EWD.js

リファレンスガイド

訳者序文

EWD.js は、MongoDB や MUMPS データベース（例えば商用の Caché、無償の GlobalsDB と GT.M）を統合して、高性能なブラウザベースのアプリケーションを構築するための、Node.js と JavaScript ベースのオープンソースフレームワークです。

JavaScript によるフレームワークの目を見張るような発展により日常使用するほとんどの事がブラウザで表現できる様になり、多くの人々が JavaScript を使って開発を行っています。

EWD.js で使用する M データベースは M 言語を構成する高性能データベースです。　M 言語は，近年に創造された新しい言語ではなく，すでに 30 余年近い歴史の中で高度に磨きあげられ，熟成されてきた言語です。M 言語は，開発当初からすでに大規模データベース・アプリケーションと，マルチユーザ・システムを開発するための，最も効率の高い方法を追求しながら作り上げられました。M 言語は医療、金融、流通などの多くの分野で使用されています。

Rob Tweed 氏は M 言語と JavaScript の親和性に注目し、M 言語のために EWD.js というオープンソースの素晴らしいフレームワークを書いてくださいました。　EWD.js は M のデーターベースを M 言語の知識が無くとも JavaScript で取り扱うことが出来る技術です。　また従来の M 言語ルーチンを併用したり、従来のコードを活用する事も可能です。　M データーベースは、ユニークでパワフルなデータベーステクノロジーの一部です。EWD.js により M 言語の技術者は新しい JavaScript のインターフェースを手にすることが出来、また M 言語を使わない技術者も高性能な M のデータベースが M 言語を勉強することなく使用できる様になるでしょう。

このマニュアルを手に是非素晴らしい EWD.js の世界を堪能して頂ければと思います。

本書は EWD.js　Reference Guide の日本語訳です。原文やプログラムは
http://ec2.mgateway.com/ewd/ws/index.html より入手してください。

尚、付録５（Raspberry Pi（イギリスの教育用ワンボードコンピュータ）への EWD.js のインストール）、付録６（MongoDB の設定）の翻訳は省略しています。

最後に本マニュアルの日本語版の作成を快諾してくださった、Rob Tweed 氏に深謝いたします。

目次

イントロダクション

背景

EWD.js は、MongoDB や MUMPS データベース（例えば Caché、GlobalsDB、GT.M）を統合して、高性能なブラウザベースのアプリケーションを構築するための、Node.js と JavaScript ベースのオープンソースフレームワークです。

EWD.js は、ブラウザベースのアプリケーションに対して完全に新しいアプローチを取っており、ブラウザと中間層である Node.js との間の通信手段として WebSocket を使用しています。 EWD.js は Node.js のための ewd.js モジュールを必要とします。 ewd.js は以下の機能を提供します。

- 静的なコンテンツを提供するためのウェブサーバとして機能します。
- WebSocket のバックエンド層を提供します。
- 選択した MongoDB または MUMPS データベースと統合するための Node.js のベースの子プロセスのプールを管理・保守します。
- ユーザーセッションを作成、管理・保守します
- MUMPS データを永続 JSON ストレージとして扱うことができるように、MUMPS データベースをネイティブの JSON データベースとして投影します。 また、MongoDB を MUMPS データベースと排他的に、あるいは一緒に EWD.js で使用することができます。 後者の構成では、他のすべてのデータベース要件のために MongoDB を使用しながら、MUMPS データベースに非常に高性能なセッション管理/持続性を提供させることができます。
- 不正使用からデータベースを保護するために必要なすべてのセキュリティを提供します。

EWD.js アプリケーションは、フロントエンドとバックエンドのロジックの両方とも完全に JavaScript で書かれています。 MUMPS 言語の知識は必要ありません。 しかし、Caché や GT.M を使用している場合は、必要に応じてバックエンド内の JavaScript のロジックからレガシーMUMPS 関数にアクセスし、呼び出すことができます。

EWD.js の背後にある思考や哲学の背景と MUMPS データベースのユニークな機能については、ブログ http://robtweed.wordpress.com/ の文書を参照してください。

このドキュメントでは、EWD.js のインストールと使用する方法について説明しています。

すぐに EWD.js システムを作成して試してみたい場合は、付録 1 から 3 をお読みください。

あなたが初めて EWD.js を使うのであれば、ある程度の時間を割いて、付録 4 のステップバイステップのチュートリアルを読んで試してください。　EWD.js の例として単純な「Hello World」を構築するプロセスを紹介しています。　これは EWD.js.の背後にある概念や仕組みを体得でき、MUMPS データベースが容易に JSON 文書を格納および取得することを理解するのに役立ちます。MongoDB を使用したい場合は、付録 6 を参照して下さい。

インストールと設定

前提条件と背景

EWD.js 環境は、以下のコンポーネントで構成されます。

- Node.js
- MongoDB 又は Mumps データベース
  (例)
  - GT.M
  - GlobalsDB
  - Caché
- Node.js 用の ewd.js モジュール

EWD.js は、Windows、Linux、または Mac OS X にインストールできます。 Caché、GlobalsDB は、これらのオペレーティングシステムのデータベースとして使用できます。 GT.M は、Linux システム上でのみ使用できます。 MongoDB は、3 つすべてのオペレーティングシステムで使用できます。

MUMPS データベースを使うために、どちらかを選択する必要がある場合は、次の点に注意して下さい。

- GlobalsDB は無料ですが、クローズドソースの製品です。 しかし、その使用条件または再配布に制限はありません。 それは本質的に Caché の MUMPS データベースエンジンのコアです。 Windows、Mac OS X および Linux で利用できます。 インターシステムズからの GlobalsDB の技術的なサポートやメンテナンスはありません。「as-is」で使用するように提供されています。 GlobalsDB は、MUMPS データベースをインストールするのに最速かつ最も簡単な、おそらく EWD.js.を初めて使用する場合の最適なオプションです。 付録 1 は、わずか数分で EWD.js の完全に動作する GlobalsDB ベースバージョンを作成する方法について説明しています。

- GT.M はフリーでオープンソースの、非常に強力な製品です。 動作は Linux システムに限定されています。 付録 2 は、わずか数分で、完全な EWD.js の GT.M ベースのバージョンを作成する方法について説明しています。 もし、VISTA と呼ばれるオープンソースの電子ヘルスケア記録（EHR）で EWD.js を使用することに興味があるなら、dEWDrop 仮想マシンをダウンロードして下さい（http://www.fourthwatchsoftware.com/）。 これは、Vista の完全に動作するバージョンを含む、設定済みの GT.M システムです。 dEWDrop 仮想マシン上の

EWD.js を起動して実行するには付録 3 の簡単な指示に従ってください。

- Caché は、Windows、 Mac OS X、そして Linux.で動作し、独自の商業ライセンスが適用される非常に強力な製品です。　EWD.js は MUMPS データベースエンジンのみ必要ですが、すでに Caché を使用している場合は、EWD.js アプリケーションの JavaScript のバックエンド· コード内から既存のコードやクラスを実行することができます。　そのため EWD.js は、Caché に付属している CSP と ZEN の Web フレームワークに大きな（はるかに簡単で軽量な）代替手段を提供し、利用可能な Caché のライセンスをより効率的に使用します。

これらのデータベースの詳細については、以下を参照してください：

- GlobalsDB: http://www.globalsdb.org/
- GT.M: http://www.fisglobal.com/products-technologyplatforms-gtm
- Caché: http://www.intersystems.com/cache/index.html

3 つのすべてのデータベースは非常に高速であり、Caché と GT.M の両者は極めて大規模な企業レベルまで拡張することができます。　GlobalsDB と Caché 用の Node.js インタフェースは、今のところ、GT.M.用の NodeM インターフェイスより約 2 倍高速です。

EWD.js を使用する際、すべての 3 つの MUMPS データベースは、アプリケーションロジック内からのデータアクセス方法や操作方法で同一に見えることに注意して下さい。　知っておくべき唯一の小さな違いは、ewd.js モジュール用に使用するスタートアップファイル内の構成設定です（後述）。それ以外は、例えば Caché で開発した EWD.js アプリケーションは、通常、GT.M システム上でコードの変更なしで動作します。

MongoDB で EWD.js を使用する場合は、2 つの選択肢があります。

- 唯一のデータベースとして、排他的に MongoDB を使用することができます。　この方法の使用を決定した場合、EWD.js は、MUMPS グローバルの MongoDB のエミュレーションを使用してユーザー· セッションを管理および維持します。 本書で説明する MUMPS データベースのすべての JSON ベースの API は、MongoDB のものと同様に動作します。　ただし、MUMPS データベースを使用している場合には、このエミュレーションのパフォーマンスが大幅に低下することを認識する必要があります。　単一の非常に限られた量の EWD.js セッションストレージを使っている場合ならば、小規模から中規模のアプリケーションには十分な性能です。もちろん、EWD.js は MongoDB に期待する同じレベルの性能で、永続的 JSON オブジェクト集合として、標準的な方法で MongoDB アクセスができます。

- 別の方法として、MUMPS データベースを使用して非常に高性能なセッションマネージメン

トを提供し、その他のデータベース動作のためには MongoDB を使用する、ハイブリッド手法が使用できます。　このモードでは、MongoDB を、その通常の方法で、JSON オブジェクトの永続的なコレクションとして、通常の期待と同じレベルの性能で使用できます。　このハイブリッド動作では、MongoDB の中でデータを維持すると同時に、レガシーMUMPS データにアクセスすることができますので、従来の医療アプリケーションの近代化と拡張のために便利です。

ブラウザ側の JavaScript フレームワークの選択に応じて、当然のことながら、フレームワーク (ExtJS, jQuery,Dojo など) をインストールする必要があります。

Node.js の MUMPS データベースとのインタフェース

Node.js のインタフェースは、3 つのすべての MUMPS データベースで使用できます。

Caché と GlobalsDB

インターシステムズは、Caché と GlobalsDB のために独自のインタフェース・ファイルを提供しています。　実際に同じファイルが両方の製品に使用され、それらは自由に入れ替えることができます。　Node.js の最新バージョン用のインタフェース･ファイルのバージョンが GlobalsDB の最新バージョンに含まれています。　Node.js の最新バージョン 0.10.x で Caché を使用したい場合は、GlobalsDB（非常に小さなダウンロードと非常に簡単なインストールプロセスです）のコピーをスペアマシンにダウンロードして、インストールし、Caché システム必要とするファイルをコピーしてください。

インタフェース･ファイルは、Caché2012.x 以降に含まれていますが、インターシステムズのリリースサイクルのために、これらのファイルは古くなっています。　さらに、Node.js のインタフェース・ファイルは、Caché の 2012.x 以前のバージョンでも動作するため、EWD.js は Caché のほぼすべてのバージョンで使用できます(訳注：動作しますが完全ではありません)。

Caché と GlobalsDB インストールの bin ディレクトリにインタフェース･ファイルがあります。インタフェース･ファイルは cache[nnn].node と名付けられています。nnn は Node.js のバージョンを示しています。　例えば、cache0100.node は Node.js のバージョン 0.10.x.ためのファイルです。　このファイルには、適切な場所(後述)にコピーし、cache.node に名前を変更する必要があります。

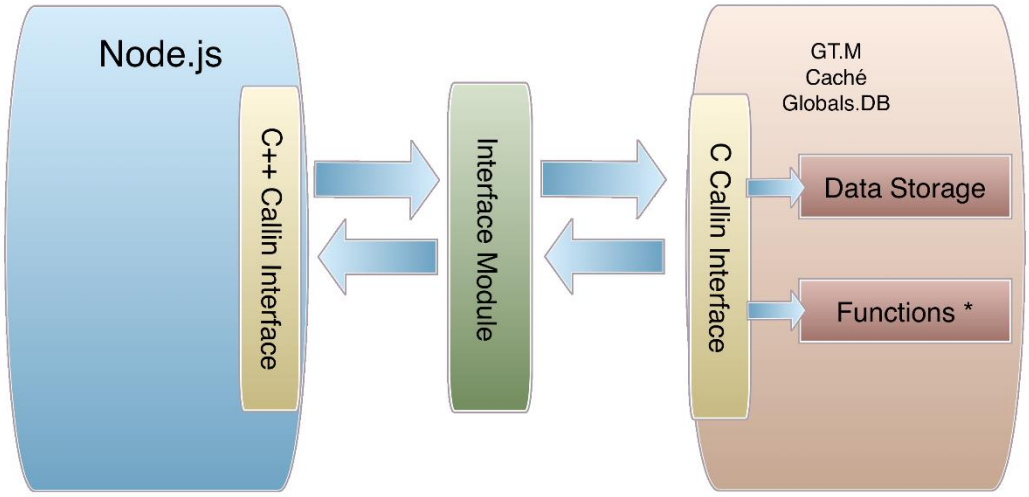
GT.M

GT.M のオープンソース Node.js インタフェースである NodeM は、David Wicksell によって作成されました。　NodeM は https://github.com/dlwicksell/nodem に置いてあります。　これはインターシステムズが提供する Node.js.用インタフェースの API との完全な互換性があります。

dEWDrop VM には、NodeM が含まれています。それ以外インストールの場合は、GitHub のページにあるインストール指示に従ってください。　NodeM インタフェースファイルには mumps.node という名前が付けられています。

アーキテクチャ

Node.js の MUMPS データベースインタフェースのアーキテクチャを以下に図示します。



インタフェースの両方のバージョンに完全非同期 API バージョンは存在しますが、EWD.js では開発者が非常に簡単に使用でき、また非同期よりよりもはるかに高速である、同期バージョンを使用しています。　これは Node.js のデータベースアクセスでの一般に認められた知恵に真っ向から対峙するやり方に見えます。　しかし後ほどお見せするように、ewd.js モジュール（EWD.js が依存する）のアーキテクチャは、ちょっと見ではありそうな同期アクセスについての問題が実際にはないことを約束します。

Node.js の MongoDB のへのインタフェース

EWD.js（下記のセクションを参照）の基礎となる ewd.js モジュールの、要求キュー/プリフォークした子プロセスのプールアーキテクチャは、データベース･アプリケーション開発者が Node.js の多くの利点を達成しながら、同期ロジックを使用できるように意図的に設計されています。　このため、EWD.js は、特別に、M/Gateway 社が開発した MongoDB のための同期インタフェイスモジュールを使用しています。　インタフェースモジュールは標準の MongoDB の API 周りのカスタムラッパーなので、すべての通常の MongoDB 機能が使用可能になっています。　それは標準的な方法で MongoDB に接続し、指定されたアドレスとポートに TCP 経由で接続します（通常デフォルトでは localhost とポート 27017）。　主な違いは、Node.js.を扱う際の標準である通常の非同期処理より、 むしろ標準的な直感的な同期ロジックを使用してすべてのデータベース処理ロジックを書くこ

とができるということです。

最初の設定手順

EWD.js を自動的に構築する手順については、このマニュアルの最後の付録 1～3 を参照してください。 これらは、EWD.js と MUMPS データベースを使用するほとんどのユーザー、特に新規のユーザーにお勧めします。

手動で EWD.js をインストール、設定する（カスタム環境用など）必要がある場合の EWD.js 環境を作成する手順は次のとおりです。

- Node.js のインストール：http://nodejs.org/ を参照してください。 EWD.js は Node.js のほとんどのバージョンで使用できますが、最新バージョン、（執筆時点で 0.10.x）をお勧めします。
- MongoDB またはあなたの選択した MUMPS データベースをインストールします。詳細については、関連する Web サイトを参照してください。
- MUMPS データベース用の適切な Node.js のインターフェイスをインストールします。

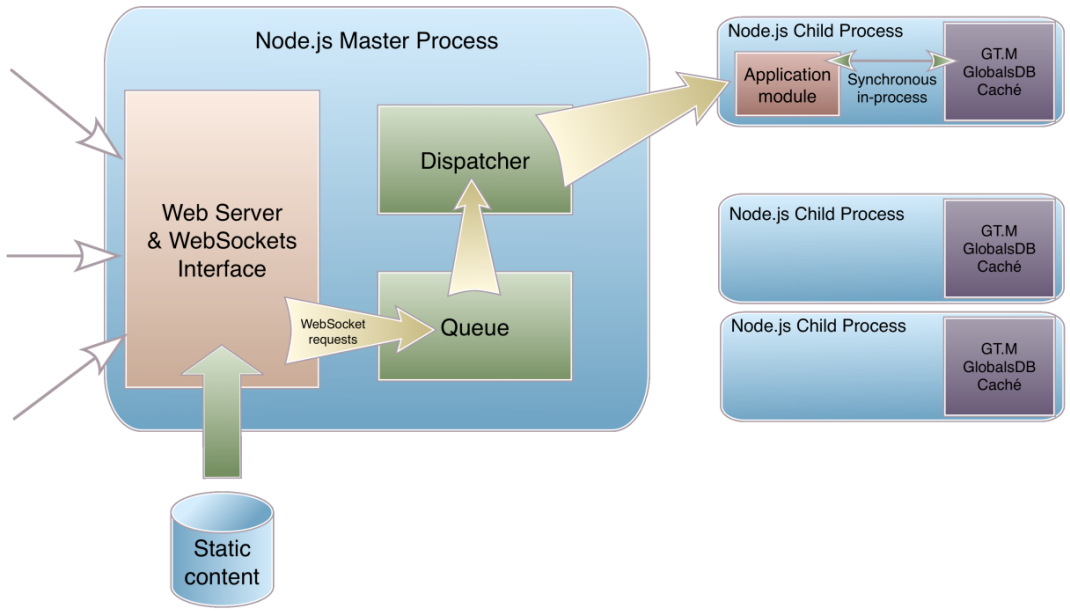
ewd.js

Node.js 用の ewd.js モジュールが EWD.js.のランタイム環境を提供します。 ewd.js は Apache2 ライセンスのオープンソースプロジェクトとして公開、保守されています。

https://github.com/robtweed/ewd.js 参照。

ewd.js のアーキテクチャ

ewd.js モジュールのアーキテクチャを、以下の図に示します。



マスターNode.js のプロセスは、Web サーバと WebSocket サーバーとして機能します。EWD.js で使用する Web サーバの要求は、静的コンテンツに対する要求に限定されます。 ewd.js 起動/設定ファイルには、プールサイズを指定します。 起動する Node.js の子プロセスの数です。 各子プロセスは自動的に MongoDB または MUMPS データベース･プロセスへの接続を作成します。 インターシステムズインタフェースと NodeM の両方とも、データベースへの内部接続プロセスを生成します。 Caché を使用している場合は、それぞれの子プロセスが Caché ライセンスを消費することに注意してください。 MongoDB の場合は、接続は標準的な TCP ベースの接続を使用していますが、API は同期接続になります。

アプリケーションからの着信の WebSocket メッセージは、自動的に処理されているキューに配置されます。Node.js の子プロセスが空きである（すなわち、すでに WebSocket のメッセージを処理していない）場合、キューに入れられた WebSocket メッセージが処理のために送信されます。 ewd.js は子プロセスが処理するメッセージを受け付けると自動的に利用可能なプールから削除して、一度に単一の WebSocket メッセージを処理することを保証します。 子プロセスは処理が完了するとすぐに、自動的に利用可能な子プロセスのプールに戻されます。

WebSocket のメッセージの処理は、Node.js のモジュールとして、JavaScript で書かれた、独自のアプリケーションロジックによって実行されます。 モジュールは MongoDB や MUMPS データベースへの完全なアクセスを行い、GT.M または Caché を使用している場合は、レガシーMUMPS 関数を呼び出すことができます。

フロントエンドマスターNode.js のプロセスから MUMPS データベースへのアクセスを切り離すために、ewd.js は以下の 2 つのことを行います。

- アーキテクチャは適切に拡張することができ、そして子プロセスは、複数のコアプロセッサの CPU を利用することができます。
- Node.js/MUMPS や Node.js/ MongoDB のインターフェイス内で、同期 API を使用することができ、 アプリケーション開発者のためのコーディングが簡単に、より直感的になります。また同期 API は非同期のものよりもはるかに高速実行するという恩恵を受けます。 複雑なデータベース操作ロジックは、深くネストされたコールバックを必要とせず、非常に簡単になります。 一方で、他のロジックのためにバックエンドモジュール内で標準の非同期コーディングを使用することができます。 EWD.js は本当にあなたに世界最高を提供します。

EWD.js は、特に ewd.js モジュールを利用するアプリケーションを開発するためのフレームワークを提供するために設計されています。

ewd.js のインストール

EWD.js のインストールは非常に簡単で、ほとんどの構成のための設定は自動化されています。このマニュアルの最後にある付録 1～3 を参照してください。

付録で示したものとは異なる独自のカスタムインストールを行う必要がある場合は、このセクションでインストール方法を説明しています。

Node.js.の一部としてインストールされる、ノードパッケージマネージャ（NPM）を使用する必要があります。

注：初めて Node.js を使用する場合に： これは主にコマンドラインプロンプトを経由して制御する製品です。 以下に説明するステップを実行するために Linux や OS X のターミナルウィンドウ、または Windows コマンドプロンプトウィンドウのいずれかを開いて下さい。

ewd.js をインストールする前に、あなたが EWD.js を実行したいと思うディレクトリパスにいることを確認してください。 場所はお使いの OS、MUMPS データベースとあなたの個人的な選択に依存します。 インストールを行うために選択したディレクトリパスの下に ewdjs という名前のサブディレクトリを作成し、そのディレクトリに移動し、ewd.js モジュールをインストールするために、npm を呼び出します。 EWD.js ホームディレクトリとして ewdjs ディレクトリを参照します。

windows

C ドライブに ewdjs ディレクトリを作成し、その中へインストールします。

(例)

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

EWD.js ホームディレクトリは C:\ ewdjs になります。

Mac OS X と Linux

ユーザーのホームディレクトリの下に ewdjs ディレクトリを作成し、その中へインストールします。

(例)

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

EWD.js ホームディレクトリは、~/ewdjs になります。

全プラットフォーム

すべてのケースで、npm が ewdjs モジュールのインストールを完了した後に、EWD.js ホームディレクトリの下に node_modules、サブディレクトリを作成されています。　この中には ewd.js モジュール、サポートファイルと EWD.js が必要とするモジュールが含まれています。

EWD.js 環境の設定

EWD.js の自動化インストールスクリプト（付録 1～3 参照）のいずれかを使用した場合は、EWD.js 環境が設定されているので、このセクションを無視してください。　EWD.js 環境の作成およびカスタムインストールを作成する方法についての詳細を知りたい場合には、このセクションが必要な情報を提供します。

インストールしたばかりの ewd.js モジュールパッケージは、EWD.js と ewd.js 環境の管理/モニタリング･インタフェースを提供するアプリケーションと EWD.js アプリケーションの書き方示すいくつかの事前に構築されたアプリケーションが含まれています。　それらを使用して独自のアプリケーションを作成して実行するためには、正しい場所にいくつかのファイルやディレクトリをコピーする必要があります。　EWD.js にはスタンドアロン Node.js と JavaScript のインストールファイルが含まれており、実行してこれらすべてのファイルを正しい場所に移動しなければなりません。EWD.js ホームディレクトリから開始すると仮定します。

```
cd node_modules/ewdjs
node install
```

インストールスクリプトは、ファイルを移動するためのベースラインを提供するパスの確認を入力するように要求します。　通常、提示されたデフォルトのパスを受け入れることをお勧めします。EWD.js ホームディレクトリの下に次のディレクトリパス構造が作成されていることがわかります。

```
- node_modules
    ◦ ewdjs (directory containing the ewd.js module)
    ◦ a number of Node.js module files
- www
    ◦ ewdjs (a directory containing the components of EWD.js that run
in the
      browser)
    ◦ 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)
     ◦ respond (contains files used for IE support of EWD.js / Bootstrap
applications)
  - ssl (This directory is where you should copy your SSL key and
certificate file
     if required. A self-signed certificate is installed for you to use
in testing)
     ◦ ssl.key
     ◦ ssl.crt
  - ewdStart-cache-win.js
  - ewdStart-globals.js
  - ewdStart-globals-win.js
  - ewdStart-gtm.js
  - ewdStart-mongo.js
  - ewdStart-pi.
  - test-gtm.js
  - test-gtm.js
```

EWD.js ディレクトリ構造の使い方

次のように EWD.js が作成したディレクトリ構造（上記のセクションを参照）を使用する必要があります。

～/ ewdjs（EWD.js ホームディレクトリ）：EWD.js スタートアップファイルはここに配置する必要があります。　追加の node.js のモジュールをインストールする必要がある場合は、このディレクトリ内から npm を実行する必要があります。

~/ewdjs/node_modules：EWD.js アプリケーション用のすべてのバックエンド· モジュールが存在する必要がある場所です。　そこには EWD.js 管理/モニタアプリケーションのバックエンドロジックを提供する ewdMonitor.js のようなサンプルがいくつかすでに存在します。　あなたのアプリケーションで必要とされている他の Node.js のモジュールもこのディレクトリにインストールする必要があります（上記参照）。

~/ewdjs/ssl：SSL キーと証明書ファイル用にこのディレクトリを使用します（必要な場合）。EWD.js では、SSL で EWD.js をテストするのに使用できる自己署名証明書をインストールしています。本番環境で使用するには適切なもので、これらのファイルを置き換えてください。

~/ewdjs/www：このディレクトリは、EWD.js.の Web サーバーのルート･パスとして機能します。このディレクトリ内のすべてのファイル、パスおよびサブパスはブラウザやリモートクライアントからアクセスできるようになります。このディレクトリ内に、アプリケーションで使用する任意の JavaScript ライブラリをインストールする必要があります。　例えば、
~/ewdjs/www/bootstrap-3.0.0 など

~/ewdjs/www/ewdjs: このディレクトリには、ブラウザで EWD.js ランタイム環境を構成する JavaScript ファイルが含まれています。　EWD.js インストールに含まれているこれらのファイルを変更しないでください。

~/ewdjs/www/ewd: すべての EWD.js アプリケーションのサブディレクトリは、このディレクトリに存在します。　インストール時に組み込まれたいくつかのアプリケーションのディレクトリ（例えば ewdMonitor、VistADemo）は、すでに作成済みです。　さらに必要な多数のアプリケーションのサブディレクトリを追加することができます。　各アプリケーションのサブディレクトリには以下の物が含まれています。

- メインの "コンテナ" HTML ファイル。　通常は index.html という名前を付けます。
- 必要に応じて 1 つ以上の「fragment」HTML ファイル。
- アプリケーションのブラウザ側の動的な機能を定義する JavaScript ファイル。　通常、app.js という名前を付けます。
- アプリケーションが必要とする他の JavaScript ファイルおよび CSS ファイル（オプション）。

~/ewdjs/www/respond：このディレクトリは、インストール時に作成され、Internet Explorer ブラウザ上の Bootstrap をサポートするために使用されている JavaScript ファイルが含まれています。

データベースインタフェースのインストールと構成

GlobalsDB

GlobalsDB をインストールした場合、インストールの/ bin ディレクトリ内にインターフェースモジュールファイルを見つけることができます。 cache0100.node という名前が付けられています。 ~/ewdjs/node_modules ディレクトリにこのファイルをコピーして cache.node に名前を変更します。（最重要）

Caché

使用している Caché のバージョンによっては、/Cache/bin ディレクトリに cache0100.node ファイルを見つけることができない場合があります。　cache0100.node が存在する場合は、

~/ewdjs/node_modules ディレクトリにコピーして cache.node に名前を変更します（最重要）。

cache0100.node が存在しない場合の最も簡単な方法は、どこか重要でない場所、例えば仮想マシン上で、GlobalsDB のコピー（Caché システムと同じプラットフォーム用である事を確認してください）をインストールして下さい。 /globalsdb/ bin ディレクトリの cache0100.node ファイルを検索します。~/ewdjs/node_modules ディレクトリにこのファイルをコピーして、cache.node に名前を変更します（最重要）。 GlobalsDB からコピーした cache0100.node ファイルは、Caché の初期のバージョンでも動作するはずです。

GT.M

GT.M の自動インストーラを使用ない場合は、NodeM モジュールをインストールする必要があります。

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

NodeM を設定する必要があります。 もし GT.M の自動インストーラを使用したのであれば、設定は終了しています。 また、第二のインストールスクリプト内を見る事により、何が必要なのか確認できます：https://github.com/robtweed/ewd.js/blob/master/gtm/install2.sh

要約すると、以下のような手順になります。

- ディレクトリ~/ewdjs/node_modules/nodem/lib に、多くのファイルがあります。これらは、Node.js の、32 ビットおよび 64 ビットバージョン用の Linux 用のインターフェースモジュールです。 32 ビット Linux を使用している場合は、mumps10.node.i686 を選択する必要があります。 64 ビット Linux を使用している場合は、mumps10.node.x8664 を選択する必要があります。 選択したファイル名を mumps.node に変更します。 mumps.node が~/ ewdjs/ node_modules/ nodem/ lib のパスに存在していることを確認してください。
- / usr/local/ lib ディレクトリに GT.M のインストールディレクトリにある libgtmshr.so ファイルへのシンボリック･リンクを張ります。

例：

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

- ~/ewdjs/node_modules/nodem/resources/calltab.ci のパスを指す GTMCI 環境変数が作成されていることを確認してください。
- gtmroutines 環境変数に~/ ewdjs/ node_modules/ nodem/ src のパスを含むように拡張されて

いることを確認してください。

注：後者の二つの工程は、あなたがご希望の場合モジュールファイル dewdrop-config.js を使用して実行時に行うことが出来ます。後頁を参照してください。

ewd.js の実行

ewd.js の起動

さあ、これから ewd.js Node.js の環境を起動しましょう。　使用している MUMPS データベースや OS などに適した ewdStart*.js ファイルを使用して、この操作を行います。

- Linux または Mac OS X 上での GlobalsDB: ewdStart-globals.js

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

- Windows 上の GlobalsDB: ewdStart-globals-win.js

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

- 付録２の方法でインストールした GT.M: ewdStart-gtm.js

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

- 付録３の方法、 dEWDrop VM でインストールした GT.M: ewdStart-gtm.js

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

- 適切にカスタマイズするための他のスタートアップファイルも提供しています。
  - ewdStart-cache-win.js: EWD.js + Windows + Caché for Windows
  - ewdStart-pi: EWD.js running in a Raspberry Pi (付録 5 参照)
  - ewdStart-mongo.js: EWD + Windows + MongoDB (MongoDB での Mumps globals のエミュレーション)

- ハイブリッド環境での MongoDB を使用したい場合は、付録 6 を参照してください。基本的に EWD.js 環境は、使用している MUMPS データベース/ OS にフォーカスを当てており、MongoDB は、作成した EWD.js アプリケーション用に追加することができます。

- 説明してきたものとは異なる構成を使用している場合は、適切な起動の JavaScript ファイルのコピーを作成し、それを適切に編集する必要があります。 Caché や GlobalsDB を使用している場合は、ewdStart-globals.js または ewdStart-globals-win を使用します。GT.M を使用している場合は、ewdStart-gtm.js を使用します。

EWD.js スタートアップパラメーターの定義

上記の EWD.js 起動ファイルのいずれかを見れば、それらがすべて共通のパターンに従っていることがわかると思います。

- require() を使って ewd.js モジュールはロードされます。
- 起動パラメータオブジェクトが定義されています。
- ewd.js start() メソッドは、引数としての起動パラメータオブジェクトを通じて呼び出されます。

EWD.js が（start() メソッドを呼び出すことにより）開始されたとき、最初の起動パラメータのデフォルトセットを作成します。 スタートアップパラメータオブジェクトで定義された任意の値は、デフォルト値を上書きするために使用されます。 ほとんどの状況では、EWD.js のデフォルトのパラメータ値の大部分を受け入れることができ、あなたは自分の環境に固有の値の僅かな数の指定をするだけです。 以下に示すように、デフォルト値をオーバーライドすることができます。

```
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'
    }
  },
  management: {
    path: '/ewdjsMgr',
    password: 'makeSureYouChangeThis!'
  }
```

たとえば、SSL を使用して、ポート 8081 リスニングで EWD.js を起動するには、次のオーバーライド起動パラメータ値を設定する必要があるでしょう。

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

他のすべてのスタートアップ値は、デフォルト値として残ります。

利用可能な起動パラメータの意味は以下のようになります。

- **database.type**: gtm | cache | mongodb
- **httpPort**: EWD.js web server がリッスンするポート番号
- **https.enabled**: true | false
- **https.keyPath**: SSL key ファイルの位置とファイル名
- **https.certificatePath**: SSL certificate ファイルの位置とファイル名
- **poolSize**: EWD.js の起動時に事前にフォークされる子プロセスの数。EWD.js が実行されている間は EWDMonitor アプリケーションで調整できます。
- **webServerRootPath**：Web サーバーのルートからの EWD.js によってマップされている物理パス。　このパスのすべてのファイルとサブディレクトリは、ブラウザや HTTP（S）クライアントからアクセス可能であることに注意してください。ほとんどの状況ではデフォル

ト設定を使用することをお勧めします

- **webSockets.externalListenerPort**：EWD.js が着信外部メッセージをリッスンする TCP ポート
- **logTo**：EWD.js のログ情報の宛先（コンソール|ファイル）
- **logFile**: logTo がファイルに設定されている場合、EWD.js によって書かれるログファイル名を定義します
- **traceLevel**: EWD.js のログレベル 0 = none, 1 = low, 2 = medium, 3 = verbose
- **modulePath**: EWD.js モジュールファイルのパス位置。　バックエンド· アプリケーション· モジュールの require()　のパスを構築するために使用される。　通常は、デフォルト設定を使用することを強くお勧めします
- **monitorInterval**: EWDMonitor アプリケーションが、情報更新の間隔のミリ秒数を指定するために使用します。
- **webService.json.path**：着信 Web サービス要求を認識するために EWD.js によって使用される URL パスのプレフィックス。
- **management.path**：着信 HTTP ベース EWD.js 管理要求を認識するために EWD.js によって使用される URL パスのプレフィックス
- **management.password**：EWDMonitor アプリケーションと HTTP ベースの管理要求のアクセスは、このパスワードを使って認証します。公的にアクセス可能な業務システムおよび EWD.js のシステムにはユニークパスワードが定義されていることを確認してください。デフォルトのパスワードは最初のテストの時だけの使用にして下さい。

EWD.js の実行

ewd.js モジュールを起動したときに、書き出される情報の量に驚かないで下さい。デフォルトのログレベルが（前のセクションの traceLevel パラメータを参照）3 の最大値に設定されているためです。　起動パラメータオブジェクト、または EWD.js の実行中に EWDMonitor アプリケーションを使用して、0（ロギングなし）から 3（最大）の間の、任意の整数値を再設定することができます。

EWD.js は設定ミスとなったすべての問題を認識するよう努め、可能な場合は適切なエラーメッセージを表示して停止します。この問題が発生した場合、EWD.js のログメッセージの中より様々な診断または助言メッセージを探し、適切に設定を修正して EWD.js.の再起動を試みてください。

EWD.js が起動しない主な理由には次のものがあります。

- アーキテクチャ対して間違った NodeM または cache.node インターフェイスファイルを使用している
- EWD.js を実行するための十分な権限を持たない場合（通常は Mike Clayton's のインストーラを使用して作成した GlobalsDB ベースのシステムに限定される）。
- 起動パラメータオブジェクトの中のパスの不一致

ewd.js の停止

ターミナルウィンドウで ewd.js を実行している場合、プロセスを停止するのに CRTL&C を使用することは避けてください。　あなたがバックグラウンドサービスとして ewd.js を実行している場合も同様に、サービスを停止しないでください。　子プロセスと結合している MUMPS データベースプロセスが回復不能な中途半端状態でハングしたままになるからです。このような方法であなたが ewd.js をシャットダウンした時には、実際、GT.M は通常は問題ありませんが、Caché と GlobalsDB プロセスは、ほとんどの場合、中途半端な状態のままになります。　この問題が発生した場合、Caché や GlobalsDB をクローズする唯一の方法は、force コマンドを使用することです。

ewd.js プロセスをシャットダウンする正しい方法は、次の 2 つの方法を使用することです。

- ewdMonitor アプリケーションを起動してログオンして（次の章を参照）、マスター· プロセス· グリッドの上に表示される Node.js のプロセス停止ボタンをクリックします。

- ewd.js プロセスへ以下の構造の HTTP（S）要求を、送信します。

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

  ewd.js インスタンスに応じた適切な値で、角括弧内のアイテムを書き換えてください。管理パスワードは ewd.js 起動ファイルで 1 つだけ定義されます（例えば ewdStart* js ファイル）（デフォルトでは keepThisSecret！に設定されています。別のパスワードに変更することを強く推奨します。）

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

これら 2 つのの方法は、同じ効果を持ちます。ewd.js マスタープロセスは、接続された子プロセスのそれぞれに MUMPS データベースをクリーンに閉じるように指示します。　すべての子プロセスが終了し次第、マスター· プロセスが終了します。

ewdMonitor アプリケーション

ewdMonitor アプリケーション

ewd.js インストールキットには、ewdMonitor という既製の EWD.js アプリケーションが含まれています。このアプリケーションには、2 つの目的があります。

- 高度なブートストラップ 3 ベースの EWD.js アプリケーションの例を提供する。
- ewd.js モジュールを監視・管理する機能と、EWD.js 環境や MUMPS データベースをさまざまな角度から調べることができるアプリケーションを提供する。

次の URL を使用して、ブラウザでアプリケーションを起動します。

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

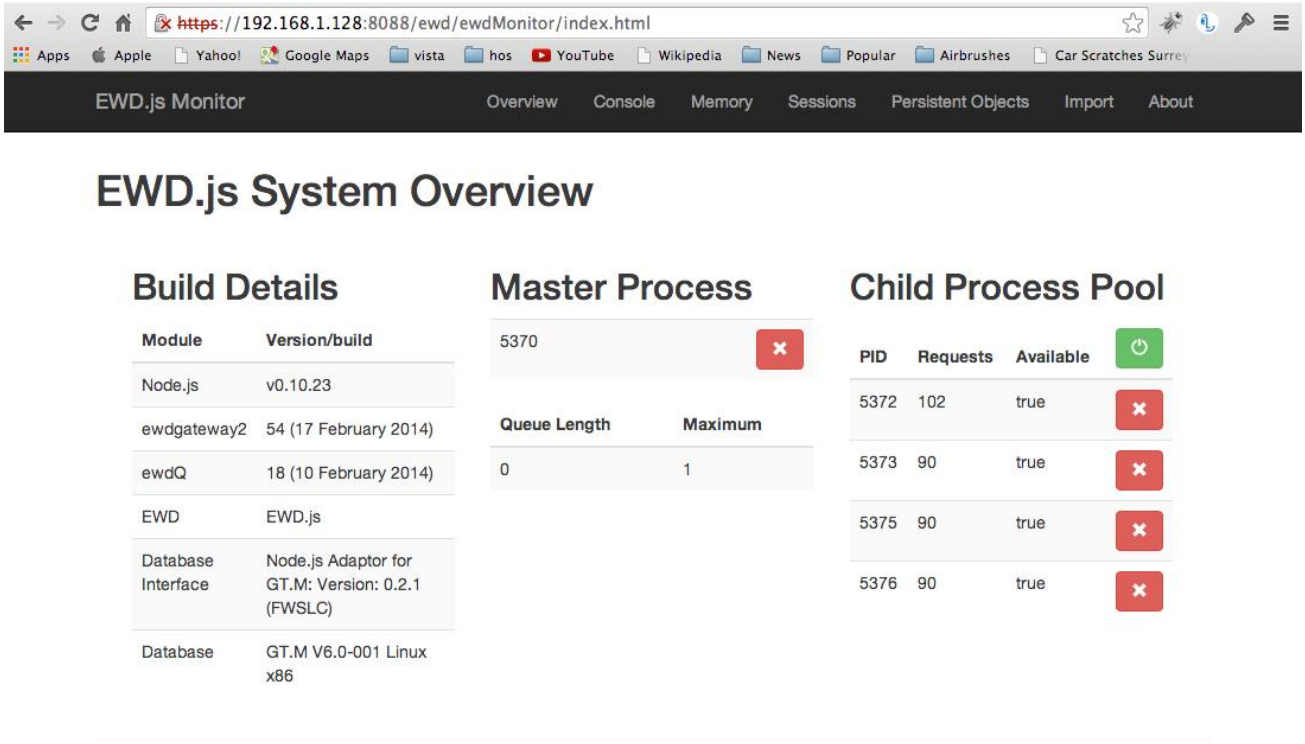
IP アドレスまたはホスト名とポートは適切に変更してください。

SSL を指定するよう EWD.js の起動パラメータを変更した場合には、https:// で始まるように上記の URL を変更する必要があります。そして、おそらく EWD.js Web サーバが使用した自己認証の SSL キーに関する警告を受け取るでしょうが、OK を押して続行します。

パスワードの入力を求められます。　パスワードは ewd.js 起動ファイルで指定されたものを使用します。以下を参考にしてください。

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

このパスワードを入力して、ログインボタンを押すと次の画面が現れます。



トップ画面では、Node.js、ewd.js、およびデータベース環境に関する基本情報が表示されます。マスターNode.js のプロセスと子プロセスのそれぞれによって処理された要求の数に関する統計が含まれます。

重要な表示は、マスター· プロセス· パネルの Node.js プロセスの Stop(停止)ボタンです。ewd.js プロセスと MUMPS データベースプロセスの接続をシャットダウンする際は常にこれを使用するようにしてください。

このアプリケーションによって提供されるその他の機能は次のとおりです：

- プロセス Id の上にマウスを移動すると、プロセスの現在のメモリ使用率を表示するパネルが表示されます。子プロセスでは、現在ロードされているアプリケーションモジュールのリストが表示されます
- ライブコンソールログ： ewd.js を実行している場合、有用なサービスです。
- ewd.js マスターと子プロセスのメモリ使用量を示すライブチャート：メモリリークをチェックするのに有用
- 現在アクティブな EWD.js セッションを示す表。各セッションの内容を表示し、セッションを終了させることができます。
- MUMPS データベースストレージの内容を表示、探索し、必要に応じて削除するためのツリーメニュー
- MUMPS データベースにデータをインポートするためのインポートオプション

- ログ出力とトレースのレベルを変更し、出力先をライブコンソールとテキストファイルの間で切り替えるためのオプション
- EWD.js ベースの Web サービスのアクセス権やセキュリティキーを管理するためのフォーム。

ある程度の時間をかけて、このアプリケーションを解析してください。これらが ewd.js と EWD.js 環境を作るための有用な手掛かりであるということが理解出来るでしょう。

EWD.js アプリケーションの作成

EWD.js アプリケーションの構造

EWD.js アプリケーションは、ブラウザ、アプリケーションのバックエンドロジックとの間の唯一のコミュニケーションの手段として WebSocket メッセージを使用します。基本的に EWD.js アプリケーションの可動部分は、JavaScript ファイルのペアのみで、一つはブラウザに、もう一つは Node.js モジュールにあります。それらはお互いに WebSocket のメッセージを送信し、受信して対応するメッセージを処理します。バックエンド Node.js のモジュールは、MongoDB またはあなたの選んだ MUMPS データベースへのアクセスをします。後者は、永続的な JavaScript のオブジェクトの集合として抽象化されます。

EWD.js アプリケーションは、いくつかのキーパーツで構成されています。

- 静的な HTML ページ：基本的なコンテナとメインの UI を提供します。
- 一つまたは複数のフラグメントファイル（オプション）：メインコンテナページに注入することができる静的な HTML マークアップのファイル。
- 通常 app.js という名前の、静的 JavaScript ファイル：以下を定義します。
  - ユーザーインターフェースより起動する送信の WebSocket メッセージ。
  - EWD.js アプリケーションのバックエンドからの着信を扱う WebSocket メッセージのハンドラ。
- バックエンド Node.js のモジュール： EWD.js アプリケーションを使用してブラウザからの着信の WebSocket メッセージを受けるためのハンドラを定義します。このモジュールファイルの名前は、通常、EWD.js アプリケーション名と同じです。

EWD.js は適切に名前の付けられたファイルが、インストールステップ中に作成されたディレクトリ･パスに適切に配置されていることを想定しています。例えば、myDemo という EWD.js アプリケーションを作成していた場合は、上記の構成要素の名前と配置は以下の様にします。

```
- 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 ]
```

EWD.js の仕組みと EWD.js アプリケーションを作成する方法を完全に理解するには、資料 4 の simple Hello World アプリケーションのチュートリアルを読んでください。

HTML コンテナーページ

すべての EWD.js アプリケーションは、メインの HTML コンテナページを必要とします。このファイルはインストール時に作成された EWD.js ホームディレクトリの下の www/ewd ディレクトリのサブディレクトリに存在しなければなりません。ディレクトリ名は EWD.js アプリケーション名と同じでなければなりません。上記の例では、myDemo です。

HTML ページでは、好きな名前を付けることができますが、慣例では index.html にすべきです。

bootstrap3 アプリケーションの場合、bootstrap3 アプリケーションフォルダで見つけるテンプレートページ（index.html）を使用する必要があります

詳細は付録 4 を参照してください。

app.js ファイル

EWD.js アプリケーションの動的挙動は、JSON 形式のコンテンツで作成され、WebSocket メッセージを経由してブラウザに配信され、そしてブラウザ側のメッセージハンドラが、その JSON 形式のコンテンツを使用して、適切な UI を変更します。

これが app.js ファイルの役割と目的です。app.js ファイルは通常の index.html ファイルと同じディレクトリにあります。

ブラウザ側の WebSocket コントローラ JavaScript ファイルは、実際には好きな名前を付けることができますが、通常の慣例では app.js.とすべきです。

bootstrap3 で使用するためのテンプレート用の app.js ファイルは bootstrap3 アプリケーションフォルダ内にあります。
また https://github.com/robtweed/ewd.js/blob/master/www/ewd/bootstrap3/app.js からも取得できます。

app.js の構成を示します。以下に示す構造を使用することを推奨します。詳細については、 付録 4 を参照してください。

```
EWD.sockets.log = true; // *** テスト終了後は false  に設定してください / 開発用
EWD.application = {
   name: 'bootstrap3', // **** 使用するアプリケーション名に変更してください
             //node_modules のファイル名（xxx.js)と一致していないと動作しません
   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: {
    // フラグメントファイルがブラウザにロードされた後、起動するハンドラを追加する
    '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: {
    // バックエンドから JSON WebSocket を受信した後鼓動するハンドラ、
    // 例：type == loggedIn のメッセージをハンドリングする場合
    loggedIn: function(messageObj) {
        toastr.options.target = 'body';
        $('#main_Container').show();
        $('#mainPageTitle').text('Welcome to VistA, ' +
messageObj.message.name);
    }
```

EWD.js WebSocket メッセージは必須の型を持ちます。EWD.js は、事前に定義したいくつかの予約

済み型名を提供し、そして またそれらに関連する事前に決められたプロパティを持ちます。しかし、main では、メッセージのタイプ名およびコンテンツ構造の決定は開発者に任されています。
WebSocket メッセージの payload(搬送データーの集合)は、JSON コンテンツとして記述します。
ブラウザから EWD.js の WebSocket のメッセージを送信するには、EWD.sockets.sendMessage API を使用します。

```
EWD.sockets.sendMessage({
    type: messageType,
    params: {
    //JSON payload（搬送データの集合）: 単純にあるいは複雑にお好きなように
    }
});
例:
EWD.sockets.sendMessage({
    type: 'myMessage',
    params: {
        name: 'Rob',
        gender: 'male'
        address: {
            town: 'Reigate',
            country: 'UK'
        }
    }
});
```

Payload(搬送データーの集合)は Params プロパティに配置する必要があることに注意してください。Params プロパティ内部のコンテンツのみがバックエンドのコードが実行される子プロセスへ搬送されます。

バックエンド Node.js モジュール

EWD.js アプリケーションの最後の部分は、バックエンドの Node.js モジュールです。

このファイルは、EWD.js のインストール手順中に作成されたホームディレクトリ(例 ~/ewdjs/node_modules)下の node_modules ディレクトリに存在する必要があります。モジュールのファイル名は、EWD.js アプリケーション名と同じでなければなりません。 たとえば、ewdMonitor というアプリケーションは ewdMonitor.js という名前のバックエンドモジュールファイルを持つことになります。

以下にバックエンド EWD.js モジュールの構成部品を示します。この構造に準拠することを推奨します。

```
//外部からアクセスされる全てのものは module.exports オブジェクトの内側に存在しなければなりません。
module.exports = {
    //ユーザーのブラウザからの着信ソケットメッセージは onMessage() 関数によって処理されます。
    onMessage: {
    //各着信メッセージタイプのハンドラを作成してください。
    // メッセージタイプの type ハンドラが    === getPatientsByPrefix の場合:
        getPatientsByPrefix: function(params, ewd) {
        // optional: 便宜のために、私は通常 EWD オブジェクトの要素の部分を抜き出します。
        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;
    //ユーザーのブラウザに WebSocket メッセージのレスポンスを返します。
    //メッセージは、同じタイプで返されます。この例では'getPatientsByPrefix。
    //応答メッセージである JSON payload(搬送データーの集合)は "message"プロパティの中に存在します。
    //従って、このメッセージのためのブラウザハンドラは maches を抽出します。
    //この例では： messageObj.message にアクセスします
      }
   }
}
```

フォームハンドリング

EWD.js は、フォーム処理のためのブラウザ側の特別な built-in function （EWD.sockets.submitForm）を持っています。もし ExtJS を使用している場合は、EWD.js はいくつかの追加の自動化を提供しますが、追加の自動化は他のフレームワークでも使用することができます。

ExtJS とこのフォームハンドリング機能を使用した場合は、自動的に ExtJS フォームコンポーネント（xtype：'form'）内のすべてのフォームフィールドの値を収集し、payload(搬送データーの集合)をメッセージとして、EWD.form のプレフィックスを付けた形で送信します。

シンタックスは以下の様になります。

```
EWD.sockets.submitForm({
   id: 'myForm', // ExtJS のフォーム･コンポーネントの ID
   alertTitle: 'An error occurred',
   // optional:バックエンドのフォーム検証よりエラーメッセージを表示する
Ext.msg.Alert パネル用の heading
   messageType: 'EWD.form.myForm'
   // バックエンドへ、このフォームの内容を送信するために使用されるメッセージタイプ。
   // フォームフィールドの値は自動的にメッセージのペイロード･オブジェクト'params'にパッケージ化されます。
   // フォームフィールド 'name' プロパティは、payload の中で使用されます。
});
```

他のフレームワークを使用している場合は、フィールドオブジェクトに定義されたフィールド値に変換する必要があります。例えば Bootstrap を使用している場合のフォームサブミットボタンハンドラは次のようになります。

注 ： toastr ウィジェットを使うことで、様々なエラーメッセージの表示を扱うことができます。

```
$('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'
```

```
                }
        });
});
```

バックエンド Node.js のモジュールは、指定されたメッセージタイプに対応するハンドラを持つことになります。 たとえば、EWD.form.login メッセージは、フィールド値にユーザ名とパスワードを持つ name プロパティを持ちます。

以下のよう書きます。

```
'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 '';
}
```

ユーザー名とパスワードフィールドの値が params を引数としてあなたのハンドラ関数に渡されることがわかります。 これには、ブラウザから送信された受信メッセージからの params の payload(搬送データーの集合)が含まれています。

EWD.js におけるフォーム処理は非常に単純です。あなたが決定した各エラー条件は、あなたが使用したいと考えるエラーメッセージのテキストの文字列を返し、ブラウザに表示されます。 EWD.form.xxx メッセージハンドラが、空文字以外の文字列を返す場合 、そして、ブラウザで実行中のアプリケーションが EWD.js/ブートストラップ 3 framwork を使用して構築されている場合、エラーメッセージが自動的にユーザブラウザの toastr ウィジェット内に表示されます。自作した HTML と JavaScript を使用している場合は、エラーは、JavaScript 警告ウィンドウとして表示されます。

しかし、フォームハンドラ関数は空文字列を返した場合、EWD.js は、ペイロードと同じタイプのメッセージ’OK:true’ （例えば前出の EWD.form.login の例では）を返します。

そのため、ブラウザ側 app.js ファイルは、タイプ EWD.form.login の着信メッセージのハンドラをログインに成功することに応答して UI を修正するために、含める必要があります。

ログインフォームを除去した例：

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

ユーザー認証コントロール

多くの EWD.js アプリケーションでは、その続行を許可する前にユーザーの認証を確立します。このような用途では、認証されていないユーザーが WebSocket のメッセージ起動を試みることができるバックドアを残していないことが重要です(例えば Chrome's Developer Tools console から)。このため、次の操作を行うことを確認してください。

1）最初のログインフォームを処理するバックエンドモジュールのロジックでは、正常に認証されたユーザーをマークするために特殊関数 ewd.session.setAuthenticated（）を使用します。デフォルトでは、ユーザーは非認証のフラグが付けられます。

2）すべてのバックエンドメッセージ処理関数は、ログイン認証機能から分離していて、ユーザーが認証されていることを確認していること。

```
getGlobals: function(params, ewd) {
      if (ewd.session.isAuthenticated) {
            //、認証されたユーザーだけのため記述
      }
},
```

注：ewd.session.isAuthenticated は、バックエンドにのみ存在するプロパティであり、開発者のロジックによって制御されている WebSocket メッセージを通してのみバックエンドでアクセスすることができます。

ewd オブジェクト

すでに、onMessage のオブジェクト内バックエンドメッセージハンドラ関数は二つの引数を持っていることを見てきました。params と ewd です。特に ewd オブジェクトの第二引数にはあなたのバックエンド･ロジックに有用であるいくつかのサブコンポーネントのオブジェクトが含まれます。

- **ewd.session**:ユーザーの EWD.js セッションへのポインタ、MUMPUS や MongoDB のデータベースに保持・保存されます。EWD.js は自動的にタイムアウトセッション毎にガベージコレクションを行います。 EWD.js セッションのデフォルトのタイムアウト時間は 1 時間です。
- **ewd.webSocketMessage**: 完了した着信の WebSocket メッセージオブジェクトへのポインタ。通

常は受信メッセージのペイロードが置かれ、params の引数を使用することになります。しかし、あなたが必要な場合には全体のメッセージオブジェクトが利用可能です。

- **ewd.sendWebSocketMsg()**:バックエンド Node.js モジュールからの WebSocket メッセージをユーザーのブラウザに送信するための関数。

この関数の構文は次のとおりです。

```
ewd.sendWebSocketMsg({
   type: messageType, // 送信のための宛先  message type
   message: payload // the message JSON payload
});
例:
ewd.sendWebSocketMsg({
   type: 'myMessage',
   message: {
      name: 'Rob',
      gender: 'male'
      address: {
         town: 'Reigate',
         country: 'UK'
      }
   }
});
```

- **ewd.mumps**: MUMPS データベースとレガシーMUMPS function にアクセスするためのポインタ（後者は Caché and GT.M のみ）。詳細は次の章を参照。

これは、あなたの Node.js のモジュール内から MUMPS データをどのように操作することができるかを示すデモアプリケーションの例です。

```
'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:アプリケーション作成で役立つ組み込み関数の集合
  - ewd.util.getSessid(token): セキュリティトークンとユニークな EWD.js セッション Id を返す
  - ewd.util.isTokenExpired(token): セッションがタイムアウトした場合に真を返す
  - ewd.util.sendMessageToAppUsers(paramsObject): 指定されたのすべてのアプリケーションのユーザーに WebSocket のメッセージを送信
    - type: the message type (string)
    - content: message payload (JSON object)
  - ewd.util.requireAndWatch(path): require()の代替。これは、指定されたモジュールをロードし、また、それを監視します。アプリケーションとモジュールの開発に有用です。

Web ブラウザからの EWD.js アプリケーションの開始

EWD.js アプリケーションは、フォームの URL を使用して開始します。

http[s]://[ip address/domain name]:[port]/ewd/[application name]/index.html
例：http://192.168.30.51:8080/ewd/ewdMonitor/index.html

- http:// 又は https://のどちらかを使用するかを機決定します。ewd.js 起動ファイルに HTTPS を有効にする設定があります。
- ewd.js を実行しているサーバーの IP アドレスかドメイン名を指定します
- ewd.js 起動ファイルで定義されたポートに対応するポートを指定します
- アプリケーション名の指定： index.html ファイルと app.js ファイルのあるディレクトリ･パス名が使用するアプリケーション名前です。大文字と小文字を区別します。

まとめ: Building an EWD.js アプリケーションの構築

付録 4 は、最初に、非常に基本的な HTML と手作りの JavaScript を使用した EWD.js アプリケーションを構築するプロセスを紹介します。次に同じデモをおよび EWD.js/Bootstrap3 フレームワークを

使用して示します。

EWD.js の応用例を参照するには、EWD.js のインストールに含まれている ewdMonitor アプリケーションを見てください。

オープンソース· VistA Electronic Healthcare Record（EHR）と EWD.js アプリケーションの統合構築例は、EWD.js のインストールに含まれている VistADemo アプリケーションを参照してください。

これらのアプリケーションのソースコードは GitHub のリポジトリ ewd.js 内にあります
https://github.com/robtweed/ewd.js

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

外部生成メッセージ

背景

ここまでは、バックエンドの MUMPS データベースとブラウザ UI を統合するための手段としての WebSocket メッセージを見てきました。しかしながら、EWD.js は外部プロセスの WebSocket メッセージを EWD.js アプリケーションの 1 つまたは複数のユーザーに送信することもできます。これらのメッセージは必要に応じて、単純な信号のみでも、複数のユーザーの複雑な JSON ペイロードを運ぶ手段にすることもできます。

これら外部メッセージは、他の Mumps または Cache プロセスで生成することができます。 または、実際には、システムのセキュリティの構成方法次第で、同じマシンまたは同じネットワーク上の他のシステムのあらゆるプロセスから生成できます。

外部メッセージ入力インタ フェース

Ewd.js モジュールには、自動的にアクティブ化され、ポート 10000（構成既定では） でリッスンする TCP ソケット サーバーのインターフェイスが含まれます。

このポートは、スタートアップ ファイル (ie　ewdStart*.js) 内のスタートアップ パラメーターオブジェクト によって変更できます。たとえば、webSockets.externalListenerPort の定義を追加することによってこれを行います。webSockets.externalListenerPort の定義を追加することによって変更を行います。

例を示します。

```
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);
```

外部プロセスは、シンプルにリスナーのポート(ie 10000、別のポートに再設定した場合を除いて)を開き、JSON 形式の文字列を書き込み、リスナー ポートを閉じるだけです。あとは EWD.js が行います。

外部生成メッセージの設定とルーティング

あなたは、以下の送信先にメッセージを送ることができます。

- すべてのアクティブな EWD.js ユーザー
- 現在アクティブな特定の EWD.js アプリケーション
- セッションの名前と、値のリストがマッチした全ての EWD.js ユーザー

メッセージの type とメッセージペイロードを指定する必要があります（EWD.js アプリケーション内の Web ソケットメッセージの場合と同じように）。セキュリティ上の理由から、すべての外部から注入するメッセージには、パスワードが含まれている必要があります、これは ewdStart*.js ファイルで指定されたものと一致する必要があります。

TCP リスナーポートへの書き込んだ JSON 文字列が、どの宛先にメッセージを送るかを決定します。JSON 形式の構造及びプロパティは、各宛先カテゴリによって若干異なります。

全ての EWD.js ユーザへの送出メッセージ

```
 {
   "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
   }
 }
```

すべての現在アクティブな EWD.js ユーザは、自分のブラウザに送信された上記のメッセージを持つことになります。

特定の EWD.js アプリケーションを使用している ユーザへの送出メッセージ

```
{
  "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
  }
}
```

myApp という名前の EWD.js アプリケーションのすべての現在アクティブなユーザは、ブラウザに送信された上記のメッセージを持つことになります。

特定の EWD.js セッションコンテンツとマッチした全てのユーザへのメッセージ

```
 {
   "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
   }
 }
```

現在アクティブな EWD.js ユーザーの rob は、自分のブラウザに送信された上記のメッセージを持

つことになります。もっと具体的かつ正確にいうと、メッセージは、usename という変数名で rob という値を持つ変数が含まれている EWD.js セッションの、すべてのユーザに送信されます。

あなたは、セッション名/値のペアを好きなだけ配列で指定することができます。ユーザのセッションでは、名前/値のペアがすべて一致した場合にのみ、メッセージが送信されます。

外部生成メッセージのハンドリング

外部で生成されたメッセージは、関連する利用者のブラウザに送信されます。

外部から注入されたメッセージを EWD.js アプリケーションで処理するために、アプリケーションごとに、ブラウザ側の JavaScript での受信メッセージ· タイプに適切なハンドラを（app.js ファイル内などに）含める必要があります。受信メッセージタイプに対してハンドラが存在しない場合は無視されます。

外部で生成されたメッセージのハンドラは通常の EWD.js の WebSocket メッセージと差異はありません。

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

着信した外部で生成されたメッセージを処理するために、バックエンドで何かをする必要がある場合は、単純に、外部で生成されたメッセージ· ペイロードの一部またはすべてと一緒にハンドラ内からバックエンドへ WebSocket メッセージを送ります：

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

```
  // ...etc
};
```

GT.M と Caché のプロセスからの外部メッセージ送信

外部の GT.M または Caché のプロセスからメッセージを送信する場合は、EWD.js リポジトリに在る MUMPS ルーチン ewdjsUtils によって提供される、組み込みメソッド（sendExternalMessage()）を利用することができます。

以下を参照してください。

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

GT.M を使用している場合、ルーチンファイルをコンパイルし、実行するためにマッピングされている適当なディレクトリにこのルーチンファイルをコピーします。Caché を使用している場合は、Caché スタジオでコードをコピー&ペーストして、ewdjsUtils という名前でコンパイルし、保存します。

ルーチンファイルの中に、sendExternalMessage()メソッドの使用例を見る事が出来ます。

```
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()は外部メッセージの 3 種類すべての演習です。　上記の MUMPS のコードは、JSON 形式の文字列を作成するかわりに、ローカル MUMPS 配列と同等の JSON の構造を構築することができます。　GT.M と Caché システム上で、EWD.js の TCP ポートを開き、閉じ、書き込みをするためのコードは、ルーチン· ファイルに含まれています。

外部の GT.M または Caché プロセスからのメッセージの送信をテストするために、以下のようなサンプルコードを呼び出すことができます。　ewd.js プロセスはデフォルトの TCP ポート 10000 を使用していると想定しています。

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

もちろん、必要に応じて 3 つのメッセージタイプのハンドラをブラウザ内に記述する必要があります。

独自の外部メッセージを作成するには externalMessageTest()プロシージャのコードを変更してください。

他の環境からの外部作成メッセージ送信

もちろん、GT.M または Caché プロセスに限定されるものではありません。EWD.js の TCP ソケットリスナーのポートを開くことができる任意のプロセスは、それへの JSON 形式のメッセージ文字列を記述して、関連する EWD.js ユーザにメッセージを送信することができます。実装の詳細は、プロセス内で使用している言語によって異なります。

JavaScript から MUMPS データへのアクセス

背景：MUMPS データベース

MUMPS データベースはデータをスキーマフリーの階層形式で格納します。各々の記憶ユニットは MUMPS の技術用語ではグローバル（グローバルスコープ変数の略）として知られています。グローバルを近代的な意味で考えると、永続的連想配列記憶ユニットと考えられます。いくつかの例を示します。

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

各々の永続的連想配列には名前があります（上記の例では、myTable、account）。次に数値またはテキスト文字列の添字がいくつか続きます。添字の数に制限はありません。

「ノード」（ノードは配列名と添え字の組み合わせによって定義されます）は、テキスト文字列であるデータ値を格納します（空の文字列が許可されています）。いつでもノードを作成し破棄することができます。それらは完全に動的であり、事前に宣言またはスキーマを必要としません。

MUMPS のデータベースは組み込みのスキーマまたはデータ･ディクショナリを持ちません。それは、より高いレベルの抽象化を設計するために、開発者に任されており、データベースは物理的に永続的配列の集合として格納されます

MUMPS データベースには、組み込みのインデックス作成がありません。インデックスは、MUMPS データベースに、効果的かつ効率的な問合せと、横断的なデータ検索を可能にするためのキーとなりますが、メインデータ配列に関連付けられたインデックスの作成、メンテは、開発者に任されています。インデックスはそれ自体が MUMPS の永続的配列です。

パワフルな NoSQL エンジンとしての MUPS データベース技術とその重要性について、もっと多くの背景を知るには、A Universal NoSQL Engine, Using a Tried and Tested Technology（http://www.mgateway.com/docs/universalNoSQL.pdf）を参照してください。

MUMPS 配列の EWD.js へのプロジェクション

ewd.js 配布物に含まれる ewdGlobals.js という名前の JavaScript ファイルがあります。ewdGlobals.js は、GlobalsDB、GT.M そして Cache の Node.js インタフェースによって提供される低レベル API の、最上位に作成された抽象化レイヤとして、EWD.js によって使用されます。ewdGlobals.js は JavaScript オブジェクトの永続的集合として MUMPS データベースに永続的な連想配

列の集合を投影します。

MUMPS 永続配列から JavaScript オブジェクトへのマッピング

ewdGlobals.js による投影の背後にある理論は、実に簡単です。　そして、その説明には例示するのが一番です。

オブジェクトとして表現したい患者レコードのセットがあるとします。　特定の実際の患者を表すトップレベル･オブジェクトを patient という名前で定義することができます。　通常、特定の患者を識別する何らかの患者識別子キーがあります。　この例での患者識別子は、単純な整数値「123456」であるとしましょう。

JavaScript のオブジェクトの表記法によると、プロパティにより順番に、さらにサブプロパティを使って入れ子にしたプロパティによって、患者の情報を表現することができます。

例を示します。

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

この患者オブジェクトは MUMPS データベースの永続配列として次のように表すことができます。

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

オブジェクトのプロパティと MUMPS 永続配列で使う添え字の間で、直接 1 対 1 の対応があることを確認することができます。　反対に、あらゆる MUMPS の永続配列は対応する JavaScript オブジェクト階層構造のプロパティで表すことができます。

通常、データが MUMPS タベースに格納されている場合、格納単位としてグローバル･ノードを参照します。　グローバル・ノードは（この場合 patient）永続配列の名前といくつかの特定の添字の組み合わせです。　グローバル・ノードは効果的に、任意の個数（ゼロを含む）の添え字を有することができます。　データは、数字または英数字形式の値で、各グローバル・ノードのリーフに

格納されます。

添字の各レベルは、個別のグローバル· ノードを表します。　郵便番号を例にとり、以下にグローバル· ノードを表してみます。

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

上の例では、データは最下位レベル（またはリーフ）のグローバル· ノードに格納されている事に注意してください。　他のすべてのグローバル· ノードは、中間ノードとして存在おり、それらは下位レベルの添え字を持っていますが、いかなるデータも持ちません。

添え字「address」と「zipcode」が、実際のグローバル· ノードのポインタである事以外、この特定の永続的な配列で使用されていることを私たちに教えてくれるものは、MUMPS データベース内に何もありません。 参照できるデータ· ディクショナリやスキーマは組み込まれていません。 逆に、この永続配列にデータを追加したい場合は、任意に好きな添え字を用いて、それを追加することができます。 それでは Country のレコードを追加してみましょう。

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

患者の体重を追加してみましょう。

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

どんな添え字でも好きなように使用できたことに注意してください。これらの特定の添え字を使用するための制限は何もありません。（アプリケーション内で気が付くと思いますが、すべてのレコードは、一貫して同じ添字方式を使用します）。

GlobalNode オブジェクト

EWD.js が提供する ewdGlobals.js の投影で、GlobalNode オブジェクトをインスタンス化することができます。例を示します。

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

GlobalNode オブジェクトは MUMPS データベースの物理的なグローバル· ノードを表すとともに、JavaScript を使って操作や参照ができる一連のプロパティとメソッドを利用可能にします。

GlobalNode オブジェクトの重要な特徴は、最初にインスタンス化されたときには、実際には物理的に存在することも存在しないこともあり、後でユーザーの EWD.js セッション内でその存在を変更することがあります。

GlobalNode オブジェクトの重要なプロパティは_value です。これは、read/write プロパティで、MUMPS データベース内の物理的なグローバル· ノードの値を参照したり設定するのに使用します。以下に例を示します。

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

あなたが_value プロパティにアクセスする時は、MUMPS データベースのディスク上にある、物理的なグローバル· ノードの値にアクセスしています。もちろん、（この場合には zipNode という名の）JavaScript オブジェクトを介してのアクセスですが。上記の例では、グローバル· ノードへのポインタを設定している最初の行は、実際に MUMPS データベースにはアクセスしないことに注意してください。　実際、ポインタが作成されたときには、物理的な MUMPS グローバルノードがデータベースに存在しないことがあります。　GlobalNode オブジェクトのメソッドが使われるのは、GlobalNode オブジェクトと物理な MUMPS グローバル· ノードの間の物理的な連結が行われた、データベースへの物理的アクセスが必要な時に限られます。

もちろん、理想的には次のことができるようにしたいと思います。

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

しかし、問題があります。　MUMPS 永続配列には動的なスキーマフリーな性質があるので、物理 MUMPS グローバル· ノードが、実際に name の添字を持つ patient の GlobalNode オブジェクトを事前に知る方法は存在しないことを意味し、そして patient.name に対応するプロパティをインスタンス化することが可能かどうかを事前に知る方法はありません。　理論的には、ewdGlobals.js はプロパティとして物理的に指定された MUMPS グローバル· ノードの下に存在するすべての添字をインスタンス化できます。しかしながらグローバル· ノードの添え字の数は数千または数万かも知れません。　それを見つけ、プロパティとしてすべての添字をインスタンス化するためには膨大な時間と膨大な処理能力が必要となり、そして Javascript で処理するためにはメモリを大量に消費することになります。　さらに、典型的な EWD.js アプリケーションでは、オブジェクトのアクセスの際、一度には少数の GlobalNode プロパティにアクセスを必要とするだけです。　すべてをインスタンス化した後、1 つまたは 2 つのインスタンスだけを使用するのは非常に時間の浪費です。

これに対処するために、 GlobalNode オブジェクトには_getProperty()、通常は$()と略す特別なメソッドがあります（jQuery の本を参照してください）。$()メソッドは以下の 2 つの事を行います。

- 親 GlobalNode オブジェクトのプロパティとして指定された添え字名をインスタンス化する。
- 下位の添え字の物理グローバル· ノードを表す別の GlobalNode オブジェクトを返す。

例を示します。

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

このコードでは、 最初に MUMPS グローバル· ノード：

patient(123456)

をポイントする GlobalNode オブジェクトを作成します。

次の行は２つの事を行います。

- patient オブジェクトに name という名のプロパティを拡張する。
- 物理的なグローバル· ノード：

  ^patient(123456, "name")

  を指す新しい GlobalNode オブジェクトを返す。

$（）メソッドを使用したこれらの 2 つの効果は、ともに非常に興味深くて強力です。最初に、name 添字へのポインタとして、使用しました。 patient の実際のプロパティとして name が正しくインスタンス化されましたので、再度$（）メソッドを使用することなく、続けて name プロパティを直接参照することができます。 そこで、上記の例を拡張して、patient の name プロパティを直接参照することにより、患者の名前を変更することができます。

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

次に、$（）メソッドは（存在しても、存在しなくても、またはデータ値を持たなくても）GlobalNode オブジェクトを返すので、好きなように深い連鎖をすることができます。 このため、以下のように、患者の town を取得することができます：

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

連鎖した$()メソッドの呼び出しはそれぞれ、添字のレベルを表す、新しい GlobalNode オブジェクトを返します。 ここで、以下の物理 MUMPS のグローバル· ノードをオブジェクトとして定義してみます。

patient - 物理 MUMPS グローバル·ノード: patient(123456) を表す。
patient.address - patient(123456,"address") を表す。
patient.address.town - patient(123456,"address","town") を表す。

ここでは、再度$()使用することなく、これらのプロパティを使用することができます。 それに対して、たとえば、郵便番号を取得するためには、（まだそれにアクセスしていませんので）郵便番号プロパティの$（）メソッドを使用する必要があります。

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

しかし、患者の town を取得したいのであれば、すでにすべてのプロパティがインスタンス化されているので、$()メソッドを使用する必要はありません。以下に例を示します。

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

この様に EWD.js の ewdGlobals.js ファイルによる投影は MUMPS データベースが永続的な JavaScript のオブジェクトの集合であったかのようにデータを処理する手段を提供します。 ディスク上に格納されたデータを操作しているという事実の大部分は隠されます。

GlobalNode プロパティ とメソッド

GlobalNode オブジェクトには MUMPS グローバル· ノードを参照、操作するために使用できる多くのメソッドとプロパティがあります。

メソッド / プロパティ	説明
$()	現在の GlobalNode オブジェクトが表す MUMPS グローバル· ノードの添字の、サブノードを表す GlobalNode オブジェクトを返します。$（）メソッドは、添字の名前を新しい GlobalNode オブジェクトとしてインスタンス化されるように明示します。
_count()	現在の GlobalNode オブジェクトで表される MUMPS グローバル· ノードの下に存在する添字の数を返します。
_delete()	現在の GlobalNode· オブジェクトのすべてのデータが物理的に削除され、指定された GlobalNode 下の下位レベルの添字を持つ MUMPS グローバル· ノードを物理的に削除します。下位レベルの添え字のためにインスタンス化した可能性のある Javascript の GlobalNode オブジェクトが存在し続けますが、その物理的な値に関連して、それらのプロパティ（例え

	ば、その_value）が変更されていることに注意してください。
_exists	GlobalNode オブジェクトが物理的に MUMPS グローバル· ノードとして存在している場合、true になります。中間のグローバル·ノードまたはグローバル· ノード実体のいずれかとして存在する可能性があることに注意してください。
_first	GlobalNode オブジェクトで表される MUMPS グローバル· ノードの下の最初の添え字の名前を返します。
_forEach()	イテレータ関数、GlobalNode オブジェクトで表される MUMPS グローバル· ノードの下に存在するすべての添字を表すコールバック関数。
_forPrefix()	イテレータ関数、MUMPS グローバル· ノードの下に存在するすべての添え字のためのコールバック関数。指定された接頭辞で始まる 添え字名を表す GlobalNode オブジェクト。
_forRange()	イテレータ関数、MUMPS グローバル·ノードの下に存在するすべての添え字のためのコールバック関数。 指定された英数字の範囲内にある添え字文字列を返します。
_getDocument()	物理的に、現在の GlobalNode オブジェクトで表される MUMPS グローバル· ノードの下に存在する MUMPS グローバル· ノードのサブツリーを取得 し、JSON ドキュメントとして返します。
_hasProperties	GlobalNode オブジェクトで表される MUMPS グローバル· ノードにおいて、その下に 1 つ以上の既存の添え字（したがって GlobalNode オブジェクトの潜在的なプロパティ）がある場合は true。
_hasValue	GlobalNode オブジェクトが物理的に MUMPS グローバル· ノードとして存在し、それに対して保存された値を持っている場合は true。
_increment()	現在の GlobalNode オブジェクトの_value プロパティのインクリメントを行います。
_last	現在の GlobalNode オブジェクトが表す MUMPS グローバル· ノードの最後の添え字の名前を返します。
_next()	現在の GlobalNode オブジェクトが表す MUMPS グローバル· ノードの指定した次の添え字の名前を返します。
_parent	GlobalNode オブジェクトとして、現在の GlobalNode オブジェクトの親ノードを返します。
_previous()	現在の GlobalNode オブジェクトが表す MUMPS グローバル· ノードの指定された一つ前の添え字の名前を返します。
_setDocument()	現在の GlobalNode オブジェクトが表す MUMPS グローバル· ノード下の MUMPS グローバル· ノードのサブツリーとして、指定された JSON ドキュメントを保存します
_value	読み取り/書き込みプロパティ、現在の GlobalNode オブジェクトで表される MUMPS グローバル· ノードに物理的な値を取得または設定するために使用。

用例

MUMPS 永続配列に以下のデータが格納されているとします。

```
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 と 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(); // 添え字 conditions の下のすべてのノードが削除されます

// 永続配列 patient に残るすべては:

// 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
```

最初の引数として、{direction: 'reverse'} を加えることにより、繰り返しの方向を逆にすることができます。

_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
```

最初の引数を置き換えることにより、繰り返しの方向を逆にすることができます。

{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
```

最初の二つの引数を置き換えることにより、繰り返しの方向を逆にすることができます。

{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
  }
```

_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);
```

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

_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
```

EWD.js 中の ewd.mumps オブジェクトで提供するその他のファンクション

以上詳細に説明してきた GlobalNode()コンストラクターに加えて、ewd.mumps オブジェクトは EWD.js アプリケーション内で使用することができる追加機能をいくつか提供します。

ewd.mumps.function()

この function は GT.M と Caché データベースで使用可能ですが、GlobalsDB では使用できません。これは、GlobalsDB は単なる MUMPS データベースエンジンであり、MUMPS 言語プロセッサが含まれていないためです。

ewd.mumps.function() API を使用すると、MUMPS 言語で記述されているレガシーコードを呼び出すことができます。この API に注目してください 唯一、MUMPS 言語コマンドを外部関数として呼び出すことができます。もしプロシージャまたは Caché クラスを呼び出したい場合は、MUMPS 外部関数ラッパーでそれらをラップする必要があります。

ewd.mumps.function() API を使用するための構文は次のとおりです。

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

```
  ここで:
  label =外部関数ラベル
  routineName =関数を含む Mumps ルーチン名
  arg1..argn = 引数
  response =関数の戻り値を含む文字列値

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

  これは以下の Mumps コードに相当します。:
  set result=$$getPatientVitals^MyEHR(patientId,date)
```

Caché を使用した場合、この API の最大の戻り値文字列の長さが 4K であることに注意してください。関数が大量のデータ、たとえば大型の JSON ドキュメントを返すようになっている場合には、ユーザーの EWD.js セッションにデータを保存してから EWD.js JavaScript モジュール内のセッションからそれを回復することをお勧めします。

これを行うには、EWD.js セッション ID などを渡すために、関数でレガシーコードをラップする必要があります。以下に例を示します。

```
  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)
    }
  }
```

訳注：作成、変更した M ルーチンを EWD.js で有効化するためには、EWD.js の再起動が必要です。

ewd.mumps.deleteGlobal()

注意して使用してください。指定した名前の MUMPS 永続アレイ全体を削除します。MUMPS データベースには元に戻す機能を提供していないので、この機能は非常に危険であることに注意してください。　一つのコマンドであなたは即座に、永久に、データベース全体を削除することができます！

使用法

```
ewd.mumps.deleteGlobal('myGlobal');
```

その場で恒久的に myGlobal という MUMPS 永続配列を削除します。

ewd.mumps.getGlobalDirectory()

MUMPS データベース内のすべての永続的な配列の名前を含む配列を返します。

ewd.mumps.version()

使用されている Node.js の MUMPS インタフェースのバージョンと MUMPS データベースのバージョンと種類の両方を識別する文字列を返します。

MUMPS データのインデックス生成

MUMPS のインデックスについて

MUMPS 永続配列を索引付けすることは、高性能、柔軟で強力なデータベースを作成するための鍵となります。　伝統的に、多くの場合 MUMPS データベースのデータ更新を扱う API や関数の中に埋め込まれた、インデックスの維持は、開発者の役割でした。

MUMPS データベース内のインデックスは、単なる別の永続な配列です。　場合によっては、同じ名前の配列を使用しますが、異なる組の添え字がインデックスのために使用されます。　別の場合では、インデックスは異なった名前の永続配列に保持します。 選択はある程度任意ですが、個人的なスタイルやデータ管理上の問題により組み合わせることがあります。　例えば、メインデータとインデックスが単一の名前の永続配列に保持されている場合は、配列が非常に大きくなり、巨大なプロセスでのバックアップが必要になります。　インデックスは、1 つまたは複数の名前に分けた配列に保持されている場合はメインのデータ配列は小さくなり、バックアップが容易になります。またジャーナリングやマッピングアレイのようなメカニズムを使用している場合、別の名前の配列でインデックスしたほうが有利です。

簡単な例を見てみましょう。（患者に関する多くの他のデータの中で）患者の姓を格納する患者データベースについて考えてみましょう。

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

ユーザーが姓で患者を検索し、一致するリストの 1 つを選択することができるルックアップ機能を作成する場合に、患者全体の配列を横断的に、 id から id へ網羅的に姓のスミスを探すような検索はしたくありません。　その代わりに、姓のインデックスを格納する並列の永続配列を作ります。配列にどのような名前を使うか、どのような添え字を使うかは自由です。　ここに一つの方法を示します。

```
patientByLastName("Jones",123457)=""
....
patientByLastName("Smith",101430)=""
patientByLastName("Smith",123456)=""
patientByLastName("Smith",128694)=""
```

```
  patientByLastName("Smith",153094)=""
  patientByLastName("Smith",204123)=""
  patientByLastName("Smith",740847)=""
  ...
  patientByLastName("
```

このようなインデックス配列では、スミスの姓を持つすべての患者を見つけるために、_forEach（）メソッドを使用して、スミスさんのレコードを反復処理して、一致するすべての患者 Ids を取得します。　患者 Id は、ユーザが選択した患者記録にアクセスするためのポインタを提供します。

例：

```
  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]);
```

特定の接頭辞、たとえば Smi で始まるすべての名前を検索したい場合は、代わりに_forPrefix（）メソッドを使用します。　また、Sma と Sme の間のすべての名前を見つけたい場合には_forRange() メソッドが使えます。

インデックスへの改良として、インデックスで使用する前に、小文字に姓を変換することなどが考えられます。

例：

```
  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
```

これにより確実に、たとえば、ハイフン付きの姓の患者を見逃さないようにできます。　もちろん、姓インデックスはおそらく患者データベース用に操作したいもっと多くの内の一つです。　例えば、誕生日のインデックス、男女別インデックスが欲しいかもしれません。　我々はこれらのために、具体的な名前の配列を作成できますが、数が多くなると、これらの維持、管理が手に負えなくなる可能性があります。　別の方法として、1 つの永続的な配列にすべてのインデックスを保持することをお勧めします。　インデックスタイプを表す最初の添え字を追加することでこれを簡単に行うことができます。

例：

```
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)=""
...
patientInde
```

これは拡張可能なことが明らかです。　単純に新しい最初の添え字を使用して、新しいインデックス･タイプを追加することができます。

姓と性別などの 2 つ以上のパラメータで検索したいと思うこともあります。　もちろん、簡単なインデックスの組み合わせを使用するようにロジックを設計することができますが、マルチパラメータのインデックスを作成・更新することもできます。

例：

```
patientIndex("genderAndName","f","Smith",101430)=""
```

```
patientIndex("genderAndName","m","Thornton",123458)=""
...etc
```

MUMPS データベースの効果的なインデックス設計は経験から来ています。　うまくいけば、それが無限に柔軟であり、インデックスの設計は設計者のスキルとアプリケーション構築方法の理解に完全に依存していることがわかります。（アプリケーションはそれを駆動するデータを中心に構築される必要があります）

ヒント：インデックスには、メインデータ配列に由来するか、メインデータ配列内のデータから取り出せるデータや値だけを保持します。　こうしておけば、メインデータ配列からインデックスを再作成または再構築することができます。

EWD.js のインデックスの保守

MUMPS のインデックスは、残念ながら、自動的に作成されません。　開発者は、MUMPS のグローバルノードを作成、変更、削除する度に、インデックスを作成・変更しなければなりません。伝統的な MUMPS 開発者にとって、データベースの更新を処理するために設計されていた API セットがない限りこれはかなり面倒な作業です。　アプリケーション全体で一貫して必要なすべてのインデックスを作成しないと、誤ったインデックスレコードの結果、誤った検索をする可能性があります。

EWD.js は、開発者が処理できるイベントを発生する Node.js の能力を利用して、多くの巧みで効率的な事を実現します。　したがって、ewdGlobals.js モジュールは、_value、_delete()と _increment()の API を使用する毎にイベントを発行します。即ち、MUMPS データベース内のデータ値を変更するメソッドです。

発行されるイベントは、次のとおりです。

- **beforesave**: MUMPS グローバル･ノードに値が設定される直前に発行されます。イベントは変更される GlobalNode オブジェクトへのポインタを渡します。

- **aftersave**: MUMPS グローバル･ノードに値が設定された直後に発行されます。イベントは対応する GlobalNode オブジェクトへのポインタを渡します。　2 つの追加プロパティ - oldValue と newValue - が GlobalNode オブジェクトで使用可能です。　save イベントの前に何も値が存在しない場合、oldValue プロパティは NULL 文字列になります。　oldValue プロパティは aftersave イベントを介して利用可能になるため、ほとんどの状況で、インデックスを更新するために aftersave イベントを使用できます。

- **beforedelete**: MUMPS グローバルノードが削除される直前に発行されます。 イベントは削除しようとしている GlobalNode オブジェクトへのポインタを渡します。

- **afterdelete**: MUMPS グローバルノードが削除された直後に発行されます。 イベントは削除された GlobalNode オブジェクトへのポインタを渡します。 追加プロパティ – oldValue プロパティは、ノードオブジェクトで使用可能です。 多く場合、afterdelete イベントが使用可能です。しかし、_delete（）メソッドは、添字の下位レベルで削除された値は、それらの値がインデックスに使用された場合に考慮される必要があり、その場合には、中間レベルのグローバル· ノードに適用されている可能性があることに注意してください。 この場合、beforedelete イベントが、削除および変更されるインデックスに関するサブノードを横断して適切に対応できます。

globalIndexer モジュール

モジュール名 globalIndexer.js は ewd.js インストールキットに含まれ、EWD.js ホームディレクトリ下の node_modules ディレクトリにあります。 このモジュールは、イベント発生時に上記に示したイベントのハンドラを提供するために、EWD.js によって自動的にロードされます。
globalIndexer.js には作成済みのスタブがあります。そこでは、必要に応じて適切に MUMPS アレイを拡張、維持するとともに、相当するインデックスを保守する必要があります。 demo アプリケーションの中に、永続的な MUMPS 配列のインデックスを作成している簡単な例があります。

globalIndexer モジュールを使用することで、すべてのメイン MUMPS データ配列のインデックスを一箇所で作成できるようになります。 アプリケーションでは、単純にメインの MUMPS データ配列の変更のみに集中すればよくなります。 ewdGlobals.js によって発行され、globalIndexer モジュール内のロジックによって処理されるイベントによって、メインのアプリケーション· ロジックを乱すことなく、自動的かつ一貫したインデックス作成ができるということです。

注：ewd.js が動作している間は、globalIndexer モジュールに加えた変更を検出して、自動的にすでにそれを使用しているすべての子プロセスにそれを再ロードします。 globalIndexer モジュールを変更するたびに ewd.js モジュールを停止し、再起動する必要はありません。

Web サービスインターフェース

EWD.js による　ウエブサービス

EWD.js は主にブラウザで実行する WebSocket ベースのアプリケーションの作成と実行のためのフレームワークですが、同時に、Web サービスとしてバックエンドの JavaScript のビジネス機能を公開するための安全なメカニズムを提供します。EWD.js の Web サービスは、（アプリケーション/JSON の）HTTP レスポンスとして JSON レスポンスを返します。

EWD.js アプリケーション開発者は、MUMPS データベースにアクセスしたり、レガシーMUMPS 関数を使用する場合には、バックエンドモジュール内でその関数を記述する必要がありますが、 標準の EWD.js のバックエンド WebSocket メッセージ処理関数の記述についても、全く同様です。
関数は、エラー文字列または JSON ドキュメントが含まれている JSON オブジェクトを返します。
EWD.js は、構文解析および処理を行った後、開発者の JSON レスポンスを HTTP レスポンスとしてパッケージ化します。着信する Web サービス HTTP リクエストの認証も行います。

ウエブサービス認証

セキュリティは、HMAC-SHA 認証メカニズムを介して提供されています。これは Amazon Web Services（例えば、SimpleDB データベース）で使用されているセキュリティモデルに基づいています。
EWD.js のセキュリティの基礎となるメカニズムの詳細については以下を参照してください。
http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/HMACAuth.htm

あるユーザーに EWD.js システム上のアプリケーションへのアクセスを許可する場合は、ewdMonitor アプリケーション（セキュリティタブをクリックします）を使用して登録します。　各ユーザーは、一意の accessId が与えられなければなりません。　以下が accessId に対して記録されます。

- 推測することが困難で長い英数字の文字列で作られている　秘密鍵
- ユーザーがアクセスを許可されている　EWD.js アプリケーションのリスト

ユーザーに accessId と秘密鍵を送付する必要があります。これは安全な方法でなければなりません。
ユーザーがこれらの資格情報を共有したり紛失しないことが重要です。

秘密鍵は、デジタル的に HTTP 要求に署名するために、ユーザのクライアント· ソフトウェアによって使用されます。　ユーザーの accessId と署名はクエリ文字列の名前/値のペアとして HTTP リクエストに追加されます。

EWD.js システムが Web サービス要求を受信する時には、

- 最初に、その accessId が認められたものかどうかをチェックします。
- 次に、その accessId が指定されたアプリケーションにアクセスする権利を有するか確認します。
- そして EWD.js は、保持している秘密鍵を使用して、HTTP 要求で受け取った accessId の、着信要求から算出した署名に対し、着信した署名をチェックします。
- 署名が一致する場合、そのユーザーは有効であるとみなされ、要求が処理されます。

これらのステップのいずれかでエラーが検出された場合、エラー応答がユーザに送信され、それ以上の処理は行われません。

EWD.js のウエブサービス生成

EWD.js Web サービスのビジネスロジックは、Node.js のモジュール内の関数として JavaScript で書かれています。 モジュールの名前は、アプリケーション名として使用されます。 あなたが望むか、それが理にかなっていれば、EWD.js Web サービス関数は、ブラウザベースのアプリケーションで使用されるモジュール内に含めることができます。 ewd.js のインストレーションキットに含まれている demo.js モジュールを参照してください。

```
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();
  }
};
```

これは、EWD.js セッション内容を JSON ドキュメントとして返すシンプルな Web サービスです。もし SessionId が無効であれば、代わりにエラーの JSON ドキュメントが送信されます。
上記の例では、EWD Web サービスのアプリケーション名は demo です。モジュール名と同じです（.js ファイル拡張子なし）。 モジュールは、お使いの EWD.js ホームディレクトリ下の node_modules ディレクトリに存在する必要があります。

メソッドは Web サービス名で呼び出されます。 この例では webServiceExample です。一つのオブジェクトが引数として、実行時に EWD.js から自動的に渡されます。便宜上そのオブジェクトを ewd と名付けています。 この EWD.js Web サービス関数のための ewd オブジェクトには、以下の

2 つのサブオブジェクトが含まれています。

mumps: MUMPS グローバルストレージおよびレガシーMUMPS 関数へのアクセスを提供します。
これは WebSocket のメッセージハンドリング機能と同じように使用されます。

query: 着信 HTTP リクエストのクエリ文字列の中に、着信の名前/値のペアを含みます。
名前/値ペアの名前や期待値は、必要とする機能を着信 HTTP リクエストによって提示する開発者が決めます。

関数を書いて、モジュールを保存すれば、直ちに EWD.js.の Web サービスとして利用可能です。

開発者が考慮することは、これで全てです。

EWD.js ウエブサービスの呼び出し

EWD.js の Web サービス呼び出しは、次に示す構造の HTTP 要求を使用します。

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

以下は、名前/値のペアとしてクエリ文字列に含まれている必要があります。

- 要求されている特定のサービス機能に必要な 名前/値のペア：さらに
- **accessId**：HTTP リクエストを送信しているユーザの登録済み accessId
- **timestamp**：JavaScript の UTCString()フォーマット
  (http://www.w3schools.com/jsref/jsref_toutcstring.asp)
  による現在の日付/時間
- **signature**：正規化された名前/値のペアの文字列から計算された HMAC-SHA256 ダイジェスト値
  （Amazon Web Services SimpleDB authentication for details of the normalisation algorithm と、EWD.js ファイル中のリンク参照）

例：

```
https://192.168.1.89:8080/json/demo/webServiceExample?
  id=1233&
  accessId=rob12kjh1i23&
  timestamp=Wed, 19 Jun 2013 14:14:35 GMT&
  signature=P0bIakNehj2TkuadxbKRsIgJCGIhY1EvntJdSce5XvQ=
```

当然のことながら、送信する前に、名前/値のペアは URI エンコードする必要があります。HTTP 要求は、実際には次のようになります。

```
https://192.168.1.89:8080/json/demo/webServiceExample?
  id=1233&
  accessId=rob12kjh1i23&
  timestamp=Wed%2C%2019%20Jun%202013%2014%3A14%3A35%20GMT&
  signature=P0bIakNehj2TkuadxbKRsIgJCGIhY1EvntJdSce5XvQ=
```

Node.js EWD.js ウエブサービス　クライアント

簡単に EWD.js Web サービスを使用できるようにするために、私たちは正しく署名された要求を送信し、その応答を処理するプロセスを自動化する、オープンソース Node.js のモジュールを作成しました。

モジュールは次の URL を参照してください：https://github.com/robtweed/ewdliteclient

インストールは非常に簡単です。npm を使います。

```
  npm install ewdliteclient
```

含まれているモジュールの例として、demo.js モジュールの中で webServiceExample サービスを呼び出す方法を示しました。　実際の EWD.js システムに合わせて、ホストの IP アドレス/ドメイン名、ポートなどを、正しいパラメータに編集する必要があります。　また、accessId を EWD.js システムに登録されているものに変更する必要があります（次のセクションを見て下さい）

Node.js の REPL (read–eval–print loop)環境で EWD Web サービスクライアントを試してみると：

```
 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
```

あなたが作成した Node.js のアプリケーションで EWD.js Web サービスクライアントを使用するには、次の手順を実行します。

```
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));
  }
});
```

同等のクライアントは、他の言語でも開発することができます。 正しい署名アルゴリズムを実装するためのガイドとして、Node.js クライアントと Amazon SimpleDB のドキュメントを使用してください。

Node.js EWD.js ウエブサービス ユーザーの登録

EWDMonitor を使用して登録可能です（前の章を参照してください）。 まずセキュリティタブをクリックしてください。 GUI を使って、新しいユーザーを追加し、または既存のものをメン

テ・削除することができます。

以下の URL を使用してブラウザから EWDMonitor アプリケーションを起動します。

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

EWD.js の構成に応じて、適切に IP アドレスまたはホスト名とポートを変更してください。

プログラムでのユーザー登録

プログラムによって登録されたユーザーを作成することが可能です。初歩的な例として https://github.com/robtweed/ewd.js/blob/master/OSEHRA/registerWSClient.js を参照してください。実際の EWD.js 構成に従ってなどのパスを変更する必要があります。また registerWSClient.js ファイル内の zewd.setDocument（）メソッド内で定義されている EWDLiteServiceAccessId オブジェクトを変更する必要があります。

あなたのバージョンは、あなたの EWD.js ホームディレクトリに保存し、そこから実行する必要があります。

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

MUMPS コードの Web サービスへの変換

EWD.js は、非常に簡単に、セキュアな JSON/ HTTP Web サービスとして、最新のコードやレガシーMUMPS コードを公開することができます。　そのために MUMPS コードに二つの簡単なラッパーを構築します。

内側のラッパー機能

この機能を使用すると、Web サービスとして公開する MUMPS コードを囲む事が出来ます。Caché クラス･メソッドを公開するためにもこのテクニックを使用できることに注意してください。ラッパー関数は 2 つの引数を持ちます。

inputs　　：MUMPS コードに必要とされる全ての入力される名前/値のペアが含まれるローカル配列。

outpits：MUMPS コードからの出力を保持するローカル配列。この配列は、複雑にできるので、あなたが望むような深さで階層化できます。EWD.js は Web サービスから返される対応する

出力配列の内容を自動的に JSON レスポンスオブジェクトに変換します。

ラッパー関数では、MUMPS コードによって作成された変数は、関数の外に漏れないことを保証する New コマンドを使用する必要があります。　さらに、MUMPS コードに必要なすべてのデータが inputs 配列によって供給されなければなりません。グローバルデータおよびローカル変数に依存することはできません。

次の例で、Vista の GET ^ VPRD を EWD.js で使用するためにラップする方法を示します。

```
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=$g(inputs("IEN"))
  ;
  ; now we can call the procedure
  ;
  do
GET^VPRD(,dfn,"demograph;allerg;meds;immunization;problem;vital;visit;ap
pointment",,,,)
  ;
  ; now transfer the results into the outputs array
  ;
  merge outputs=^TMP("VPR",$j)
  set outputs(0)="VISTA Extract for patient IEN "_dfn
  ;
  ; tidy up and finish
  ;
  kill ^TMP("VPR",$j)
  QUIT ""
```

```
 ;
```

^ vistADemo という Cache ルーチン名または vistADemo.m という GT.M ルーチン· ファイル名でこれを保存してください。

これでレガシーVista のプロシージャを呼び出して、明快で、自己完結、再利用可能な機能として使うことができます。

外側のラッパーファンクション

次のステップでは、二番目の外側の関数ラッパーを作成します。これは、EWD.js が呼び出すことができる正規化された標準インタフェースを提供します。　それは EWD.js が使用できる一時的なグローバルへ、inputs と outputs の配列をマッピングすることです。　このグローバルの名前は ^%zewdTemp です。　定義する最初の添え字としてプロセス Id を付加することで確実に、複数の EWD.js プロセスが、
この一時的なグローバルに書き込まれたデータを壊さないようにすることができます。プロセス Id は EWD.js.によって自動的に外側ラッパー関数に渡されます。

私たちが既存の Cache ルーチンまたは GT.M ルーチンファイルに追加する必要があることは、これだけです。

```
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
```

Node.js のモジュールから外部ラッパーの呼び出し

EWD.js にはラップされた MUMPS コードを呼び出すことが可能な組み込み関数が用意されています。

```
ewd.invokeWrapper(MumpsFunctionRef, ewd);
```

ここでは、先にラップされた Vista の機能を呼び出すために使用するバックエンド Node.js のモジュールの例を示します。

```
module.exports = {
  getVistAData: function(ewd) {
    return ewd.invokeWrapper('vistaExtractWrapper^vistADemo', ewd);
  }
}
```

Web サービスとしての呼び出し

MUMPS コードは、今やいずれの EWD.js Web サービスとも同じ方法で呼び出すことができます。下の例では、 上記に示したバックエンド Node.js のモジュールに vistATest.js と名付けて、何かを探すためにクライアントシステムから HTTP（S）リクエスト（URL エンコードは、わかりやすくするために使用していません）を送っています。 IP アドレス、accessId、ポートおよびタイムスタンプと署名の値は、もちろん適切に変更してください。

```
https://192.168.1.89:8080/json/vistATest/getVistAData?
  IEN=123456&
  accessId=rob12kjh1i23&
  timestamp=Wed, 19 Jun 2013 14:14:35 GMT&
  signature=P0bIakNehj2TkuadxbKRsIgJCGIhY1EvntJdSce5XvQ=
```

これは vistATest モジュール内の getVistAData メソッドを呼び出し、IEN は入力値として自動的に渡されます。 URL 内のすべての名前/値のペア（accessid、タイムスタンプ、署名など）は自動的に ewd.invokeWrapper（）関数から呼び出される MUMPS ラッパー関数の inputs 配列にマッピングされます。 このため MUMPS 関数がもっと多くの inputs を必要とするのであれば、単純に HTTP リクエストの名前/値のペアのリストに追加します。 名前は大文字と小文字が区別されることに注意してください。

この Web サービスから返る受信応答は、MUMPS 関数の出力配列の内容と等価の JSON オブジェクトとして表現された、JSON 形式のペイロードとして配信されます

Node.js からの Web サービスの呼び出し

Node.js のモジュール内の MUMPS コードより EWD.js Web サービス･インタフェースを起動したいなら（例えば、リモート･システム上で実行される EWD.js から）、この章で前述した ewdliteclient モジュールを使用することが可能です。　このモジュールは、開発者が呼び出しインタフェースを定義することに集中することができるように、自動的にすべてのデジタル署名メカニズムの後になります。

以下のように、上記の Web サービスは、リモート EWD.js（または他の Node.js の）システムから呼び出すことができます。

```
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));
  }
});
```

他の言語または環境からの Web サービスの呼び出し

あなたが他の言語や環境から、MUMPS コードに EWD.js Web サービスのインタフェースを呼び出すためにやらなければならないことは、先に示されているタイプの適切に署名された HTTP リクエストを構築することです。 Amazon Web サービスのドキュメントに記載されたルールに従って、署名ルールを実装する必要があります。

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

ewdliteclient モジュール内の JavaScript 実装がガイドとして使えます。

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

あなたのクライアントインターフェイスは JSON レスポンスを受け取ることが出来ます。

EWD.js のアップデート

EWD.js のアップデート

ewd.js は、継続的に強化されているので、アップデートがリリースされているかどうかを確認するために GitHub の上で調べて見て下さい。　告知は、Twitter 上で行いますので　@rtweed をフォローして下さい。

EWD.js のアップデートするには、ewd.js モジュールを更新してください。

Node.js と npm によって ewd.js をとても簡単に更新することが出きます。あなたの EWD.js ホームディレクトリに移動して、npm アップデートを使用するだけです。

例：

```
cd ~/ewdjs
npm update ewdjs
```

ewdMonitor アプリケーションの更新と拡張機能がインストールパッケージに含まれているので、それらのさまざまなコンポーネントファイルを適切な宛先ディレクトリへコピーするようにしてください。　標準的な方法でインストール、設定した EWD.js では、以下を実行することにより、それらを正しい位置に移動させることができます。

```
cd node_modules/ewdjs
node update
```

もとのインストールによって作成された、EWD.js アプリケーション、スタートアップファイルなどは、この更新プロセス中に置き換えられますので注意してください。　独自のアプリケーションや名前の異なったファイルは変更されません。　EWD.js によってインストールされたオリジナルのファイルのどれかを変更した場合には、名前の変更やバックアップを取って変更内容が失われないように、更新を実行する前に確認してください。

付録１

スクラッチからの GlobalsDB-based EWD.js/Ubuntu システムの作成

はじめに

EWD.js にはわずか数分で、完全に動作する GlobalsDB ベース EWD.js システムをスクラッチから構築するための自動インストールスクリプトが含まれています。インストールは、64 ビット Ubuntu のデスクトップまたはサーバシステムから開始します。物理マシン、仮想マシンまたは EC2 インスタンスのいずれでもかまいません。EWD.js インストーラは以下をインストールし、設定します。

- NVM NVM (Node.js のバージョンマネージャ, Node.js の迅速かつ簡単な更新を可能にします)
- Node.js
- GlobalsDB
- NodeM
- EWD.js

インストーラの実行

Ubuntu マシンでターミナル･セッションを起動し、次の操作を行います。

```
cd ~
sudo apt-get install -y subversion
svn export https://github.com/robtweed/ewd.js/trunk/globalsdb globalsdb
source globalsdb/install.sh
```

EWD.js は使用できる状態になりました！

ewd.js の起動

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

ewdMonitor アプリケーションの実行

お使いのブラウザで、URL（あなたの Ubuntu マシンに割り当てられた適切な IP アドレスの変更）を入力します。

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

SSL を使用する場合は、以下に示すようにデフォルトの ewdStart-globals.js ファイルを変更して下さい：

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

SSL 証明書に関する警告が表示された場合は、OK を押してください。これは ewd.js が自己署名証明書を使用しているためです。

ewdMonitor アプリケーションはパスワードの入力を求めます。このパスワードは ewdStart-globals.js 起動ファイルで定義され、既定では「keepThisSecret!」となっています。[特に dEWDrop VM がパブリックアクセス権を持っている場合は、できるだけ早くこのパスワードを変更することをお勧めします。しかし、今のところは、デフォルトのままにしておきます。]

パスワードを入力し、ログインボタンをクリックすると、正常に起動し、EWD.js と ewdMonitor アプリケーションが実行されます。

EWD.js アプリケーションを構築する準備が整いました。付録 4 へ進んでください。

付録２

スクラッチからの GT.M-based EWD.js/Ubuntu 14.04 システムの作成

はじめに

Ubuntu の 14.04 には GT.M.ための新しい apt-get のインストーラが導入されています。EWD.js の新バージョンは、このインストーラを使った GT.M に対する完全に動作するシステムが作成できるインストール･スクリプトが含まれています。インストールは、Ubuntu の 14.04 のデスクトップまたはサーバシステムから開始します。物理マシン、仮想マシンまたは EC2 インスタンスのいずれでもかまいません。EWD.js インストーラは以下をインストールし、設定します

- NVM (Node.js のバージョンマネージャ, Node.js の迅速かつ簡単な更新を可能にします)
- Node.js
- GT.M
- NodeM
- EWD.js

インストーラの実行

Ubuntu14.04 マシンでターミナル･セッションを起動し、次の操作を行います。

```
cd ~
sudo apt-get install -y subversion
svn export https://github.com/robtweed/ewd.js/trunk/gtm gtm
source gtm/install.sh
```

EWD.js は使用できる状態になりました！

ewd.js の起動

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

ewdMonitor アプリケーションの実行

お使いのブラウザで、URL（あなたの Ubuntu マシンに割り当てられた適切な IP アドレスの変更）を入力します。

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

SSL を使用する場合は、以下に示すようにデフォルトの ewdStart-gtm.js ファイルを変更して下さい。

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

SSL 証明書に関する警告が表示された場合は、OK を押してください。これは ewd.js が自己署名証明書を使用しているためです。

ewdMonitor アプリケーションはパスワードの入力を求めます。このパスワードは ewdStart-gtm.js 起動ファイルで定義され、既定では「keepThisSecret!」となっています。[特に dEWDrop VM がパブリックアクセス権を持っている場合は、できるだけ早くこのパスワードを変更することをお勧めします。しかし、今のところは、デフォルトのままにしておきます。]

パスワードを入力し、ログインボタンをクリックすると、正常に起動し、EWD.js と ewdMonitor アプリケーションが実行されます。

Ubuntu 上で、EWD.js アプリケーションを構築する準備が整いました。付録 4 へ進んでください。

付録３

dEWDrop v5 サーバーで動作する EWD.js のインストール

はじめに

構築済みの、GT.M ベースの dEWDrop v5 仮想マシン（VM）で、EWD の古い「クラシック」バージョンを含む、Vista ベースの ready-to-run 環境を提供します。

以下の手順によって、dEWDrop v5 VM 上の新しい、更新された完全に動作する EWD.js 環境をすぐに作成できます。

dEWDrop VM のインストール

すでに dEWDrop VM が稼働している場合は、次のセクション：dEWDrop VM のアップデートに進んでください。

まだ dEWDrop VM をインストールしていない場合は、次の手順を実行します。さまざまな仮想マシンのホスティングパッケージを使用することができますが、無料の VMware Player を使用すると仮定します。

Step1

以下の URL から最新の dEWDrop VM（執筆時点ではバージョン 5）をダウンロードしてください。

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

ファイルサイズは 1.6Gb です。

Step2

仮想マシン用のディレクトリを作成します。

~/Virtual_Machines/dEWDrop5

Windows マシンでは以下の様になります。

c:\Virtual_Machines\dEWDrop5

Step3

作成したディレクトリにダウンロードした zip ファイルを移動します。

Step4

zip ファイルを展開します。展開には 7-ZIP エキスパンダを使用します。

ホストマシンが Ubuntu Linux の場合、以下を使用してインストールします。

sudo apt-get install p7zip

Windows または Mac OS X を使用している場合は、以下を参照して下さい。

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

注：ご使用のホスト･マシンが Windows を使用している場合は、メインファイル（dEWDrop.vmdk）は 8.7Gb 程度になるため、NTFS でフォーマットされているドライブに Virtual_Machines ディレクトリに配置する必要があります。（PC の FAT ファイルシステムは４GB までだったための注意）

Ubuntu の Linux を実行しているホストマシンの場合は、次のように dEWDrop VM ファイルを展開します。

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

Step5

次のステップに進むためには VMware Player がインストールされている必要があります。VMware Player がインストールされていない場合は、以下を参照してダウンロード、インストールを行ってください。

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

Step 6:

VMWare Player を起動し、新しいファイルを開くメニューオプションを選択します。
Virtual_Machines/ dEWDrop5 ディレクトリに移動し、以下のファイルを選択します。

dEWDrop.vmx

（代わりに、ホストマシンのファイル･マネージャ･ユーティリティから dEWDrop.vmx をダブルクリックする事も出来ます）。

注：VMWare　Player が仮想ディスクファイルを見つけることができないことことがあります。その様な場合は、ファイル dEWDrop.vmdk を探して選択します。

dEWDrop VM を最初に起動したときには、移動かまたは、コピーしたかどうかを、VMWare のかを尋ねられます。　“Copied It”を選択してください。　VMware Tools をダウンロードするかどうかを尋ねられた場合は、それは気にしないように伝えることができます。

Step7

VMware Player のコンソールより dEWDrop VM を起動します。起動するとユーザー名の入力を要求されます。

ユーザー名：　vista

続いてパスワードを要求されます

パスワード：　ewd

これで、Ubuntu の 12.04 システム上の dEWDrop VM にログイン出来ました。

Step8

コマンドを入力して、ホストのマシンが仮想マシンに割り当てた IP アドレスを調べます。

ifconfig

eth0 というセクションを探し、inet addr で始まる行を探します。　ここで IP アドレスを知ることが出来ます。

例：　192.168.1.68 など

Step9

ここでコンソールを離れて、端末または puTTY を起動し、dEWDrop サーバーに SSH 接続を行う必要があります。　Linux/ Unix のターミナルから Step8 などで調べた IP アドレスを使用して、接続します。

```
ssh vista@192.168.1.68
```

パスワード：ewd を入力します。

これで、dEWDrop VM マシンを起動して実行する事が出来ました。

dEWDrop VM　のアップデート

dEWDrop VM 上のターミナルセッションから、以下の手順に従います。

```
cd ~
sudo apt-get install subversion
svn export https://github.com/robtweed/ewd.js/trunk/dEWDrop dewdrop
source dewdrop/upgrade.sh
```

dEWDrop VM サーバーは完全に更新され、EWD.js.で使用するための準備ができました。

新しい EWD.js 作業環境は、ディレクトリ~/ ewdjs に作成され、すべての EWD.js 関連の仕事はこのディレクトリの下で行われます。　インストールスクリプトによって EWD.js と既存 GT.M データベース（完全に動作する Vista のシステム）との間のアクセスが自動的に設定されます。

存在する　EWD.js アプリケーションのアップデート

すでに EWD.js の古いバージョン（例えば ewdgateway2 モジュールを使用したバージョン）を使用して独自の EWD.js のアプリケーションを作成している場合は、新しいアプリケーションディレクトリ~/ewdjs/www/ewd にこれらのアプリケーションをコピーする必要があります。そしてそれらのバックエンドモジュールファイルは~/ewdjs/node_modules へコピーします。

また、既存の index.html ファイルを変更する必要があります。　/ewdLite/EWD.js と /ewdLite/ewdBootstrap3.js への参照は、それぞれ　/ewdjs/EWD.js と/ewdjs/ewdBootstrap3.js に変更する必要があります。

ewd.js の起動

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

ewdMonitor アプリケーションの実行

お使いのブラウザで、URL（あなたの dEWDrop VM マシンに割り当てられた適切な IP アドレスへ変更してください）を入力します。

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

SSL を使用する場合は、以下に示すようにデフォルトの ewdStart-gtm.js ファイルを変更して下さい。

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

SSL 証明書に関する警告が表示された場合は、OK を押してください。これは ewd.js が自己署名証明書を使用しているためです。

ewdMonitor アプリケーションはパスワードの入力を求めます。このパスワードは ewdStart-gtm.js 起動ファイルで定義され、既定では「keepThisSecret!」となっています。[特に dEWDrop VM がパブリックアクセス権を持っている場合は、できるだけ早くこのパスワードを変更することをお勧めします。しかし、今のところは、デフォルトのままにしておきます。]

パスワードを入力し、ログインボタンをクリックすると、正常に起動し、EWD.js と ewdMonitor アプリケーションが実行されます。

dEWDrop VM サーバー.上で EWD.js アプリケーションを構築する準備が整いました。　付録 4 へ進んでください。

付録４

初心者のためのＥＷＤ．ＪＳガイド

簡単な「Ｈｅｌｌｏ　Ｗｏｒｌｄ」アプリケーション

この付録は、フレームワークを背景に基本原理を実証し、非常に単純なアプリケーションを構築する方法について説明し、EWD.js 初心者向けに記述されています。　単純な（シンプルな HTML ページと、2 個のボタン以外にはほとんど何も使用しない）「Ｈｅｌｌｏ　Ｗｏｒｌｄ」アプリケーションを構築しましょう。　装飾的な JavaScript フレームワークを使用しません。　２つの基本的な関数を用いて「Ｈｅｌｌｏ　Ｗｏｒｌｄ」アプリケーションを作りましょう。

- ひとつ目は、ブラウザ中の HTML ページから、バックエンドモジュールに WebSocket メッセージを送って、Mumps の永続配列へメッセージを保存するボタンを作成します。
- ふたつ目は、最初のメッセージのペイロードを検索してブラウザに返すように指示する、バックエンドモジュールへの WebSocket メッセージを送るボタンを作成します。

このガイドは、あなたが既に EWD.js のインストールやシステム設定を済ませていると仮定しています。まだこれをしていない場合は、このドキュメントの本文、あるいは付録１～３にある自動化されたインストールに従ってください。

さあ、始めましょう。

EWD.js ホームディレクトリー

この付録の全体にわたって、あなたの EWD.js ホームディレクトリーを参照することになります。このドキュメントの大部分がこれを説明しています。この EWD.js ホームディレクトリーは、ewd.js をインストールした、つまり以下を実行した時の、ディレクトリーです。

```
npm install ewdjs
```

したがって、ewd.js モジュールは node_modules という名のサブディレクトリーにあります。

これ以降、EWD.js ホームディレクトリーは、*~/ewdjs*　のように表示していきます。

Windows マシンにおいては、このパスは次のようになります。

c:\ewdjs

ewd.js モジュールを始めよう

適切なスタート・ファイルを使用してください。以下に例を示します。

付録 3 に述べられていたインストーラーを使用して構築された GT.M システム上では：

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

あるいは VM の dEWDrop 上では：

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

HTML ページ

あなたの環境の　*~/ewdjs/www/ewd* の下に、helloworld と名付けたサブディレクトリを作成します。

~/ewdjs/www/ewd/helloworld ディレクトリーに、以下の内容で index.html というファイルを作成してください。

```
<!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>
```

Build 54 の時点で EWD.js は、jQuery を使用するよう要求していることに注意してください。これは、インターネット・エクスプローラーの古いバージョンのような古いブラウザをサポートしたからです。

ブラウザを起動してください。

理想は Crome です。もし Crome を使っているなら、以下のようにして、開発者ツール・コンソールを開いてください。

- 右上メニューのアイコンをクリックしてください（３本の短い横棒が表示されてます）。
- ドロップダウン・メニューからツールを選びます。
- サブメニューから JavaScript コンソールを選びます。
- 最初は、コンソールパネルは Chrome ウィンドウの下に隠れています。右から２番目のアイコン（影付きのコンピュータ端末のような）をクリックします。

さて、URL（あなたのシステムの IP アドレスに、適切に変更してください）を使って HTML ページを実行してください。

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

あなたのブラウザに以下のように表示されるでしょう。

EWD.js Hello World Application

Chrome の JavaScript コンソールを見ると、下記のように表示されるでしょう。

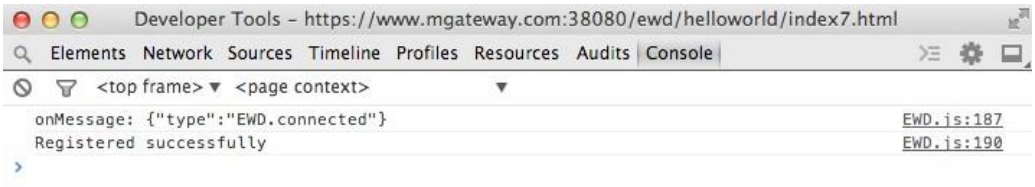
エラーメッセージが EWD.js framework によって返されます。以下のように index.html ページを編集することにより、それを解決することができます。

```
<!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>
```

これは、helloworld という名のアプリケーションを起動している事を EWD.js に伝えます。 最後のラインは、JavaScript コンソールへその動作を記録するように EWD.js に命令します(そうでなければ、EWD.js はランタイムエラーとして何も記録しません)。 ブラウザをリフレッシュしてください。そうすれば、今回は、JavaScript コンソールでメッセージがこのように見えるでしょう。



Hello World アプリケーションは、今、EWD.js で適切に動作しています。 また、他のアプリケーションを実行する準備ができています。

app.js ファイル

次へ進む前に、ベストプラクティスを採用しましょう。あなたの index.html ファイル内にインラインの JavaScript を作成する代わりに、分離した JavaScript ファイルへそれらをすべて移行させましょう。既定では、このファイルを app.js と名付けます。 つまり、index.html ファイルと同じディレクトリーに app.js という名の新しいファイルを作成し、その中へ JavaScript を移動させます。 app.js は以下のようになります。

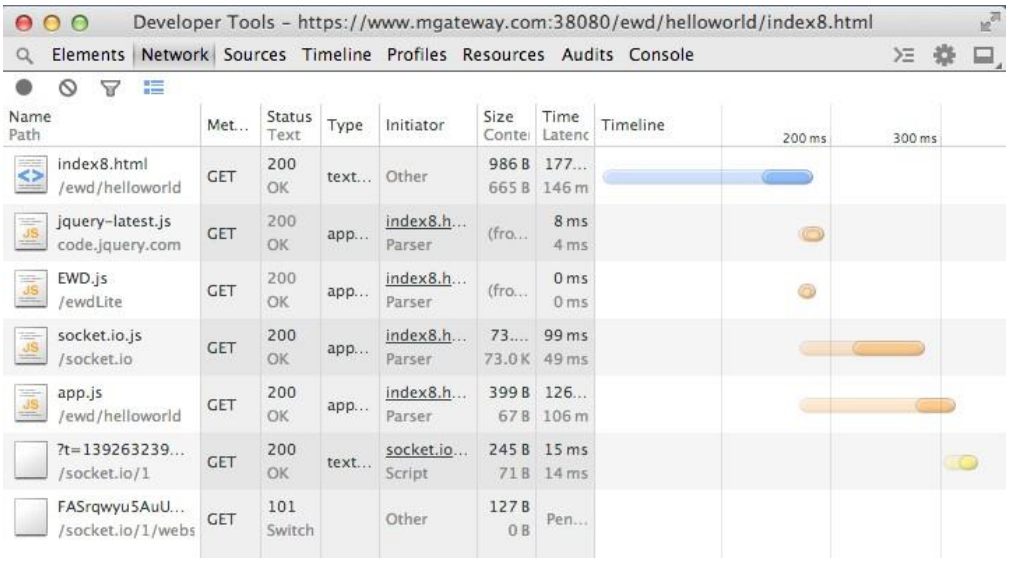
```
EWD.application = {
  name: 'helloworld'
};
EWD.sockets.log = true;
```

app.js をロードするように、index.html ファイルを編集してください。

```
<!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>
```

ブラウザ中の URL をリロードしてみます。アプリケーションは、同様に動作するはずです。JavaScript コンソールの一番上のネットワークタブのクリックにより、app.js ファイルを正しくロードしていることを確認することができます。



5行目において、app.js がロードされていることが分かります。

最初の WebSockt メッセージ送信

ボタンをページの body に付加して、それに *onClick()*イベントハンドラーを割り当てます。

```
<!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>
```

さて、app.js ファイルに sendMessage()という名のハンドラー関数を定義して下さい。 このハンドラー関数は EWD.js WebSocket メッセージをバックエンドへ送るでしょう。 メッセージのタイプは sendHelloWorld として定義しました。　メッセージはそのペイロードに多くの名前と値のペアを含みます。　ペイロード内容および構造を決定することは私たちの責任です。 そのペイロードは params という名のプロパティに入れます。

```
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;
```

ブラウザをリロードすると、「Send Message」と書いてあるボタンが見えます。

ボタンをクリックしてください。すると、次のエラーメッセージが JavaScript コンソールに表示されます。

このエラーはまさに良い知らせです！ それは、メッセージが EWD.js のバックエンドへの送信に成功したことを伝えています。 エラーが伝えていることは、EWD.js は、~/ewdjs/node_modules ディレクトリーに helloworld.js という名のモジュールを見つけることができませんでしたということです。 そう、以前に app.js に加えた、helloworld という名のモジュールを実際に捜す以下のコマンドのために。

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

さあ、helloworld アプリケーション用のバックエンドモジュールを作成しましょう。

helloworld バックエンドモジュール

~/ewdjs/node_modules ディレクトリーに、下記の helloworld.js という名のファイルを作成してください。

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

};
```

最初にするべきことは、helloworld アプリケーションの任意のユーザから入って来る全ての WebSocket メッセージを受信することと、ewd.js コンソールへのそれらの内容を記録することです。忘れてはいけません。 この JavaScript・モジュールはブラウザ中ではなくバックエンドで実行します！

ブラウザで index.html ページをリロードして、ボタンをクリックしてください。すると、ewd.js モジュールに次のように見えます（メッセージによって相当後ろにスクロールする必要があるかもしれません）。

```
*** 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!
```

上部には、送信したメッセージがあります!

Type-specific Message Handler の追加

OK、それでは、タイプ sendHelloWorld のメッセージ用の specific handler を加えましょう。 このことにより、onSocketMessage()関数を onMessage という名のオブジェクトに入れ替えて、それに対し、特定のメッセージ・ハンドラーを定義します。 sendHelloWorld メッセージを扱うために、helloworld.js ファイルの内容を以下のように置き換えてください。

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

この編集されたバージョンのファイルを保存してください。

モジュールで発生したエラーのデバッグ

helloworld.js モジュールの編集されたバージョンを保存した時、ewd.js コンソールで注意深く見れば、下記がログに現われるのが見えるでしょう。

```
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
```

バックエンドモジュールがその子プロセスのうちの 1 つにロードされる場合は常に、ewd.js はイベントハンドラーを自動的に加えます。モジュールに変更が行なわれる場合は常に、子プロセスはリロードすることを試みるでしょう。 モジュールにシンタックスエラーがあった場合にはリロードに失敗することに注意してください。お気の毒ですが、子プロセスは、多くの場合あなたになぜそれがロードしなかったかのどんな意味のある診断も与えることができません。

その場合のヒントです。モジュールがロードしない場合、新しい端末セッションを立ち上げて、Node.js REPL を開始してください。（最初に、あなたの EWD.js ホームディレクトリーにいることを確かめてください）

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

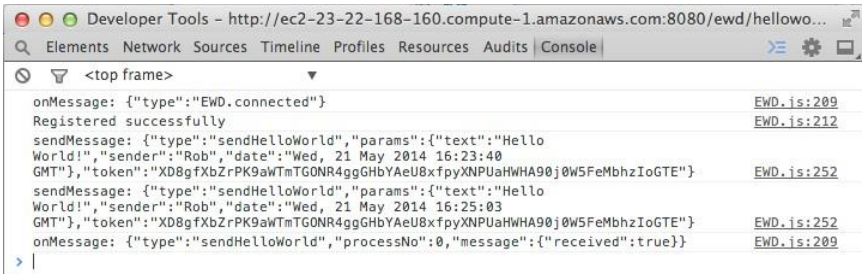
さて require()関数を使用して、REPL の中のモジュールをロードしてみてください。
構文エラーがあれば、それらが何であるか確かめることができるでしょう。

```
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
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

The Type-specific Message Handler in Action

ブラウザ中の index.html ページをリロードして、再びボタンをクリックしてください。
今回は、次のメッセージが JavaScript コンソールに記述されたのが見えるでしょう。



もはやエラーはありません。 最後のメッセージは、私たちのバックエンドメッセージ・ハンドラーに加えたリターンの結果として現われました。

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

バックエンドモジュールがその returnValue として JSON オブジェクトを返す場合、受信メッセージと同じタイプの WebSocket メッセージが、ブラウザに返されます。
returnValue オブジェクトは、ブラウザによって受け取られた WebSocket メッセージのメッセージプロパティにあります。

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

この例では、バックエンドからあなたの好きなだけ複雑に深く入れ子した JSON オブジェクトを返す代わりに、非常に単純なオブジェクトを返したことに注意してください。

さて、受信メッセージを正確に扱うことができ、ブラウザに対する反応を返すことができました。次に Mumps データベースへどのように受信メッセージを格納することができるかを見ましょう。

Mumps データベースへレコードを格納する

この練習では、複雑なことはまだ何もしていません。 _setDocument()メソッドの使用により、Mumps の永続配列へ受信メッセージ・オブジェクトを直接保存したことのみです。 これから%AMessage という名の配列を使用するつもりです。なぜなら、これが ewdMonitor アプリケーションにおける永続オブジェクト名のリストのトップの近くで現われるからです。

そこで、以下のように helloworld.js モジュールファイルを編集してください。

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

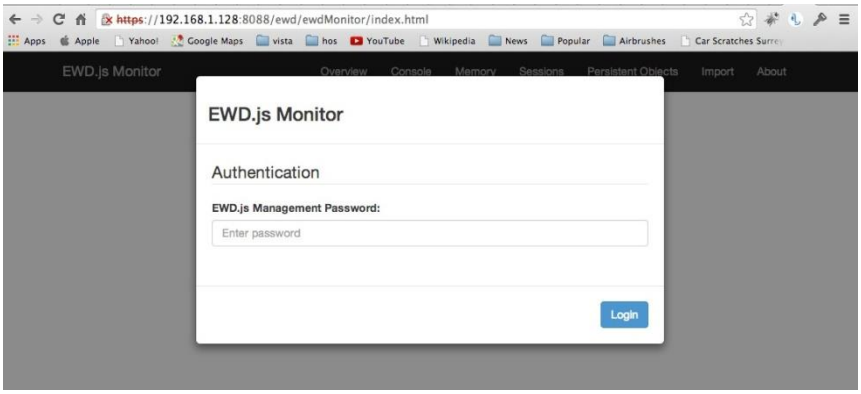
ページをリロードして、ボタンをクリックしてください。 修正済の応答メッセージが Javascript コンソールに現われるのが見えるでしょう。 メッセージによって何がバックエンドで起こったか分かるでしょう。 この確認には、ewdMonitor アプリケーションを使用します。

Mumps データベースを確認するために ewdMonitor アプリケーションを使用する

新しいブラウザ・タブを開き、URL を使って ewdMonitor アプリケーションを開始します。(適切な IP アドレスに修正します)

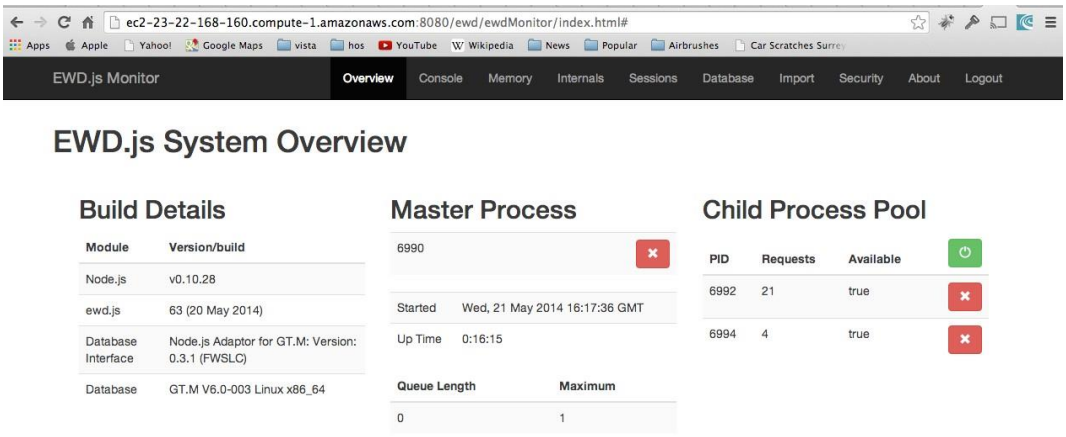
```
http://192.168.1.101:8080/ewd/ewdMonitor/index.html
```

ewdMonitor アプリケーションは起動するとパスワードを聞いてきます。

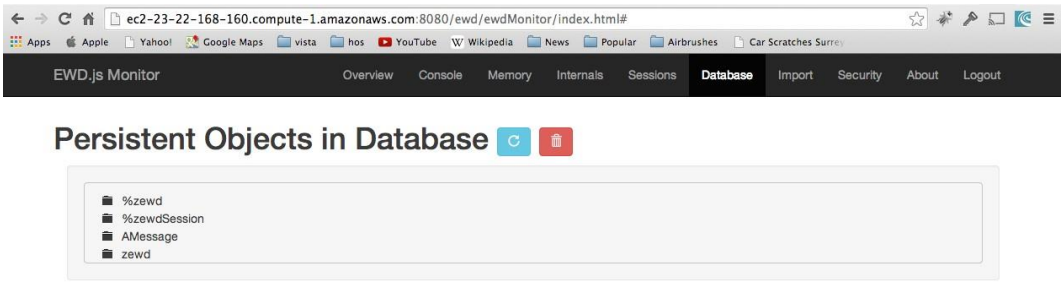


もし、開始時に ewd.js のパスワードを変更しなかったならば（EWD.js を使用し始めるときに一度変更はすべきですが）デフォルト・パスワード keepThisSecret!を入力します!

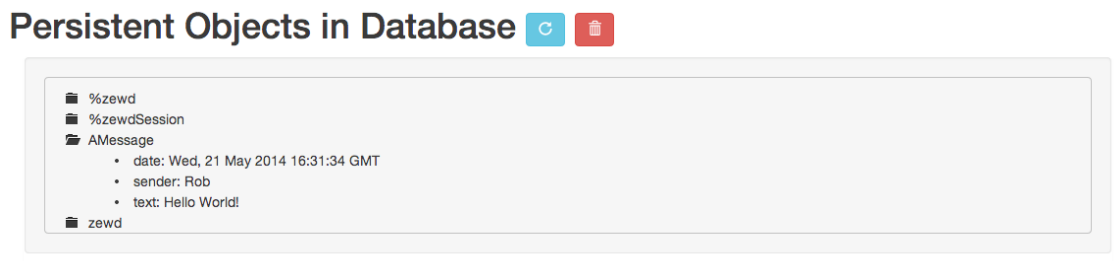
ewdMonitor アプリケーションが動作始めます。



データベースという名のタブをクリックしてください。 Mumps Globals のリストが表示されます - この表示されたリストは、実行しているシステムのタイプに依存するでしょう。また、リストをスクロールする必要があるかもしれません。



リスト中に、作成した永続オブジェクト：AMessage を見つけるでしょう。
それを広げるには名前の隣のフォルダーシンボルをクリックしてください。
そして、そのレベルをすべて広げるために繰り返します。
それがブラウザから送った WebSocket メッセージの構造および内容を正確に映すことを理解して下さい。



これと WebSocket メッセージとの差異は、このバージョンがディスク上に永久に格納されるということです（少なくとも私たちがそれを削除するか変更することに決めるまで）。

Mumps データベースへこのデータを保存するために何もあらかじめ宣言する必要はなかったことに気づきます。また、スキーマを定義する必要はありませんでした。 単に永続オブジェクトの名前を決めて、_setDocument ()メソッドを使って、それへ JSON ドキュメントを保存しました。

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

Mumps データベースへ JSON ドキュメントを格納することは自明で単純です!

再びブラウザ中のボタンをクリックしてみます。ewdMonitor アプリケーションの持続オブジェクト・パネル中のリフレッシュ・ボタンを押して、再び AMessage オブジェクトの内容を調べて下さい。 日付が最新のメッセージの新しい値で上書きされたことを理解して下さい。

ブラウザ中の応答メッセージの扱い

私たちの WebSocket メッセージサンプルの応答時間を作成するために必要な、最後の 1 ステップは、ブラウザに返されたレスポンスを適切に扱うことです。
それをするために、app.js への WebSocket メッセージハンドラー、および index.html ファイル中のメッセージを表示するためのプレースホルダーを加える必要があります。

div タグを index.html ファイルに以下のように加えてください。

```
<!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>
```

次に、メッセージ・ハンドラーを app. js に加えてください。

```
  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()
      }
    });
  };
```

EWD. application. onMessage オブジェクトは EWD. js バックエンドから入って来る WebSocket メッセージ用の定義ハンドラーに使用されます。 ハンドラーはそれぞれ、名前が受信メッセージのタイプ特性と一致する機能です。この場合は「sendHelloWorld」です。 ハンドラー関数はそれぞれ単一の引数を持っています。受信メッセージ・オブジェクトです。 通常、メッセージ・ハンドラーは、受信メッセージの JSON ペイロードに応じた方法で UI を修正します。 この例では、2 秒後に再び消える確認メッセージを示しています。

もちろん、ハンドラーは望みどおりに複雑にできます。 可能なことの考えを思いつくために ewdMonitor アプリケーションの動作を見てください。 それは、受信 WebSocket メッセージのすべての種類に応答する EWD. js アプリケーションです。

保存されたメッセージを検索ためのふたつ目のボタン

さあ、ふたつ目のボタンを HTML ページに加えましょう。また、クリックされる場合は常に、保存されたメッセージを検索させてください。これは実際、かなり単純です。
最初に以下のように index.html ファイルを編集してください。

```
<!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>
```

getMessage()を app.js のクリックハンドラー関数に加えてください。

```
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"

  });
};
```

これが今何を行うか、たぶんお分かりでしょう。 ふたつ目のボタンをクリックすると、バックエンドに getHelloWorld タイプの WebSocket メッセージを送るでしょう。 このメッセージについては、ペイロードを送っていないことに注意してください。 基本的に格納されたメッセージを取って来るために私たちが望むバックエンドに信号を送るメッセージを使用しています。

ふたつ目のメッセージ用のバックエンドメッセージ・ハンドラーの追加

さらにこの受信メッセージに応答し、保存されたメッセージを検索する、バックエンドイベント・ハンドラーを加える必要があります。 したがって、バックエンドモジュール(つまり helloworld.js)ファイルを以下のように編集します。

```
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();
    }
  }
};
```

新バージョンを実行してみます

このファイルを保存し、ブラウザへ index.html ファイルをリロードしてください。 このように表示されます。

Retrieve にセーブされたメッセージ・ボタンをクリックしてみて、JavaScript コンソールを見

てください。このように表示されるでしょう。

Mumps データベースへ保存したオリジナルの JSON メッセージがあります。 ご覧のように、受け取られたメッセージには getHelloWorld のタイプおよび保存された JSON の内容が、そのメッセージプロパティにあります。

メッセージ・ハンドラーをブラウザに加えます

したがって、今私たちがしなければならないのは、ブラウザ中の検索されたメッセージの詳細を表示するためにハンドラーを app. js ファイルに加えることです。

```
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"
  });
};
```

今、Retrieve にセーブされたメッセージをクリックすると、ブラウザに結果が表示されるでしょう。

上記の例において、生の JSON メッセージをダンプしているところです。 そのコンポーネントを分離させて、ブラウザにそれらを適切に表示してみてください。

さらに、Send メッセージそして次に Retrieve にセーブされたメッセージ・ボタンをクリックしてみてください。これをするごとに、メッセージの新バージョンが表示されるでしょう。 日付の値が異なることが言えるでしょう。

暗黙のハンドラーとバックエンドから送るマルチプル・メッセージ

最後に試みるべき事があります。 バックエンドメッセージハンドラーはリターンメッセージを

送る必要はありません。 例えば、sendHelloWorld メッセージ・ハンドラーを、暗黙「fire and forget」ハンドラーとして作ることができました。

```
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();
    }
  }
};
```

更に、バックエンドメッセージ・ハンドラーはそれがブラウザに好きなだけメッセージを送ることができます。 したがって、例えば、それ自身のタイプを備えた個別のメッセージ中の保存されたメッセージ内容を送り、メッセージが成功に検索されたとブラウザに単に合図するためにハンドラーの returnValue を使用することができました。

```
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};
    }
  }
};
```

index.html ページをリロードし、Send Message ボタンをクリックしてください。 今回は、認識がブラウザに現われません。また、どんな受信メッセージも開発者用ツール・コンソールに現われないでしょう。

さて Retrieve Saved Message ボタンをクリックして、JavaScript コンソールを見てください。下記のように表示されます。

ブラウザには今、タイプ savedMessage および getHelloWorld と共に、バックエンドハンドラーから 2 つの別個のメッセージが送られました。

これが、非常に強力で(使用するのが簡単である)、柔軟な、フレームワークであることがすぐに分かります。あなたは、それを利用してアプリケーションを開発できます。

さらに、単純な HTML ページと共に EWD.js のメッセージおよびバックエンド Mumps ストレージ・エンジンをどのように使用することが出来るかわかりました。 それは任意の JavaScript フレームワークと共に使用することができます。 好きなフレームワークを選んで、EWD.js で開発を始めましょう。

EWD.js /Bootstrap フレームワークの使い方

Bootstrap（http://getbootstrap.com/）と呼ばれる JavaScript フレームワークを使用している場合、EWD.js の Bootstrap に対する特に高度なサポート機能が使えます。 ewdMonitor アプリケーションは Bootstrap を使用して構築されています。 次のセクションでは、EWD.js.の Bootstrap 関連機能を使用して、アプリケーションをさらに改良することができるかを検討してみましょう。

The Bootstrap 3 テンプレートファイル

~/ ewdjs/www/ewd ディレクトリに bootstrap3 というアプリケーションがあります。これは実際に動作するアプリケーションではありません。これは任意の Bootstrap3 EWD.js アプリケーションの出発点として使用できるサンプルページやフラグメントファイルのセットです。

Helloworld, Bootstrap-スタイル

helloworld アプリケーション用に新しい index.html ページを作成します – 名前は indexbs.html にします。 bootstrap3 アプリケーションディレクトリから index.html ファイルをコピーして作成します。それは以下の様になります。

```
 <!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><![e
ndif]-->
    <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>
```

このファイルにはロードされるものがたくさんがあることがわかります。それらは、すべてのアプリケーションのために必要とされる訳ではありませんが、このチュートリアルでは、すべてを残しておいても害はありません。すべての JavaScript と CSS ファイルがコンテンツ配信ネットワーク（CDN）のリポジトリからフェッチされることに注意してください。 Bootstrap3 で使用されるさまざまなコンポーネントのフレームワークおよびユーティリティに慣れてくると、ローカルコピーをインストールし、<script>と<ink>タグを、適切に変更することができます。

次に、bootstrap3 アプリケーションディレクトリ内の main.html という名前のファイルを見つけて、helloworld ディレクトリにコピーします。

あなたが EWD.js / Bootstrap 3 アプリケーションで行う作業の大半は、app.js ファイルおよび index.html ファイルにフェッチ・注入される、フラグメント· ファイルに対して行われます。 index.html に、殆ど変更を加える必要はありません。 デモ HelloWorld アプリケーションでは、これから作成する main.html とフラグメントファイルを中心に単一のフラグメントファイルで作って行きます。

次に、bootstrap3 アプリケーションディレクトリから app.js ファイルをコピーして、app.js の以前のバージョンを置き換えます（おそらく最初のオリジナルのコピーを作成します）。
app.js は次のようになります。

```
 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);
    }
  }
};
```

app.js は、以下のように編集します（太字で示す線で示した部分）。

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

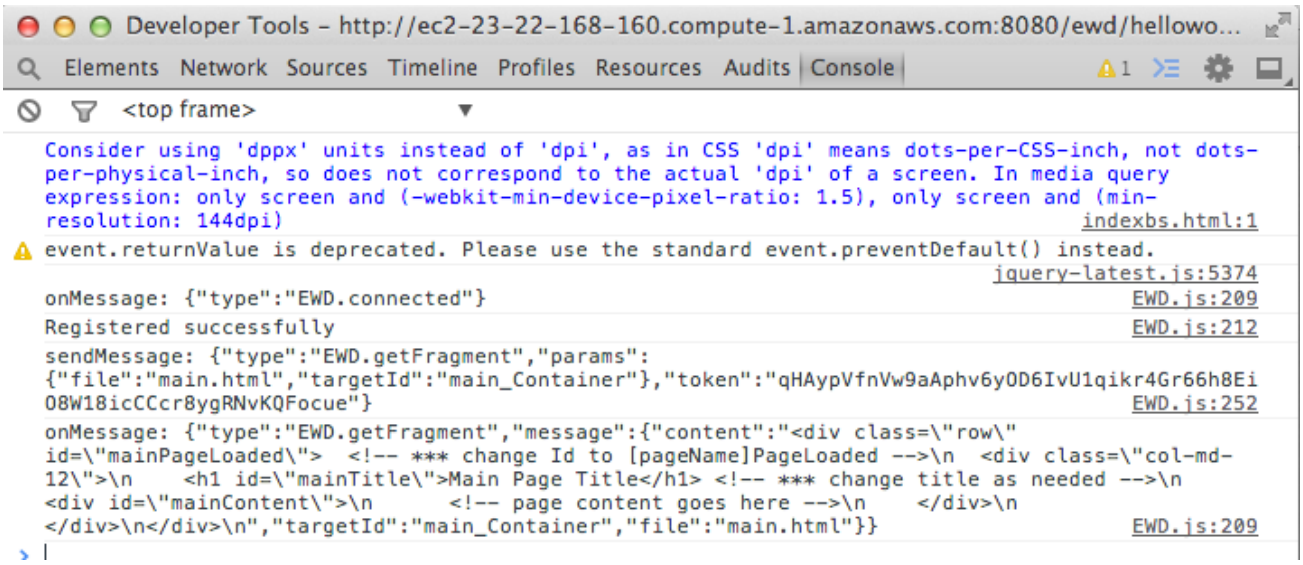
```
development
 EWD.application = {
   name: 'helloworld',
   timeout: 3600,
   login: false,
   labels: {
     'ewd-title': 'Hello World',
     'ewd-navbar-title-phone': 'Hello World App',
     'ewd-navbar-title-other': 'Hello World Application'
   },
   navFragments: {
     main: {
       cache: true
     },
     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
  }
};
```

まだバックエンドモジュールについては心配しないでください。 お使いのブラウザに indexbs.html をロードすると以下のページが見えます。

Main Page Title

Google chrom の JavaScript コンソールには、次のように表示されます。



main.html フラグメントファイルの内容の配信を示す、最終のメッセージに注目してください。また、アプリケーションが自動的に登録されていることがわかります。

アプリケーションの機能の追加を開始する前に、再び app.js ファイルを見てみましょう。以下のセクションを見て下さい。

onStartup：このファンクションは、index.html ファイルとすべての関連する JavaScript と CSS ファイルが読み込まれ初期化され、かつ、socket.io ライブラリがバックエンドへの WebSocket 接続が成立し、EWD.js がアプリケーション登録するまでの間に自動的にトリガされます。 この機

能を使用して、index.html ファイル内で定義されている任意のボタンなどに初期のハンドラを定義します。今回のアプリケーションでは EWD.getFragment（）関数を使用して main.html フラグメントページを取得します。

onPageSwap：このオブジェクトは、Bootstrap ナビゲーションバー内のナビゲーションタブがクリックされたときに起動するすべてのハンドラ関数を含みます。当初、ナビゲーションバーを使用する予定はありませんので、これを無視して下さい。

onFragment：このオブジェクトは、フラグメントファイルがブラウザにロードされたときに起動するすべてのハンドラ関数を含みます。フラグメントファイルで定義されるボタンなどの、任意のハンドラを定義する場所です。これは EWD.js の本当に素晴らしく、重要な特徴です。静的なフラグメント･ファイルでない動的な、実行時機能の追加を可能にします。

onMessage：このオブジェクトは EWD.js アプリケーションのバックエンドモジュールから送信された着信 WebSocket メッセージの全てのハンドラ関数を含みます。

Bootstrap 3 を使用する場合、基本的な Hello World アプリケーションで見たよりも、イベントを処理するために使用するより多くの洗練されたメカニズムを目にすることが出来ます。

Bootstrap3 ページからのメッセージの送信法

まず、オリジナルの Hello World アプリケーションと同様に、バックエンドにメッセージを送信するためのボタンをページに追加してみましょう。　main.html ファイルの変更を行いますが、一緒に、main.html とがロードされるときにフェッチされる独自のフラグメント･ファイルのボタンを定義してみましょう。少し過剰ですが、app.js.中で onFragment オブジェクトを使用する良いデモンストレーションです。

以下のスクリプトを含む、button1.html という名前の新しいファイルを helloworld のディレクトリに作成します。

```
<button class="btn btn-info" type="button" id="sendMsgBtn" data-
toggle="tooltip" dataplacement="top" title="" data-original-title="Send
Message">
   <span class="glyphicon glyphicon-send"></span>
</button>
```

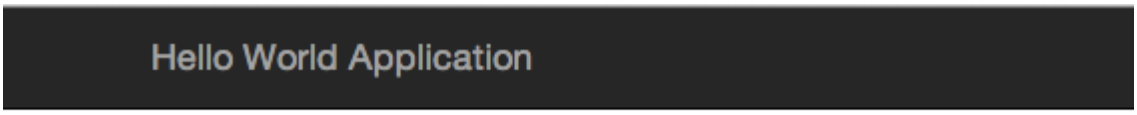
app.js ファイル内、onFragment オブジェクト部分の太字で示した所を編集して下さい。

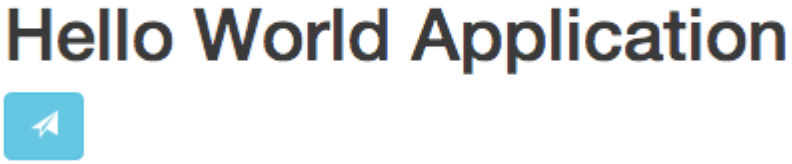
```
onFragment: {
```

```
  // add handlers that fire after fragment contents are loaded into
browser
  'main.html': function() {
    $('#mainTitle').text('Hello World Application');
    EWD.getFragment('button1.html', 'mainContent');
   }
 },
```

main.html フラグメントページがロードされた後、この関数は起動し、次の 2 つのことを行います。

- main.html フラグメントのタイトルテキストを変更します。main.html は静的に編集できるだけですが、静的に編集されたフラグメントファイルの内容を動的に変更することができる方法を示しています。
- button1.html フラグメントを main.html 内の同じ ID を持つ タグにロードします。mainContent は indexbs.html ファイルをブラウザに再ロードします。画面には以下の様に表示されるはずです。





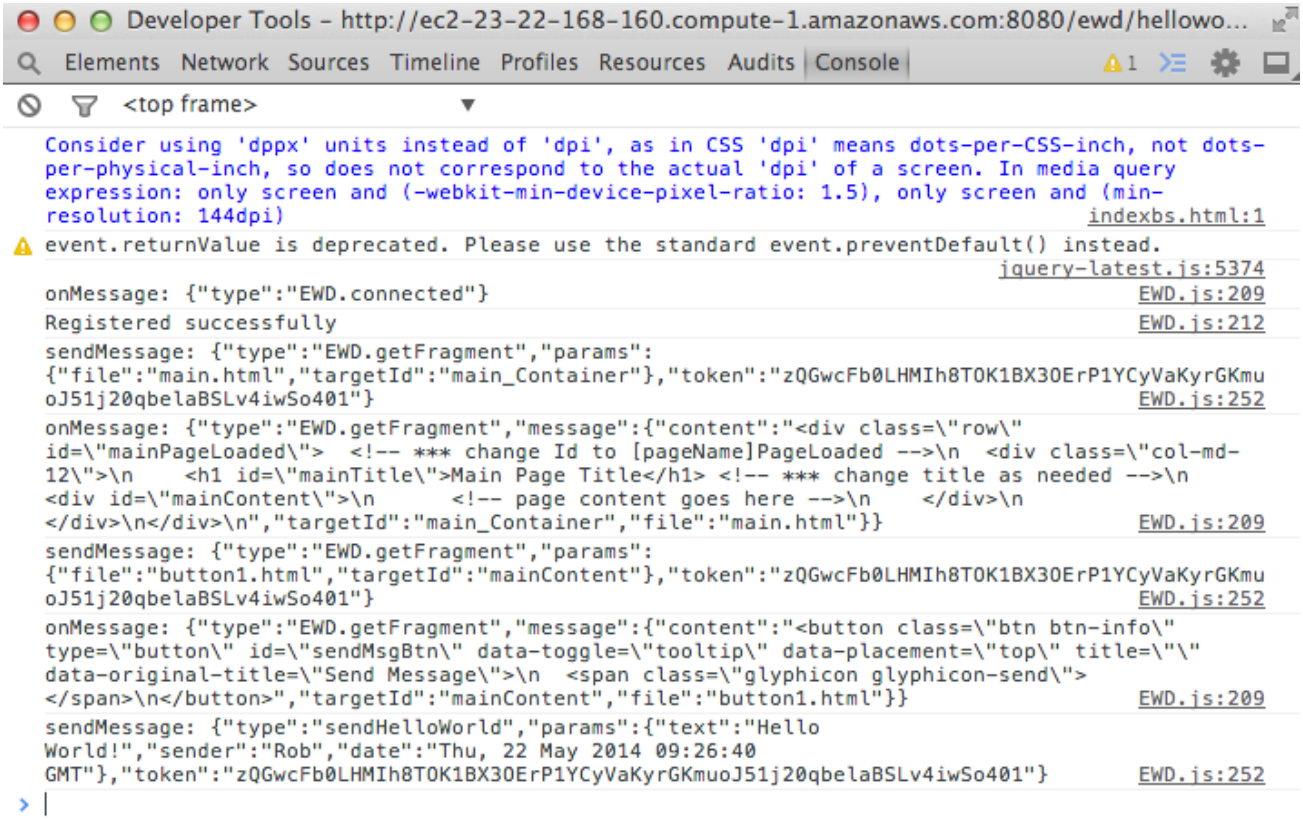
素敵な Bootstrap 3 ボタンがありますが、これから二つのことを追加する必要があります。

- ツールチップを含んだ button タグの作成。ツールチップが有効になっていないため、まだ表示されません。
- ボタンをクリックした時、バックエンドに sendHelloWorld メッセージを送信する必要があります。

button1.html フラグメントがロードされ、実行開始された後に別の onFragment ハンドラ関数を追加することにより、次の二つのことを行うことができます。太字で示されているように app.js の onFragment セクションを編集します。

```
  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()
          }
        });
      });
    }
  },
```

indexbs.html ページを再読み込みしてみてください。ボタンの上にポインタを置くと、ツールチップが表示されるはずです。ボタンをクリックすると、オリジナルの基本的なデモアプリケーションと同じように、sendHelloWorld メッセージを送信します。Google chrom の JavaScript コンソールには、次のように表示されます。



バックエンドから戻ってくる応答メッセージはありませんが、最後のところは、silent fire-and-forget ハンドラであるとことに注目して下さい。　動作していることを証明するために、に node_modules ディレクトリにある helloworld.js ファイルを編集してみましょう。

```
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};
```

```
    }
  }
};
```

app.js 内の onMessage オブジェクトに以下を追加します。

```
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);
  }
}
```

試してみてください。保存ボタンをクリックすると、メッセージが保存されたことを確認するトースター（ポップアップ）メッセージが右上隅に表示されるはずです。toastr ユーティリティは EWD.js/Bootstrap3 フレームワークで、事前ロードするウィジェットの一つです。

Bootstrap3 を使用したデータの取得方法

最後に、保存したメッセージを取得する２番目のボタンを追加します。ボタンが送信ボタンを一度クリックした後にのみ表示されるようにしてみましょう。私たちがこれを行うにはいくつかの方法がありますが、次の手順を実行する事にしてみましょう。　以下の button1.html の太字で示されている所を編集します。

```
<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>
```

app.js の強調文字部分を追加します:

```
 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));
    }
  }
};
```

試してみましょう。最初のボタンをクリックすると第 2 ボタンが表示されます。第二のボタンを

クリックすると、保存されたメッセージを検索し、トースター（ポップアップ）表示します。

ナビゲーションタブの追加

それでは EWD.js.の Bootstrap 3 のフレームワークに含まれている別の機能を試してみましょう。

最初に、bootstrap3 アプリケーションディレクトリに navlist.html という名前のファイルを見つけ、それを helloworld アプリケーションディレクトリにコピーします。 Main と About という 2 つの定義されたタブが用意されています。すでに動作するメインページを持っていますが、動作させるには bootstrap3 アプリケーションディレクトリ内の about.html を探し、helloworld アプリケーションディレクトリにコピーする必要があります。

indexbs.html ファイル内に、about フラグメント用のプレースホルダコンテナ div を見つけることができます。 app.js ファイル内の navFragments オブジェクト中に、Main と About フラグメントをナビゲーションタブで動作させるために何が必要なのかを見ることができます。 どちらの場合も、それらは最初のフェッチの後、キャッシュされますので、2 回目以降のナビゲーション･タブのクリックは再びそれをフェッチすることなく、ページ内の既存のコンテンツを表示します。

ナビゲーションタブを有効にするために app.js に二つの簡単な変更を加える必要あります。

- アプリケーション起動時の navlist.html フラグメントのフェッチ（onStartup）
- Navlist フラグメントがロードされた後の nav のタブの活性化

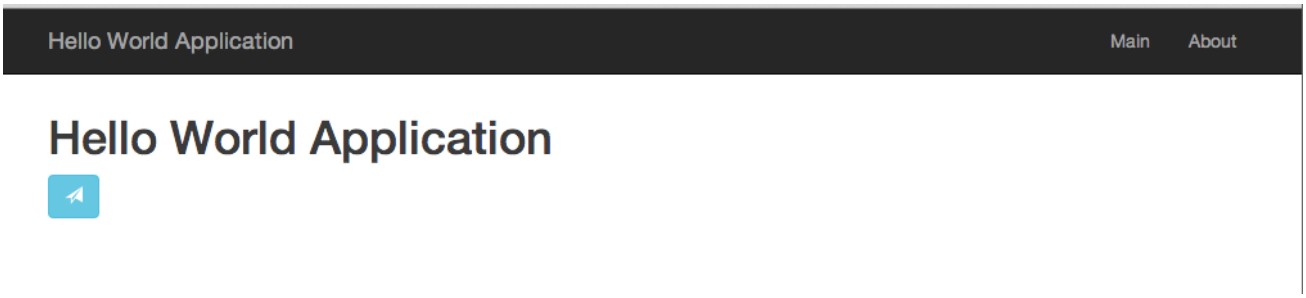
app.js の強調文字部分を変更するだけです。

```
 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: {
```

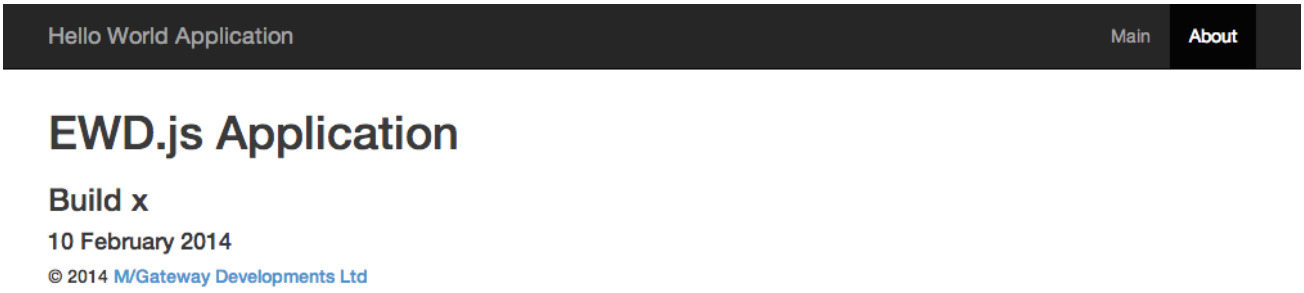
```
      cache: true
    },
    about: {
      cache: true
    }
  },
  onStartup: function() {
    // remove everything in here apart from this line:
    EWD.getFragment('main.html', 'main_Container');
    EWD.getFragment('navlist.html', 'navList');
  },
  onPageSwap: {
    // add handlers that fire 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();
    }
  }
};
```

試してみましょう。 2 つのナビゲーション· タブが表示されます。



[About]タブをクリックすると、その内容が表示されます。



お好みに合わせて about.html の内容を変更編集してください。

自由に、ナビゲーションオプションを追加することができます。navlist.html に新しいものを単純にコピーして貼り付けます。それらがロードされる indexbs.html 内のプレースホルダのコンテナがあり、それらは app.js ファイル内の onPageSwap オブジェクトから追加することを確認してください。
キャッシングは非常に単純なルールによって決定されます。フラグメントの最初のタグは、フォームの ID を持つ必要があります。

[navName]PageLoaded

例として、main.html と about.htm を参照してください。それらの最初のタグは、それぞれ mainPageLoaded と aboutPageLoaded の ID を持っているのが判ります。

結語

これでこのチュートリアルは終了しました。　あなたは今やどのように EWD.js アプリケーションが動作し、どうしたら構築できるかを理解することが出来たことでしょう。また、レスポンシブ Bootstrap 3 アプリケーションを構築するために必要な基本的なほとんどの知識を体得できたと思います。より高度な技術については、EWD.js.に含まれている ewdMonitor· アプリケーションのソース· コードを調べて下さい。

EWD.js 使用したアプリケーションの開発をお楽しみください！

EWD.js リファレンスガイド　(bild 0.67)

2014 年 10 月 23 日　第 1 版　発行

原著者	Rob Tweed	M/Gateway Developments
監修	土屋喬義	日本 M テクノロジー学会
訳者	土屋喬義	日本 M テクノロジー学会
	西山強	日本 M テクノロジー学会
	伊藤章	日本 M テクノロジー学会

発行者　土屋喬義