EWD.js

Guide de référence

Traduction française non-officielle par Yrelay / Un-official French translation by Yrelay
http://gradvs1.mgateway.com/download/EWDjs.pdf

Table of Contents

Introduction

Contexte

EWD.js est un framework Open Source, basé sur Node.js / JavaScript pour construire des applications basées sur un navigateur de haute performance qui intègrent avec MongoDB et/ou des bases de données Mumps (par exemple, Caché, GlobalsDB et GT.M). Les environnements EWD.js fonctionnent comme des applications Node.js en serveur/conteneur : ce qui représente un équivalent JavaScript d'Apache Tomcat.

EWD.js utilise une approche complètement nouvelle pour les applications basées sur un navigateur, et utilise WebSockets comme moyen de communication entre le navigateur et le Node.js intermédiaire. EWD.js nécessite le module Node.js *ewd.js*. *ewd.js* fournit les fonctionnalités suivantes :

- il agit comme un serveur web pour servir des contenus statiques
- il fournit le niveau WebSockets backend
- il gère et maintient un pool de processus fils à base de Node.js, dont chacun est responsable de l'intégration avec le MongoDB et/ou base de données Mumps que vous avez choisi d'utiliser
- il crée, gère et maintient des sessions utilisateur
- il projette une base de données Mumps comme une base de données native JSON, tels que le stockage de données Mumps qui peut être traitée comme un stockage persistant JSON. Vous pouvez également utiliser EWD.js avec MongoDB, soit exclusivement ou conjointement avec une base de données Mumps. Dans cette dernière configuration, vous pouvez laisser la base de données Mumps fournir de très haute performance de gestion de session/persistance, tout en utilisant MongoDB pour tous vos autres besoins de base de données.
- il gère toute la sécurité nécessaire pour protéger votre base de données à partir de l'utilisation non autorisée
- il fournit un environnement de serveur d'application dans lequel les applications multiples EWD.js peuvent être exécutées simultanément, et dans laquelle les applications peuvent être éditées et changées sans la nécessité pour le serveur d'application pour être redémarré

Les applications EWD.js sont entièrement écrits en JavaScript, tant pour leur front-end et back-end logique. Aucune connaissance du langage Mumps n'est nécessaire. Toutefois, si vous utilisez Cache ou GT.M, vous pouvez accéder et invoquer hérités les fonctions Mumps à partir de votre back-end JavaScript logique si nécessaire.

Sur le fond de la pensée et de la philosophie de EWD.js et les capacités uniques de la base de données Mumps, voir les nombreux articles de blog sur http://robtweed.wordpress.com/. Ce document explique comment installer et utiliser EWD.js.

Si vous voulez créer rapidement un système EWD.js, essayer de lire les annexes 1 à 3.

Si vous êtes nouveau sur EWD.js vous devez également passer un peu de temps à lire et suivre le tutoriel étape par étape à l'Annexe 4 : il vous emmène à travers le processus de construction d'un exemple simple de EWD.js « Hello World ». Il devrait vous aider à vous familiariser avec les concepts et mécanique derrière EWD.js, et apprécier la facilité avec laquelle une base de données Mumps stocke et récupère les documents JSON. Si vous préférez utiliser MongoDB, vous devriez consulter l'annexe 6.

Installation & Configuration

Prérequis et Contexte

Un environnement EWD.js se compose des éléments suivants:

- Node.js
- MongoDB et/ou une base de données Mumps, par exemple :
  - GT.M
  - GlobalsDB
  - Caché
- Le module *ewd.js* pour Node.js

EWD.js peut être installé sur Windows, Linux ou Mac OS X. Caché ou GlobalsDB peut être utilisé comme la base de données sur l'un de ces systèmes d'exploitation. GT.M ne peut être utilisé sur les systèmes Linux. MongoDB est disponible pour tous les trois systèmes d'exploitation.

Si vous souhaitez utiliser une base de données Mumps et la nécessité de choisir lequel utiliser, garder ceci à l'esprit :

- GlobalsDB est logiciel libre, mais fermée. Cependant, il n'a pas de limites en termes de son utilisation ou de re-distribution. Il est essentiellement le moteur de base de données de base Mumps de Caché. Des versions sont disponibles pour Windows, Mac OS X et Linux. Aucun support technique ou de maintenance n'est disponible pour GlobalsDB InterSystems : il est prévu pour une utilisation «tel quel». GlobalsDB est la plus rapide et la plus simple des bases de données Mumps à installer, et est probablement la meilleure option à utiliser si vous êtes nouveau sur EWD.js. Lire l'Annexe 1, qui décrit comment créer une version de travail EWD.js entièrement basé sur GlobalsDB en quelques minutes.
- GT.M est un logiciel Open Source, produit gratuit, de puissance industrielle. Il est limité aux systèmes Linux. Lire l'annexe 2, qui décrit comment créer une version de travail EWD.js entièrement basé sur GT.M en quelques minutes. Si vous êtes intéressé pour utiliser EWD.js avec Open Source Electronic Healthcare Record (EHR) appelé Vista, alors vous pouvez télécharger la machine virtuelle EWDrop (http://www.fourthwatchsoftware.com/). Cela vous donnera avec un système pré-construit, pré-configuré avec GT.M qui comprend une version de de travail Vista. Suivez les instructions à l'annexe 3 pour obtenir EWD.js installé et opérationnel sur une VM EWDrop.
- Caché est un produit de licence commerciale, de puissance industrielle, avec des versions disponibles pour Windows, Mac OS X et Linux. EWD.js exige uniquement son moteur de base de données Mumps, mais si vous êtes déjà un utilisateur Caché, vous pouvez exécuter un code ou des classes déjà existantes au sein du code JavaScript de vos applications EWD.js back-end. EWD.js fournit donc un grand (et beaucoup plus simple et léger) alternative à la CSP et les frameworks web Zen qui viennent avec Caché, et utilise vos licences de cache disponible beaucoup plus efficacement.

Pour plus de détails au sujet de ces bases de données, voir :

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

Tous les trois bases de données sont extrêmement rapides, et les deux Caché et GT.M peuvent évoluer à très grands niveaux de l'entreprise. L'interface Node.js pour GlobalsDB et Caché compte actuellement environ deux fois la vitesse de performance que l'interface modem pour GT.M.

Notez que lorsque vous utilisez EWD.js, tous les trois bases de données Mumps semblent être identiques pour la façon dont vous accédez et manipuler des données à partir de votre application. La seule différence que vous devez connaitre est une petite différence dans les paramètres de configuration dans le fichier de démarrage vous allez utiliser pour le module de ewd.js (voir plus loin). Sinon, EWD.js applications que vous développez pour, disons, Caché, devrait normalement travailler, sans aucune modification de codage sur un système de GT.M.

Si vous souhaitez utiliser EWD.js avec MongoDB, vous avez deux choix :

- Vous pouvez utiliser MongoDB exclusivement comme la seule base de données. Si vous décidez d'utiliser cette approche, puis EWD.js va gérer et maintenir les sessions utilisateur en utilisant une émulation de MongoDB globales Mumps - toutes les API basées sur des bases de données JSON Mumps qui sont décrits dans le présent document fonctionnera à l'identique avec MongoDB. Cependant, vous devez être conscient que la performance de cette émulation est actuellement nettement plus lente que si vous utilisez une base de données Mumps. Pourvu que vous utilisez seulement une quantité très limitée de stockage dans une session EWD.js, la performance devrait être suffisante pour la plupart des applications de petite et moyenne taille. Bien sûr, EWD.js vous permet d'utiliser et de l'accès MongoDB de manière standard, comme les collections persistantes d'objets JSON, avec le même niveau de performance que vous pourriez normalement attendre de MongoDB.
- Sinon, vous pouvez utiliser une approche hybride, utilisant une base de données Mumps pour assurer la gestion de la session de très haute performance, et vous permettant d'utiliser MongoDB pour toutes les autres activités de base de données. Dans ce mode, vous utiliserez MongoDB dans sa façon normale, comme les collections persistantes d'objets JSON, avec le même niveau de performance que vous auriez normalement attendre de MongoDB. Dans cette opération hybride, vous pouvez également accéder aux données Mumps existantes dans le même temps que le maintien de données MongoDB : utile pour moderniser et d'étendre les anciennes applications de soins de santé.

En fonction de votre choix de navigateur framework JavaScript, vous aurez bien sûr besoin d'installer également (par exemple, ExtJS, jQuery, Dojo, etc.).

Interface Node.js pour les bases de données Mumps

L'interface Node.js est disponible pour tous les trois bases de données Mumps.

Caché & GlobalsDB

InterSystems fournit son propre fichier d'interface pour Caché et GlobalsDB. Le même fichier est effectivement utilisé pour les deux produits, et ils peuvent être librement échangées. Vous verrez que les versions du fichier d'interface pour les versions les plus récentes de Node.js sont inclus avec la dernière version de GlobalsDB. Si vous voulez utiliser Caché avec la version 0.10.x Node.js, par exemple, puis téléchargez et installez une copie de GlobalsDB (il représente

un très petit téléchargement et un processus d'installation très simple) sur une machine de test et copiez le fichier dont vous avez besoin sur votre système de Caché.

Les fichiers d'interface sont inclus avec Caché 2012.x et sur les versions suivantes, mais, en raison des cycles de publication InterSystems, ces fichiers ont tendance à être à jour. En outre, les fichiers d'interface Node.js peuvent être utilisés avec presque toutes les versions de cache, y compris ceux datant d'avant 2012.x, donc vous devriez être en mesure d'utiliser avec la plupart des version de EWD.js de Caché.

Vous trouverez les fichiers d'interface dans le répertoire bin d'une installation Caché et GlobalsDB : ils sont nommés cache *[nnn] .node* où *nnn* indique la version Node.js. Par exemple, *cache0100.node* est le dossier pour la version Node.js 0.10.x. Ce fichier doit être copié à l'emplacement approprié (voir plus loin) et renommé *cache.node*.

GT.M

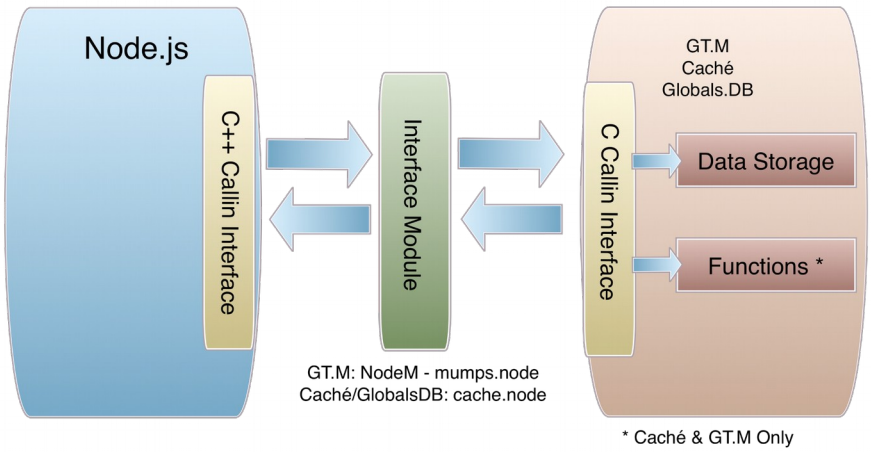
Une interface Node.js Open Source pour GT.M, nommé *NodeM*, a été créée par David Wicksell. Vous le trouverez sur https://github.com/dlwicksell/nodem. Ceci est entièrement compatible avec les API fournies par l'interface InterSystems pour Node.js.

La VM EWDrop comprend déjà NodeM. Sinon, suivez les instructions d'installation sur la page Github.

Le fichier d'interface NodeM est nommé *mumps.node*.

Architecture

L'architecture de l'interfaçage de la base de données Node.js/Mumps est comme ci-dessous :



Bien qu'il existe des versions totalement asynchrone de l'API dans les deux versions de l'interface, EWD.js utilise les versions synchrones, car ceux-ci sont beaucoup plus simples à utiliser par les développeurs et sont nettement plus rapide que leurs homologues asynchrones. Cela peut sembler aller à l'encontre de la sagesse acceptée pour l'accès Node.js/de base de données, mais, comme vous le verrez plus tard, l'architecture du module *ewd.js* (sur lequel EWD.js dépend) veille à ce que cet accès synchrone est pas réellement le problème qu'il pourrait sembler à première vue.

Interface Node.js pour MongoDB

La demande file d'attente/embranchement des processus enfants de l'architecture du module ewd.js sous-entendu EWD.js (voir la section ci-dessous) est délibérément conçu pour permettre aux développeurs d'applications de base de données à utiliser la logique synchrone, tout en atteignant les nombreux avantages de Node.js. Pour cette raison, EWD.js utilise un module d'interface synchrone pour MongoDB qui a été spécialement développé par M/Gateway Developments Ltd. Il fournit un conteneur (wrapper) personnalisé autour des API MongoDB standard, de sorte toutes les fonctionnalités de MongoDB habitude soit disponible pour vous, et il se connecte à MongoDB de façon standard : via TCP à une adresse et port spécifié (localhost et le port 27017 par défaut comme d'habitude).

La principale différence est que vous pouvez écrire d'une manière logique les manipulations de votre base de données en utilisant la norme, soit une logique synchrone intuitive, plutôt que la logique asynchrone habituelle qui est la norme quand on travaille avec Node.js.

Premières étapes de la configuration

Voir les annexes 1 à 3 à la fin de ce document pour obtenir des instructions sur la construction automatiquement un certain nombre de configurations de référence - ceux-ci sont recommandés pour la plupart des utilisateurs, en particulier ceux qui découvrent EWD.js et les les bases de données Mumps.
Si vous avez besoin d'installer et de configurer manuellement EWD.js (par exemple pour un environnement personnalisé), les premières étapes de la création d'un environnement EWD.js sont :

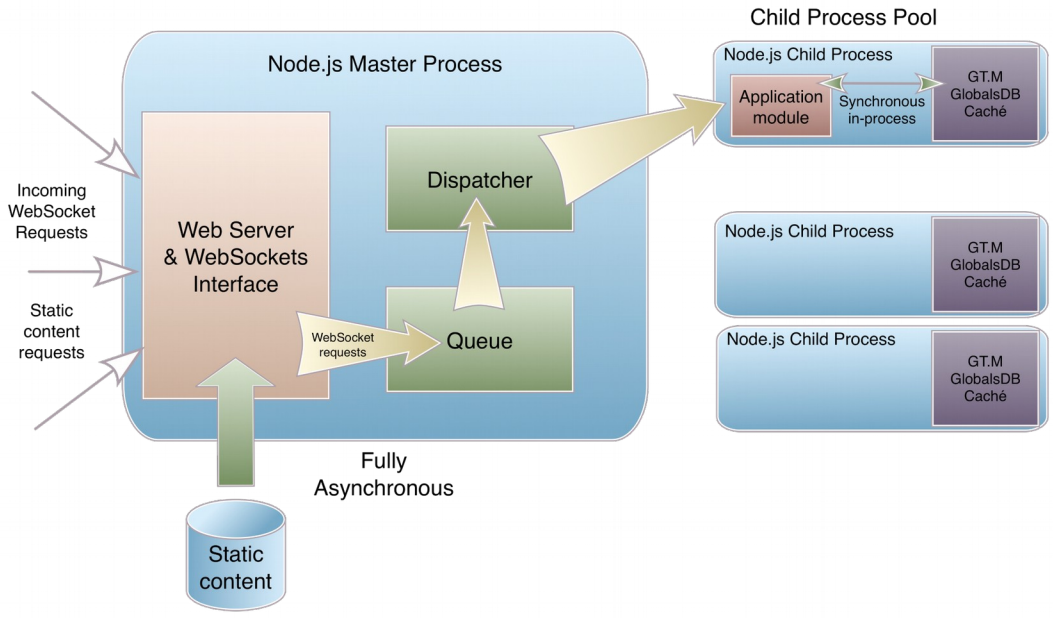
- Installez Node.js : Voir http://nodejs.org/ EWD.js peut être utilisé avec la plupart des versions de Node.js, mais la dernière version est recommandée (0.10.x au moment de la rédaction).
- Installez MongoDB et/ou votre base de données Mumps choisie : voir le site web pour plus de détails pertinents.
- Installer/configurer l'interface Node.js approprié pour votre base de données Mumps.

ewd.js

Le module de *ewd.js* pour Node.js fournit l'environnement d'exécution pour EWD.js. Il est publié et maintenu avec une licence de projet Apache, Voir https://github.com/robtweed/ewd.js

Architecture ewd.js

L'architecture du module de *ewd.js* est résumé dans le schéma ci-dessous.



Le processus maître Node.js agit comme un serveur web-serveur et websocket. Lorsqu'il est utilisé avec EWD.js, les demandes du serveur Web sont limitées aux demandes de contenu statique. Dans votre fichier de démarrage/configuration *ewd.js* spécifier une taille d'expansion : le nombre de processus enfants Node.js qui sera démarré. Chaque processus enfant créera automatiquement une connexion à MongoDB et/ou un processus de base de données Mumps, ceci tant l'interface InterSystems et NodeM créeront des connexions sur la base de données. Notez que si vous utilisez Caché, chaque processus enfant consomme une licence de Caché. Si vous utilisez MongoDB, la connexion utilise la connexion basé sur TCP standard, mais les API sont synchrones.

Le messages WebSocket entrants de votre application sont placés sur une file d'attente qui est traitée automatiquement : si un processus enfant Node.js est libre (ie ne traite pas encore les messages websocket), un message websocket de file d'attente est envoyé par le traitement. *ewd.js* assure qu'un processus enfant ne traite un message de websocket qu'unique à un moment en le supprimant automatiquement de la file d'attente disponible dès qu'il reçoit un message à traiter. Dès que le processus est terminé, le processus fils est renvoyé immédiatement et automatiquement à la file d'attente du processus enfant disponible.

Le traitement des messages WebSocket est effectué par votre propre application, écrite par vous, en JavaScript, comme un module Node.js. Votre module a un accès complet à MongoDB et/ou votre base de données Mumps, et, si vous utilisez GT.M ou Caché, peut également invoquer les fonctions Mumps héritées.

En découplant l'accès à la base de données Mumps du processus frontal maître Node.js, *ewd.js* réalise deux choses :

- il permet d'utiliser une architecture à une échelle appropriée, et les processus enfants peuvent utiliser les processeurs multi-core.
- il permet d'utiliser des API synchrones dans Node.js/Mumps ou l'interface Node.js/MongoDB, rendant le code plus facile et plus intuitive pour le développeur de l'application, et profitant du fait que les API synchrones des résultats significativement plus vite que les asynchrones. La logique de manipulation de base de données complexe devient très simple, sans aucun besoin de rappels profondément imbriqués. Vous pouvez néanmoins utiliser le codage asynchrone standard pour toute autre logique au sein de votre module de back-end - EWD.js vous donne vraiment le meilleur de tous les mondes.

EWD.js est spécialement conçu pour vous offrir un cadre pour développer des applications qui font usage du module *ewd.js*.

Installation de ewd.js

L'installation de EWD.js est très simple et largement automatisé pour la plupart des configurations. Voir les annexes 1 à 3 à la fin de ce document.

Si vous avez besoin pour créer votre propre installation personnalisée qui diffère de celles couvertes par les appendices, cette section explique les « modules mobiles » et la façon dont ils doivent être installés.

Vous devez utiliser le gestionnaire de package Node (*npm*) qui est installé en même temps que Node.js.

Remarque: si vous n'avez pas utilisé Node.js avant, c'est un produit que vous gérer et contrôler principalement en ligne de commande. Ainsi, vous aurez besoin d'ouvrir soit une fenêtre de terminal Linux ou OS X, ou une fenêtre invite de commande Windows pour effectuer les étapes décrites ci-dessous.

Avant d'installer *ewd.js*, vous devez vous assurer que vous êtes dans le chemin du répertoire où vous voulez installer EWD.js. Cela dépendra de votre système d'exploitation, de la base de données Mumps et de votre choix personnel. Il est recommandé de créez un sous-répertoire nommé *ewdjs* sous le chemin du répertoire de votre choix, de naviguer dans ce répertoire puis invoquez *npm* pour installer le module de *ewd.js*. Nous devons indiquer que le répertoire *ewdjs* est le répertoire Home de EWD.js. Par exemple :

Windows

Créez le répertoire *ewdjs*, par exemple, depuis votre lecteur C et l'installer à partir de ce répertoire, par exemple :

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

Votre répertoire d'origine pour EWD.js est *c:\ewdjs*

Mac OS X or Linux

Créez le répertoire *ewdjs*, par exemple, depuis votre répertoire home et l'installer à partir de ce répertoire, par exemple :

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

Votre répertoire Home pour EWD.js est *~/ewdjs*

Pour toutes les plates-formes

Pendant le processus d'installation, vous serez invité à confirmer le chemin dans lequel EWD.js a été installé. Juste accepter la valeur par défaut qu'il suggère en appuyant sur la touche Entrée, à savoir:

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

Les sous-éléments essentiels de EWD.js seront installés par rapport à ce chemin.

Il vous sera alors demandé si vous souhaitez installer en option les sous-composants, supplémentaires pour EWD.js :

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

Si vous êtes nouveau sur EWD.js, tapant *Y* (suivie de la touche *Entrée*) option recommandée,

Dans tous les cas, après que *npm* ait fini d'installer le module de *ewdjs*, vous verrez que un sous-répertoire nommé *node_modules* a été créé sous votre répertoire Home de EWD.js. Il contient le module *ewd.js* et les fichiers de support et des modules requis par EWD.js

Configuration de l'environnement EWD.js

Si vous avez utilisé l'un des scripts d'installation automatisés pour EWD.js (comme décrit dans les annexes 1 à 3), l'environnement EWD.js aura été mis en place pour vous, dans ce cas, vous pouvez ignorer cette section. Si vous voulez en savoir plus sur la façon dont l'environnement EWD.js est créé et/ou besoin de créer une installation personnalisée, cette section fournira les informations souhaitées.

Comme décrit dans la section précédente, lorsque vous avez installé EWD.js, le script d'installation vous demande d'entrer/confirmer le chemin qui a fourni la base pour déplacer des fichiers supplémentaires essentiels et facultatifs. Vous trouverez maintenant sous votre répertoire Home de EWD.js la structure répertoire suivante :

```
- node_modules
    o ewdjs (répertoire contenant le module ewd.js)
    o un certain nombre de fichiers de module Node.js

- www
   o ewdjs (répertoire contenant les composants de EWD.js qui fonctionnent dans le
navigateur)

   o ewd (répertoire contenant un certain nombre d'application pré-construite. Tout
processus côté navigateur, JavaScript et CSS pour vos applications résideront dans ce
répertoire)

   o respond (Contient des fichiers utilisés pour le support IE des applications
EWD.js/Bootstrap)

   o services (peut contenir des fichiers de micro-service front-end)

- ssl (Ce répertoire est l'endroit où vous devez copier votre clé SSL et fichier de
certificat si nécessaire. Si vous installez les extras en option, un certificat auto-
signé est installé pour l'utiliser dans vos tests)
   o ssl.key

   o ssl.crt

  Si vous installez les options disponibles, vous trouverez un ensemble de fichiers de
démarrage d'exemple dans le répertoire Home, par exemple:

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

Utilisation de la structure de répertoire EWD.js

La structure de répertoire que EWD.js a créé (voir section ci-dessus) doit être utilisé comme suit :

~/ewdjs (votre répertoire Home EWD.js) : Vos fichiers de démarrage EWD.js doit résider ici. Si vous avez besoin d'installer des modules supplémentaires Node.js, vous devez exécuter *npm* au sein de ce répertoire.

~/ewdjs/node_modules: Cette est l'endroit où tous les modules de back-end pour vos applications EWD.js doivent résider. Vous en aurez déjà présents quelques exemples tels que ewdMonitor.js qui fournissent la logique back-end pour l'application Gestionnaire EWD.js/Monitor. Tous les autres modules Node.js qui sont requis par vos applications devraient également être installés dans ce répertoire (voir note ci-dessus).

~/ewdjs/ssl: Utilisez ce répertoire pour vos fichiers de clés et de certificats SSL (si nécessaire). EWD.js installe un certificat auto-signé que vous pouvez utiliser pour tester EWD.js avec SSL. Remplacer ces fichiers avec ceux appropriés pour une utilisation en production.

~/ewdjs/www: Ce répertoire agit comme le chemin racine du serveur web pour EWD.js. Tous les fichiers, chemins et sous-chemins dans ce répertoire seront accessibles depuis les navigateurs et les clients distants. Vous devez installer les bibliothèques JavaScript que vous souhaitez utiliser dans vos applications au sein de ce répertoire, par exemple *~/ewdjs/www/bootstrap-3.0.0*

~/ewdjs/www/ewdjs*:* Ce répertoire contient les fichiers JavaScript qui créent l'environnement EWD.js run-time dans le navigateur. Ne pas modifier l'un de ces fichiers qui sont inclus dans l'installation de EWD.js.

~/ewdjs/www/ewd: Toutes vos applications EWD.js résident dans ce répertoire. Lors de l'installation un certain nombre de ceux prédéfinis auront déjà été créées pour vous (par exemple ewdMonitor, VistADemo). Vous pouvez ajouter autant d'autres sous-répertoires d'applications que vous le souhaitez. Chaque sous-répertoire de l'application contient :

- un fichier principal de "conteneur" HTML, normalement nommé *index.html*
- facultativement, un ou plusieurs "fragment" fichiers HTML
- un fichier JavaScript qui définit les fonctionnalités dynamique côté navigateur. Ce fichier est normalement nommé *app.js*
- éventuellement d'autres fichiers JavaScript et / ou les fichiers CSS requis par l'application.

~/ewdjs/www/respond: Ce répertoire est créé pour vous lors de l'installation et contient les fichiers JavaScript qui sont utilisés pour le support Bootstrap sur les navigateurs Internet Explorer.

~/ewdjs/www/services: Ce répertoire est créé pour vous lors de l'installation et contient un exemple frontal Microservice JavaScript module.

Installation et configuration des interfaces de base de données

GlobalsDB
Lorsque vous installez GlobalsDB, vous trouverez les fichiers de module d'interface dans le répertoire */bin* de l'installation - il est nommé *cache0100.node*. Vous devez copier ce fichier dans le répertoire *~/ewdjs/node_modules* et, **surtout le renommer** en *cache.node*.

Caché
Selon la version de cache que vous utilisez, vous pouvez ou ne pouvez pas être en mesure de trouver un fichier *cache0100.node* dans votre */Cache/bin*. S'il est présent, vous devez le copier dans le répertoire *~/ewdjs/node_modules* et, **surtout le renommer** en *cache.node*.
S'il est absent, alors la meilleure chose à faire est d'installer une copie de GlobalsDB quelque part sans importance, par exemple sur une machine virtuelle
- Faire juste attention que ce soit pour la même plateforme que votre système Caché. Trouver le fichier *cache0100.node* dans le répertoire*/globalsdb/bin*. Vous devez copier ce fichier dans le répertoire */globalsdb/bin* et, **surtout le renommer** en *cache.node*. Le fichier *cache0100.node* de GlobalsDB devrait fonctionner avec la plupart des premières versions de Caché.

GT.M
Si vous n'avez pas utilisé le programme d'installation automatisé pour GT.M, vous devez installer le module NodeM :

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

Vous aurez besoin de configurer NodeM. Si vous avez utilisé les installateurs automatisés pour GT.M, alors cela aura été fait pour vous. Sinon, vous pouvez jeter un œil dans le second script installation pour voir ce qui est nécessaire : https://github.com/robtweed/ewd.js/blob/ master/gtm/install2.sh

En résumé. Les étapes sont les suivantes :

- Dans le répertoire *~/ewdjs/node_modules/nodem/lib*, vous trouverez un certain nombre de fichiers : ce sont des modules d'interface pour les différentes versions de Node.js pour 32 bits et Linux 64 bits. Si vous utilisez Linux 32 bits, vous devez sélectionner les fichiers *mumps10.node.i686*. Si vous utilisez Linux 64 bits, vous devez sélectionner les fichiers *mumps10.node.x8664*. Renommez le fichier sélectionné à *mumps.node*. Assurez-vous que *mumps.node* réside toujours dans le *~/ewdjs/node_modules/nodem/lib*.

- Dans le répertoire */usr/local/lib, créer* un lien symbolique vers le fichier *libgtmshr.so* que vous trouverez dans votre répertoire d'installation de GT.M, par exemple :

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

- Assurez-vous que la variable d'environnement GTMCI est créée et pointe vers le chemin *~/ewdjs/node_modules/nodem/resources/calltab.ci*

- Assurez-vous que la variable *d*'environnement *gtmroutines* est étendue pour inclure le chemin *~/ewdjs/node_modules/nodem/src*

Remarque : pour les deux dernières étapes, si vous le souhaitez, vous pouvez le faire en exécutant les fichiers de module *dewdrop-config.js*. Voir plus tard.

Fonctionnement de ewd.js

Démarrage de ewd.js

Vous pouvez maintenant démarrer l'environnement *ewd.js*. Vous le faites en utilisant le fichier *ewdStart*.js* qui est approprié à la base de données Mumps et à l'OS vous utilisez, par exemple:

- GlobalsDB sur Linux ou Mac OS X : *ewdStart-globals.js*

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

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

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

- GT.M, installé conformément à l'annexe 2 : *ewdStart-gtm.js*

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

- GT.M fonctionne dans la VM dEWDrop conformément à l'annexe 3 :*ewdStart-gtm.js*

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

- D'autres fichiers de démarrage ont été créés pour être utiliser ou personnaliser de manière appropriée :

  ewdStart-cache-win.js: EWD.js + Windows + Caché pour Windows

  ewdStart-pi: EWD.js fonctionne sur Raspberry Pi (voir l'annexe 5)

  ewdStart-mongo.js: EWD + Windows + MongoDB (en utilisant l'émulation MongoDB de globales Mumps)

- Si vous souhaitez utiliser MongoDB dans un environnement hybride, voir l'annexe 6. Essentiellement l'environnement EWD.js est axé sur la base de données Mumps/OS que vous utilisez, et MongoDB peut ensuite être ajouté pour la totalité ou certaines applications EWD.js que vous créées.

- Si vous utilisez une configuration différente de celle que je viens de décrire, vous aurez besoin pour créer une copie du fichier de démarrage JavaScript approprié et de le modifier de manière appropriée. Si vous utilisez le Caché ou GlobalsDB, utilisez *ewdStart-globals.js* ou *ewdStart-globalswin.js* comme votre point de départ. Si vous utilisez GT.M, utiliser *ewdStart-gtm.js* comme votre point de départ.

Définition des paramètres de démarrage EWD.js

Si vous regardez l'un des fichiers de démarrage EWD.js décrites ci-dessus, vous verrez que tous suivent un modèle commun :

- Le module de *ewd.js* est chargée en utilisant *require ()*
- Un objet de paramètre de démarrage est défini
- La méthode ewd.js *start ()* est appelée, en passant l'objet de paramètre de démarrage comme argument.

Lorsque EWD.js est démarré (en invoquant sa méthode *start ()*), il crée d'abord un ensemble de paramètres de démarrage par défaut. Toutes les valeurs définies dans votre objet paramètre de démarrage sont ensuite utilisés pour remplacer les valeurs par défaut. Dans la plupart des situations, vous pouvez accepter la majorité des valeurs de paramètres EWD.js par défaut , et il vous suffit de spécifier un petit nombre de valeurs qui sont spécifiques à votre environnement. Vous pouvez remplacer l'une des valeurs par défaut qui sont définis comme suit :

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

Ainsi, par exemple, pour démarrer EWD.js à l'aide de SSL et d'écouter sur le port 8081, vous auriez à définir les valeurs des paramètres de démarrage suivantes :

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

Toutes les autres valeurs de démarrage resteraient identique à leurs valeurs par défaut.

La signification des paramètres de démarrage disponibles est la suivante.

- **database.type** : GTM | cache | MongoDB (Remarque : la valeur *Caché* est utilisée à la fois pour Caché et GlobalsDB)

- **http Port** : le port sur lequel le serveur Web EWD.js écoutera
- **https.enabled** : true | false
- **https.keyPath** : le chemin pour votre fichier de clé SSL si *https.enabled* est vrai
- **https.certificatePath** : le chemin pour votre fichier de certificat SSL si *https.enabled* est vrai
- **poolSize** : le nombre de processus fils qui sera pré-définit quand EWD.js démarre. L'application EWDMonitor peut être utilisé pour ajuster ce paramètre durant le fonctionnement de EWD.js.
- **webServerRootPath** : le chemin physique qui est mappé par EWD.js à la racine du serveur Web. Notez que tous les fichiers et sous-répertoires de ce chemin sont accessibles depuis les navigateurs et HTTP(S) clients. Il est recommandé d'utiliser le paramètre par défaut dans la plupart des circonstances
- **webSockets.externalListenerPort** : le port TCP sur lequel EWD.js écoutera les messages entrants externes. Mettre à *false* pour désactiver ce port et sa fonctionnalité
- **logTo** : la destination pour les informations d'enregistrement de EWD.js (console | fichier)
- **logFile** : si *logTo* est réglé sur un fichier, cela définit le nom du fichier dans lequel les journaux sont écrits par EWD.js
- **traceLevel** : le niveau de détail de la journalisation de EWD.js. 0 = none, 1 = low, 2 = medium, 3 = verbose
- **modulePath** : le chemin dans lequel les fichiers de module EWD.js résident. Il est utilisé pour construire le chemin *require()* pour vos modules d'application back-end. Il est fortement recommandé d'utiliser le paramètre par défaut dans la plupart des circonstances
- **monitorInterval** : utilisé par l'application EWDMonitor pour spécifier le nombre de millisecondes entre chaque mise à jour de l'information
- **webService.json.path** : le préfixe de chemin d'URL qui est utilisé par EWD.js de reconnaître les demandes de services Web entrants
- **management.path** : le préfixe de chemin d'URL qui est utilisé par EWD.js de reconnaître les demandes de gestion des EWD.js basés sur HTTP entrantes
- **management.password** : accès à l'application EWDMonitor et aux demandes de gestion basées sur HTTP requiert une authentification en utilisant ce mot de passe. **Assurez-vous que les systèmes et/ou EWD.js systèmes de production qui sont accessibles au public ont un mot de passe unique défini.** Utilisez uniquement le mot de passe par défaut pour le test initial.

Lancement de EWD.js

Lorsque vous démarrez le module de *ewd.js*, ne soyez pas surpris par la quantité d'information qu'il génère : en effet, le niveau d'enregistrement par défaut (voir le paramètre traceLevel dans la section précédente) a été mis à la valeur maximale de 3 . Vous pouvez réinitialiser cette valeur toute entière comprise entre 0 (pas de log) et 3 (maximum), soit dans votre objet paramètres de démarrage ou en utilisant l'application EWDMonitor durant l'exécution de EWD.js,

EWD.js va essayer de reconnaître les problèmes dus à une mauvaise configuration et affiche un message d'erreur approprié si possible et s'arrêtera. Si cela se produit, recherchez les messages de diagnostic ou de conseil dans les journaux EWD.js, modifier la configuration appropriée et essayer de redémarrer EWD.js.

Les raisons habituelles pour que EWD.js ne démarre pas sont :

- utilisation d'un fichier NodeM ou *cache.node* avec une interface pour votre architecture mauvaise

- des privilèges insuffisants pour exécuter EWD.js (généralement limités à des systèmes basés sur GlobalsDB créés à l'aide du programme d'installation de Mike Clayton).

- l'inadéquation de chemin dans l'objet de paramètres de démarrage

Arrêt de ewd.js

Si vous utilisez *ewd.js* dans une fenêtre de terminal, vous pouvez simplement taper *CTRL et C* pour arrêter en toute sécurité les processus de base et d'enfant.

Vous pouvez également arrêter le processus de *ewd.js* utilisant les trois méthodes suivantes :

- Démarrer et se connecter à l'application *ewdMonitor* (voir chapitre suivant) et cliquez sur le bouton *Stop Node.js Process* que vous allez voir ci-dessus la grille du *Master Process*.

- Si vous utilisez EWD.js comme un service, et/ou si vous voulez arrêter les processus EWD.js d'un autre processus :

    a) Identifier le pid du processus maître. Le plus simple est de taper :

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

Chercher celle qui fait référence à votre fichier de démarrage. par exemple, dans l'exemple ci-dessous, il est le premier (pid 12259), se référant à *ewdStart-globals* :

```
ubuntu@ip-172-30-1-88:~/ewdjs$ ps -ax | grep node
12259 pts/0    Sl+    0:21 node ewdStart-globals

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

    b) Maintenant, tout simplement invoquer une commande kill pour ce pid, par exemple:

```
kill 12259
```

Cela envoie un signal SIGTERM au processus maître de EWD.js, qui déclenche la procédure d'arrêt progressif. Notez que l'appel d'un kill pour l'un des processus enfants sera ignoré par le processus enfant.

• Envoyer un HTTP(S) demande au processus de *ewd.js*, de la structure suivante :

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

    Remplacez les éléments entre crochets avec des valeurs appropriées à votre instance de *ewd.js*. Le mot de passe de gestion est celui défini dans le fichier de démarrage *ewd.js* (par exemple *ewdStart *.js*) (Par défaut il est réglé sur *keepThisSecret!*, mais pour des raisons évidentes, il est fortement recommandé de modifier le mot de passe par autre chose). Par exemple :

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

Toutes ces méthodes ont le même effet : le processus maître de *ewd.js* instruit chacun de ses processus fils connectés à fermer la base de données Mumps proprement. Dès que tous les processus enfants l'ont fait, le processus maître sort, à son tour, provoque les processus enfants à également sortir.

L'application ewdMonitor

L'application ewdMonitor

Inclus dans le kit d'installation de *ewd.js* est une application prête à l'emploi EWD.js nommé *ewdMonitor*. Cette application sert à deux fins :

- il fournit un bon exemple d'une application avancée Bootstrap 3 basée sur EWD.js
- il vous fournit une application qui permet de surveiller et de gérer le module *ewd.js*, et avec lequel vous pouvez inspecter divers aspects de votre environnement de base de données Mumps et EWD.js.

Vous démarrez l'application dans un navigateur en utilisant l'URL :

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

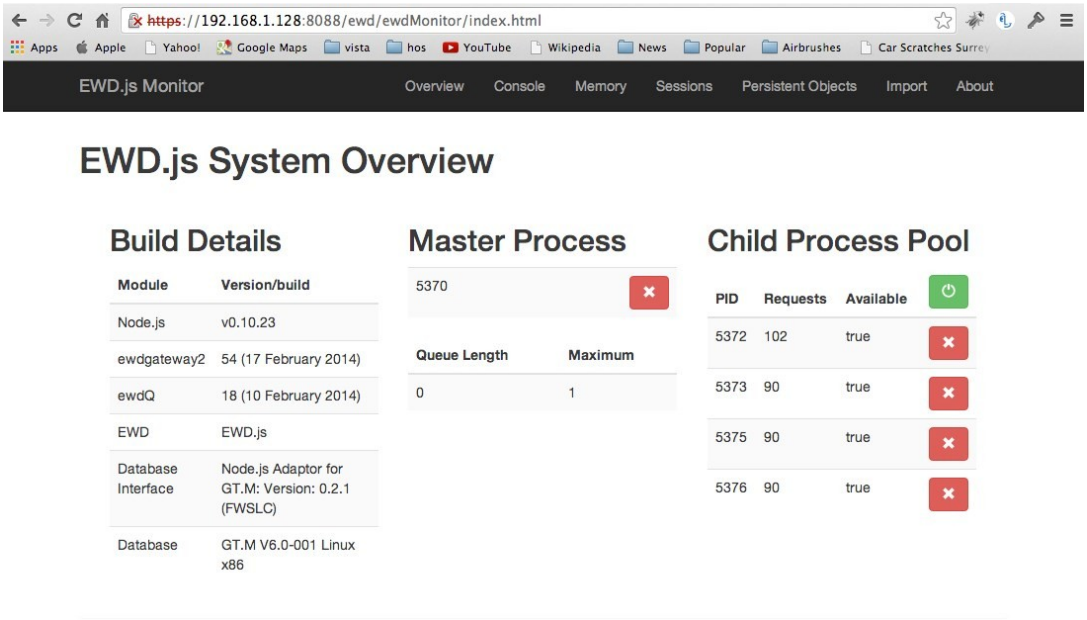
Modifier l'adresse IP ou le nom d'hôte et le port de manière appropriée.

Si vous avez modifié les paramètres de démarrage de EWD.js en précisant SSL, vous aurez besoin de changer l'URL ci-dessus pour commencer par *https://* et vous aurez probablement un avertissement sur l'auto-certificat SSL utilisé par votre serveur web *ewd.js* : lui dire qu'il est OK pour continuer.

On vous demandera un mot de passe : utiliser celui spécifié dans le fichier *ewd.js* de démarrage, soit comme spécifié dans cette section :

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

Entrez le mot de passe et l'application devrait s'ouvrir et ressembler à ceci :



Dans le panneau supérieur, vous verrez des informations de base sur les Node.js, *ewd.js* et les environnements de base de données, y compris des statistiques sur le processus Node.js maître et le nombre de demandes traitées par chacun des processus enfants.

Une caractéristique importante est le bouton **Stop Node.js Process** dans le panneau **Master Process**. Vous devez toujours essayer de l'utiliser pour arrêter le processus de *ewd.js* et ses connexions associées aux processus de base de données Mumps.

Autres fonctionnalités fournies par cette application sont :

- en planant au-dessus des processus Ids, on fait apparaître un panneau qui affiche l'utilisation actuelle de la mémoire du processus. Pour les processus enfant, vous verrez aussi la liste des modules d'application qui sont actuellement chargé
- un journal en direct accessible en console : utile si vous êtes en cours d'exécution de *ewd.js* en tant que service
- un graphique en temps réel, montrant l'utilisation de la mémoire par les processus de base et les enfants de *ewd.js* : utile pour vérifier les sur-utilisations de la mémoire
- un tableau montrant actuellement les sessions EWD.js actifs. Vous pouvez afficher le contenu de chaque séssion et/ou terminer des sessions
- un menu en arbre qui vous permet de visualiser, explorer et éventuellement supprimer le contenu de votre espace de stockage de base de données Mumps
- une option d'importation pour importer des données dans votre base de données Mumps
- des options pour changer l'enregistrement/le niveau de traçage et la commutation entre l'exploitation de la console en direct et un fichier texte
- un formulaire pour le maintenir les droits d'accès et les clés de sécurité pour les services Web basés sur EWD.js.

Passez un peu de temps à explorer cette application : vous devriez trouver que cette fenêtre est utile dans *ewd.js* et les environnements EWD.js.

Création d'une application EWD.js

Anatomie d'une application EWD.js

Les applications EWD.js utilisent les messages WebSocket comme seul moyen de communication entre le navigateur et la logique back-end de l'application. Essentiellement, les seules pièces mobiles d'une application EWD.js sont une paire de fichiers JavaScript, un dans le navigateur et l'autre un module Node.js. Ils envoient des messages WebSocket les uns aux autres et traitent les messages correspondants qu'ils reçoivent. Le module Node.js back-end a accès à MongoDB et/ou votre base de données Mumps sélectionnée, celle-ci étant abstraite comme une collection d'objets JavaScript persistants.

Une application EWD.js se compose d'un certain nombre d'éléments clés :

- Une page HTML statique qui fournit le conteneur de base et l'interface utilisateur principale
- Éventuellement, un ou plusieurs fichiers fragmentés : fichiers de balisage HTML statique qui peut être injecté dans la page conteneur principal
- Un fichier JavaScript statique, normalement nommés app.js, qui définit :
- les messages WebSocket sortants, déclenchées par des événements dans l'interface utilisateur
- les gestionnaires pour les messages entrants WebSocket de l'application back-end EWD.js
- un module Node.js back-end qui définit les gestionnaires pour message websocket entrant à partir de navigateurs utilisant l'application EWD.js. Le nom de ce fichier de module est généralement le même que le nom de l'application EWD.js.

EWD.js a besoin que vos créations soient nommés et placés de manière appropriée dans les chemins de répertoire que vous avez créé au cours des étapes d'installation. Par exemple, si nous étions en train de créer une application nommée EWD.js *myDemo*, vous nommer et placer les composants ci-dessus comme suit (par rapport à votre répertoire Home, par exemple *~/ewdjs*) :

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

Pour bien comprendre le commentaire EWD.js fonctionne et comment créer des applications EWD.js, nous vous encourageons à lire et à suivre la simple application tutoriel **Hello World** à l'Annexe 4.

La Page Conteneur HTML

Chaque demande EWD.js a besoin d'un Page principale du conteneur HTML. Ce fichier doit résider dans un sous-répertoire du répertoire *www/ewd* qui a été créé à l'origine au cours des étapes d'installation sous votre EWD.js répertoire Home. Le nom du répertoire doit être le même que le nom de l'application EWD.js, par exemple, dans l'exemple ci-dessus, *myDemo*.

La page HTML peut avoir n'importe quel nom que vous voulez, mais la convention normale est de le nommer *index.html*.

Pour les applications Bootstrap 3, vous devez utiliser le modèle de page (*index.html*) que vous trouverez dans le dossier de l'application de bootstrap3, ou au https://github.com/robtweed/ewd.js/blob/master/www/ewd/bootstrap3/index.html Voir l'annexe 4 pour plus de détails.

Le fichier app.js

Le comportement dynamique d'une application EWD.js est créé par le contenu JSON livré dans le navigateur via des messages WebSocket, après quoi vos gestionnaires de messages navigateur secondaires utiliser ce contenu JSON et modifier l'interface utilisateur de manière appropriée.

Tel est le rôle et le but du fichier *app.js*. Il réside normalement dans le même répertoire que le fichier *index.html*.

Le contrôleur JavaScript websocket côté navigateur peut effectivement avoir le nom que vous voulez, mais la convention est de le nommer *app.js*.

Un modèle *app.js* fichier pour une utilisation avec Bootstrap 3 est prévu dans le dossier de l'application de bootstrap3, ou au https://github.com/ robtweed/ewd.js/blob/master/www/ewd/bootstrap3/app.js

Ses éléments constitutifs sont les suivantes. Il est recommandé que vous respectiez la structure représentée ci-dessous. Pour plus d'informations, voir l'annexe 4.

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

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

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

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

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

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

  },
   onMessage: {

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

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

};
```

Les messages Websocket EWD.js ont un type obligatoire. EWD.js fournit un certain nombre de noms de types prédéfinis réservés, et ces propriétés sont également prédéterminées qui leur sont associés. Dans l'ensemble, cependant, il appartient au développeur de déterminer les noms de type et la structure du contenu des messages. Les informations utiles sous forme de messages WebSocket sont décrits comme contenu JSON.

Pour envoyer un message websocket EWD.js à partir du navigateur, utilisez l'API EWD.sockets.sendMessage avec la syntaxe :

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

eg:

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

Notez que l'information utile doit être placé dans la propriété *params*. Seul le contenu à l'intérieur de la propriété *params* est acheminé vers le processus enfant où votre code back-end se déroulera.

Back-end du Module Node.js

La dernière pièce dans une application EWD.js est le module Node.js back-end.

Ce fichier doit résider dans le répertoire *node_modules* qui a été créé à l'origine au cours des étapes d'installation sous votre répertoire Home EWD.js (par exemple *~/ewdjs/node_modules*). le nom de fichier du module doit être le même que le nom de l'application EWD.js, par exemple, un moniteur de *ewdMonitor* nommé aura un fichier de module de back-end nommé *ewdMonitor*.

Les éléments constitutifs du module d'un back-end sont les suivants. Il est recommandé que vous respectiez la structure représentée ci-dessous :

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

 module.exports = {

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

   onMessage: {

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

    getPatientsByPrefix: function(params, ewd) {

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

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

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

  }

};
```

Formulaire de la transaction

EWD.js a une fonction intégrée spéciale côté navigateur (le formulaire *EWD.sockets.submit*) pour le traitement des formulaires. Si vous utilisez ExtJS, puis EWD.js fournit une certaine automatisation supplémentaire, mais il peut être utilisé avec tout autre framework, y compris le vôtre.

Si vous utilisez cette fonction de formulaire avec ExtJS, il recueille automatiquement les valeurs de tous les champs de formulaire dans un élément de forme ExtJS (ie xtype: 'form') et les envoie comme la charge utile d'un message dont le type doit être précédé *EWD.form*. La syntaxe est la suivante :

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

Si vous utilisez d'autres frameworks, vous devez tagguer les valeurs de formulaire qui ont besoin de la soumission en définissant les champs objet, par exemple, si vous utilisez Bootstrap, soumettre une forme de gestionnaire de bouton pourrait ressembler à ceci. Notez la façon dont un formulaire widget peut être spécifié pour gérer l'affichage de messages d'erreur :

```
$('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'
    }
  });
});
```

Le module Node.js back-end aura un gestionnaire correspondant pour le type de message spécifié, par exemple pour gérer un message *EWD.form.login* pour un formulaire avec des champs ayant des propriétés de nom avec les valeurs nom d'utilisateur et mot de passe, nous pourrions faire ce qui suit :

```
  '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 '';
   }
```

Vous pouvez voir que les valeurs de champ de *username* et *password* sont transmis à votre fonction comme argument *params* : il contient l'information utile *params* du message entrant qui a été envoyé à partir du navigateur.

Le traitement du formulaire dans EWD.js est très simple : pour chaque condition d'erreur que vous déterminer, il suffit de retourner la chaîne que vous souhaitez utiliser le texte d'un message d'erreur apparaît dans le navigateur. Si un gestionnaire de messages *EWD.form.xxx* renvoie une chaîne non vide et de l'application en cours d'exécution dans le navigateur a été construit en utilisant EWD.js/Bootstrap 3 framwork, le message d'erreur apparaîtra automatiquement sur le navigateur de l'utilisateur dans un toastr widget. Si vous utilisez HTML fabriqués à la main et de JavaScript dans le navigateur, l'erreur apparaîtra comme une fenêtre d'alerte JavaScript.

Cependant, si une fonction sous forme de gestionnaire renvoie une chaîne vide, EWD.js renvoie un message du même type (par exemple *EWD.form.login* dans l'exemple ci-dessus) avec le *ok* comme la charge utile *ok: true*.

Par conséquent, côté navigateur le fichier app.js doit inclure un gestionnaire pour un message entrant de Type *EWD.form.login* afin de modifier l'interface utilisateur en réponse à une connexion réussie - par exemple en retirant le formulaire de connexion, par exemple :

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

Contrôle d'Authentification de l'Utilisateur

Dans de nombreuses applications EWD.js vous voudrez pour établir les informations d'authentification utilisateur avant de leur permettre de continuer. Dans de telles applications, il est important que vous ne laissez pas une porte de derrière qui permet à un utilisateur non-authentifié d'essayer en invoquant des messages WebSocket (par exemple à partir des outils du développeur de la console de Chrome). Pour ce faire, assurez-vous d'effectuer les opérations suivantes :

1)Dans votre logique des modules back-end qui traite le formulaire de connexion initiale, utilisez la fonction spéciale *ewd.session.setAuthenticated()* pour marquer l'utilisateur comme ayant été authentifié avec succès. Par défaut, les utilisateurs sont marqués comme non authentifié.

2)Assurez-vous que toutes les fonctions de gestion des messages d'arrière-plan, en dehors de la fonction d'authentification de connexion, vérifier que l'utilisateur a été authentifié, par exemple,

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

Remarque : *ewd.session.isAuthenticated* est une propriété qui existe seulement en back-end et back-end ne peut être accessible via un message WebSocket dont le traitement est entièrement contrôlée par la logique du développeur.

L'Objet ewd

Vous avez vu que vos fonctions de message de gestionnaire back-end au sein de l'objet *onMessage* ont deux arguments: *params* et *ewd*. Le second argument, *ewd*, est un objet qui contient plusieurs objets sous-composants qui sont utiles dans votre logique back-end, en particulier :

- **ewd.session** : un pointeur vers EWD.js la session de l'utilisateur qui est stocké et maintenu dans la base de données Mumps ou MongoDB. EWD.js automatiquement délesté de toutes les sessions expirées. Les sessions EWD.js ont un délai d'expiration par défaut de 1 heure.
- **ewd.webSocketMessage** : un pointeur vers l'objet de message WebSocket entrant complète. Normalement vous utilisez l'argument *params*, puisque c'est là la charge utile du message entrant est normalement placé. Cependant, l'objet du message entier est mis à votre disposition pour les cas où vous pourriez l'exiger ;
- **ewd.sendWebSocketMsg()** : une fonction pour envoyer des messages à partir du module WebSocket Node.js back-end pour le navigateur de l'utilisateur. La syntaxe de cette fonction est :

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

eg:

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

- **ewd.mumps** : un pointeur qui vous donne accès à la base de données Mumps et Legacy Mumps fonctions (celui-ci est disponible sur Caché et GT.M seulement). Cette opération est décrite en détail dans le chapitre suivant.

Voici un exemple de l'application de démonstration de la façon dont les données Mumps peuvent être manipulés à partir de votre module Node.js :

```
   '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** : une collection de fonctions prédéfinis qui peuvent être utiles dans vos applications, y compris :
- **ewd.util.getSessid(token) :** retourne la session de EWD.js identifiant unique associé à un jeton de sécurité
- **ewd.util.isTokenExpired(token) :** renvoie vrai si la session a expiré
- **ewd.util.sendMessageToAppUsers(paramsObject) :** envoie un message websocket à tous les utilisateurs d'une application spécifique. Les propriétés de l'objet *params* sont les suivantes :
  - **type :** le type de message (string)
  - **content :** message de charge utile (JSON object)

- **ewd.util.requireAndWatch(path) :** alternative à *require()*. Ceci charge le module spécifié et définit une montre sur elle aussi. Si le module est édité, il est automatiquement rechargé. Utile pendant le développement application/module.

Démarrage des Applications EWD.js dans un Navigateur

Les applications EWD.js sont démarrés en utilisant une URL de la forme :

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

- Spécifiez http:// ou https:// en fonction de si oui ou non votre fichier de démarrage de *ewd.js* permet HTTPS ou non
- Réglez l'adresse IP ou le nom de domaine selon le serveur sur lequel fonctionne *ewd.js*
- Spécifiez le port qui correspond au port défini dans votre fichier de démarrage de *ewd.js*
- Indiquez le nom de l'application qui correspond, sensible à la casse, le nom utilisé comme chemin de répertoire pour vos fichiers *index.html* et *app.js*.

Les Applications de débogage EWD.js utilisant Node Inspector

Node Inspector (https://github.com/node-inspector/node-inspector) est devenu un utilitaire standard pour le débogage des applications Node.js. EWD.js en offre une intégration, de sorte que vous puissiez l'utiliser pour parcourir et examiner comment votre application EWD.js se comporte en fonctionnement (ou à défaut!).

Il suffit de suivre ces étapes :

- Vous devez utiliser EWD.js Build 67 ou les versions suivantes. Améliorez votre installation EWD.js si nécessaire
  Installez Node Inspector :
  - *cd ~/ewdjs (ou là où se trouve votre répertoire Home EWD.js)*
  - *npm install -g node-inspector*

- Commencez Node Inspector dans un terminal/SSH
  - *cd ~/ewdjs (ou là où se trouve votre répertoire Home EWD.js)*
  - *noeud-inspecteur --web-port 8081*

- Commencez EWD.js de la manière habituelle (s'il n'a pas déjà en cours d'exécution)
  - *cd ~/ewdjs (ou là où se trouve votre répertoire Home EWD.js)*

En utilisant Chrome, mettre en place l'application *ewdMonitor* et se connecter. Dans la vue d'ensemble du panneau principal, vous verrez un bouton dans la section de la liste des processus fils qui va commencer un nouveau processus fils en mode débogage. Cliquez dessus, puis fermez les autres processus enfant de sorte que le seul processus est votre nouveau debug actif. Cela garantit que toutes les activités sera assurée par ce processus

Cliquez sur le bouton de mise au point bleu pour le processus enfant. Un nouvel onglet du navigateur s'ouvrira et l'interface débogueur Node Inspector apparaît - il faut quelques secondes pour charger complètement. Si vous regardez attentivement, il est presque identique console JavaScript/Outils de développeur Chrome. Vous pouvez maintenant ajouter des points d'arrêt et examiner votre module(s) en fonctionnement !

Lorsque vous avez terminé, lancer de nouveaux, les processus enfants normaux (pas en mode debug) et fermer celle que vous avez activé en mode débogage. CTRL et C pour arrêter le processus Node Inspector dans sa fenêtre de terminal

Remarque : Si vous voulez changer le web-port utilisé par Node Inspector, cliquez sur l'onglet **Internes** dans l'application de *ewdMonitor* et cliquez sur le nouveau bouton, vous trouverez là-bas

Si vous avez pas l'habitude de Node Inspector, il vaut la peine de lire quelques-uns des nombreux tutoriels et d'articles sur son utilisation. Par exemple: https://www.youtube.com/watch?v=03qGA-GJXjI

Mise en commun : Construction d'une application EWD.js

L'annexe 4 vous emmène à travers le processus de création d'une application EWD.js, d'abord un HTML très basique et l'application JavaScript, puis la même application de démonstration en utilisant le framework EWD.js/Bootstrap 3.

Pour voir des exemples d'applications EWD.js, jetez un œil à l'application *ewdMonitor* qui est inclus dans l'installation EWD.js.

Pour voir un exemple de création d'une application EWD.js intégration avec l'Open Source VistA Electronic Healthcare Record (EHR), voir l'application VistADemo qui est également inclus dans l'installation de EWD.js.

Le code source de ces applications peut être trouvé dans le *ewd.js* Github référentiel au https://github.com/robtweed/ewd.js, par exemple:

Application ewdMonitor :
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

Application VistADemo :
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

Dorsaux fonctions oreillons pour accéder Vista (remarciment à Chris Casey):

https://github.com/robtweed/ewd.js/blob/master/OSEHRA/ZZCPCR00.m

Support Micro-Service dans EWD.js

Que sont les Micro-Services?

Les Micro-Services sont les derniers gros buzz dans le monde du web et le développement d'application basée sur un navigateur. Comme avec beaucoup de mots à la mode dans l'informatique, le terme est un peu vague et difficile à spécifiquement pin-point, mais il vaut la peine de lire le journal définitive par Martin Fowler sur le sujet :

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

En un mot, les Micro-Services sont tous les énormes applications web monolithique/navigateur qui nécessitent de la mise au point et de la place pour leur création. Elles sont composées d'un assemblage de nombreux petits, composants, réutilisables. Tout est question de créer un environnement où diverses équipes disparates dans une organisation peut se concentrer uniquement sur leur part d'une entreprise, et construire le micro-Service(s) que d'autres peuvent utiliser dans leurs applications.

L'exemple classique est le site Web d'Amazon qui ressemble à une grosse application monolithique à l'utilisateur, mais qui est en fait un ensemble de micro-services pour lesquels chaque département ou une section au sein de Amazon est individuellement responsable. Les pages Web qui composent le site Amazon que nous, voyons en tant que utilisateur, est simplement un mécanisme pour tirer ensemble les micro-services d'une manière cohérente.

EWD.js Build 85 (et versions suivantes) comprend maintenant une architecture Micro-service afin de vous permettre de travailler de façon similaire. Il vous permet effectivement de créer et de travailler avec deux types de Micro-Services :

- Services back-end (côté serveur)
- Services front-end (côté navigateur)

Une demande EWD.js peut utiliser un ou des deux types.

Les Micro-Services Back-end (côté serveur)

Ce sont les plus simples et les plus faciles à comprendre et à utiliser. L'idée est que, au lieu d'une application EWD.js ayant un seul module back-end qui contient l'ensemble de ses gestionnaires de messages et associé logique métier back-end, l'application EWD.js peut également faire usage d'une ou plusieurs modules back-end, chacun d'entre eux se concentrent sur un domaine fonctionnel particulier ou une entreprise. Par exemple, votre application peut nécessiter un ensemble de services de back-end pour :

- comptes
- personnel
- contrôle du stock
- facturation
- service client
- etc

Un back-end Micro-service est tout simplement un ensemble de gestionnaires de messages, mais il peut être écrit de manière isolée et utilisée par une application EWD.js qui en a besoin.

Par exemple, ici un simple exemple de service back-end :

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

Il est sauvegardé dans le répertoire *~/ewdjs/node_modules* avec tous les autres modules de back-end. Supposons que nous enregistrons comme *exampleService.js* (Vous pouvez nommer ce que vous voulez, mais il est une bonne idée d'utiliser un nom significatif qui décrit la zone fonctionnelle couverte, par exemple *accounts.js*, *stockControl.js* etc).

Pour utiliser ce Micro-service dans votre application, il suffit d'ajouter une fonction de services à son module de back-end :

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

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

Cela indique à EWD.js la liste des micro-services back-end que notre demande est autorisé à utiliser. Chaque nom dans le tableau doit correspondre au nom d'un back-end du module Micro-Service (sans l'extension de fichier *.js*). Ce *service()* est important parce que nous avons besoin pour empêcher un utilisateur malveillant d'essayer d'envoyer des messages voyous qui tentent d'utiliser des micro-services qui ne sont pas normalement disponibles pour l'application !

Maintenant, le dossier de front-end de notre application peut envoyer des messages au module *exampleService* Micro-Services, simplement en ajoutant une nouvelle propriété nommé *service* à la () à l'objet d'entrée de la fonction *EWD.sockets.sendMessage()* :

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

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

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

Les Micro-Services Front-end (côté navigateur)

EWD.js vous permet également de définir un ensemble d'actions et de gestionnaires de messages qui peuvent être utilisés par le front-end de toutes les applications EWD.js. Tout comme avec les back-end Micro-Services, l'idée est que l'auteur d'un tel front-end Micro-service n'a pas besoin de savoir quelque chose sur les applications EWD.js il sera utilisé, de sorte qu'ils sont complètement réutilisable.

Un front-end Micro-service peut également inclure éventuellement un fichier de fragment qui peut être chargé dans le *index.html* d'une application EWD.js, mais des fragments peut encore être fait sur mesure par l'auteur de l'application afin d'assurer qu'ils sont correctement structurées et de style pour adapter à la conception de la page globale.

Un front-end Micro-service peut être conçu pour se concentrer sur la manipulation d'un élément particulier de la fonctionnalité de l'entreprise (par exemple, comment gérer les listes de contrôle des stocks), ou ils peuvent se concentrer sur un ou plusieurs interface utilisateur widgets (par exemple, un calendrier ou une grille de contrôle générique) . Ils sont un moyen de créer des front-end au comportement et la fonctionnalité réutilisables.

Front-end Micro-Services sont un peu plus complexe que les Back-end Micro-Services, mais le truc pour les utiliser est de faire usage de leur modèle standard. Utilisez les exemples ci-dessous sous forme de modèles que vous modifiez par la suite et à étendre.

Lorsque vous installez EWD.js build 85 (ou version ultérieure), vous trouverez un nouveau répertoire sous ~/ewdjs/www :

~/ewdjs/www/services

C'est l'endroit où vous devez déposer tous vos modules front-end Micro-service et leurs fichiers de fragments associés. Vous trouverez un exemple d'un là-dedans: *demoPatientProfile.js* et son fichier de fragment associé *demoPatientProfile.html* Le schéma général d'un front-end fichier de module Micro-service est comme suit :

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

   // put any private functions in here

   return {

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

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

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

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

Un front-end Micro-service est chargé dans le navigateur de l'application en utilisant la fonction intégrée : *EWD.require()*. Cela rend l'utilisation de *require.js* JavaScript module loader (http://requirejs.org), qui doit donc être chargé dans la page *index.html* de votre application. Une utilisation typique de *EWD.require()* est indiqué ci-dessous. Ce serait appelée depuis fichier *app.js* de votre application EWD.js:

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

- ***serviceName*** spécifie le nom du Micro-service front-end que vous voulez charger. EWD.js va chercher un fichier du même nom dans votre répertoire des services *~ewdjs/www/services* avec une extension de fichier *.js* - si dans l'exemple ci-dessus, il va charger */services/ patientProfile.js*
- ***targetSelector*** précise l'élément(s) HTML cible dans lequel le fragment du service sera chargé. Si une valeur de chaîne simple est utilisé, EWD.js va charger le fragment dans l'élément qui a un ID de cette valeur. Sinon, vous pouvez utiliser un sélecteur jQuery, dans ce cas, le fragment sera chargé dans l'élément(s) qui correspondent au sélecteur jQuery spécifié. Ainsi, dans l'exemple ci-dessus, le fragment du service sera chargé dans l'élément avec un *id* de *mainContainer*

- ***fragmentName*** peut ou ne peut pas être spécifié. Si non spécifié (comme dans l'exemple ci-dessus), EWD.js tentera de charger un fichier du même nom que le *serviceName* avec une extension *.html* de votre répertoire */services* - si dans l'exemple ci-dessus, il va charger */services/patientProfile.html*. Si *fragmentName* est spécifié, EWD.js tentera de charger un fichier du nom que vous spécifiez, avec une extension *.html*, de la propre répertoire de l'application (par exemple */www/ewd/{applicationName})*. Vous pouvez éventuellement supprimer le chargement du fragment de service en spécifiant *fragmentName: false*

Quand un front-end Micro-service est chargé en utilisant *EWD.require()*, quatre choses se passent dans un ordre strict (synchrone) :

- le fragment est chargé
- le gestionnaire correspondant à *onFragment()*, s'il est définie dans le fichier *app.js* de l'application, s'active
- le service *init()* s'active
- la fonction d*one()* dans le *EWD.require()*, si elle est définie, s'active

Typiquement, vous allez utiliser chaque gestionnaire de la manière suivante :

- dans la plupart des cas, le développeur de l'application ne sera pas dans la nécessité de fournir un *onFragment()*. Cependant, si le développeur comprend le service en détail, il/elle peut fournir un *onFragment()* pour personnaliser le fragment, modifier son balisage, étendre ses gestionnaires, etc.

- la fonction *init()* du service est où l'auteur de service évoque code qui définit le comportement du service en termes de gestionnaires, événements, etc. Notez que la fonction *init()* a accès à toutes les fonctions privées (le cas échéant) qui sont définis au partie supérieure du module de service. La fonction *init()* peut également être utilisé pour étendre *EWD.application* avec des méthodes supplémentaires qui sont alors disponibles pour *app.js* et toute application EWD.js qui charge le service.

- la fonction *EWD.require()* est l'endroit où l'auteur de l'application (à savoir l'utilisateur du service) peut éventuellement invoquer un code spécifique à l'utilisation pour le service de l'application. Au moment où ce code est exécuté, le fragment est complètement chargée et le service est initialisé. Nous aimerions attendons à ce que la fonction *done()* peut être ignoré pour la plupart des services, et ne doit être utilisé par des utilisateurs expérimentés qui comprennent en détail le service qu'ils sont en cours de chargement.

Si l'auteur de l'application décide d'utiliser son/sa propre fragment au lieu de celui fourni par le service (le cas échéant), alors il est de la responsabilité de l'auteur de l'application d'alimenter le gestionnaire *onFragment()*. Une fois de plus, nous aurions seulement prévoyons utilisateurs avancés qui comprennent parfaitement le fonctionnement d'un service de spécifier leur propre fragment. Notez que cela ne sera pas autrement affecter la séquence d'événements décrite ci-dessus.

Exemple de Démonstration

Si vous installez ou mettez à jour EWD.js Build 85 (ou version ultérieure), vous trouverez un exemple de démonstration :

~/ewdjs/www/ewd/demoMicroServices

Vous pouvez exécuter ceci en utilisant l'URL habituelle :

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

Vous devriez voir apparaître une page avec quelques détails de profil de patient. Qu'est-ce qui est réellement arrivé est que le contenu de la page de profil est venu à partir d'un front-end Micro-service, avec des données via un back-end Micro-service.

Jetez un oeil dans le fichier *~/ewdjs/www/ewd/demoMicroServices/app.js*. Dans son gestionnaire *onStartup()*, vous verrez où il charge le front-end Micro-service nommé *demoPatientProfile* :

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

Vous trouverez ce Micro-service dans *~/ewdjs/www/services :*

- son fragment est nommé *demoPatientProfile.html*
- la logique est *demoPatientProfile.js*

Parce *fragmentName* n'a pas été précisée dans le *EWD.require()*, le fragment est automatiquement chargé et la propriété *targetSelector* dans le *EWD.require()* indique EWD.js de l'injecter dans l'élément dont l'id est *main_Container*.

Maintenant, jetez un œil à l'intérieur *demoPatientProfile.js* et de trouver sa fonction *init()*.

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

Vous verrez qu'il envoie un message à l'arrière-plan avec un type de *getUserData*. Cependant, notez la propriété de service de *demoPatientProfile*. Cela signifie qu'il est l'accès à un back-end Micro-service nommé *demoPatientProfile*.

Jetez un œil sur le module de back-end pour notre application demoMicroServices :
~/ewdjs/node_modules/demoMicroServices.js :

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

Parce que le front-end Micro-service utilisé par notre application *demoMicroServices* utilise le back-end Micro-service nommé *demoPatientProfile*, votre demande doit d'abord être autorisé à l'utiliser : cela est fait dans les *services()* function ci-dessus.

Maintenant, jetez un œil à Micro-Service back-end dans *~/ewdjs/node_modules/demoPatientProfile.js* :

```
module.exports = {
   onMessage: {

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

Cela ressemble à aucun EWD.js standards du module de back-end. Sa réponse est renvoyée à l'extrémité avant, mais au lieu d'être manipulés par les *app.js* de l'application, elle est traitée à l'intérieur du service *demoPatientProfile*. Pour comprendre pourquoi, regardez à nouveau à quel point il a envoyé ce message au back-end Micro-Services :

```
    init: function(serviceName) {
      console.log('demoPatientProfile service: init() firing');
      // lets call upon the profile back end service now

EWD.sockets.sendMessage({         type:'getUserDat
a',         service:'demoPatientProfile',
frontEndService: serviceName
      });
      console.log('demoPatientProfile service: message sent to demoPatientProfile back-end service');
},
```

Remarquez la propriété *frontEndService* réglé sur la valeur de *serviceName* - cela indique à EWD.js que le gestionnaire pour cette réponse de message est dans ce front-end service. Plus bas vous trouverez ce gestionnaire :

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

Cela remplit le fragment et l'affiche. Voilà le service accompli, et la fonction *done()* dans le *EWD.require()* qui a chargé le Micro-service sera désormais actif :

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

L'utilisation de la fonction *done()* est facultative: nous l'utilisons ici juste pour afficher un message de journal dans la console pour confirmer la fin de la séquence des événements. Nous aurions pu laissé complètement, et dans la plupart des cas, vous aurez probablement le faire.

C'est tout ! Notre application a chargé et utilisé avec succès un front-end et back-end Micro-service. La chose intéressante est que le front-end et back-end Micro-Services ont été écrits sans aucune connaissance de la façon dont ils seront utilisés dans l'application, et tout développeur d'application peut simplement charger ce front-end Micro-service en utilisant simplement *EWD.require()* et de le traiter comme une « boîte noire ».

Essayez de lancer l'application à nouveau, cette fois avec la console JavaScript du navigateur. Vous verrez toute une série de messages du journal en cours d'écriture sur: elles ont été délibérément ajoutées à l'application *demo* et de front-end Micro-service. Vous pouvez les trouver dans les différents fichiers JavaScript et confirmer par vous-même comment la séquence des événements aura lieu.

Conclusion

Voilà donc EWD.js Micro-Services. Ils ont un concept très puissant, et permettent un véritable développement de l'équipe dans toute l'entreprise et la réutilisation du code.

Après vous être familiarisé avec EWD.js, leur utilisation est fortement recommandée.

Messages générés de façon externe

Contexte

Jusqu'à présent, nous avons examiné les messages WebSocket comme un moyen d'intégrer une interface de navigateur avec une base de données back-end Mumps. Cependant, EWD.js permet également aux procédés externes d'envoyer des messages WebSocket à un ou plusieurs utilisateurs d'applications EWD.js. Ces messages peuvent être soit un signal simple ou un moyen de fournir aussi complexe une charge utile JSON que vous le souhaitez à un ou plusieurs utilisateurs.

Les processus qui génèrent ces messages externes peuvent être d'autres Mumps ou Cache processes, ou, en fait, tout processus sur la même machine, ou, selon la façon dont vous configurez la sécurité de vos systèmes, autre système sur le même réseau.

L'interface d'entrée de messages externes

Le module *ewd.js* comprend une interface de serveur de socket TCP qui est automatiquement activé et configuré, par défaut, à l'écoute sur le port 10000.

Vous pouvez modifier ce port en modifiant les paramètres de démarrage objet dans votre fichier de démarrage (votre fichier *ewdStart*.js*). Pour ce faire, en ajoutant la définition *webSockets.externalListenerPort*, par exemple:

```
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);
```

Un processus externe a simplement besoin d'ouvrir le port d'écoute (par exemple 10000, sauf si vous avez reconfiguré pour utiliser un port différent), écrire une chaîne de format JSON et ensuite fermer le port d'écoute. EWD.js fera le reste.

Remarque : Pour désactiver le port d'écoute externe, il suffit de définir la valeur des fichiers de démarrage du paramètre à *false*, par exemple :

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

La définition et de routage d'un message généré de façon externe

Vous pouvez envoyer des messages à:

- tous les utilisateurs actuellement actifs de EWD.js
- tous les utilisateurs actuels d'une application EWD.js spécifiée
- tous les utilisateurs dont EWD.js session correspondent à une liste de noms et les valeurs de session

Vous devez spécifier un message *type* (comme vous le feriez pour les messages web sockets sein d'une application EWD.js) et une charge utile de message. Pour des raisons de sécurité, tous les messages de l'extérieur-injectés doivent inclure un mot de passe: cela doit correspondre à celui spécifié dans le fichier *ewdStart*.js*.

La chaîne JSON que vous écrivez sur le port TCP d'écoute détermine votre message une destination est requise. La structure de JSON et les propriétés diffère légèrement pour chaque catégorie de destination :

Les messages destinés à tous les utilisateurs

```
{
 "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
 }
}
```

Tous les utilisateurs EWD.js actuellement actifs auront le message ci-dessus envoyé à leur navigateur.

Les messages destinés à tous les utilisateurs d'une application spécifiée EWD.js

```
{
 "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
 }
}
```

Tous les utilisateurs actuellement actifs d'une application EWD.js nommée *myApp* auront le message ci-dessus envoyé à leur navigateur.

Les messages destinés à tous les utilisateurs spécifiés correspondants au contenu d'une session EWD.js

```
{
 "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
 }
}
```

Tous les utilisateurs EWD.js actuellement actifs dont le nom d'utilisateur est Rob auront le message ci-dessus envoyé à leur navigateur. Plus précisément et avec précision, le message est envoyé à tous les utilisateurs dont EWD.js session contient un identifiant de variable nommée dont la valeur est Rob.

Vous pouvez spécifier autant de paires session nom/valeur que vous le souhaitez dans le tableau. Le message ne sera envoyé si toutes les paires nom/valeur correspondent en session d'un utilisateur.

Gestion des messages générés de façon externe

Les messages générés de l'extérieur sont envoyés au navigateur des utilisateurs concernés.

Pour que les messages injectés de l'extérieur soient traites par les applications EWD.js, vous devez inclure un gestionnaire JavaScript approprié pour le type de message entrant côté navigateur pour chaque application (par exemple dans le fichier *app.js*). Si aucun gestionnaire existe pour le type de message entrant, il sera ignoré.

Les gestionnaires pour les messages externes générés ne sont pas différents de ceux des messages EWD.js WebSocket normales, par exemple :

```
onMessage: {

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

  // ...etc

};
```

Si vous avez besoin de faire quelque chose en back-end afin de traiter un message entrant généré de façon externe, il suffit d'envoyer un message de WebSocket en back-end à partir de votre gestionnaire ainsi que tout ou partie de la charge utile du message généré de façon externe, par exemple :

```
onMessage: {

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

  // ...etc

};
```

Envoi de Messages avec les Processus GT.M et Caché

Si vous voulez envoyer des messages externe à partir des processus GT.M ou Caché, vous pouvez utiliser la méthode de pré-construit (*sendExternalMessage()*) qui est fournie par les *ewdjsUtils* nommés routine Mumps qui est inclus dans le référentiel EWD.js. Voir: https://github.com/robtweed/ewd.js/blob/master/mumps/ewdjsUtils.m

Si vous utilisez GT.M, copiez ce fichier de routine dans un répertoire approprié qui est mappé pour compiler et exécuter des fichiers de routine. Si vous utilisez Caché, de copier et coller le code dans Caché Studio Compiler/Enregistrer en utilisant le nom *ewdjsUtils*.

Vous trouverez un exemple de la façon d'utiliser la méthode *sendExternalMessage()* dans le fichier de routine, à savoir :

```
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() exerce les trois types de message externe. Au lieu de créer une chaîne de format JSON, le code Mumps vous permet de construire la structure JSON comme un tableau locale Mumps équivalent. Le code pour l'ouverture, la fermeture et l'écriture sur le port TCP sur les systèmes EWD.js GT.M et Caché est inclus dans le fichier routine.

Donc, pour tester l'envoi d'un message GT.M externe ou processus Caché, vous pouvez appeler la fonction suivante, qui suppose que votre processus de *ewd.js* utilise le port TCP par défaut, 10000 :

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

Bien sûr, vous aurez besoin d'écrire des gestionnaires dans le navigateur pour les trois types de messages si vous voulez qu'ils fassent quelque chose.

Modifier le code dans la procédure de test de *externalMessageTest()* pour créer des messages externes de votre choix.

Envoi de messages générés de façon externe à partir d'autres environnements

Bien sûr, vous n'êtes pas limité à des processus de GT.M ou Caché. Tout processus qui peut ouvrir le port TCP socket EWD.js en écoute peut écrire un message au format JSON et donc envoyer des messages aux utilisateurs pertinents de EWD.js. Les détails mise en œuvre varient en fonction de la langue que vous utilisez au sein de ces processus.

Accès JavaScript aux dpnnées Mumps

Contexte de la base de données Mumps

La base de données Mumps stocke les données dans un format hiérarchique sans schéma. Dans la technologie de langage Mumps, l'unité individuelle de stockage est connu comme un Global (une abréviation de Globally-Scoped Variables). Étant donné le sens moderne habituel du terme Global, il est peut-être mieux de penser à l'unité de stockage sous forme de tableau associatif persistante. Quelques exemples seraient :

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

Chaque tableau associatif persistante a un nom (par exemple myTable, comme l'exemples ci-dessus). Vient ensuite un certain nombre d'indices dont les valeurs peuvent être des chaînes numériques ou de texte. Vous pouvez avoir un nombre quelconque d'indices.

Chaque "nœud" (un nœud est défini par un nom de tableau et un ensemble spécifique d'indices) stocke une valeur de données qui est une chaîne de texte

(Les chaînes vides sont autorisés). Vous pouvez créer ou détruire nœuds quand vous le souhaitez. Ils sont entièrement dynamiques et ne nécessitent aucun pré-déclaration ou schéma.

Une base de données Mumps n'est pas intégré dans le schéma ou le dictionnaire de données. Il appartient au développeur de concevoir l'abstraction de plus haut niveau et le sens d'une base de données qui est physiquement stocké sous forme d'un ensemble de tableaux persistants.

Une base de données Mumps n'a pas non plus d'indexation intégré. Les indices sont la clé pour être en mesure de manière efficace et efficiente de recherche, d'interrogation et les données traversent dans une base de données Mumps, mais il est au développeur de concevoir, de créer et de maintenir les indices qui sont associés avec les principaux réseaux de données. Les indices sont, eux-mêmes, stocké dans les tableaux Mumps persistants.

Vous pouvez lire plus de fond à la technologie de base de données Mumps et son importance en tant que moteur de base de données NoSQL puissant dans un document intitulé A NoSQL moteur universel, en utilisant une technologie éprouvée: http://www.mgateway.com/docs/universalNoSQL.pdf.

Extrapolation de EWD.js sur les tableaux Mumps

Inclus dans la distribution de *ewd.js* est un fichier Javascript nommé *ewdGlobals.js*. *ewdGlobals.js* est utilisé par EWD.js pour créer une couche d'abstraction au-dessus des API de bas niveau fournies par les interfaces Node.js à GlobalsDB, GT.M et Caché, *ewdGlobals.js* projette la collection de tableaux associatifs persistantes dans une base de données Mumps comme une collection d'objets JavaScript persistants.

Cartographie des Tableaux persistants Mumps en objets JavaScript

La théorie derrière la projection utilisée par *ewdGlobals.js* est vraiment très simple, et mieux expliquée et illustrée par un exemple.

Supposons que nous ayons un ensemble de dossiers des patients que nous souhaitons représenter comme des objets. Nous pourrions définir un objet de niveau supérieur du patient nommé qui représente un patient physique particulière. Habituellement, nous aurions une sorte de clé d'identification des patients qui distingue notre patient particulier: aux fins de cet exemple, disons que identifiant patient est un nombre entier simple : 123456.

La notation d'objet JavaScript permettrait de représenter les informations du patient à l'aide des propriétés qui pourraient, à leur tour, être imbriquées à l'aide des sous-propriétés, par exemple:

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

Cet objet patient pourrait être représentée comme suit dans une base de données Mumps que le tableau persistant qui suit:

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

Ainsi, vous pouvez voir qu'il y a une correspondance directe un-à-un qui peut être faite entre les propriétés d'un objet et les indices utilisés dans un tableau persistant Mumps. L'inverse est également vrai : toute tableau persistant Mumps existante peut être représentée par la hiérarchie d'un objet JavaScript correspondante de propriétés.

Lorsque les données sont stockées dans une base de données Mumps, nous avons tendance à nous référer à chaque unité de stockage comme un *Global Node*. Un *Global Node* est la combinaison d'un nom tableaux persistant (dans ce cas, *patient*) et un certain nombre d'indices spécifiques. Un *Global Node* peut effectivement y avoir un certain nombre d'indices, y compris zéro. Données, sous la forme de valeurs numériques ou alphanumériques, sont stockées à chaque feuille d'un *Global Node.*

Chaque niveau de l'abonnement représente un *Global Node* individuel. Donc, en prenant l'exemple de *zipcode* ci-dessus, nous pouvons représenter les *Global Node* suivants:

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

A noter que les données n'ont pas été stockées au niveau le plus bas (ou feuilles) *Global Node* indiqué ci-dessus. Tous les autres *Global Node* existent mais sont simplement *Node* intermédiaires: ils ont des indices de niveau inférieur, mais ne possèdent pas toutes les données.

Il n'y a rien dans la base de données des Mumps qui nous dira que les indices de “address” et “zipcode” ont été utilisés dans ce tableau persistant particulière autre que par l'introspection des réels *Global Node* : autrement dit, il n'y a pas de dictionnaire intégré de données ou le schéma que nous pouvons faire référence. Inversement, si nous voulons ajouter plus de données à ce tableau persistant, nous pouvons simplement ajouter, en utilisant arbitrairement quelque indices nous souhaitons. Donc, on pourrait ajouter un enregistrement *County* :

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

Ou nous pourrions ajouter le poids du patient:

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

Notez que je pourrais avoir utilisé toute subscripting me plaisait : il n'y avait rien qui m'a forcé à utiliser ces indices particuliers (bien que dans votre application vous vous assurez que tous les dossiers soient toujours utilisés de la même stransaction).

L'Object GlobalNode

La projection *ewdGlobals.js* fournies par EWD.js vous permet d'instancier un objet *GlobalNode* . Par exemple:

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

Un objet *GlobalNode* représente un *GlobalNode* physique dans une base de données Mumps et a à sa disposition un ensemble de propriétés et les méthodes qui lui permettent d'être manipulé et examiné en utilisant JavaScript. Une caractéristique importante d'un objet *GlobalNode* est *qu'il peut ou ne peut pas réellement exister physiquement lors de la première instancié*, mais cela peut ou peut ne pas changer plus tard au cours de son existence au sein EWD.js la session d'un utilisateur.

Une propriété essentielle d'un objet *GlobalNode* est *_value*. Ceci est une lecture/écriture des biens qui vous permet d'inspecter ou de définir la valeur du *GlobalNode* physique dans la base de données Mumps, par exemple :

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

Lorsque vous accédez à la propriété *_value*, vous accédez à la valeur de la physique *Global Node* sur le disque dans la base de données Mumps, mais bien sûr, nous faisons donc via un objet JavaScript, dans ce cas appelé *zipNode*. Notez que dans l'exemple ci-dessus, la première ligne qui met en place le pointeur sur le *Global Node* ne fait pas accéder à la base de données Mumps : en effet la *Global Node* Mumps physique peut même ne pas exister dans la base de données lorsque le pointeur est créé. Il l'est seulement quand une méthode d'un objet *Global Node* est utilisé qui nécessite un accès physique à la base de données d'une liaison physique entre l'objet *Global Node* et le *Global Node Mumps* physique est faite.

Bien sûr, idéalement, nous aimerions être en mesure de faire ce qui suit :

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

Mais il y a un problème: le caractère libre de schéma dynamique des tableaux persistants Mumps signifie qu'il n'y a aucun moyen à l'avance de savoir que le *Global Node* Mumps physique représentant l'objet *GlobalNode patient* effectivement a un indice de *name* et donc aucun moyen de savoir à l'avance qu'il est possible d'instancier une propriété correspondant de *patient* appelé *name*. En théorie, *ewdGlobals.js* pourrait instancier comme une propriété tout indice qui existe physiquement sous un *GlobalNode* Mumps spécifié. Toutefois, un *GlobalNode* pourrait avoir des milliers ou des dizaines de milliers d'indices: il pourrait prendre beaucoup de temps et de puissance de traitement pour trouver et instancier chaque indice comme une propriété et il consomme beaucoup de mémoire au sein de Javascript pour le faire. En outre, dans une application EWD.js typique, vous ne devez avoir accès qu'à un petit nombre de propriétés *GlobalNode* à tout moment, de sorte qu'il serait très coûteux de les avoir tous instanciés et alors seulement utilisent un ou deux.

Pour faire face à cela, un objet *GlobalNode* a une méthode spéciale à sa disposition appelé *_getProperty()*, normalement abrégé *$()* (en prenant une feuille hors du livre de jQuery!). La méthode de *$()* fait deux choses :

- instancie le nom d'indice spécifié comme une propriété de l'objet *GlobalNode* mère
- renvoie un autre objet *GlobalNode* qui représente la plus faible en indice *GlobalNode* physique.

Par exemple:

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

Ce que fait ce code est de créer d'abord un objet *GlobalNode* qui pointe vers le *Global Node Mumps physique* :

patient(123456) GlobalNode

La deuxième ligne fait deux choses :

- étend l'objet *patient* avec une propriété nommée *name ;*
- renvoie un nouvel objet de *GlobalNode* qui pointe vers le *Global Node* physique :

^patient(123456, "name")

Ces deux effets de l'utilisation de la méthode *$()* sont à la fois très intéressant et puissant. Tout d'abord, maintenant que nous avons utilisé une fois comme un pointeur vers l'indice *name*, le *name* a été correctement instancié comme une propriété réelle de patients, afin que nous puissions ensuite se référer directement à la propriété du *name* au lieu d'utiliser le *$()* méthode à nouveau . Ainsi, pour poursuivre l'exemple ci-dessus, nous pouvons maintenant changer le nom du patient en se référant directement au nom de la propriété du patient :

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

Deuxièmement, parce que la méthode *$()* retourne un objet *GlobalNode* (qui peut ou peut ne pas exister ou avoir une valeur de données), nous pouvons enchaîner aussi profond que nous voulons. Ainsi, nous pouvons obtenir la ville du *patient* comme ceci :

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

Chacun des appels enchaînes de la méthode *$()* a retourné un nouvel objet *GlobalNode*, représentant ce sous niveau, nous avons donc désormais la *Global Node* Mumps physique suivante définis comme des objets :

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

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

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

Donc, nous pouvons maintenant utiliser ces propriétés au lieu d'utiliser *$()* de nouveau. Par exemple, pour obtenir le zip code, nous avons juste besoin d'utiliser la méthode *$()* pour la propriété zipcode (puisque nous n'y avons pas encore accédé):

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

Mais si nous voulons signaler le town du patient, nous avons déjà toutes les propriétés instancié, donc on n'a pas besoin d'utiliser la méthode *$()* du tout, par exemple:

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

Comme vous pouvez le voir, par conséquent, la projection fournie par le fichier *ewdGlobals.js* de EWD.js fournit un moyen de traitement des données au sein d'une base de données Mumps comme si elle était une collection d'objets JavaScript persistants. Le fait que vous manipulez des données stockées sur le disque est largement caché.

Propriétés et Méthodes GlobalNode

Un objet *GlobalNode* a un certain nombre de méthodes et propriétés que vous pouvez utiliser pour inspecter et manipuler le *Global Node* Mumps physique qu'il représente :

Method / Property	Description
$()	Retourne un objet *GlobalNode* qui représente un sous-noeud indicé du *GlobalNode* Mumps représenté par l'Object en cours *GlobalNode*. La méthode $() indique le nom de l'indice à instancier comme un nouveau objet *GlobalNode*
_count()	Renvoie le nombre d'indices qui existent dans le cadre du *GlobalNode* Mumps représenté par l'Object en cours *GlobalNode*
_delete()	Supprime physiquement toutes les données pour l'objet *GlobalNode* actuel et physiquement supprime tous les *Global Node* Mumps de niveau inférieur spécifié par *GlobalNode*. Notez que tous les objets Javascript *GlobalNode* que vous avez peut-être instanciés pour les indices de niveau inférieur continuent d'exister, mais leurs propriétés qui se rapportent à leurs valeurs physiques auront changé (par exemple leur *_value*).
_exists	true si l'objet *GlobalNode* existe physiquement comme *Global Node Mumps*. Notez qu'il peut exister sous forme d'intermédiaire ou de feuille *Global Node*
_first	Retourne le nom du premier sous indice *GlobalNode* Mumps représenté par l'Object *GlobalNode* en cours.
_forEach()	Fonction itérant, tirant une fonction de rappel pour chaque indice qui existe sous le *Global Node* Mumps représenté par l'objet *GlobalNode*.
_forPrefix()	Fonction itérant, tirant une fonction de rappel pour chaque indice qui existe sous le *Global Node* Mumps représenté par l'objet *GlobalNode*, où le nom de l'indice commence par le préfixe spécifié.
_forRange()	Fonction itérant, tirant une fonction de rappel pour chaque indice qui existe sous le *Global Node* Mumps représenté par l'objet *GlobalNode*, où le nom de l'indice se situe dans une gamme alphanumérique spécifié.
_getDocument()	Récupère la sous-arborescence du *Global Node* Mumps qui existe physiquement dans le cadre du *Global Node* Mumps représenté par le objet *GlobalNode* actuel, et le renvoie en tant que document JSON.
_hasProperties	true si le *Global Node* Mumps représenté par l'objet *GlobalNode* a un ou plusieurs indices existants (et donc des propriétés potentielles de l'objet *GlobalNode*) en dessous.
_hasValue	true si l'objet *GlobalNode* existe physiquement comme *Global Node* Mumps et a une valeur enregistrée contre lui.
_increment()	Effectue un incrément de la propriété *_value* de l'objet courant *GlobalNode*.
_last	Retourne le nom du dernier sous indice *GlobalNode* Mumps représenté par l'object en cours *GlobalNode.*
_next()	Retourne le nom du prochain indice qui suit celui spécifié, sous le *Global Node* Mumps représenté par l'object en cours *GlobalNode*.
_parent	Renvoie le *GlobalNode* parent de l'object en cours *GlobalNode* comme un objet *GlobalNode.*

_previous()	Retourne le nom du prochaine indice précédant celui spécifié, dans le *Global Node* Mumps représenté par l'object en cours *GlobalNode*.
_setDocument()	Enregistre le document JSON spécifié comme un sous-arbre de *Global Nodes* Mumps sous le *Global Nodes* Mumps représenté par l'object en cours *GlobalNode*.
_value	Propriété de lecture/écriture, utilisée pour obtenir ou définir la valeur physique du *Global Node* Mumps représenté par l'object en cours *GlobalNode*.

Exemples

Supposons que nous ayons les données suivantes stockées dans un tableau persistant Mumps :

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

_count()

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

_delete()

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

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

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

_exists

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

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

_first

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

_forEach()

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

 birthdate: -851884200
 conditions: intermediate node
```

Vous pouvez inverser le sens des itérations en ajoutant {direction: 'reverse'} comme premier argument de la fonction forEach().

_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
```

Vous pouvez inverser le sens des itérations en remplaçant le premier argument avec un objet :

{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
```

Vous pouvez inverser le sens des itérations en remplaçant les deux premiers arguments avec un objet :

{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
    }
```

Remarque : *_getDocument()* a deux arguments optionnels :

- **base** : La valeur par défaut est zéro, et il définit le premier indice dans le global node Mumps à laquelle le premier élément correspondant dans le tableau JavaScript devraient être cartographiés. Si vous utilisez *_getDocument()* pour chercher un héritage du nœud Mumps, vous aurez probablement envie de mettre la base à 1, puisque, par convention, les plus anciens tableaux Mumps sont basés sur 1;

- **useArrays** : Cette valeur par défaut *true*, signifie qu'il va vérifier si un indice numérique peut correspondre à un tableau JavaScript. Pour ce faire, il réalise les premières itérations à travers toutes les valeurs de cet indice jusqu'à ce qu'il trouve soit une pause dans la séquence numérique ou une valeur non numérique, auquel cas il optera pour la map de objet. Cela vient inévitablement à un certain coût en termes de performances. Si vous êtes heureux pour *_getDocument()* pour créer toujours des objets pour chaque niveau de l'indice dans la hiérarchie du Global Node (même si un indice aurait mis en correspondance avec un tableau), alors vous pouvez définir *useArrays* à *false*. Cela permettra d'améliorer très sensiblement les performances de *_getDocument()*. Notez que si *useArrays* est faux, l'argument de base est ignoré.

```
Example:

If the persistent Mumps data contained the following:

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


Then:

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

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


And:

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

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

_hasProperties

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

_hasValue

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

_increment()

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

_last

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

_next()

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

_parent

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

var patient = conditions._parent;

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

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

_previous()

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

_setDocument()

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

};

patient._setDocument(doc, true);

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


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

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

Remarque : *_setDocument()* a deux arguments optionnels supplémentaires :

- **fast** : Ce second arguments par défaut est à *false*, ce qui signifie qu'il va créer des objets GlobalNode pour chaque Global Node physique qu'il crée comme il traverse l'objet JavaScript. Cela vient inévitablement avec un coût de performance et de mémoire qui peut rapidement devenir significatif si l'objet JavaScript que vous enregistrez est très grand. **Dans la plupart des cas, il est une bonne idée de mettre ce second argument de *true***, qui arrête la création d'objets GlobalNode intermédiaires.
- **base** : Ce troisième argument est par défaut à zéro, et il définit le premier indice dans le global node Mumps à laquelle le premier élément correspondant dans le tableau JavaScript doit être mappé. Si vous utilisez *_setDocument()* pour écrire les données d'un tableau Global Mumps, vous aurez probablement envie de mettre la base à 1, puisque, par convention, les plus anciens tableaux Mumps sont basés sur 1.

_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
```

Autres fonctions fournies par les objets *ewd.mumps* dans EWD.js

En plus du constructeur *GlobalNode()* qui a été décrit en détail ci-dessus, l'objet *ewd.mumps* fournit plusieurs fonctions supplémentaires qui peuvent être utilisés dans vos applications EWD.js.

ewd.mumps.function()

Il est disponible pour une utilisation sur GT.M et bases de données Caché, mais pas pour GlobalsDB. En effet, GlobalsDB se trouve un moteur de base de données Mumps et ne comprend pas un processeur de langage Mumps.

L'API *ewd.mumps.function()* vous permet d'invoquer le code existant qui est écrit en langage Mumps. Notez que cette API ne vous permet d'invoquer ce qu'on appelle dans le jargon du langage Mumps des fonctions extrinsèques. Si vous voulez invoquer des procédures ou des classes Caché, vous aurez besoin de les envelopper dans une fonction Mumps extrinsèque wrapper.

La syntaxe pour utiliser l'API *ewd.mumps.function()* est la suivante:

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

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

eg:

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

   This is the equivalent of the Mumps code:

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

Notez que pour returnValue la longueur maximale de la chaîne pour cette API pour l'utilisation de Caché est juste 4k. si la fonction doit renvoyer un grand nombre de données, par exemple un document volumineux JSON, alors il est une bonne idée d'enregistrer les données dans EWD.js session de l'utilisateur, puis le récupérer à partir de la session dans vos EWD.js module JavaScript.

Pour ce faire, vous aurez besoin d'envelopper le code existant dans la fonction à laquelle vous passez l'Id session EWD.js, par exemple :

```
onSocketMessage: function(ewd) {

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

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

}
```

ewd.mumps.deleteGlobal()

Cela devrait être utilisé avec précaution. Il va supprimer un ensemble de tableau persistant Mumps nommé. Notez que les bases de données Mumps ne fournissent pas de capacité de Undelete, cette fonction peut être extrêmement dangereuse : en une seule commande, vous pouvez instantanément supprimer définitivement une base de données entière !

Utilisation : *ewd.mumps.deleteGlobal('myGlobal') ;* supprime immédiatement et définitivement un tableau persistant Mumps nommé myGlobal

ewd.mumps.getGlobalDirectory()

Retourne un tableau contenant les noms de tous les tableaux persistants au sein de votre base de données Mumps.

ewd.mumps.version()

Renvoie une chaîne qui identifie à la fois la version de l'interface Node.js/Mumps et la version et le type de base de données Mumps utilisé.

Indexation des données Mumps

A propos des Indexes Mumps

L'indexation des tableaux persistants Mumps est la clé pour créer des bases de données de haute performance, souple et puissantes. Traditionnellement, la maintenance les indices étaient à la charge du développeur, souvent enterrés à l'intérieur des fonctions ou des API qui a géré la mise à jour des données dans une base de données Mumps.

Un index dans une base de données Mumps est juste un autre tableau persistant. Parfois, le même tableau nommé est utilisé, mais un ensemble différent d'indices sont utilisés pour l'indice. À d'autres moments, les indices peuvent être organisées dans des tableaux persistants des noms différents. Le choix est quelque peu arbitraire, mais peut être une combinaison de style personnel et/ou des problèmes de gestion de données. Par exemple, si les données principales et les indices ont lieu dans un seul tableau persistant nommé, le tableau peut devenir très important et nécessitera la sauvegarde dans un très grand processus. Si les indices sont conservés dans un ou des réseaux plus séparément nommés, le principal réseau de données peut être plus petit, plus facile à sauvegarder. Garder indices dans des tableaux séparément nommés peut également être bénéfique si vous utilisez des mécanismes tels que tourillons ou de cartographie des tableaux.

Prenons un exemple simple: envisager une base de données des patients où nous stockons nom d'un patient (parmi beaucoup d'autres données sur un patient).

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

Si nous voulons créer un centre de recherche où l'utilisateur peut rechercher des patients par nom et sélectionner un dans une liste de correspondance, ce que nous ne voulons pas avoir à faire est de parcourir l'ensemble du réseau du patient, identifiant par id, à la recherche, de manière exhaustive pour tous lastName de *Smith*. Au lieu de cela, nous pouvons créer et de maintenir un tableau persistant parallèle qui stocke un indice lastName. Il est à nous ce que nous nommons ce tableau, et ce que les indices que nous utilisons, mais voici une façon cela pourrait se faire :

```
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
```

Avec un tel éventail d'index, pour trouver tous les patients avec un nom de famille de Smith, tout ce que nous devons faire est d'utiliser la méthode _forEach() pour parcourir les dossiers Smith pour obtenir les identifiants de patients pour tous les patients correspondants. L'identification du patient nous fournit ensuite avec le pointeur nous avons besoin pour accéder au dossier du patient que l'utilisateur a choisi, par exemple :

```
 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]);
```

Si nous voulions trouver tous les noms commençant par un préfixe particulier, par exemple *Smi*, nous pourrions utiliser la méthode *_forRange()* à la place, ou nous pourrions utiliser la méthode *_forRange()* pour trouver tous les noms entre Sma et Sme.

Une amélioration de l'indice pourrait être de convertir le nom en minuscules avant de l'utiliser dans l'index, par exemple :

```
 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
```

Cela permettrait d'assurer que vous ne manquez pas les patients qui ont, par exemple, un trait d'union dans le nom de famille.

Bien sûr, un index par nom de famille est juste un des nombreux vous aurez probablement envie de maintenir une base de données patient. Par exemple, nous pourrions avoir un index par date de naissance, et l'autre par le sexe. Nous pourrions créer des tableaux spécifiquement nommé pour ceux-ci, mais cela pourrait devenir lourd et difficile à maintenir et à se rappeler si nous avons beaucoup d'entre eux. Comme alternative, nous pourrions maintenir tous les indices dans un seul tableau persistant. Nous pourrions le faire facilement en ajoutant un premier indice qui indique le type d'index, par exemple :

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

  ...etc


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

Ceci est clairement extensible aussi : nous pouvons ajouter de nouveaux types d'index en utilisant simplement un nouveau premier indice.

Parfois, vous aurez envie de chercher à travers deux ou plusieurs paramètres, par exemple nom et le sexe. Vous pouvez, bien sûr, la conception de votre logique d'utiliser des combinaisons d'indices simples, mais vous pouvez, bien sûr, créer et maintenir des indices multi-paramètres, par exemple:

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

 ...etc
```

La conception d'index efficace pour les bases de données Mumps est quelque chose qui vient avec l'expérience. Vous pouvez nous l'espérons voir qu'il est infiniment flexible, et la conception de votre index dépend entièrement de votre compétence et de compréhension de la façon dont vos applications ont besoin d'être construit autour des données qui l'anime.

Astuce: ne détiennent que des données ou des valeurs dans un index qui sont originaires de la principale matrice de données ou peut être dérivée des données dans le principal réseau de données. Si vous faites cela, vous pouvez recréer ou reconstruire vos indices de juste le principal réseau de données.

Maintien des indexes dans EWD.js

Les indexes Mumps ne se créent pas, malheureusement. Il est à la charge du développeur, de les créer et de les maintenir les indexes dont vous avez besoin à chaque fois que vous créez, modifier ou supprimer les *Global Nodes* Mumps. Pour un développeur traditionnelle Mumps, cela représentait une corvée, sauf si un ensemble d'API a été conçu pour gérer les mises à jour de base de données. A défaut de créer tous les indices nécessaires cohérente dans une application pourrait conduire à des dossiers d'index manquant et des recherches donc erronées.

EWD.js rend les choses beaucoup plus simple et efficace en utilisant la capacité de Node.js pour émettre des événements qui peuvent être traitées par le développeur. Par conséquent, le module *ewdGlobals.js* émet des événements chaque fois que vous utilisez *_value*, *_delete()* et l'API *_increment()* : à savoir les méthodes qui changent les valeurs de données dans votre base de données Mumps.

Les événements émis sont :

- **beforesave :** émise immédiatement avant une valeur est définie dans un *Global Node Mumps*. L'événement que vous passe un pointeur à l'objet *GlobalNode* qui est sur le point d'être modifié
- **aftersave :** émise immédiatement après une valeur est définie dans un *Global Node Mumps*. L'événement passe un pointeur à l'objet *GlobalNode* correspondant. Deux propriétés supplémentaires - *oldValue* et *newValue* - sont mis à disposition dans l'objet *GlobalNode*. *oldValue* sera une chaîne vide si aucune valeur existait auparavant avant l'événement sauver. Parce que le *oldValue* est mis à disposition par l'intermédiaire de l'événement *aftersave*, dans la plupart des cas, vous serez en mesure d'utiliser simplement l'événement *aftersave* de mettre à jour vos indices.
- **beforedelete :** émise immédiatement avant qu'un *Global Node* Mumps soit supprimé. L'événement que vous passe un pointeur à l'objet *GlobalNode* qui est sur le point d'être supprimé.
- **afterdelete :** émise immédiatement après qu'un *Global Node Mumps soit* supprimé. L'événement passe un pointeur à l'objet *GlobalNode* qui a été supprimé. Une propriété supplémentaire - *oldValue* est mis à disposition dans l'objet *GlobalNode*. Dans de nombreux cas, vous serez en mesure de l'utiliser simplement l'après *afterdelete*. Cependant, rappelez-vous que la méthode *_delete()* peut avoir été appliqué à un niveau intermédiaire de Global Node, dans ce cas, ils peuvent devoir être pris en compte si ces valeurs ont été utilisés dans les indexes des valeurs supprimées au plus bas niveaux de l'aboressance. Dans ce cas, l'événement *beforedelete* vous permettra de parcourir les sous-noeuds qui sont sur le point d'être supprimé et modifier les indexes de manière appropriée.

Le Module globalIndexer

Inclus dans le kit d'installation de *ewd.js* est un module nommé *globalIndexer.js* que vous trouverez dans le répertoire *node_modules* sous votre répertoire Home EWD.js. Ce module est automatiquement chargé par EWD.js de fournir des gestionnaires d'événements pour les événements ci-dessus chaque fois qu'ils se produisent.

Vous trouverez que les bouchons ont été créés dans *globalIndexer.js* que vous devriez étendre et maintenir de manière appropriée aux tableaux Mumps vous souhaitez conserver et les indexes correspondants vous aurez besoin pour eux. Vous verrez un exemple simple qui a été défini pour l'indexation du tableau persistant Mumps utilisé dans l'application *demo*.

En utilisant le module *globalIndexer*, vous serez en mesure de préciser l'indexation de tous vos principaux tableaux de données Mumps en un seul endroit. Vos applications doivent simplement se concentrer sur les changements qui doivent être apportés à votre principale matrice de données Mumps. Les événements émis par *ewdGlobals.js* et traitées par votre logique dans le module de *globalIndexer* signifie que l'indexation peut être géré automatiquement et de manière cohérente, sans encombrer votre logique principale de l'application.

Notez que *ewd.js*, lors de l'exécution, détecte les modifications apportées au module de *globalIndexer* et recharge dans tout processus enfant qui utilisent déjà automatiquement. Vous ne devriez pas avoir à arrêter et redémarrer le module de *ewd.js* chaque fois que vous modifiez le module de *globalIndexer*.

Interface Web Service

EWD.js à base de Services Web

Alors que EWD.js est avant tout un cadre pour la création et l'exécution d'applications à base de WebSocket qui fonctionnent dans un navigateur, il fournit également un mécanisme sécurisé pour exposer back-end JavaScript des fonctions professionnelles en tant que services Web. services web EWD.js renvoient une réponse JSON comme une réponse HTTP *application/json*.

Le développeur d'applications EWD.js a simplement besoin d'écrire une fonction dans un module de back-end qui accède à la base de données mumps et/ou utilise hérités fonctions Mumps, exactement de la même manière que il/elle aurait si l'écriture EWD.js standards un message backend websocket fonctions manutention de l'ouvrage. La fonction doit retourner un objet JSON qui soit contient une chaîne d'erreur ou un document JSON. EWD.js regarde après la demande HTTP analyse et la manutention, l'emballage jusqu'à la réponse du développeur JSON comme une réponse HTTP, et authentifier les requêtes HTTP de services Web entrants.

Authentication Web Service

La sécurité est assurée par l'intermédiaire d'un mécanisme d'authentification HMAC-SHA qui est basé sur le modèle de sécurité utilisé par Amazon Web Services, par exemple, leur base de données SimpleDB : Voir http://docs.aws.amazon.com/AmazonSimpleDB/latest/ DeveloperGuide/HMACAuth.htm pour plus de détails sur la mécanique sur lesquels repose la sécurité de EWD.js.

Si un utilisateur est d'avoir accès à des applications sur un système EWD.js, ils sont enregistrés en utilisant l'application de *ewdMonitor* (cliquez sur l'onglet Sécurité). Chaque utilisateur doit recevoir une *accessId* unique. Dans ce *accessId* sont enregistrées :

- une clé secrète, qui devrait être une longue chaîne alphanumérique qui est difficile à deviner
- une liste d'applications EWD.js que l'utilisateur est autorisé à accéder

L'utilisateur doit être envoyé leur *accessId* et clé secrète: cela doit être fait de manière sécurisée. Il est important que l'utilisateur ne partage ni égarer ces informations d'identification.

La clé secrète est utilisée par le logiciel de client de l'utilisateur pour signer numériquement ses requêtes HTTP. l'ID et la signature *accessId* de l'utilisateur sont ajoutés à la requête HTTP comme queryString paires nom/valeur.

Lorsque le système EWD.js reçoit une demande de service Web :

- le *accessId* est d'abord vérifié pour voir s'il est reconnu ou non
- ensuite, un contrôle est effectué pour s'assurer que le *accessId* a le droit d'accéder à l'application spécifiée
- EWD.js vérifie ensuite la valeur de signature entrant sur la valeur qu'il calcule à partir de la demande entrante, en utilisant la clé secrète qu'elle détient pour l'*accessId* qu'il a reçu dans la requête HTTP.
- Si les signatures correspondent, l'utilisateur est considéré comme valide et la demande est traitée.

Si une erreur est détectée dans l'une de ces étapes, un message d'erreur est envoyé à l'utilisateur et sans que le traitement n'ait lieu.

Création d'un Service Web EWD.js

La logique métier d'un service Web EWD.js est écrit en JavaScript en fonction dans un module Node.js. Le nom du module est utilisé comme nom de l'application. Si vous le souhaitez, et s'il est logique de le faire, les fonctions de service Web EWD.js peuvent être inclus dans un module qui est également utilisé pour une application basée sur un navigateur. Vous verrez un exemple de cela dans le module *demo.js* qui est inclus dans le kit d'installation de *ewd.js* :

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

Ceci est un service Web simple qui va retourner le contenu d'une session EWD.js spécifié comme un document JSON. Si l'Id de session est invalide, un document JSON d'erreur est envoyé à la place.

Dans l'exemple ci-dessus, le service Web EWD *Application Name* est *demo*: le même que le nom du module (sans l'extension de fichier .js). Le module doit se trouver dans le répertoire *node_modules* sous votre répertoire Home EWD.js.

La méthode à appeler est connu comme le Web *Service Name*, dans ce cas *webServiceExample*. Il a un seul argument qui est un objet qui est automatiquement transmis à EWD.js au moment de l'exécution. Par convention, nous appelons cet objet *ewd*. L'objet ewd pour une fonction de service Web EWD.js contient deux sous-objets :

- **mumps** : donnant accès au stockage des globals Mumps et l'héritage des fonctions. Il est utilisé de manière identique aux fonctions de gestion des messages Websocket.
- **query** : contenant les paires entrants nom/valeur qui étaient dans la requête HTTP entrant queryString. Il appartient au développeur de déterminer les noms et les valeurs attendues des paires nom/valeur requises par son/sa fonction qui doivent être fournis par les requêtes HTTP entrantes.

Dès que la fonction est écrite et le module enregistré, il est disponible en tant que service Web dans EWD.js.

Cela est tout ce qu'il y a à faire dans la mesure où le développeur est concerné.

Invocation d'un Service Web EWD.js

Un service web EWD.js est appelée à l'aide d'une requête HTTP avec la structure suivante :

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

Ce qui suit doit être inclus dans la chaîne de requête sous forme de paires nom/valeur:

- les paires nom/valeur requises par la fonction de service spécifique qui est demandé, plus :
- **accessId**: le code d'accès enregistré pour l'utilisateur qui envoie la requête HTTP
- **timestamp:** la date/l'heure actuelle au format JavaScript *toUTCString()* (http://www.w3schools.com/jsref/jsref_toutcstring.asp)
- **signature:** la valeur HMAC-SHA256 empreinte calculée à partir de la chaîne de couple normalisée nom/valeur (voir le lien plus haut pour l'authentification Amazon Web Services SimpleDB pour plus de détails sur l'algorithme de normalisation qui est également utilisé dans EWD.js). par exemple :

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

Bien sûr, les paires nom/valeur doivent être codées URI avant de les envoyer, si la demande se fait comme suit :

```
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=
```

Le Client Node.js EWD.js Web Service

Afin de le rendre facile à utiliser les services Web EWD.js, nous avons créé un module Open Source Node.js qui permettra d'automatiser le processus d'envoi des demandes correctement signés et le traitement de leurs réponses.

Le module est publié à l'adresse: https://github.com/robtweed/ewdliteclient installation est très simple: l'utilisation npm :

```
npm install ewdliteclient
```

Est inclus dans le module un exemple montrant comment appeler le service *webServiceExample* qui est défini dans le module *demo.js*. Vous aurez besoin de modifier les paramètres d'adresse IP/nom du domaine, le port etc pour correspondre à votre système de EWD.js. Vous aurez également besoin de changer le code d'accès à celui que vous avez enregistré sur votre système de EWD.js (voir la section suivante).

Pour essayer le service client Web EWD dans le Node.js REPL :

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

Pour utiliser le client Web Service EWD.js dans votre propre application Node.js, vous procédez comme suit :

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

Des clients équivalents peuvent être développés pour d'autres langages. Utilisez le client Node.js et la documentation Amazon SimpleDB comme un guide pour la mise en œuvre de l'algorithme de signature correcte.

Enregistrement d'un Utilisateur sur le Service Web EWD.js

Vous pouvez utiliser l'application EWDMonitor (voir le chapitre précédent sur l'utilisation de cette application). Il suffit de cliquer sur l'onglet Sécurité : vous pouvez ajouter de nouveaux utilisateurs et de maintenir ou de supprimer ceux qui existent déjà au sein de votre interface utilisateur qui devrait être auto-explicatif.

Vous démarrez l'application EWDMonitor dans un navigateur en utilisant l'URL:

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

Modifier l'adresse IP ou le nom d'hôte et le port de manière appropriée en fonction de votre configuration EWD.js.

Les inscriptions aux programmes

Il est possible de créer un utilisateur enregistré par programme. Pour un exemple, voir https://github.com/robtweed/ewd.js/blob/ master/OSEHRA/registerWSClient.js comme point de départ. Vous devrez modifier les chemins etc selon votre configuration EWD.js. Vous aurez également besoin de modifier l'objet *EWDLiteServiceAccessId* qui est défini dans la méthode *zewd.setDocument()* dans le fichier de registerWSClient.js.

Votre version doit être enregistré et exécuté à partir de votre répertoire Home EWD.js, par exemple :

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

Conversion du code Mumps dans un service Web

EWD.js rend très simple à exposer de nouvelles ou un héritage de code Mumps en tant que service Web JSON/HTTP qui peut être invoqué en toute sécurité. Il consiste à construire deux Wrapper simples autour de votre code Mumps :

Fonction intérieur Wrapper

Cette fonction doit entourer le code Mumps que vous souhaitez exposer en tant que service Web. Notez que vous pouvez également utiliser cette technique pour exposer les méthodes de classe Caché. La fonction wrapper a deux arguments :

- **inputs**: un réseau local qui contient toutes les paires nom/valeur d'entrée requis par votre code Mumps
- **outputs**: un réseau local qui contient les résultats de votre code Mumps. Ce tableau peut être aussi complexe et aussi profondément imbriquée que vous le souhaitez. EWD.js convertira automatiquement vos sorties contenu du tableau à un objet de réponse JSON correspondante qui sera renvoyée par le service Web.

La fonction wrapper doit utiliser *New* commandes à veiller à ce que les variables créées par votre code Mumps ne fuit pas hors de la fonction. En outre, toutes les données requises par votre code Mumps doivent être fournis par le tableau des *inputs* : **vous ne pouvez pas** compter sur les données à l'échelle des Globals être disponibles et les fuites dans la fonction wrapper.

L'exemple ci-dessous montre comment la procédure VistA GET^VPRD peut être enroulé pour être utilisé avec EWD.js :

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

Enregistrer ce dans une routine de Caché nommée ^vistADemo ou dans un fichier de routine GT.M nommé vistADemo.m

Nous avons donc maintenant, une fonction claire, autonome réutilisable que l'on peut utiliser pour appeler notre procédure héritée de Vista.

Fonction extérieur Wrapper

L'étape suivante consiste à créer une seconde, fonction externe wrapper. Ceci fournit une interface standard, normalisée que EWD.js peut invoquer. Il mappe les entrées et sorties d'un tableau dans Global temporaire qui EWD.js sait utiliser. Ce Global est nommé *^%zewdTemp*. En ajoutant un premier indice qui définit le processus Id, nous pouvons veiller à ce que plusieurs processus EWD.js ne seront pas encombrés par des données écrites sur ce Global temporaire. Le processus Id sera automatiquement répercuté à la fonction wrapper extérieure par EWD.js.

Ceci est tout ce que nous devons ajouter à la routine de Caché ou au fichier routine GT.M que nous avons créé plus tôt :

```
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
```

Invoquant Wrapper extérieure de votre module Node.js

EWD.js fournit une fonction intégrée EWD.js qui peut invoquer votre code Mumps wrappered:

ewd.util.invokeWrapperFunction(MumpsFunctionRef, ewd);

Voici un exemple d'un module de back-end Node.js qui peut être utilisé pour appeler la fonction VistA :

```
module.exports = {

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

}
```

Invoquant comme un service Web

Votre code des Mumps peut maintenant être invoqué de la même manière que tout service Web EWD.js. Par exemple, si le module de back-end Node.js ci-dessus est nommé *vistATest.js*, nous pourrions envoyer une requête HTTP(S) demande à partir d'un système client à la recherche de quelque chose comme ça (encodage d'URL ne sont pas utilisées pour plus de clarté). L'adresse IP, le port et les valeurs pour l'accès Id, horodatage et signature varieraient bien sûr :

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

Cette commande appelle la méthode *getVistAData* dans le module *vistATest* et *IEN* sera adoptée automatiquement une entrée. Toutes les paires nom/valeur dans l'URL (en dehors de *accessid, timestamp* et *signature*) sont automatiquement mises en correspondance avec les tableau *inputs* de toute fonction wrapper Mumps que vous appelez via *ewd.invokeWrapper()*. Donc, si votre fonction Mumps nécessite plusieurs entrées, il suffit de les ajouter à la liste de paire nom/valeur de la requête HTTP. Notez que les noms d'entrée sont sensibles à la casse.

La réponse qui sera reçu en retour de ce service web sera livré comme une charge utile JSON, contenant le contenu du tableau de sortie de la fonction Mumps, exprimée en tant qu'objet JSON équivalent.

Invoquer le Web Service de Node.js

Si vous voulez invoquer une interface de service Web EWD.js à votre code Mumps au sein d'un module Node.js (par exemple de EWD.js fonctionnant sur un système distant), alors vous pouvez utiliser le module *ewdliteclient* qui a été décrit plus tôt dans ce chapitre. Ce module sera automatiquement occuper tous les mécaniciens de signature numérique, vous permettant simplement de se concentrer sur la définition de l'interface d'appel.

Par exemple, le service Web décrit ci-dessus peut être appelé à partir d'un EWD.js distants (ou d'autres) Node.js système comme suit :

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

Invoquer le Web Service avec d'autres langages et environnements

Tout ce que vous devez faire pour invoquer une interface de service Web EWD.js à votre code Mumps de toute autre langue ou de l'environnement est de construire une requête HTTP bien-signé du type indiqué précédemment.

Vous aurez besoin de mettre en œuvre les règles de signature selon les règles décrites dans la documentation Amazon Web Services: Voir

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

Vous pouvez utiliser l'application JavaScript dans le module de *ewdliteclient* comme un guide :

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

l'interface client doit attendre une réponse JSON.

Désactivation le Service de Sécurité Web

Pendant le développement et le test de vos services web, il peut être pratique de désactiver la sécurité basée sur la signature numérique intégré.

En outre, si vous utilisez le EWD REST serveur pour l'accès externe (https://github.com/robtweed/ewdrest) et si (**et seulement si**) le serveur EWD.js est protégé/pare-feu d'un accès externe, il y a des avantages de performance de la désactivation du mécanisme de sécurité.

Pour désactiver la sécurité de service Web, ajoutez le paramètre suivant dans votre fichier de démarrage EWD.js :

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

Assurez-vous que vous comprenez les risques de sécurité de le faire! La désactivation de la sécurité intégré permet à quiconque d'accéder et invoquer des services Web que vous avez défini !

Mise à jour de EWD.js

Mise à jour de EWD.js

ewd.js est continuellement améliorée, donc il vaut mieux vérifier sur GitHub pour voir si une mise à jour a été publiée. Des annonces seront faites sur Twitter : *@rtweed* pour les recevoir.

EWD.js est mis à jour en mettant à jour le module *ewd.js*.

Node.js et *npm* rendent trivialement simple à mettre à jour *ewd.js*. Il suffit d'aller dans le répertoire Home de EWD.js et d'utiliser *npm install*, par exemple:

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

Des mises à jour et des améliorations de l'application *ewdMonitor* seront inclus dans le package d'installation. Vous serez invité à confirmer le chemin dans lequel EWD.js a été installé à l'origine, par exemple :

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

Si le chemin suggéré est correct, appuyez simplement sur la touche Entrée, dans le cas contraire tapez le bon chemin. Les dernières versions des sous-composantes essentielles de EWD.js seront installés par rapport à cette voie.

Vous serez alors invites à dire si vous souhaitez installer les sous-composants supplémentaires, en option dans EWD.js :

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

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

Suivez les instructions et EWD.js va maintenant être entièrement mis à jour et prêt à redémarrer.

Notez que les applications EWD.js, les fichiers de démarrage, etc qui ont été créés à l'origine par les EWD.js processus d'installation sera remplacé par cette procédure de mise à jour. Vos propres applications et/ou des fichiers nommés différemment resteront inchangés. Si vous avez modifié des fichiers qui ont été installés à l'origine par EWD.js, assurez-vous de renommer, copier et/ou de les sauvegarder avant de lancer la mise à jour afin d'éviter de perdre vos modifications.

Annexe 1

Création de GlobalsDB basé sur EWD.js/système Ubuntu de Scratch

Contexte

EWD.js comprend un script d'installation automatisée pour la construction d'un environnement, entièrement travailler système complet de EWD.js basés sur GlobalsDB à partir de zéro en seulement quelques minutes. Le point de départ est une Ubuntu bureau ou un serveur système 64 bits, soit une machine physique, machine virtuelle ou une instance EC2. Le programme d'installation EWD.js va installer et configurer :

- NVM (Le Node.js Version Manager, permettant la mise à jour simple et rapide de Node.js à l'avenir)
- Node.js
- GlobalsDB
- NodeM
- EWD.js

Charger et exécuter l'Installer

Démarrez une session de terminal sur votre machine Ubuntu et procédez comme suit :

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

Au cours du processus d'installation, il vous sera invité à confirmer le chemin dans lequel EWD.js a été installé. Simplement accepter le défaut qu'il suggère en appuyant sur la touche Entrée, à savoir :

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

Les sous-composants essentiels de EWD.js seront installés par rapport à ce chemin.

Un invite vous demandera si vous souhaitez installer les sous-composants supplémentaires, en option pour EWD.js :

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

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

Si vous êtes nouveau à EWD.js, tapant Y (suivie de la touche Entrée) est recommandée.

EWD.js est maintenant prêt à l'emploi !

Démarrez ewd.js

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

Exécuter l'application *ewdMonitor*

Dans votre navigateur, entrez l'URL (modifier l'adresse IP de manière appropriée par rapport à celle attribuée à votre machine Ubuntu) :

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

Si vous souhaitez utiliser SSL, modifier les paramètres par défaut de l'objet dans le fichier *ewdStart-globals.js* comme indiqué ci-dessous :

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

Si vous obtenez un avertissement sur les certificats SSL, dites au navigateur qu'il est OK : parce *ewd.js* utilise des certificats auto-signés.

L'application *ewdMonitor* devrait maintenant apparaitre et vous sera demander un mot de passe. Il est défini dans le fichier de démarrage *ewdStartglobals.js* et est pré-défini à *keepThisSecret! [Il est une bonne idée de changer ce mot de passe dès que possible, surtout si la VM dEWDrop a un accès public. Cependant, pour l'instant, laisser par défaut.]*

Entrez ce mot de passe, cliquez sur le bouton *Login* et vous devriez être connecté avec succès avec EWD.js et l'application *ewdMonitor*.

Vous êtes maintenant prêt à construire vos propres applications EWD.js sur votre VM dEWDrop. Aller à l'Annexe 4.

Annexe 2

Création d'une GT.M basée sur EWD.js/Système Ubuntu 14.04 Scratch

Contexte

Ubuntu 14.04 introduit le nouvel apt-get installateur pour GT.M. La nouvelle version de EWD.js comprend un script d'installation, permettant la création d'un système complet, pleinement opérationnel. Le point de départ est un ordinateur de bureau ou un serveur système Ubuntu 14.04, soit une machine physique, machine virtuelle ou une instance EC2. Le programme d'installation EWD.js va installer et configurer :

- NVM (Le Node.js Version Manager, ce qui permet la mise à jour simple et rapide de Node.js à l'avenir)
- Node.js
- GT.M
- NodeM
- EWD.js

Charger et exécuter Installer

Démarrez une session de terminal sur votre machine Ubuntu 14.04 et procédez comme suit :

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

Au cours du processus d'installation, il vous sera demandé de confirmer le chemin dans lequel EWD.js a été installé. Simplement accepter le défaut qu'il suggère en appuyant sur la touche Entrée, à savoir :

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

Les sous-composants essentiels de EWD.js seront installés par rapport à ce chemin.

Vous sera alors demandé si vous souhaitez installer les sous-composants supplémentaires, en option pour EWD.js :

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

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

Si vous êtes nouveau à EWD.js, tapant Y (suivie de la touche Entrée) ce qui est recommandée.

EWD.js est maintenant prêt à l'emploi !

Démarrez ewd.js

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

Exécutez l'application *ewdMonitor*

Dans votre navigateur, entrez l'URL (modifier l'adresse IP de manière appropriée pour que celle attribuée à votre machine Ubuntu) :

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

Si vous souhaitez utiliser SSL, modifier les paramètres par défaut dans le fichier objet *ewdStart-gtm.js* comme indiqué ci-dessous :

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

Si vous obtenez alors un avertissement sur les certificats SSL, dites-le navigateur qu'il est OK : parce *ewd.js* d'utilise des certificats auto-signés.

L'application *ewdMonitor* devrait maintenant surgir dans la vie et vous demander un mot de passe. Ceci est défini dans le fichier *ewdStart-gtm.js* et est pré-défini comme *keepThisSecret! [Il est une bonne idée de changer ce mot de passe dès que possible, surtout si votre VM dEWDrop a accès public. Cependant, pour l'instant, laisser par défaut.]*

Entrez ce mot de passe, cliquez sur le bouton *Login* et vous devriez être avec succès en service avec EWD.js et l'application *ewdMonitor*.

Vous êtes maintenant prêt à construire vos propres applications EWD.js sur la VM dEWDrop. Tourner à l'Annexe 4.

Annexe 3

Installation de EWD.js sur un serveur dEWDrop v5

Contexte

La machine virtuelle (VM) dEWDrop pré-construit basée sur GT.M fournit un environnement de prêt-à-l'emploi, basé sur VistA qui comprenait l'ancienne version "classique" de EWD.

Les instructions suivantes devraient vous permettre de créer rapidement une nouvelle mise à jour complète de l'environnement de travail EWD.js sur une VM dEWDrop v5.

Installation d'une VM dEWDrop

Si vous avez déjà une VM dEWDrop en place et qui fonctionne, passez à la section suivante : *Updating a dEWDrop VM*.

Toutefois, si vous n'avez pas déjà installé la VM dEWDrop, procédez comme suit. Bien que vous pouvez utiliser une variété de forfaits d'hébergement de machine virtuelle, je suppose ici que vous utiliserez la connexion VMWare Player :

Étape 1 :

Téléchargez une copie de la dernière VM dEWDrop (version 5 au moment de la rédaction) à partir de

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

Ce fichier a une taille d'environ 1,6 Go.

Étape 2 :

Créez un répertoire pour la machine virtuelle que vous allez décompresser à partir du fichier zip, par exemple:

~/Virtual_Machines/dEWDrop5

ou sur les machines Windows, quelque chose comme ça :

c:\Virtual_Machines\dEWDrop5

Étape 3 :

Déplacez le fichier zip téléchargé dans ce répertoire

Étape 4 :

Décompressez le fichier zip. Vous aurez besoin d'utiliser l'outil de décompression 7-zip.

Si votre machine hôte est en cours d'exécution Ubuntu Linux, vous pouvez en installer un à l'aide de :

sudo apt-get install p7zip

Si vous utilisez Windows ou Mac OS X, l'outil de décompression Stuffit fera le travail, ou, pour les utilisateurs Windows, consultez :

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

Remarque : si vous utilisez Windows comme machine hôte, vous devez placer votre répertoire machines virtuelles sur un lecteur qui est formaté en NTFS, car le fichier principal (dEWDrop.vmdk) se décompresse à environ 8.7Gb.

Si votre machine hôte en cours d'exécution est Ubuntu Linux, décompresser les fichiers de la VM dEWDrop VM comme suit :

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

Étape 5 :

Vous aurez besoin d'avoir VMWare Player qui est nécessaire pour procéder aux étapes suivantes. Si vous n'avez pas déjà installé VMWare Player, vous trouverez les détails sur la façon de le télécharger et l'installer à l'adresse :

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

Étape 6 :

Démarrez VMWare Player et sélectionnez l'option de menu pour ouvrir un nouveau fichier.

Accédez à votre répertoire *Virtual_Machines/dEWDrop5* et sélectionnez le fichier :

dEWDrop.vmx

(Vous pouvez également utiliser l'utilitaire de gestion de fichiers de votre machine hôte et double-cliquez sur dEWDrop.vmx).

Remarque : VMWare Player peut vous dire qu'il ne peut pas trouver le fichier de disque virtuel. Si oui, recherchez et sélectionnez le fichier dEWDrop.vmdk.

La première fois que vous démarrez la VM dEWDrop, VMWare vous demandera si vous l'avez déplacé ou copié.

Sélectionnez *"I Copied It"* Si elle vous demande si vous voulez télécharger VMware Tools, vous pouvez lui dire de

ne pas le faire.

Étape 7 :

La VM dEWDrop ne devrait pas démarrer à l'intérieur de la console VMWare Player. Quand elle vous demande l'*Username*, entrez :

vista

et quand elle demande le *Password*, entrez :

ewd

Vous devriez maintenant être connecté à la VM dEWDrop, qui est un système Ubuntu 12.04.

Étape 8 :

Trouver l'adresse IP qui a été affectée à la VM par la machine de votre hôte en entrant la commande :

ifconfig

Cherchez la section intitulée eth0 et vous devriez voir une ligne commençant: *inet addr*. Cela devrait vous dire l'adresse IP, par exemple 192.168.1.68

Étape 9 :

Vous devez laisser la console à ce stade et lancez une session de terminal ou de puTTY et établir une connexion SSH au serveur dEWDrop. Utilisez l'adresse l'IP que vous avez découvert à l'étape 8. par exemple à partir d'un terminal Linux/Unix, se connecter en utilisant :

ssh vista@192.168.1.68

Puis entrez le mot de passe: *ewd*

Vous devriez maintenant avoir une VM dEWDrop en état opérationnel.

Mise à jour de la VM dEWDrop

Suivez les étapes ci-dessous à partir d'une session de terminal sur la VM dEWDrop :

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

Au cours du processus d'installation, il vous sera demandé de confirmer le chemin dans lequel EWD.js a été installé. Simplement accepter le défaut qu'il suggère en appuyant sur la touche Entrée, à savoir :

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

Les sous-composants essentiels de EWD.js seront installés par rapport à ce chemin.

Il vous sera alors demandé si vous souhaitez installer les sous-composants supplémentaires, en option pour EWD.js:

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

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

Si vous êtes nouveau sur EWD.js, tapez Y (suivie de la touche Entrée) est recommandée.

Le serveur dEWDrop est maintenant entièrement mis à jour et prêt à l'emploi avec EWD.js.

Le nouveau environnement de travail EWD.js a été créé dans le répertoire ~/ewdjs et tous les travaux liés à EWD.js devrait avoir lieu à l'intérieur et en dessous de ce répertoire. L'accès entre EWD.js et la base de données existante GT.M (avec un système de VistA entièrement fonctionnel) a été automatiquement installé et configuré pour vous par les scripts d'installation.

Mise à jour des Applications Existantes EWD.js

Si vous aviez déjà créé vos propres applications EWD.js à l'aide de l'ancienne version de EWD.js (à savoir la version qui a utilisé le module de ewdgateway2), vous devez copier ces applications dans le nouveau répertoire de l'application : *~/ewdjs/www/ewd* et copier leurs fichiers de module de back-end dans *~/ewdjs/node_modules*.

Vous aurez aussi besoin de modifier les fichiers index.html existants: les références à */ewdLite/EWD.js* et */ewdLite/ewdBootstrap3.js* doit être modifiée par */ewdjs/EWD.js* et */ewdjs/ewdBootstrap3.js* respectivement.

Start Up ewd.js

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

Exécutez l'Application *ewdMonitor*

Dans votre navigateur, entrez l'URL (modifier l'adresse IP de manière appropriée par celle attribuée à votre VM dEWDrop) :

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

Si vous souhaitez utiliser SSL, modifier les paramètres par défaut dans le fichier objet *ewdStart-gtm.js* comme indiqué ci-dessous :

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

Si vous obtenez un avertissement sur les certificats SSL, dites au navigateur qu'il est OK: il l'est parce *ewd.js* utilise des certificats auto-signés.

L'application *ewdMonitor* devrait maintenant s'afficher et vous demander un mot de passe. Ceci est défini dans le fichier de démarrage *ewdStart-gtm.js* et est pré-défini comme *keepThisSecret! [Il est une bonne idée de changer ce mot de passe dès que possible, surtout si votre VM dEWDrop a un accès public. Cependant, pour l'instant, laisser le par défaut.]*

Entrez ce mot de passe, cliquez sur le bouton *Login* et vous devriez être connecté avec succès avec EWD.js et l'application *ewdMonitor*.

Vous êtes maintenant prêt à construire vos propres applications EWD.js sur votre VM *dEWDrop*. Aller à l'Annexe 4.

Appendix 4

Guide d'Initiation à EWD.js

Une simple Application Hello World

Cette annexe fournit un guide complet pour le débutant EWD.js, expliquant comment construire une application très simple qui illustre les principes de base. Nous allons construire une simple application Hello World en utilisant rien de plus qu'une simple page HTML et un couple de boutons : nous n'alons pas utiliser de frameworks fantaisie JavaScript. Nous allons créer deux fonctions de base dans notre application Hello Word :

- Nous allons d'abord créer un bouton qui envoie un message de WebSocket de la page HTML dans le navigateur à un module de back-end où nous allons sauver le contenu du message dans un tableau persistant Mumps ;
- Deuxièmement, nous allons créer un bouton qui envoie un message de WebSocket back-end où il lui donne instruction pour récupérer la première charge utile de message et de le retourner au navigateur.

Ce guide suppose que vous avez déjà installé et configuré EWD.js sur votre système. Si vous ne l'avez pas encore fait, suivez les instructions dans les principaux chapitres de ce document, ou des annexes 1 à 3 pour des installations automatisées rapides.

Alors, commençons.

Votre Répertoire Home EWD.js

Tout au long de la présente annexe, je ferai référence à votre répertoire Home EWD.js. Ceci est expliqué dans le contenu principal de cette documentation, mais en résumé, votre répertoire Home EWD.js est le répertoire où vous étiez lorsque vous avez installé *ewd.js*, soit lors de l'exécution :

```
npm install ewdjs
```

Il contient donc le répertoire ou se trouve le sous-répertoire nommés *node_modules* nommés, à l'intérieur du quel se trouve le module *ewd.js*.

A partir de maintenant, nous allons désigner le répertoire Home de EWD.js par *~/ewdjs*

Sur une machine Windows, ce chemin peut correspondre à :

c:\ewdjs

Lancer le Module *ewd.js*

Utilisez le fichier de démarrage approprié. Par exemple :

Sur un système construit sur GT.M utiliser le programme d'installation décrite à l'annexe 3 :

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

Ou sur une VM dEWDrop :

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

La Page HTML

Créer un nouveau répertoire nommé *helloworld* sous votre répertoire *~/ewdjs/www/ewd*.

Maintenant, créez un fichier nommé *index.html* dans le répertoire *~/ewdjs/www/ewd/helloworld* contenant les éléments suivants :

```
<!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>
```

Notez que comme la Build 54, EWD.js nécessite l'utilisation de JQuery: cela a simplifié la maintenance pour les anciens navigateurs tels que les anciennes versions d'Internet Explorer.

Lancez un navigateur, idéalement Chrome. Si vous utilisez Chrome, ouvrir sa console Outils de développement en procédant comme suit :

- Cliquez sur l'icône de droite du menu supérieur (représenté par trois lignes horizontales)
- Sélectionnez Outils dans le menu déroulant
- Sélectionnez console JavaScript à partir du sous-menu
- Initialement, le panneau de la console est amarré au fond de la fenêtre Chrome. Cliquez sur la 2e icône plus à droite (il ressemble un peu à un terminal d'ordinateur avec une ombre)

Maintenant, lancez votre page HTML en utilisant l'URL (modifier l'adresse IP appropriée pour votre système) :

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

doit apparaître dans votre navigateur ce qui suit :

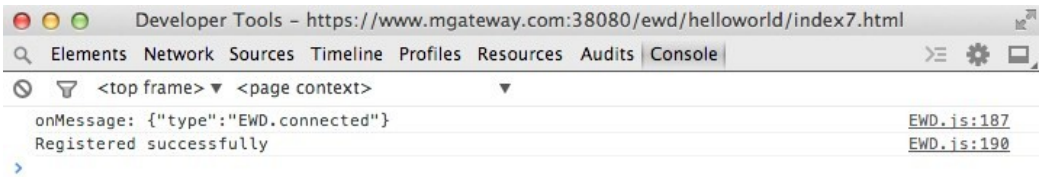
EWD.js Hello World Application

Si vous regardez dans la console JavaScript de Chrome, vous devriez voir ce qui suit:

Ce message d'erreur a été renvoyée par le framework EWD.js. Nous pouvons le résoudre en modifiant la page index.html comme suit :

```
<!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>
```

Cela indique à EWD.js que vous utilisez une application nommée *helloworld*. La dernière ligne indique à EWD.js que vous connecter son activité dans la console JavaScript (sinon EWD.js ne rapporte rien à part des erreurs d'exécution). Actualisez le navigateur et cette fois vous devriez voir un message semblable à celui dans la console JavaScript :



Notre application Hello World ne fonctionne pas correctement avec EWD.js. Maintenant, nous sommes prêts à construire notre application.

Le Fichier app.js

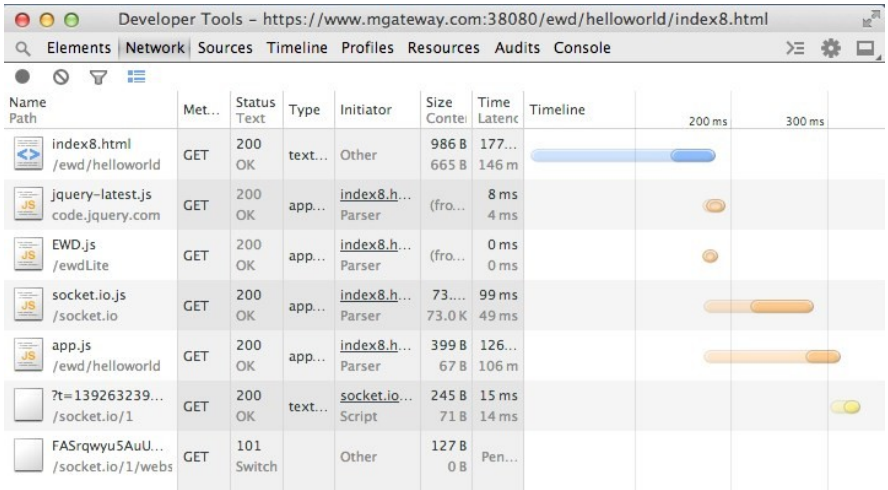
Avant d'aller plus loin, nous allons adopter les meilleures pratiques, et au lieu de créer en ligne JavaScript au sein de votre fichier *index.html*, nous allons passer tout cela dans un fichier JavaScript distinct. Par convention, nous appelons ce fichier *app.js*. Donc, créer un nouveau fichier *app.js* nommées dans le même répertoire que votre fichier *index.html* et déplacer ce Javascript dedans. *app.js* devraient ressembler à ceci :

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

Modifiez votre fichier index.html afin qu'il charge *app.js* :

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

Essayez de recharger l'URL dans le navigateur: l'application doit être exécutée de manière identique. Nous pouvons prouver le fichier *app.js* est le chargé correctement en cliquant sur l'onglet Network en haut de la console JavaScript. Vous devriez voir quelque chose comme cela dans la console :



Sur la 5e ligne, vous pouvez voir que *app.js* a été chargé.

Envoi de Notre Premier Message WebSocket

Ajouter un bouton sur le corps de la page, affecter un *onClick()* gestionnaire d'événement :

```
<!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>
```

Maintenant définir la fonction de gestionnaire *sendMessage()* dans le fichier *app.js*. Cette fonction de gestionnaire va envoyer un message EWD.js WebSocket au back-end. Nous avons décidé de définir le type de message que *sendHelloWorld*. Le message contient un certain nombre de paires nom/valeur dans sa charge utile. Il est à nous de décider du contenu et de la structure de charge utile. La charge utile va dans la propriété nommée *params* :

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

Lorsque vous rechargez le navigateur, vous devriez maintenant voir un bouton qui contient le texte *Send Message*.

Cliquez sur le bouton et vous devriez voir le message d'erreur suivant apparaît dans la console JavaScript.

Eh bien, cette erreur est en fait une bonne nouvelle ! Il nous dit que notre message a été envoyé à l'EWD.js back-end avec succès. Que l'erreur nous dit est que EWD.js n'a pas pu trouver un module *helloworld.js* nommées dans votre répertoire *~/ewdjs/node_modules*. Il est en fait à la recherche d'un module *helloworld* nommé en raison de cette commande, nous avons ajouté plutôt à *app.js* :

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

Donc, il ne faut pas créer le module back-end pour notre application *helloworld*.

Le Module Back-end *helloworld*

Créer un fichier nommé *helloworld.js* dans votre répertoire *~/ewdjs/node_modules* contenant les éléments suivants :

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

Initialement à tout cela va être d'écouter pour tous les messages entrants WebSocket de tous les utilisateurs de l'application *helloworld* et de connecter leurs contenus à la console de *ewd.js*. Rappelez-vous : ce module JavaScript fonctionne en back-end, et pas dans le navigateur !

Recharger la page *index.html* dans le navigateur et cliquez sur le bouton. Maintenant, regardez le module *ewd.js* et vous devriez voir ce qui suit (vous devrez peut-être revenir en arrière à travers les messages assez loin) :

```
*** 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!
```

Le message que nous avons envoyé est en haut !

Ajout d'un gestionnaire de messages de type spécifique

OK alors maintenant nous allons ajouter un gestionnaire spécifique pour les messages de type *sendHelloWorld*. Pour ce faire, remplacez la fonction onSocketMessage() avec un objet nommé *onMessage,* dans lequel nous définissons les gestionnaires de messages spécifiques. Ainsi, afin de gérer notre message de sendHelloWorld, remplacer le contenu du fichier de *helloworld.js* comme suit :

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

Enregistrer cette version modifiée du fichier.

Erreurs de débogage dans votre module

Si vous regardez attentivement la console *ewd.js* lorsque vous avez enregistré la version modifiée du module de *helloworld.js*, vous devriez voir l'apparition suivant dans le journal (en fait, il peut apparaître plus d'une fois) :

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

ewd.js ajoute un gestionnaire d'événements automatiquement chaque fois un module back-end est chargé dans l'un de ses processus enfants : à chaque fois qu'une modification est apportée au module, le processus fils va tenter de le recharger. Notez que si vous avez des erreurs de syntaxe dans votre module, il ne parviendra pas à être rechargé mais malheureusement, le processus enfant est souvent incapable de vous donner tout diagnostic significatif à savoir pourquoi il n'a pas pu charger.

Voici donc un conseil : si votre module ne se charge pas, lancez une nouvelle session de terminal et lancer le Node.js REPL (assurez-vous que dans votre le rpemier répertoire Home de EWD.js) :

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

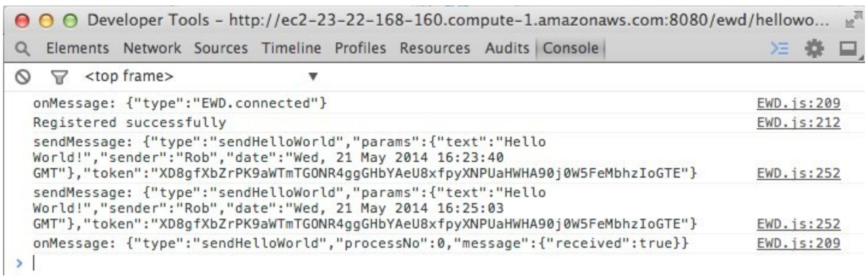
Maintenant, essayez de charger le module dans le REPL en utilisant la fonction *require()*. S'il y a des erreurs de syntaxe, vous serez en mesure de voir ce quoi i s'agit, par exemple :

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

Vous pouvez également utiliser Node Inspector pour déboguer votre application. Pour plus de détails, voir la section intitulée *débogage des Applications à l'aide de Node Inspector* dans le chapitre Création d'une application EWD.js plus haut dans ce document.

Le Gestionnaire de Messages spécifiques de Type Action

Recharger la page *index.html* dans le navigateur et cliquez à nouveau sur le bouton. Cette fois, vous devriez voir le message suivant enregistré dans la console JavaScript :



Nous obtenons plus d'erreurs. Ce dernier message est apparu à la suite du *return*, nous avons ajouté à notre gestionnaire de messages en back-end :

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

Lorsque le module back-end revient avec un objet JSON *returnValue*, un message de WebSocket du même type que le message entrant est renvoyé au navigateur. L'objet *returnValue* est dans la propriété *message* du message de WebSocket reçu par le navigateur, à savoir :

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

Notez que bien que, dans cet exemple, nous sommes rentrés un objet très simple, l'objet JSON nous revenant du back-end peut être aussi complexe et aussi profondément imbriquées que l'on peut les aimer.

Donc, nous avons été en mesure de traiter le message entrant correctement et renvoyer une réponse au navigateur. Maintenant, regardons comment nous pouvons stocker ce message entrant dans la base de données Mumps.

Stockage de votre enregistrement dans la base de données Mumps

Dans cet exercice, on ne va pas faire quelque chose de trop complexe. Tout ce que nous allons faire est de sauvegarder l'objet *message* entrant directement dans un tableau persistant Mumps en utilisant la méthode *_setDocument()*. Nous allons utiliser un nom de tableau de*%AMessage* parce que cela va apparaître dans le haut de la liste des noms des objets persistants quand on va le chercher à l'application ewdMonitor.

Donc, éditez le fichier de module *helloworld.js* comme suit :

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

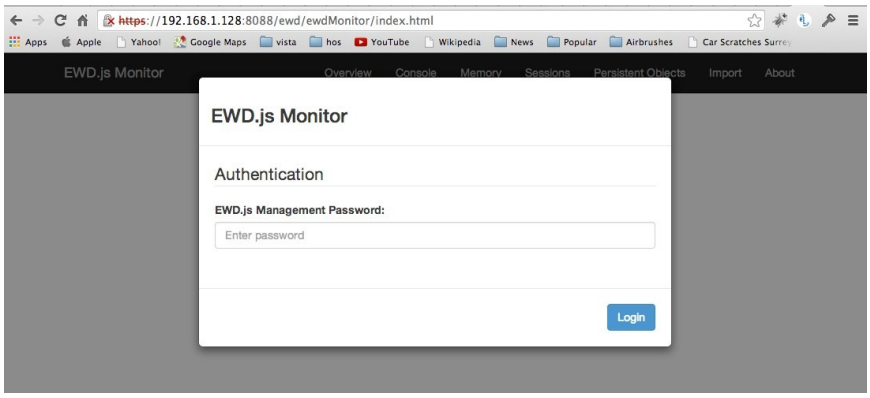
Recharger la page en clickant sur le bouton. Vous devriez voir se modifier le message JavaScript figurant dans la console. Maintenant, Nous allons découvrir ce qui est arrivé au message back-end. Pour cela, nous allons utiliser l'application *ewdMonitor*.

Utilisation de l'Application Monitor EWS pour inspecter la base de données Mumps

Ouvrir un nouvel onglet du navigateur et démarrer l'application ewdMonitor en utilisant l'URL (modifier l'adresse IP, le cas échéant) :

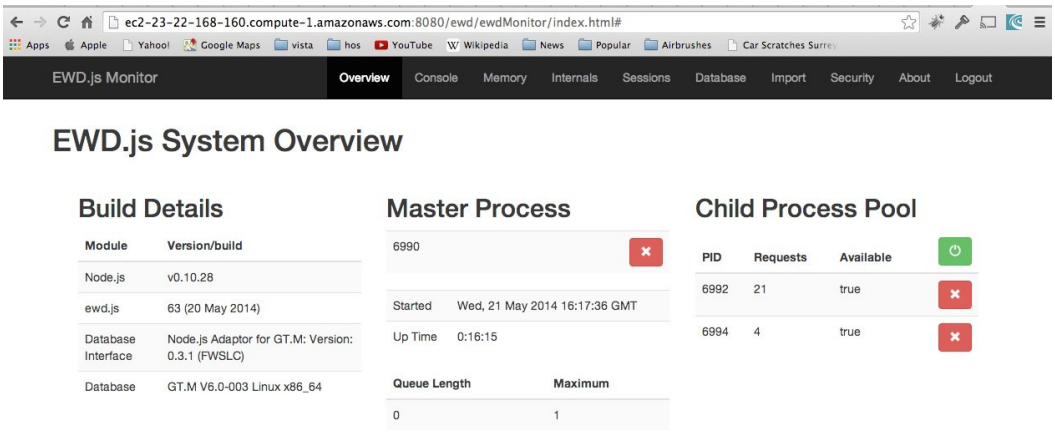
```
http://192.168.1.101:8080/ewd/ewdMonitor/index.html
```

L'application *ewdMonitor* devrait démarrer et vous demander un mot de passe.

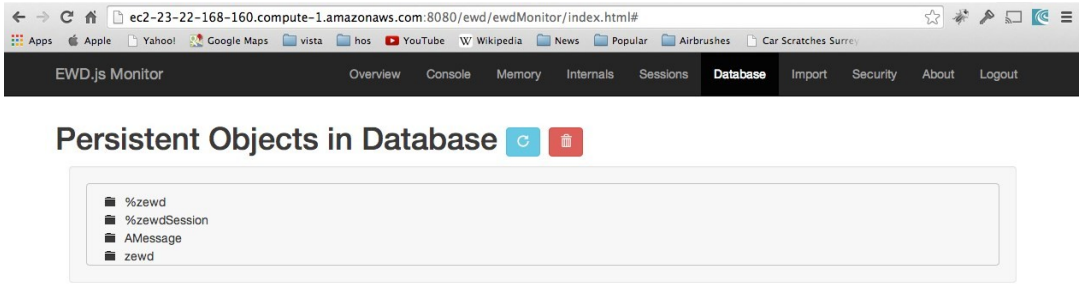


Si vous ne modifiez le mot de passe dans le fichier de démarrage pour *ewd.js* (que vous devez faire une fois que vous commencer à utiliser EWD.js sérieusement), vous devez entrer le mot de passe par défaut qui est *keepThisSecret!*

L'application *ewdMonitor* apparaître dans le navigateur :



Cliquez sur l'onglet nommé *Database*. Vous devriez voir apparaître une liste des Globals Mumps - la liste que vous voyez dépend du type de système que vous utilisez et vous pouvez avoir besoin de faire défiler la liste :



Dans la liste, vous trouverez l'objet persistant que nous avons créé : *AMessage*. Cliquez sur le symbole de dossier à côté du nom pour l'étendre, et répéter pour développer tous ses niveaux. Vous devriez voir qu'il reflète exactement la structure et le contenu du message de WebSocket vous avez envoyé à partir de votre navigateur :

La différence entre ce dernier et le message WebSocket est que cette version est stocké de façon permanente sur le disque (au moins jusqu'à ce que nous décidons de le supprimer ou le modifier).

Notez que nous n'avons pas effectuer de pré-déclarer ou quoi que ce soit pour enregistrer ces données à la base de données Mumps, et on n'a pas eu non plus à définir un schéma : nous avons simplement décidé sur un nom pour notre objet persistant et enregistré le document JSON dedans en utilisant la méthode *_setDocument()* :

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

Le stockage des documents JSON dans une base de données Mumps est trivialement simples !

Essayez de cliquer sur le bouton dans le navigateur, appuyez sur le bouton *Refresh* dans les applications *ewdMonitor* les objets persistants panel et examiner à nouveau le contenu de l'objet *AMessage* : vous devriez voir que la date a été remplacée par la nouvelle valeur du dernier message.

Gestion de la Réponse du Message dans le Navigateur

Il y a juste une dernière étape, que nous devons faire afin de créer un aller-retour complet pour notre exemple de message WebSocket, et qui est de gérer correctement la réponse qui a été renvoyé au navigateur. Pour ce faire, nous avons besoin d'ajouter un gestionnaire de message WebSocket au fichier *app.js*, et un espace réservé pour l'affichage d'un message dans le fichier *index.html*.

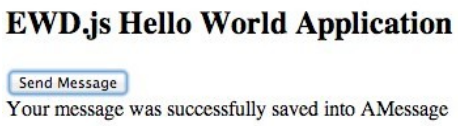
Ajouter une balise *div* à votre fichier *index.html* comme suit :

```
<!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>
```

Puis ajouter le gestionnaire de message à *app.js* :

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

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

EWD.sockets.log = true;

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

L'objet *EWD.application.onMessage* est utilisé pour définir des gestionnaires pour les messages entrants WebSocket du EWD.js back-end. Chaque gestionnaire est une fonction dont le nom correspond à la propriété type du message entrant : dans ce cas *sendHelloWorld*. Chaque fonction de gestionnaire a un seul argument : l'objet du message entrant. Habituellement, vos gestionnaires de messages modifier l'interface utilisateur d'une certaine façon en réponse à la charge message JSON entrant. Dans cet exemple, nous sommes en train de montrer un message de confirmation qui disparaît à nouveau au bout de 2 secondes.



Bien sûr, votre gestionnaire peut être aussi complexe que vous le souhaitez. Jetez un œil au comportement de l'application *ewdMonitor* pour avoir une idée de ce qui est possible de faire : il s'agit d'une application EWD.js qui peut répondre à toutes sortes de messages WebSocket entrants.

Un deuxième bouton pour récupérer le message Sauvegardé

Ajoutons maintenant une seconde bouton à notre page HTML, et le faire pour récupérer le message enregistré chaque fois qu'il est cliqué. Ceci est vraiment très simple. D'abord modifier le fichier *index.html* comme suit :

```
<!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>
```

Ajouter la fonction *getMessage()* qui gérera le click avec *app.js* :

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

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

EWD.sockets.log = true;

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

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

Vous pouvez probablement voir ce que cela va faire maintenant : en cliquant sur le deuxième bouton, il enverra un message de WebSocket de Type *getHelloWorld* au back-end. Notez que pour ce message, nous n'avons envoi aucune charge utile : nous l'avons essentiellement fait au moyen d'un message à envoyer pour signaler au back-end que nous voulons chercher le message enregistré.

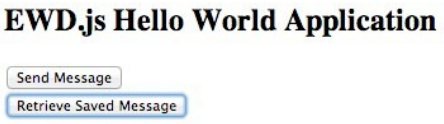
Ajouter un Gestionnaire de Messages Back-end pour le Second Message

Il est clair que nous devons aussi ajouter le gestionnaire d'événements back-end qui répondra à ce message entrant et récupérer le message enregistré. Donc modifier le fichier du module de back-end (à savoir les *helloworld.js*) comme suit :

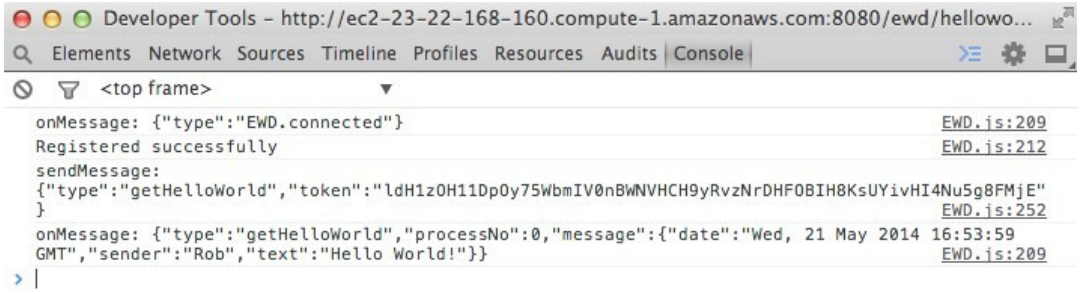
```
module.exports = {
  onMessage: {
    sendHelloWorld: function(params, ewd) {
      var wsMsg = params;
      var savedMsg = new ewd.mumps.GlobalNode('AMessage', []);
      savedMsg._setDocument(params);
      return {savedInto: 'AMessage'};
    }, // don't forget this comma!
    getHelloWorld: function(params,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

Essayez de Lancer la Nouvelle Version

Enregistrez ce fichier et rechargez le fichier *index.html* dans votre navigateur. Il devrait maintenant apparaître comme ceci :



Essayez de cliquer sur le bouton *Retrieve Saved Message* et de regarder la console JavaScript. Vous devriez voir ceci :



Il correspond au message originale JSON que nous avions enregistrés dans la base de données Mumps. Comme vous pouvez le voir, le message reçu a un type *getHelloWorld* et le contenu JSON sauvé a une structure de type propriété de message.

Ajouter un Gestionnaire de Messages au Navigateur

Alors maintenant, tout ce que nous avons à faire est d'ajouter un gestionnaire à notre fichier *app.js* pour afficher les détails du message récupérés dans le navigateur :

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

Maintenant, lorsque vous cliquez sur le Retrieve Saved Message vous verrez les résultats dans le navigateur :

Dans l'exemple ci-dessus, je suis juste le transfert du message brut JSON. Essayez de séparer ses composants et de les afficher correctement dans le navigateur.

Aussi, essayez de cliquer sur *Send Message* puis le bouton *Retrieve Saved Message* : chaque fois que vous faites cela, vous aurez une nouvelle version du message à revenir. Vous serez en mesure de le vérifier parce que la valeur de date sera différente.

Gestionnaire Silencieux et Envoi de Messages Multiples Back-end

Il y a une dernière chose à essayer. Un gestionnaire de messages back-end n'a pas à envoyer un message de retour du tout : par exemple, nous avons pu faire le gestionnaire silencieux de messages *sendHelloWorld*, à savoir :

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

En outre, un gestionnaire de messages back-end peut envoyer autant de messages qu'il aime au navigateur. Ainsi, par exemple, nous pourrions envoyer le contenu du message enregistré dans un message séparé avec son propre type, et utiliser le retour du gestionnaire de valeur pour simplement signaler au navigateur que le message a été récupéré avec succès, par exemple :

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

Essayez de recharger la page *index.html*, cliquez sur le bouton *Send Message*. Cette fois, vous ne devrait pas recevoir un accusé de réception apparaissant dans le navigateur et vous ne verrez pas de message entrant apparaissant dans la console *Developer Tools.*

Maintenant, cliquez sur le bouton *Retrieve Saved Message* et jeter un œil à la console JavaScript. Vous devriez maintenant voir ce qui suit :

Comme vous pouvez le voir, il a été envoyé au navigateur deux messages distincts de notre gestionnaire back-end, avec des types *savedMessage* et *getHelloWorld*.

Vous pouvez nous l'espérons voir que c'est un cadre très souple et puissant, mais simple à utiliser : il est maintenant à vous de définir comment l'exploiter et comment l'utiliser dans vos applications.

Vous avez vu comment communique EWD.js et comment le moteur de stockage Mumps en back-end peut-être utilisé avec une simple page HTML. Il peut être bien sûr utilisé avec n'importe quel framework JavaScript : Choisissez n'importe votre framwork et commencer à développer avec EWD.js.

Utilisation de EWD.js avec le Framework Bootstrap

Si vous utilisez le framework JavaScript appelé Bootstrap (http://getbootstrap.com/), vous verrez que EWD.js a un support particulièrement avancé avec lui. L'application *ewdMonitor* a été construit en utilisant Bootstrap. Dans la section suivante, nous allons examiner comment votre application peut encore être affinée en utilisant les fonctions liées à Bootstrap avec EWD.js.

Les fichiers modèles Bootstrap 3

Dans votre répertoire ~/ewdjs/www/ewd existe une application nommée bootstrap3. Cela ne veut pas en fait dire que c'est une application fonctionnelle. Au contraire, il s'agit d'un ensemble d'exemples de pages et fichiers fragments que vous pouvez utiliser comme point de départ pour toutes les applications Bootstrap 3 avec EWD.js que vous créez.

Helloworld, avec le style Bootstrap

Créer une nouvelle page *index.html* pour votre application *helloworld* - nommons le *indexbs.html*. Nous ferons cela en copiant le fichier *index.html* du répertoire de l'application bootstrap3. Ça devrait ressembler à ça :

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

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

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

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

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

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

    </style>

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

  </head>
```

```
    <body>

    <!-- Modal Login Form -->

    <div id="loginPanel" class="modal fade"></div>

    <!-- Main Page Definition -->

    <!-- NavBar header -->
    <nav class="navbar navbar-inverse navbar-fixed-top">
      <div class="container">
        <div class="navbar-header">
          <div class="navbar-brand visible-xs" id="ewd-navbar-title-phone"></div>
          <div class="navbar-brand hidden-xs" id="ewd-navbar-title-other"></div>
          <button class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </button>
        </div>
        <div class="navbar-collapse collapse navbar-ex1-collapse">
          <ul class="nav navbar-nav pull-right" id="navList"></ul>
        </div>
      </div>
    </nav>

    <!-- Main body -->
    <div id="content">

      <!-- main CONTAINER -->
      <div id="main_Container" class="container in" style="display: none"></div>

      <!-- about CONTAINER -->

      <div id="about_Container" class="container collapse"></div>

    </div>

    <!-- Modal info panel -->
    <div id="infoPanel" class="modal fade"></div>

    <div id="confirmPanel" class="modal fade"></div>

    <div id="patientSelectionPanel" class="modal fade"></div>

    <!-- Placed at the end of the document so the pages load faster -->
    <script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
    <script src="//www.fuelcdn.com/fuelux/2.4.1/loader.js" type="text/javascript"></script>
    <script type="text/javascript" src="//code.jquery.com/ui/1.10.4/jquery-ui.js"></script>
    <script type="text/javascript" src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></script>
    <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/select2/3.4.5/select2.js"></script>
    <script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/toastr.js/2.0.1/js/toastr.min.js"></script>
    <!--[if lte IE 8]><script language="javascript" type="text/javascript"
src="//cdnjs.cloudflare.com/ajax/libs/flot/0.8.2/excanvas.min.js"></script><![endif]-->
    <script language="javascript" type="text/javascript"
src="//cdnjs.cloudflare.com/ajax/libs/flot/0.8.2/jquery.flot.min.js"></script>
    <script src="/ewdjs/EWD.js"></script>
    <script type="text/javascript" src="/ewdjs/ewdBootstrap3.js"></script>

    <script type="text/javascript" src="app.js"></script>

  </body>
</html>
```

Vous pouvez immédiatement voir qu'il y a beaucoup de choses qui ont chargé dans ce fichier. Tout n'est pas nécessaire pour toutes les applications, mais pour le but de ce tutoriel, cela ne fera pas de mal. Notez que tous les fichiers JavaScript et CSS sont extraits des référentiels Content Delivery Network (CDN). Comme vous vous familiariser avec les différents composants frameworks et les services publics qui sont utilisés avec Bootstrap 3, vous pouvez installer des copies locales et modifier les balises <*script*> et le <*link*> d'une manière appropriée.

Ensuite, trouver le fichier nommé *main.html* dans *bootstrap3* répertoire de l'application et copiez-le dans votre répertoire *helloworld*.

La plupart du travail que vous faites dans les applications EWD.js/Bootstrap 3 a lieu dans le fichier *app.js* et dans les fichiers de fragments qui seront récupéré et injecté dans le fichier *index.html*. Vous devriez vraiment pas avoir besoin de faire beaucoup chose, le cas échéant, des modifications de *index.html*. Pour notre application *helloworld* de démonstration, nous allons simplement utiliser un fichier de fragment unique que nous allons construire autour du fichier de fragment de *main.html*.

Ensuite, copiez le fichier *app.js* à partir du répertoire de l'application *bootstrap3*, le remplacement de la version précédente de *app.js* (peut-être faire une copie du premier original). *app.js* devrait ressembler à ce qui suit :

```
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);
    }
  }

};
```

Modifier *app.js* comme suit (désigné par les lignes en gras) :

```
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
  }

};
```

Ne vous inquiétez pas pour l'instant sur le module back-end. Essayez le chargement de *index.html* dans votre navigateur. Vous devriez voir ce qui suit :

Main Page Title

Jetez un œil dans la console JavaScript. Vous devriez voir ce qui suit :

Notez le message final qui montre la distribution du contenu du fichier de fragment de *main.html*. Notez également que l'application elle-même a enregistré automatiquement.

Avant de commencer à ajouter de la fonctionnalité à votre application, jetez un œil au fichier *app.js* à nouveau. Notez les sections suivantes :

- ***onStartup***: Cette fonction est déclenchée automatiquement lorsque le fichier *index.html*, tous les fichiers JavaScript et le fichier CSS associés ont été chargé et initialisé, et non moment où la bibliothèque *socket.io* a fait une connexion WebSocket au back-end et que EWD.js a enregistré la demande. Cette fonction est l'endroit où vous devez définir vos gestionnaires initiaux pour tous les boutons, etc qui sont définis dans le fichier *index.html*. Dans notre application, c'est à cet endroit où nous récupérons le fragment de *main.html* en utilisant la fonction *EWD.getFragment()*.
- ***onPageSwap***: Cet objet contient toutes les fonctions de gestionnaire que vous souhaitez déclencher lorsque les onglets de navigation dans la barre de navigation Bootstrap sont cliqués. Au départ, on ne va pas utiliser une barre de navigation, de sorte que nous pouvons ignorer ce point pour le moment.
- ***onFragment***: Cet objet contient toutes les fonctions de gestionnaire que vous voulez récupérer quand un fichier de fragment est chargé dans le navigateur. Ceci est l'endroit idéal pour définir tous les gestionnaires pour les boutons, etc qui sont définis dans le fichier de fragment. Ceci est une caractéristique très agréable et important dans EWD.js: il vous permet d'ajouter des fonctionnalités dynamiques, d'exécution à des fichiers fragments autrement statiques.
- ***onMessage***: Cet objet contient toutes les fonctions de gestionnaire de messages WebSocket entrants qui ont été envoyés à partir de EWD.js module back-end de votre application.

Vous pouvez voir que lorsque nous utilisons Bootstrap 3, nous avons beaucoup plus et combinaisons de mécanismes à utiliser pour la gestion des événements que nous avons vu dans notre première version de l'application de base Hello World.

Envoi d'un Message depuis la Page Bootstrap

Tout comme dans notre application original Hello World, nous allons d'abord ajouter à la page un bouton qui va envoyer un message au back-end. Nous pourrions le faire en modifiant le fichier *main.html*, mais soyons un peu plus aventureux et définissons un bouton dans son propre fichier de fragment qui sera récupéré lorsque *main.html* est chargé. Ceci est un peu avancé, mais nécessaire pour bien démontrer l'utilisation de l'objet *onFragment* dans app.js.

Donc, d'abord, créez un nouveau fichier nommé *button1.html* dans le répertoire *helloworld*, contenant les éléments suivants :

```
<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>
```

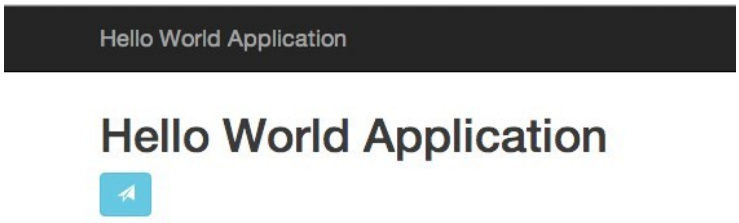
Maintenant modifier *app.js* comme indiqué en caractères gras :

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

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

  },

  onStartup: function() {

    EWD.getFragment('main.html', 'main_Container');

  },

  onPageSwap: {
    // add handlers that fire after pages are swapped via top nav menu
    /* eg:
    about: function() {
      console.log('"about" menu was selected');
    }
    */

  },

  onFragment: {
    // add handlers that fire after fragment contents are loaded into browser

    'main.html': function() {
      $('#mainTitle').text('Hello World Application');
      EWD.getFragment('button1.html', 'mainContent');
    }

  },

  onMessage: {
    // add handlers that fire after JSON WebSocket messages are received from back-end

  }

};
```

Cette fonction se déclenche après le fragment de *main.html* a chargé et fait deux choses :

- modifie le texte du titre du fragment *main.html*. Nous pourrions simplement éditer en dur *main.html*, mais cela montre comment le contenu des fichiers de fragments codés en dur peuvent être modifiés dynamiquement ;
- charger notre nouveau fragment de *button1.html* dans la balise <*div*> au sein de *main.html* qui a l'id: *mainContent*

Essayez de recharger le fichier *index.html* dans le navigateur et vous devriez maintenant voir :



Voici notre belle touche Bootstrap 3, mais il y a deux choses que nous devons ajouter maintenant :

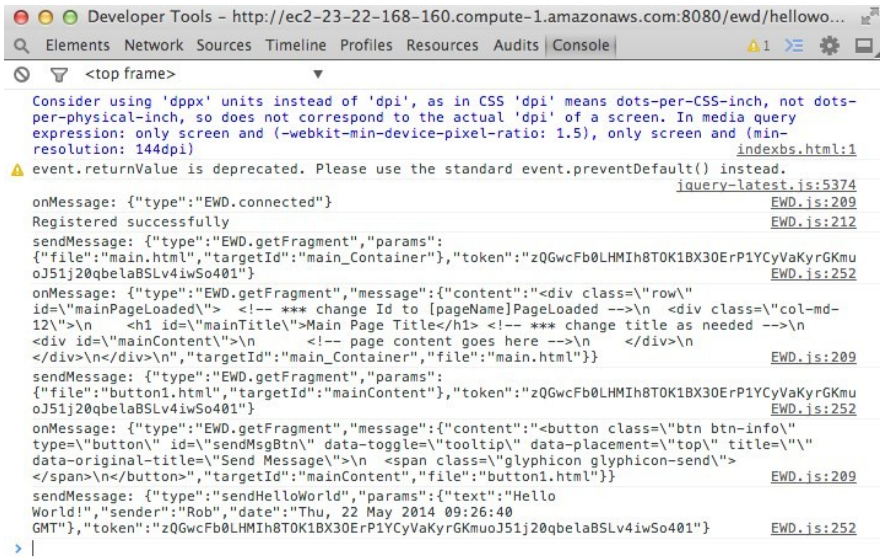
- la balise *button* inclus une info-bulle. Cela n'apparaît pas encore parce que nous avons besoin d'activer l'infobulle

- lorsque vous cliquez dessus, le bouton doit envoyer notre message *sendHelloWorld* au back-end.

Nous pouvons faire ces deux choses en ajoutant une autre fonction de gestionnaire *onFragment*, cette fois-ci il se déclenche après que le fragment *button1.html* ait été chargé. Donc, modifiez la section *onFragment* de app.js comme indiqué en gras ci-dessous :

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

},
```

Maintenant, essayez de recharger la page *indexbs.html* : vous devriez maintenant voir l'info-bulle lorsque vous survolez le bouton. Lorsque vous cliquez sur le bouton, il va envoyer notre message *sendHelloWorld*, tout comme dans notre application originale de démonstration de base. La console JavaScript doit indiquer ce qui suit :



Il n'y a pas de message de réponse revenant du back-end, mais si vous vous souvenez, la dernière chose que nous avons fait était de faire un gestionnaire silencieux. Juste pour prouver que cela fonctionne, nous allons remettre le fichier *helloworld.js* dans *node_modules* à la façon dont il était avant :

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

Et, pour faire bonne mesure, ajouter ce qui suit à l'objet *onMessage* au sein de *app.js* :

```
  onMessage: {

    // add handlers that fire after JSON WebSocket messages are received from back-end

    sendHelloWorld: function(messageObj) {
      toastr.clear();
      toastr.info('Message saved into ' + messageObj.message.savedInto);
    }
  }
```

Essayez-le maintenant - lorsque vous cliquez sur le bouton *save*, vous devriez voir un message Toaster apparaissent dans le coin supérieur droit, ce qui confirme que le message a été enregistré. L'utilitaire *toastr* est l'un des widgets pré-chargés dans le framework EWD.js/Bootstrap 3.

Récupération des Données en Utilisant Bootstrap 3

Enfin, nous allons ajouter un second bouton qui va récupérer notre message précédemment enregistré. Faisons le apparaître seulement après que le bouton *Send* a été cliqué une fois. Nous pourrions faire cela, mais nous allons faire ce qui suit de plusieurs façons. Première modifier *button.html* comme indiqué en gras ci-dessous :

```
<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>
```

Maintenant modifier *app.js* comme indiqué ci-dessous en gras :

```
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));
    }
  }

};
```

Essayez-le - vous devriez voir le deuxième bouton apparaît après la réponse reçue lorsque l'on a cliquer sur le premier bouton. En cliquant sur le deuxième bouton cela permet de récupérer le message enregistré et l'afficher dans le Toaster.

Ajout d'onglets de navigation

Montrons une autre installation qui est inclus dans le framework Bootstrap 3 pour EWD.js.

Tout d'abord, trouver le fichier nommé *navlist.html* dans le répertoire de l'application *bootstrap3* et copiez-le dans le répertoire de l'application *helloworld*. Il est livré avec deux onglets définis : *Main et About*. Nous avons déjà la page principale en fonctionnement, mais vous devez maintenant retrouver *about.html* dans le répertoire de l'application *bootstrap3* et le copier dans le répertoire de l'application *helloworld*.

Si vous regardez dans le fichier *indexbs.html*, vous trouverez pour le sujet de fragment un espace réservé *Container* div. Vous verrez également dans le fichier *app.js*, dans l'objet *navFragments*, ce qui est nécessaire pour faire le travail des onglets Nav avec les fragments *Main et About*. Dans les deux cas, ils seront mis en cache après la première extraction, afin que les clics ultérieures sur les onglets Nav permettent simplement de révéler le contenu existant dans la page, sans avoir à aller chercher l' co,tenu de nouveau.

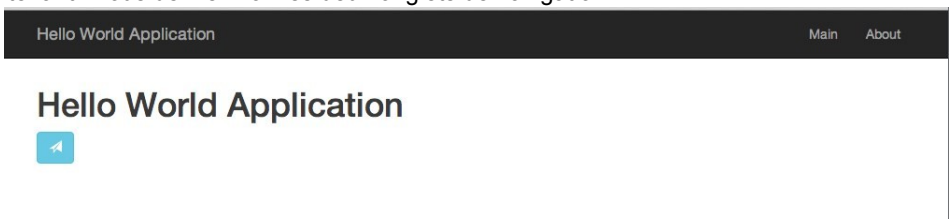
Nous devons juste faire deux changements simples à *app.js* pour permettre le fonctionnement des onglets Nav :

- chercher le fragment de *navlist.html* lorsque l'application a démarré (OnStartup)
- activer les onglets nav après que le fragment *navlist* ait été chargé

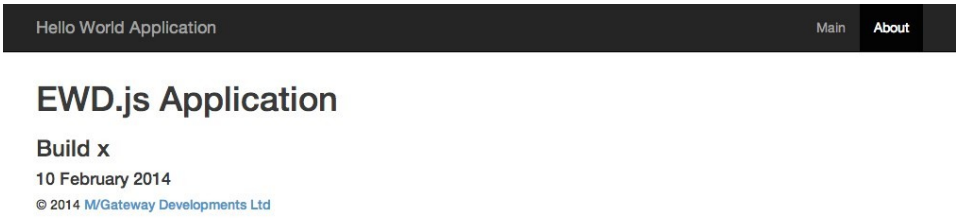
Il suffit de faire les changements indiqués ci-dessous en gras :

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

Essayez-le maintenant - vous devriez voir les deux onglets de navigation :



En cliquant sur l'onglet *About,* cela affiche son contenu qui est initialement comme indiqué :



Il suffit de modifier *about.html* pour modifier son contenu avec tout ce que vous préférez voir !

Vous pouvez ajouter plus d'options Nav que vous souhaitez. Il vous suffit de copier et coller de nouvelles en *navlist.html*. Assurez-vous que vous avez un conteneur d'espace réservé dans *indexbs.html* dans lequel ils seront chargés, et de les ajouter dans l'objet *onPageSwap* dans le fichier *app.js*. La mise en cache est déterminé par une règle très simple: la première étiquette d'un fragment devrait avoir un identifiant de la forme :

[navName]PageLoaded

Par exemple, regardez *about.html et main.html* : leurs premières balises ont les ids de respectivement *mainPageLoaded* et *aboutPageLoaded*.

Conclusions

Voilà la fin de ce tutoriel. Nous espérons que vous pouvez maintenant comprendre comment les applications EWD.js fonctionnent et peuvent être construits. Vous avez également la plupart des informations de base dont vous avez besoin pour construire des applications Bootstrap 3. Pour les techniques plus avancées, examiner le code source de l'application *ewdMonitor* qui est inclus avec EWD.js.

Nous espérons que vous apprécierez le développement d'applications utilisant EWD.js !

Annexe 5

Installation de EWD.js sur Raspberry Pi

Contexte

Raspberry Pi est un ordinateur basé sur une architecture ARM unique à très faible coût qui est devenu extrêmement populaire. Il existe deux modèles A et B qui coûtent environ 25 $ et 35 $ respectivement. Il peut être utilisé avec EWD.js

EWD.js nécessite une base de données Mumps pour fonctionner, mais ni GT.M, Cache ni GlobalsDB ont été porté sur le processeur ARM. Cependant, EWD.js comprend maintenant un module nommé Node.js qui émule une base de données Mumps via un ensemble identique d'API à celles utilisées pour les bases de données réelles Mumps. noDB.js utilise un objet JavaScript pour contenir les données qui autrement seraient stockées et manipulées dans une base de données Mumps. Cet objet JavaScript est régulièrement copié dans un fichier texte nommé noDB.txt pendant le fonctionnement normal de EWD.js. Lorsque EWD.js est (re) démarré, le contenu du fichier noDB.txt sont utilisés pour pré-remplir l'objet JavaScript utilisé par noDB.js. Par conséquent, noDB.js est entièrement persistant et se comporte comme si il était une base de données Mumps mono-utilisateur.

La seule limitation est que lorsque vous utilisez noDB.js, EWD.js ne doit être configuré pour générer un processus enfant unique. Étant donné que nous allons être en cours d'exécution EWD.js sur un Raspberry Pi, ce qui est vraiment pas une limitation importante.

Première Etape avec Raspberry Pi

Lorsque votre Raspberry Pi arrive, vous découvrirez que par défaut, il n'est pas livré avec un système d'exploitation (OS). De plus au lieu d'un disque dur, il utilise une carte SD pour son stockage permanent. Procurez-vous vite une carte SD : la class 10 est actuellement la plus rapide. Choisir une taille de 4Gb ou plus. J'ai aquis une SD de classe 10 de 16Gb pour moins de 10 £ et il fonctionne très bien avec Raspberry Pi.

En utilisant un autre ordinateur, la première chose que vous devez faire est de formater votre carte SD et copier un package d'installation du système d'exploitation appelé NOOBS sur la carte SD. Vous mettez ensuite la carte SD dans la fente SD du Raspberry Pi, brancher tous les câbles, etc et allumez-le. Le programme d'installation de NOOBS se déclenche et il vous sera offert un choix de système d'exploitation à installer. Vous devez choisir Raspbian.

Ces étapes sont décrites en détail ici :

http://www.raspberrypi.org/wp-content/uploads/2012/04/quick-start-guide-v2_1.pdf

Soyez patient : il faut un certain temps à Raspbian pour réaliser l'installation, mais tout devrait fonctionner pour vous.

Pour vous connecter au Raspberry Pi :

username: pi
password: raspberry

Installation de Node.js

Lorsque vous démarrez Raspbian pour la première fois, un ensemble de configurations options apparaîtront : vous devez choisir l'option avancée qui vous permet d'activer le serveur SSH et le client. Cela vous permettra d'accéder à la Raspberry Pi à distance sur votre réseau.

Maintenant, vous êtes prêt à installer Node.js. Ceci est maintenant un processus incroyablement simple. Il suffit d'utiliser les deux commandes décrites,

ici: http://www.raspberrypi.org/phpBB3/viewtopic.php?f=66&t=54817 En résumé, procédez comme suit :

```
cd ~
wget http://node-arm.herokuapp.com/node_latest_armhf.deb sudo dpkg -i node_latest_armhf.deb
```

Il s'agit d'une procédure d'installation assez rapide, surtout si vous avez une carte SD rapide.

Vérifiez l'installation lorsqu'elle est terminée en tapant :

```
node -v
```

Vous pouvez vérifier le numéro de version (0.10.28 au moment de la rédaction).

Voici ce que mon installation donne comme résultat :

```
pi@raspberrypi ~ $ wget http://node-arm.herokuapp.com/node_latest_armhf.deb --2013-11-15 14:56:12--
http://node-arm.herokuapp.com/node_latest_armhf.deb Resolving node-arm.herokuapp.com (node-
arm.herokuapp.com)... 184.73.160.229, 184.73.212.128, 50.17.229.49, ...
Connecting to node-arm.herokuapp.com (node-arm.herokuapp.com)|184.73.160.229|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5537702 (5.3M) [application/x-debian-package]
Saving to: `node_latest_armhf.deb'

100%[======================================>] 5,537,702  1.63M/s  in 3.2s

2013-11-15 14:56:15 (1.63 MB/s) - `node_latest_armhf.deb' saved [5537702/5537702]

pi@raspberrypi ~ $ sudo dpkg -i node_latest_armhf.deb
Selecting previously unselected package node.
(Reading database ... 65274 files and directories currently installed.)
Unpacking node (from node_latest_armhf.deb) ...
Setting up node (0.10.22-1) ...
Processing triggers for man-db ...
pi@raspberrypi ~ $ node -v
v0.10.22
```

Installation de ewd.js

Maintenant, vous êtes prêt à installer le module *ewd.js* :

```
cd ~
mkdir ewdjs
cd ewdjs
npm install ewdjs
```

Soyez patient. Il faut un certain temps à Raspberry Pi pour construire les bibliothèques *socket.io*.

Au cours du processus d'installation, il vous sera demandé de confirmer le chemin dans lequel EWD.js a été installé. Acceptez la valeur par défaut qu'il suggère en appuyant sur la touche Entrée, à savoir :

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

Les sous-éléments *essentiels* de EWD.js seront installés par rapport à ce chemin.

Il vous sera alors demandés si vous souhaitez installer les sous-options *extra* de EWD.js :

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

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

Si vous êtes nouveau sur EWD.js, tapez Y (suivi de la touche Entrée) est recommandée.

Quand c'est terminé, vous devriez trouver que *ewd,js* est installé dans le chemin du répertoire */home/pi/ewdjs/node_modules*. Cela devrait être - vous devriez être prêt à utiliser EWD.js.

Démarrage de EWD.js sur votre Raspberry Pi

Il suffit d'entrer les commandes suivantes :

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

C'est parti :

```
pi@raspberrypi ~ $ cd ewdjs
pi@raspberrypi ~/ewdjs $ node ewdStart-pi

******************************************
**** EWD.js Build 63 (20 May 2014) *******
******************************************

Master process: 2355
1 child Node processes will be started...
Child process 2356 has started
child process returned response {"type":"childProcessStarted","pid":2356,"processNo":0}
Sending initialise request to 2356: {
  "type": "initialise",
  "params": {
    "httpPort": 8080,
    "database": {
      "type": "gtm",
      "nodePath": "noDB"
    },
    "webSockets": {
      "path": "/ewdWebSocket/",
      "socketIoPath": "socket.io",
      "externalListenerPort": 10000,
      "maxDisconnectTime": 3600000
    },
    "ewdGlobalsPath": "./ewdGlobals",
    "traceLevel": 3,
    "logTo": "console",
    "logFile": "ewdLog.txt",
    "startTime": 1400765253179,
    "https": {
      "enabled": false,
      "keyPath": "/home/pi/ewdjs/ssl/ssl.key",
      "certificatePath": "/home/pi/ewdjs/ssl/ssl.crt"
    },
    "webServerRootPath": "/home/pi/ewdjs/www",
    "management": {
      "path": "/ewdjsMgr",
      "password": "keepThisSecret!"
    },
    "no": 0,
    "hNow": 5471674054,
    "modulePath": "/home/pi/ewdjs/node_modules",
    "homePath": "/home/pi"
  }

}

database loaded:

....

**** dbStatus: true
requireAndWatch: /home/pi/ewdjs/node_modules/globalIndexer.js loaded by process 2356
child process returned response {"pid":2356,"type":"log","message":"requireAndWatch:
/home/pi/ewdjs/node_modules/globalIndexer.js loaded by process 2356","processNo":0}
** Global Indexer loaded: /home/pi/ewdjs/node_modules/globalIndexer.js
child process returned response {"pid":2356,"type":"log","message":"** Global Indexer loaded:
/home/pi/ewdjs/node_modules/globalIndexer.js","processNo":0}
child process returned response {"pid":2356,"type":"firstChildInitialised","processNo":0}
First child process started.  Now starting the other processes....
Starting web server...
HTTP is enabled; listening on port 8080
   info  - socket.io started
```

Vous pouvez maintenant essayer les applications de démonstration EWD.js livrées avec le kit d'installation, par exemple essayez d'exécuter *ewdMonitor* dans votre navigateur :

http://192.168.1.112:8080/ewd/ewdMonitor/index.html

Remarque : modifier l'adresse IP de l'URL pour correspondre à celle attribuée à votre Raspberry Pi. Pour la connaître, utilisez la commande :

• *ifconfig*

Le mot de passe *ewdMonitor* est spécifié dans le fichier *ewdStart-pi.js*, et est initialement fixé à *keepThisSecret!*

Vous pouvez maintenant commencer à construire vos propres applications EWD.js. Créez-les dans le répertoire */home/pi/ewdjs/www/ewd/*. Suivez les instructions dans le corps principal de ce document, ou mieux encore, passez maintenant à l'Annexe 4 et suivre le tutoriel.

Installation de MongDB

Non seulement vous pouvez exécuter EWD.js utilisant noDB.js pour simuler une base de données Global, vous pouvez maintenant utiliser MongoDB soit comme une base de données exclusive ou un environnement hybride avec noDB. Dans les deux cas, vous aurez besoin d'installer MongoDB sur votre Raspberry Pi. Ce processus peut être long et problématique - l'approche normale recommandée est de construire MongoDB depuis la source. Cependant, quelqu'un a gentiment fait des binaires pré-compilés disponibles, ce qui rend le processus très rapide et simple.

Vous trouverez ces binaires sur :

https://github.com/brice-morin/ArduPi/tree/master/mongodb-rpi/mongo/bin

Tout ce que vous devez faire est de créer le chemin d'accès :

/opt/mongodb/bin

et copier les quatre fichiers à partir du référentiel Github dans le répertoire *bin* nouvellement créé. Ensuite, délicatement, définissez les autorisations des quatre fichiers pour être exécutable :

```
cd /opt/mongodb/bin
sudo chmod 775 *
```

Maintenant, vous pouvez démarrer le démon MongoDB :

```
cd /opt/mongodb/bin
./mongod -dbpath="/home/pi/mongodb"
```

Cela va créer une nouvelle base de données dans */home/pi/mongodb*. Si le chemin n'existe pas, il sera créé. Vous devriez voir quelque chose comme ceci :

```
pi@raspberrypi /opt/mongodb/bin $ ./mongod --dbpath="/home/pi/mongodb"
db level locking enabled: 1
Sat Jan  4 15:48:56
Sat Jan  4 15:48:56 warning: 32-bit servers don't have journaling enabled by default. Please use --journal if you
want durability.
Sat Jan  4 15:48:56
warning: some regex utf8 things will not work.  pcre build doesn't have --enable-unicode-properties
Sat Jan  4 15:48:56 [initandlisten] MongoDB starting : pid=4416 port=27017 dbpath=/home/pi/mongodb 32-bit hos-
t=raspberrypi
Sat Jan  4 15:48:56 [initandlisten]
Sat Jan  4 15:48:56 [initandlisten] ** NOTE: This is a development version (2.1.1-pre-) of MongoDB.
Sat Jan  4 15:48:56 [initandlisten] **       Not recommended for production.
Sat Jan  4 15:48:56 [initandlisten]
Sat Jan  4 15:48:56 [initandlisten] ** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of
data
Sat Jan  4 15:48:56 [initandlisten] **       see http://blog.mongodb.org/post/137788967/32-bit-limitations
Sat Jan  4 15:48:56 [initandlisten] **       with --journal, the limit is lower
Sat Jan  4 15:48:56 [initandlisten]
Sat Jan  4 15:48:56 [initandlisten] db version v2.1.1-pre-, pdfile version 4.5
Sat Jan  4 15:48:56 [initandlisten] git version: 47fbbdceb21fc2b791d22db7f01792500647daa9
Sat Jan  4 15:48:56 [initandlisten] build info: Linux raspberrypi 3.2.27+ #102 PREEMPT Sat Sep 1 01:00:50 BST
2012 armv6l BOOST_LIB_VERSION=1_49
Sat Jan  4 15:48:56 [initandlisten] options: { dbpath: "/home/pi/mongodb" }
Sat Jan  4 15:48:56 [initandlisten] waiting for connections on port 27017
Sat Jan  4 15:48:56 [websvr] admin web console waiting for connections on port 28017
```

Installation de l'interface Node.js pour MongoDB

Si vous souhaitez utiliser MongoDB avec EWD.js, vous devez installer notre interface synchrone Node.js.

Procédez comme suit :

```
cp /home/pi/ewdjs/node_modules/ewdjs/mongoDB/node-0.10/raspberryPi/mongo.node
/home/pi/ewdjs/node_modules/mongo.node
```

Exécution de EWD.js avec MongoDB exclusivement

EWD.js démarrage en utilisant le fichier de démarrage *ewdStart-pi-mongo.js* :

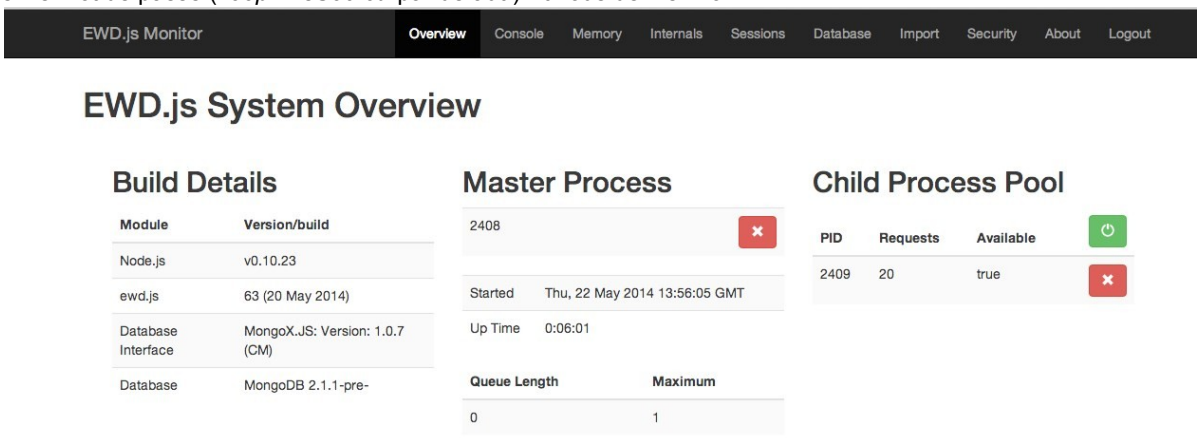
```
cd ~/ewdjs
node ewdStart-pi-mongo
```

EWD.js utilise maintenant MongoDB pour émuler une base de données Global, mais vous pouvez également accéder à MongoDB directement à partir de vos propres applications EWD.js. Voir l'annexe 6 pour plus de détails.

Si vous exécutez l'application ewdMonitor, vous devriez voir les rapports qui utilisent MongoDB comme base de données (modifier l'adresse IP correspondant à votre Raspberry Pi) :

http://192.168.1.113:8080/ewd/ewdMonitor/index.html

Entrez le mot de passe (*keepThisSecret!* par défaut!) Et vous devriez voir :



Essayez de cliquer sur Sessions depuis EWD et sur l'onglet des objets persistants. Bien que EWD.js pense que les deux sont stockées sous forme de Globals dans une base de données Global, les données que vous voyez sont réellement stockées dans MongoDB.

Notez que si vous utilisez MongoDB comme base de données exclusive, vous pouvez, en théorie, utiliser plus d'un processus enfant.

Lancer EWD.js comme un environnement hybride

Vous pouvez également utiliser MongoDB aux côtés de Node.js pour créer un environnement hybride (voir l'annexe 6 pour plus de détails sur les environnements hybrides). Au lieu d'utiliser le fichier MongoDB spécifique EWD.startup, modifier *ewdStart-pi.js* comme suit (les lignes supplémentaires, que vous devez ajouter sont en caractères gras) :

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

var params = {
      poolSize: 1,
      httpPort: 8080,
         https: {
               enabled: false,
      },
      traceLevel: 3,
      database: {
        nodePath: 'noDB',
        also: ['mongodb']
      },
      management: {
        password: 'keepThisSecret!'
     } };
ewd.start(params);
```

Lorsque vous démarrez EWD.js en utilisant ce fichier, il fonctionnera normalement en utilisant Node.js avec un processus enfant unique. Cependant, MongoDB sera disponible pour vos applications. Voir l'annexe 6 pour plus de détails.

Notez que lorsqu'il est exécuté en mode hybride, vous ne pouvez utiliser qu'un processus enfant unique.

Amusez-vous avec votre Raspberry Pi et EWD.js !

Annexe 6

Configuration de EWD.js pour une utilisation avec MongoDB

Modes de fonctionnement

EWD.js peut être configuré pour travailler avec MongoDB d'une des deux manières :

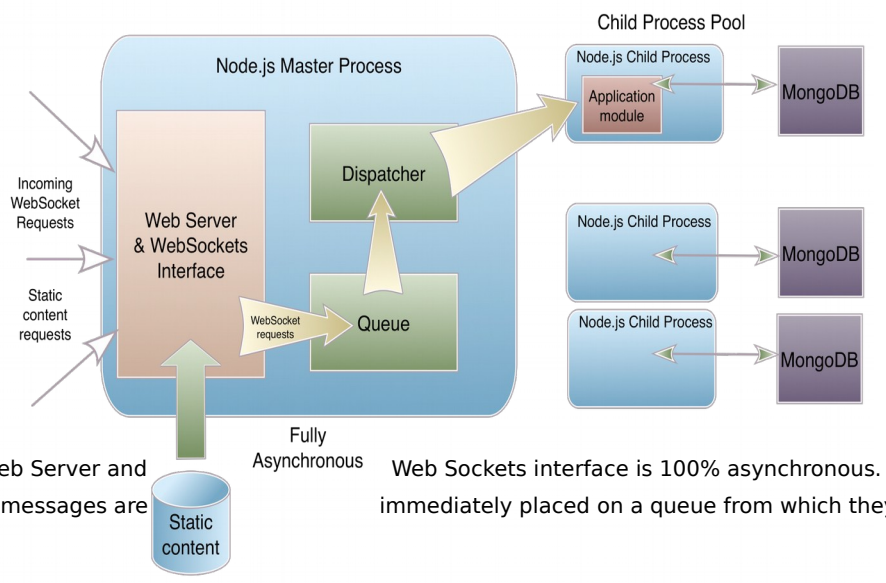
Exclusivement en utilisant MongoDB (sans une base de données Mumps). Dans ce mode de fonctionnement, la mécanique interne de EWD.js qui comptent normalement sur une base de données Mumps et la fonctionnalité intégrée de gestion de session sont fournies par MongoDB. Pour ce faire, EWD.js fait usage d'une émulation des Globals Mumps basé sur MongoDB.

Un environnement hybride, dans lequel une base de données Mumps est utilisée pour soutenir les mécanismes internes de EWD.js et la fonctionnalité intégrée de gestion de session. Par ailleurs, le développeur de l'application peut utiliser MongoDB en utilisant les API synchrones fournies par EWD.js.

API Node.js Synchrones pour MongoDB ?

L'utilisation des API synchrones pour accéder à MongoDB à l'intérieur d'un environnement Node.js peut sembler quelque peu hérétique ! Normalement la sagesse admise est que Node.js doit utiliser des I/O non-bloquant et donc l'utilisation de la logique asynchrone lors de l'accès à une ressource lente, comme une base de données. Alors pourquoi EWD.js utilisent des API synchrones et ne privilégie pas la performance ?

Pour répondre à la deuxième question : non, ils ne le faut pas. Ceci est dû à l'hybride, l'architecture découplée utilisée par EWD.js :



the main Web Server and Web Sockets interface is 100% asynchronous. Incoming WebSocket messages are immediately placed on a queue from which they are dispatched to

Le processus maître de EWD.js fournit l'un processus Node.js enfant. Dès qu'une demande est transmise à un processus enfant, il se retire de la liste d'attente disponible, et retourne à la liste d'attente dès que le traitement est terminé. Chaque processus enfant EWD.js Node.js traite donc qu'une demande seulement de WebSocket à la fois. Cela apporte plusieurs avantages :

- puisque chaque processus enfant n'est seulement manipulé que par la demande d'un seul utilisateur WebSocket, on peut se permettre d'utiliser les I/O bloqués à synchrone, car il ne traite qu'un à la fois

- en déléguant à WebSocket le traitement des demandes de processus enfants, EWD.js peut faire usage de tous les cœurs de processeur disponibles. traitement de calcul intensif n'a aucune incidence sur la performance du processus maître ou le traitement qui a lieu dans d'autres processus de l'enfant

- en étant capable d'utiliser des API synchrones, la logique de l'application back-end peut être écrit de manière intuitive. Le développement d'applications est plus rapide et le code résultant est nettement plus maintenable que ce qui est possible avec la logique asynchrone. En outre, la gestion des erreurs appropriée try/catch peut être utilisé.

EWD.js permet aux processus de l'enfant d'être démarrés et arrêtés dynamiquement sans avoir à arrêter et redémarrer le processus maître.

Utilisation de MongoDB exclusivement avec EWD.js

Si vous voulez utiliser EWD.js juste avec MongoDB seul, il y a trois étapes clés :

- Installez Node.js 0.10.x sur votre serveur. Au moment de la rédaction, la dernière version est 0.10.28.
- Installez MongoDB sur votre serveur
- Installez le module du *ewd.js*. Voir la section *Installing ewd.js* à la page 7 de ce document ainsi que la section *Setting up the EWD.js Environment* pages 8-9.

Allez sur le répertoire *node_modules* sous votre répertoire Home de EWD.js (par exemple *~/ewdjs*), puis allez sur le chemin :

node_modules/ewdjs/mongoDB/node-0.10 et vous trouverez quatre sous-répertoires :

- linux32
- linux64
- raspberryPi
- winx64

A l'intérieur de chacun de ces répertoires, vous trouverez un fichier nommé *mongo.node* qui est une version binaire pré-compilés de notre interface MongoDB synchrone pour Linux 32 bits, Linux 64 bits, le Raspberry Pi et Windows 64 bits, respectivement. Copiez le fichier approprié pour votre serveur dans le répertoire *node_modules* sous votre répertoire de EWD.js (par exemple *~/ewdjs/node_modules*).

Par exemple, si vous installez sur une machine Windows 64 bits, votre répertoire EWD.home est probablement *c:\ewdjs*, auquel cas la copie *c:\ewdjs\node_modules\ewdjs\mongoDB\node-0.10\winx64\mongo.node* vers *c:\ewdjs\node_modules \mongo.node*

Si vous avez correctement suivi les étapes d'installation antérieures (pages 7-9), vous devez également avoir un fichier *mongoGlobals.js* nommé dans le même répertoire *node_modules* comme *mongo.node*. *mongoGlobals.js* regardez après l'émulation des Globals Mumps utilisant MongoDB.

Vous pouvez maintenant commencer EWD.js. Vous trouverez un exemple de fichier de démarrage nommé *ewdStart-mongo.js* dans votre répertoire Home EWD.js. Cela a été écrit en supposant que vous utilisez EWD.js sur un serveur Windows, avec un répertoire de EWD.js de *c:\node*. Tout ce que vous devez faire est de modifier la valeur de *ModulePath* pour correspondre au chemin complet du répertoire *node_modules* de votre serveur (par exemple, sur une machine Linux, il peut être */home/username/node/node_modules*). Démarrez EWD.js de façon normale en utilisant ce fichier :

Linux :

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

Windows:

```
cd c:\ewdjs
node ewdStart-mongo
```

Vous devriez être en mesure d'exécuter l'application ewdMonitor (voir page 15). Il devrait être signalé l'utilisation de la base de données comme étant MongoDB. Tout devrait se comporter comme si vous utilisez une base de données Mumps.

Vous pouvez accéder à MongoDB au sein de vos modules back-end via l'objet ewd.mongoDB qui donne accès à toutes les API d'interface, par exemple :

```
module.exports = {
  onMessage: {
    getVersion: function(params, ewd) {
      var version = ewd.mongoDB.version();
      return {version: version};
    }
  }
};
```

Voir plus loin pour plus de détails sur les API MongoDB disponibles.

Création d'un Hybride Mumps/System MongoDB EWD.js

L'avantage d'un système hybride est que le mécanique interne au sein de EWD.js qui supposent l'utilisation d'une base de données Mumps ne sont pas ralentis par l'inefficacité de l'émulation MongoDB des globals Mumps. Au lieu de cela, il fonctionnera à pleine vitesse. Vous serez également en mesure d'utiliser la session EWD.js sans se soucier de la performance.

En outre, vous aurez plein accès à MongoDB, mais, si vous moderniser l'héritage demande Healthcare Mumps , vous serez en mesure d'accéder à la fois des données Mumps existants et aux données MongoDB.

Pour configurer un environnement hybride, décider d'abord sur la version Mumps que vous souhaitez utiliser. Si vous travaillez avec un ancien système Healthcare, vous devez probablement savoir quelle version vous devez utiliser : par exemple Caché ou GT.M. Si vous voulez juste quelque chose à regarder après la mécanique EWD.js, puis GlobalsDB est probablement la meilleure option : il est gratuit, très rapide, très facile à installer et disponible pour tous les systèmes d'exploitation : parfaite en tant que moteur de session EWD.js !

Utilisez les instructions ailleurs dans cette documentation pour installer et configurer Node.js, la base de données Mumps et le module *ewd.j*s, comme si vous alliez à utiliser une base de données Mumps.

Ensuite, installez MongoDB (si vous ne l'avez pas déjà).

Ensuite, allez vers le chemin: *~/ewdjs/node_modules/ewdjs/mongoDB/node-0.10* et vous trouverez quatre sous-répertoires :

- linux32
- linux64
- raspberryPi
- winx64

A l'intérieur de chacun de ces répertoires, vous trouverez un fichier nommé *mongo.node* qui est une version binaire pré-compilé de notre interface MongoDB synchrone pour Linux 32 bits, Linux 64 bits, le Raspberry Pi et Windows 64 bits, respectivement. Copiez le fichier approprié pour votre serveur dans le répertoire *node_modules* sous votre répertoire de EWD.js.

Par exemple, si vous installez sur une machine Windows 64 bits, votre répertoire Home EWD est probablement *c:\ewdjs*, auquel cas la copie *c:\ewdjs\node_modules\ewdjs\mongoDB\node-0.10\winx64\mongo.node* vers *c:\ewdjs\node_modules* \mongo.node

Enfin, vous devez modifier votre fichier EWD.js de démarrage et ajoutez le database.also['mongodb'], par exemple :

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

var params = {
      poolSize: 1,
      httpPort: 8080,
          https: {
                enabled: false,
      },
      traceLevel: 3,
      database: {
        nodePath: 'noDB',
        also: ['mongodb']
      },
      management: {
        password: 'keepThisSecret!'
     }
};

ewd.start(params);
```

Vous pouvez maintenant démarrer EWD.js comme d'habitude, par exemple :

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

Bien que votre base de données spécifiée Mumps est à la recherche après EWD.js, gestion de session, etc., vous pouvez accéder à MongoDB au sein de vos modules back-end via l'objet *ewd.mongoDB* qui donne accès à toutes les API d'interface, par exemple :

```
module.exports = {
  onMessage: {
    getVersion: function(params, ewd) {
      var version = ewd.mongoDB.version();
      var result = ewd.mongoDB.retrieve("db.ccda", {ccdaNo: 1});
      return {version: version, result: result};
    }
  }
};
```

Résumé de l'API Synchrone MongoDB

Ce qui suit est un résumé des API les plus courantes que vous utiliserez avec EWD.js et MongoDB. Une liste détaillée sera publiée ailleurs en temps opportun. Dans les coulisses, ces API invoquent les API MongoDB standard, donc si vous êtes familier avec MongoDB vous aurez probablement les reconnaître et comprendre leur comportement.

open(connectionObject)

Ouvre la connexion à MongoDB. Vous n'aurez pas besoin d'accéder à cette API - EWD.js le fait pour vous.

```
mongoDB.open({address: "localhost", port: 27017});
```

insert(collectionName, object)

Ajoute un objet à une collection

```
var result1 = mongoDB.insert("db.test", {department: "mumps", key: 11, name: "Chris Munt", address: "Banstead",
phone: 5456312727});
console.log("Insert Record (MongoX sets new _id): Created: " + user.object_id_date(result1._id).DateText);
console.log("Insert Result: " + JSON.stringify(result1, null, '\t'));
```

update(collectionName, matchObject, replacementObject)

Modifie un objet dans une collection. L'objet de correspondance définit les paires nom/valeur qui correspondent à l'objet à être mis à jour.

```
mongoDB.update("db.test", {department: "mumps", key: 3}, {department: "mumps", key: 3, name: "John Smith",
phone_numbers: [{type: "home", no: 909090}, {type: "work", no: 111111}]});
```

retrieve(collectionName, matchObject)

Trouve un ou plusieurs objets dans une collection et le renvoie comme un tableau d'objets. L'objet de correspondance définit les paires nom/valeur qui correspondent à l'objet(s) à trouver.

```
var result = mongoDB.retrieve("db.test", {department: "mumps"});
console.log("Data Set: " + JSON.stringify(result, null, '\t'));

console.log("Get OBJECT by ID (" + result1._id + ")\n");
result = mongoDB.retrieve("db.test", {_id: result1._id});
console.log("Data Set: " + JSON.stringify(result, null, '\t'));
```

remove(collectionName, matchObject)

Trouve un ou plusieurs objets dans une collection et supprime le/les collection(s). L'objet de correspondance définit les paires nom/valeur qui correspondent à l'objet(s) à supprimer.

```
mongoDB.remove("db.test", {department: "mumps", key: 3});

// remove everything from the collection:

mongoDB.remove("db.test", {});
```

createIndex(collectionName, indexObject, [params])

Indices d'une collection, basée sur les paires nom/valeur spécifiées dans indexObject

```
var result = mongoDB.create_index("db.testi", {key: 1}, "key_index",
"MONGO_INDEX_UNIQUE, MONGO_INDEX_DROP_DUPS");

console.log("Result of create_index(): " + JSON.stringify(result, null, '\t'));
console.log("\nGet the new status information for db.testi ... \n");
var result = mongoDB.command("db", {collStats : "testi"});
```

command([params])

Invoque un certain nombre de commandes. Quelques exemples sont énumérés ci-dessous. *listCommands* vous fournira une liste des commandes disponibles :

```
var result = mongoDB.command("db", {listCommands : ""});
console.log("Command (listCommands) : " + JSON.stringify(result, null, '\t'));

result = mongoDB.command("db", {buildInfo : ""});
console.log("Command (buildInfo): " + JSON.stringify(result, null, '\t'));

result = mongoDB.command("db", {hostInfo : ""});
console.log("Command (hostInfo) : " + JSON.stringify(result, null, '\t'));
```

version()

Renvoie la version de l'API MongoDB

```
console.log(mongoDB.version());
```

close()

Ferme la connexion à l'API MongoDB. Remarque: vous n'avez normalement pas besoin d'utiliser cette API lorsque vous utilisez EWD.js

```
mongoDB.close();
```