi.org/phpBB3/viewtopic.php?f=66&t=54817

In summary, do the following:

```
cd ~
wget http://node-arm.herokuapp.com/node_latest_armhf.deb
sudo dpkg -i node_latest_armhf.deb
```

It's a pretty quick installation procedure, especially if you have a fast SD card.

Check the installation when it completes by typing:

```
node -v
```

It should report back the version number (0.10.28 at the time of writing).

Here's what my installation looked like:

```
pi@raspberrypi ~ $ wget http://node-arm.herokuapp.com/node_latest_armhf.deb
--2013-11-15 14:56:12--  http://node-arm.herokuapp.com/node_latest_armhf.deb
Resolving node-arm.herokuapp.com (node-arm.herokuapp.com)... 184.73.160.229, 184.73.212.128, 50.17.229.49, ...
Connecting to node-arm.herokuapp.com (node-arm.herokuapp.com)|184.73.160.229|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5537702 (5.3M) [application/x-debian-package]
Saving to: `node_latest_armhf.deb'

100%[======================================>] 5,537,702   1.63M/s   in 3.2s

2013-11-15 14:56:15 (1.63 MB/s) - `node_latest_armhf.deb' saved [5537702/5537702]

pi@raspberrypi ~ $ sudo dpkg -i node_latest_armhf.deb
Selecting previously unselected package node.
(Reading database ... 65274 files and directories currently installed.)
Unpacking node (from node_latest_armhf.deb) ...
Setting up node (0.10.22-1) ...
Processing triggers for man-db ...
pi@raspberrypi ~ $ node -v
v0.10.22
```

Installing ewd.js

Now you're ready to install the *ewd.js* module:

```
cd ~
mkdir ewdjs
cd ewdjs
npm install ewdjs
```

Be patient. It takes a while for the Raspberry Pi to build the *socket.io* libraries.

During the installation process, you'll be asked to confirm the path in which EWD.js has been installed. Just accept the default that it suggests by pressing the Enter key, ie:

```
Install EWD.js to directory path (/home/pi/ewdjs):
```

The essential sub-components of EWD.js will be installed relative to this path.

You'll then be asked if you want to install the extra, optional sub-components for EWD.js:

```
EWD.js has been installed and configured successfully

Do you want to install the additional resources from the /extras directory?
If you're new to EWD.js or want to create a test environment, enter Y
If you're an experienced user or this is a production environment, enter N
Enter Y/N:
```

If you're new to EWD.js, typing Y (followed by the Enter key) is recommended.

When it completes, you should find that it's installed *ewd.js* into the directory path */home/pi/ewdjs/node_modules.* That should be it - you should be ready to fire up EWD.js

Starting EWD.js on your Raspberry Pi

Just enter the following commands:

```
cd ~/ewdjs
node ewdStart-pi
```

It should burst into life:

```
pi@raspberrypi ~ $ cd ewdjs
pi@raspberrypi ~/ewdjs $ node ewdStart-pi

******************************************
**** EWD.js Build 63 (20 May 2014) *******
******************************************

Master process: 2355
1 child Node processes will be started...
Child process 2356 has started
child process returned response {"type":"childProcessStarted","pid":2356,"processNo":0}
Sending initialise request to 2356: {
  "type": "initialise",
  "params": {
    "httpPort": 8080,
    "database": {
      "type": "gtm",
      "nodePath": "noDB"
    },
    "webSockets": {
      "path": "/ewdWebSocket/",
      "socketIoPath": "socket.io",
      "externalListenerPort": 10000,
      "maxDisconnectTime": 3600000
    },
    "ewdGlobalsPath": "./ewdGlobals",
    "traceLevel": 3,
    "logTo": "console",
    "logFile": "ewdLog.txt",
    "startTime": 1400765253179,
    "https": {
      "enabled": false,
      "keyPath": "/home/pi/ewdjs/ssl/ssl.key",
      "certificatePath": "/home/pi/ewdjs/ssl/ssl.crt"
    },
    "webServerRootPath": "/home/pi/ewdjs/www",
    "management": {
      "path": "/ewdjsMgr",
      "password": "keepThisSecret!"
    },
    "no": 0,
    "hNow": 5471674054,
    "modulePath": "/home/pi/ewdjs/node_modules",
    "homePath": "/home/pi"
  }
}
database loaded:
....

**** dbStatus: true
requireAndWatch: /home/pi/ewdjs/node_modules/globalIndexer.js loaded by process 2356
child process returned response {"pid":2356,"type":"log","message":"requireAndWatch:
/home/pi/ewdjs/node_modules/globalIndexer.js loaded by process 2356","processNo":0}
** Global Indexer loaded: /home/pi/ewdjs/node_modules/globalIndexer.js
child process returned response {"pid":2356,"type":"log","message":"** Global Indexer loaded:
/home/pi/ewdjs/node_modules/globalIndexer.js","processNo":0}
child process returned response {"pid":2356,"type":"firstChildInitialised","processNo":0}
First child process started.  Now starting the other processes....
Starting web server...
HTTP is enabled; listening on port 8080
   info  - socket.io started
```

You can now try out the demo EWD.js applications that come with the installation kit, e.g. try running *ewdMonitor* in your browser:

http://192.168.1.112:8080/ewd/ewdMonitor/index.html

Note: change the IP address of the URL to match that allocated to your Raspberry Pi. To find out what it is, use the command:

• *ifconfig*

The *ewdMonitor* password is specified in the *ewdStart-pi.js* file, and is initially set to *keepThisSecret!*

You can now start building your own EWD.js applications. Create them in the */home/pi/ewdjs/www/ewd/* directory. Follow the instructions in the main body of this document, or better still, skip now to Appendix 4 and follow the tutorial.

Installing MongDB

Not only can you run EWD.js using noDB.js to simulate a global database, you can now also use MongoDB either as an exclusive database or as a hybrid environment with noDB. In both cases, you'll need to install MongoDB on your Raspberry Pi. This can be a long and problematic process - the normal recommended approach is to build MongoDB from source. However, someone has kindly made pre-built binaries available, making the process really quick and simple.

You'll find these binaries at:

https://github.com/brice-morin/ArduPi/tree/master/mongodb-rpi/mongo/bin

All you need to do is create the directory path:

/opt/mongodb/bin

and copy the four files from the Github repository into the newly-created *bin* directory. Then, critically, set the permissions of the four files to be executable:

```
cd /opt/mongodb/bin
sudo chmod 775 *
```

Now you can start the MongoDB daemon:

```
cd /opt/mongodb/bin
./mongod --dbpath="/home/pi/mongodb"
```

This will create a new database in the */home/pi/mongodb* path. If the path doesn't exist, it will create it for you. You should see something like this:

```
pi@raspberrypi /opt/mongodb/bin $ ./mongod --dbpath="/home/pi/mongodb"
db level locking enabled: 1
Sat Jan  4 15:48:56
Sat Jan  4 15:48:56 warning: 32-bit servers don't have journaling enabled by default. Please use --journal if you
want durability.
Sat Jan  4 15:48:56
warning: some regex utf8 things will not work.  pcre build doesn't have --enable-unicode-properties
Sat Jan  4 15:48:56 [initandlisten] MongoDB starting : pid=4416 port=27017 dbpath=/home/pi/mongodb 32-bit hos-
t=raspberrypi
Sat Jan  4 15:48:56 [initandlisten]
Sat Jan  4 15:48:56 [initandlisten] ** NOTE: This is a development version (2.1.1-pre-) of MongoDB.
Sat Jan  4 15:48:56 [initandlisten] **       Not recommended for production.
Sat Jan  4 15:48:56 [initandlisten]
Sat Jan  4 15:48:56 [initandlisten] ** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of
data
Sat Jan  4 15:48:56 [initandlisten] **       see http://blog.mongodb.org/post/137788967/32-bit-limitations
Sat Jan  4 15:48:56 [initandlisten] **       with --journal, the limit is lower
Sat Jan  4 15:48:56 [initandlisten]
Sat Jan  4 15:48:56 [initandlisten] db version v2.1.1-pre-, pdfile version 4.5
Sat Jan  4 15:48:56 [initandlisten] git version: 47fbbdceb21fc2b791d22db7f01792500647daa9
Sat Jan  4 15:48:56 [initandlisten] build info: Linux raspberrypi 3.2.27+ #102 PREEMPT Sat Sep 1 01:00:50 BST
2012 armv6l BOOST_LIB_VERSION=1_49
Sat Jan  4 15:48:56 [initandlisten] options: { dbpath: "/home/pi/mongodb" }
Sat Jan  4 15:48:56 [initandlisten] waiting for connections on port 27017
Sat Jan  4 15:48:56 [websvr] admin web console waiting for connections on port 28017
```

Installing the Node.js interface for MongoDB

If you want to use MongoDB with EWD.js, you must install our synchronous Node.js interface.

Do the following:

```
cp /home/pi/ewdjs/node_modules/ewdjs/mongoDB/node-0.10/raspberryPi/mongo.node
/home/pi/ewdjs/node_modules/mongo.node
```

Running EWD.js with MongoDB exclusively

Start up EWD.js using the *ewdStart-pi-mongo.js* startup file:

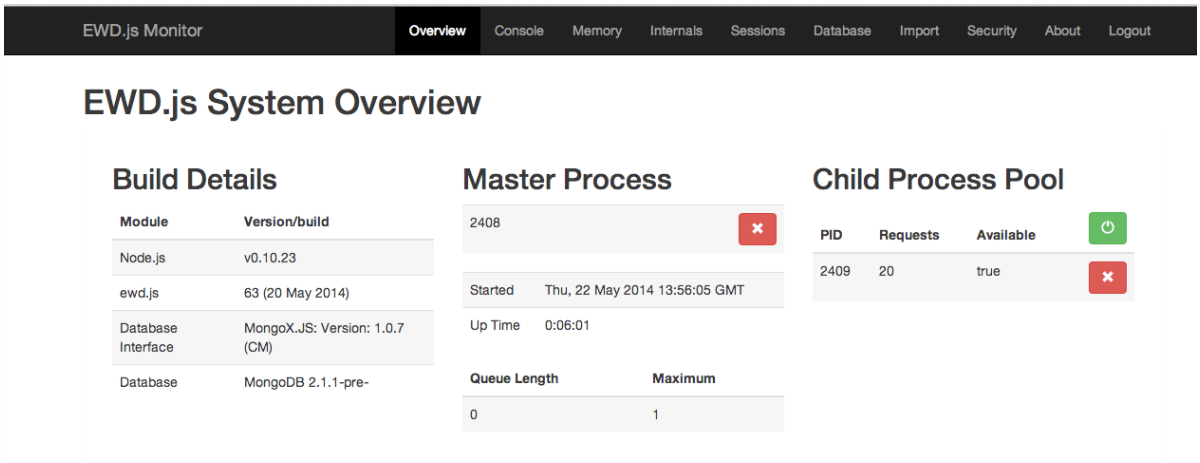
```
cd ~/ewdjs
node ewdStart-pi-mongo
```

EWD.js is now using MongoDB to emulate a global database, but you can also access MongoDB in its own right from within your own EWD.js applications. See Appendix 6 for further details.

If you run the ewdMonitor application, you should see it reporting that it is using MongoDB as its database (change the IP address as appropriate for your Raspberry Pi):

```
http://192.168.1.113:8080/ewd/ewdMonitor/index.html
```

Enter the password (keepThisSecret! by default) and you should see:



Try clicking the EWD Sessions and Persistent Objects tabs. Although EWD.js thinks both are stored as globals in a global database, the data you see is actually being stored in MongoDB.

Note that if you use MongoDB as the exclusive database, you can, in theory, use more than one child process.

Running EWD.js as a hybrid environment

You can also use MongoDB alongside noDB.js to create a hybrid environment (see Appendix 6 for full details about hybrid environments). Instead of using the MongoDB-specific EWD.startup file, modify *ewdStart-pi.js* as follows (additional line you should add is shown in bold):

```
var ewd = require('ewdjs');

var params = {
      poolSize: 1,
      httpPort: 8080,
             https: {
               enabled: false,
      },
      traceLevel: 3,
      database: {
        nodePath: 'noDB',
        also: ['mongodb']
      },
      management: {
        password: 'keepThisSecret!'
     }
};

ewd.start(params);
```

When you start EWD.js using this file, it will run normally using noDB.js with a single child process. However, MongoDB will be available for your applications. See Appendix 6 for details.

Note that when run in hybrid mode, you can only use a single child process.

Have fun with your Raspberry Pi and EWD.js!

Appendix 6

Configuring EWD.js for use with MongoDB

Modes of Operation

EWD.js can be configured to work with MongoDB in one of two ways:

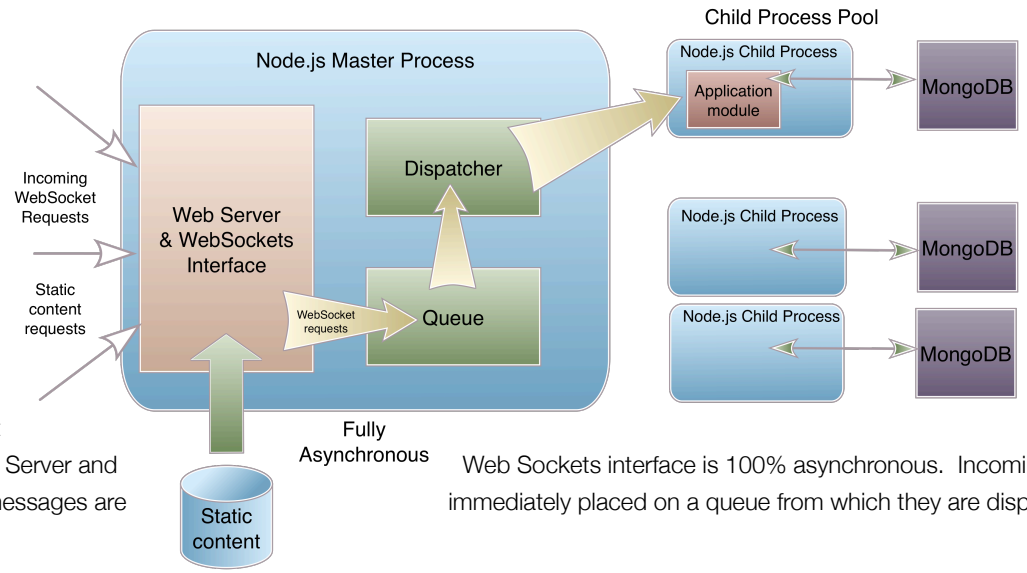
Exclusively using MongoDB (ie without a Mumps database). In this mode of operation, the internal mechanics of EWD.js that normally rely on a Mumps database and the built-in Session Management functionality are provided by MongoDB. To make this possible, EWD.js makes use of a MongoDB-based emulation of Mumps Globals.

A hybrid environment, whereby a Mumps database is used to support the internal mechanics of EWD.js and the built-in Session Management functionality. Separately, the application developer can make use of MongoDB using the synchronous APIs provided by EWD.js.

Node.js Synchronous APIs for MongoDB?

The use of synchronous APIs to access MongoDB from within a Node.js environment may seem somewhat heretical! Normally-accepted wisdom is that Node.js should use non-blocking I/O and therefore the use of asynchronous logic when accessing a slow resource such as a database. So why does EWD.js use synchronous APIs and don't they affect performance?

To answer the second question first: no they don't. This is because of the hybrid, decoupled architecture used by EWD.js:



The master process provides in EWD.js that Web Sockets interface is 100% asynchronous. Incoming the main Web Server and immediately placed on a queue from which they are dispatched to WebSocket messages are

one of a pool of pre-forked child Node.js processes. As soon as a request is passed to a child process it removes itself from the available pool, and returns itself to the pool as soon as processing is completed. Each EWD.js Node.js child process therefore processes only a single WebSocket request at a time. This brings several benefits:

- since each child process is only ever handling a single user's WebSocket request, it can afford to use synchronous, blocking I/O since it isn't holding anyone else up in doing so

- by devolving WebSocket request processing to child processes, EWD.js can make use of all the available CPU cores. Compute-intensive processing doesn't affect the performance of the master process or the processing taking place in other child processes

- by being able to use synchronous APIs, the back-end application logic can be written in an intuitive way. Application development is faster and the resulting code is significantly more maintainable than is possible with asynchronous logic. Additionally, proper try/catch error handling can be used.

EWD.js allows child processes to be started and stopped dynamically without having to stop and restart the master process.

Using MongoDB exclusively with EWD.js

If you want to use EWD.js just with MongoDB alone, then there are three key steps:

- Install Node.js 0.10.x on your server. At the time of writing, the latest version is 0.10.28.
- Install MongoDB on your server
- Install the *ewd.js* Node.js module. See the section *Installing ewd.js* on page 7 of this document and also the section *Setting up the EWD.js Environment* on pages 8-9.

Navigate to the node_modules directory under your EWD.js Home Directory (eg ~/ewdjs) , and then navigate down the path: *node_modules/ewdjs/mongoDB/node-0.10* and you'll find four sub-directories:

- linux32
- linux64
- raspberryPi
- winx64

Inside each of these directories you'll find a file named *mongo.node* which is a pre-built binary version of our synchronous MongoDB interface for 32-bit Linux, 64-bit Linux, the Raspberry Pi and 64-bit Windows respectively. Copy the appropriate file for your server to the *node_modules* directory under your EWD.js home directory (eg *~/ewdjs/node_modules*).

For example, if you are installing on a 64-bit Windows machine, your EWD.home directory is probably *c:\ewdjs*, in which case copy *c:\ewdjs\node_modules\ewdjs\mongoDB\node-0.10\winx64\mongo.node* to *c:\ewdjs\node_modules\mongo.node*

Assuming you properly followed the earlier installation steps (pages 7-9) you should also have a file named *mongoGlobals.js* in the same *node_modules* directory as *mongo.node*. *mongoGlobals.js* looks after the emulation of Mumps Globals using MongoDB.

You can now start EWD.js. You'll find an example startup file named *ewdStart-mongo.js* in your EWD.js Home Directory. This has been written assuming you're using EWD.js on a Windows server, with an EWD.js Home Directory of c:\node. All you need to do is modify the modulePath value to match the full path of your server's node_modules directory (eg on a Linux machine, it may be */home/username/node/node_modules*). Start EWD.js in the normal way using this file:

Linux:

```
cd ~/ewdjs
node ewdStart-mongo
```

Windows:

```
cd c:\ewdjs
node ewdStart-mongo
```

You should be able to run the ewdMonitor application (see page 15). This should report the database as being MongoDB. Everything should behave as if you are using a Mumps database.

You can access MongoDB within your back-end modules via the *ewd.mongoDB* object which provides access to all the interface APIs, eg:

```
module.exports = {
  onMessage: {
    getVersion: function(params, ewd) {
      var version = ewd.mongoDB.version();
      return {version: version};
    }
  }
};
```

See later for details on the available MongoDB APIs.

Creating a Hybrid Mumps/MongoDB EWD.js System

The advantage of a hybrid system is that the internal mechanics within EWD.js that assume the use of a Mumps database aren't slowed down by the inefficiencies of the MongoDB emulation of Mumps globals. Instead, it will run at full speed. You'll also be able to use the EWD.js Session without worrying about performance.

Additionally you'll have full, normal access to MongoDB, but, if you're modernising a legacy Mumps Healthcare application, you'll be able to access both legacy Mumps data and MongoDB data.

To set up a hybrid environment, first decide on the Mumps version you want to use. If you're working with a legacy healthcare system, you'll probably know which version you have to use: eg Cache or GT.M. If you just want something to look after the EWD.js mechanics, then GlobalsDB is probably the best option: it's free, very fast, very easy to install and available for all operating systems: perfect as an EWD.js Session engine!

Use the instructions elsewhere in this documentation to install and configure Node.js, the Mumps database and the *ewd.js* module, just as if you were going to use a Mumps database on its own.

Next install MongoDB (if you haven't already).

Next, navigate to the path: ~/ewdjs/node_modules/ewdjs/mongoDB/node-0.10 and you'll find four sub-directories:

- linux32
- linux64
- raspberryPi
- winx64

Inside each of these directories you'll find a file named *mongo.node* which is a pre-built binary version of our synchronous MongoDB interface for 32-bit Linux, 64-bit Linux, the Raspberry Pi and 64-bit Windows respectively. Copy the appropriate file for your server to the *node_modules* directory under your EWD.js home directory.

For example, if you are installing on a 64-bit Windows machine, your EWD.home directory is probably *c:\ewdjs*, in which case copy *c:\ewdjs\node_modules\ewdjs\mongoDB\node-0.10\winx64\mongo.node* to *c:\ewdjs\node_modules\mongo.node*

Finally, you need to modify your EWD.js startup file and add the database.also['mongodb'] property, eg:

```
var ewd = require('ewdjs');

var params = {
      poolSize: 1,
      httpPort: 8080,
             https: {
               enabled: false,
      },
      traceLevel: 3,
      database: {
        nodePath: 'noDB',
        also: ['mongodb']
      },
      management: {
        password: 'keepThisSecret!'
      }
};

ewd.start(params);
```

You can now start EWD.js as usual, eg

```
cd ~/ewdjs
node ewdStart-gtm
```

Although your specified Mumps database is looking after EWD.js, session management etc, you can access MongoDB within your back-end modules via the *ewd.mongoDB* object which provides access to all the interface APIs, eg:

```
module.exports = {
  onMessage: {
    getVersion: function(params, ewd) {
      var version = ewd.mongoDB.version();
      var result = ewd.mongoDB.retrieve("db.ccda", {ccdaNo: 1});
      return {version: version, result: result};
    }
  }
};
```

A Summary of the Synchronous MongoDB APIs

The following is a summary of the most common APIs that you'll use with EWD.js and MongoDB. A detailed list will be published elsewhere in due course. Behind the scenes, these APIs invoke the standard MongoDB APIs, so if you're familiar with MongoDB you will probably recognise them and understand their behaviour.

open(connectionObject)

Opens the connection to MongoDB. You won't need to access this API - EWD.js looks after it for you.

```
mongoDB.open({address: "localhost", port: 27017});
```

insert(collectionName, object)

Adds an object to a collection

```
var result1 = mongoDB.insert("db.test", {department: "mumps", key: 11, name: "Chris Munt", address: "Banstead",
phone: 5456312727});
console.log("Insert Record (MongoX sets new _id): Created: " + user.object_id_date(result1._id).DateText);
console.log("Insert Result: " + JSON.stringify(result1, null, '\t'));
```

update(collectionName, matchObject, replacementObject)

Modifies an object in a collection. The match object defines the name/value pairs that match the object to be updated.

```
mongoDB.update("db.test", {department: "mumps", key: 3}, {department: "mumps", key: 3, name: "John Smith",
phone_numbers: [{type: "home", no: 909090}, {type: "work", no: 111111}]});
```

retrieve(collectionName, matchObject)

Finds one or more objects in a collection and returns it/them as an array of objects. The match object defines the name/value pairs that match the object(s) to be found.

```
var result = mongoDB.retrieve("db.test", {department: "mumps"});
console.log("Data Set: " + JSON.stringify(result, null, '\t'));

console.log("Get OBJECT by ID (" + result1._id + ")\n");
result = mongoDB.retrieve("db.test", {_id: result1._id});
console.log("Data Set: " + JSON.stringify(result, null, '\t'));
```

remove(collectionName, matchObject)

Finds one or more objects in a collection and removes it/them from the collection. The match object defines the name/value pairs that match the object(s) to be removed.

```
mongoDB.remove("db.test", {department: "mumps", key: 3});

// remove everything from the collection:

mongoDB.remove("db.test", {});
```

createIndex(collectionName, indexObject, [params])

Indexes a collection, based on the name/value pairs specified in the indexObject

```
var result = mongoDB.create_index("db.testi", {key: 1}, "key_index", "MONGO_INDEX_UNIQUE, MONGO_IN-
DEX_DROP_DUPS");

console.log("Result of create_index(): " + JSON.stringify(result, null, '\t'));
console.log("\nGet the new status information for db.testi ... \n");
var result = mongoDB.command("db", {collStats : "testi"});
```

command([params])

Invokes one of a number of commands. Some examples are listed below. *listCommands* will provide you with a list of available commands:

```
var result = mongoDB.command("db", {listCommands : ""});
console.log("Command (listCommands) : " + JSON.stringify(result, null, '\t'));

result = mongoDB.command("db", {buildInfo : ""});
console.log("Command (buildInfo): " + JSON.stringify(result, null, '\t'));

result = mongoDB.command("db", {hostInfo : ""});
console.log("Command (hostInfo) : " + JSON.stringify(result, null, '\t'));
```

version()

Returns the MongoDB API version

```
console.log(mongoDB.version());
```

close()

Closes the connection to MongoDB API. Note: you normally don't need to use this API when using EWD.js

```
mongoDB.close();
```