ewd-document-store

Installation & Reference Guide

Version 1
5 March 2016

What is ewd-document-store?

ewd-document-store is a Node.js module that uses a Global Storage database such as InterSystems' Cache or GT.M to provide:

- persistent JavaScript objects; and, simultaneously
- a fine-grained Document Database where the "unit of storage" can be anything from an entire JSON document to an individual name/value pair anywhere within a JSON document.

Note: ewd-document-store requires synchronous access to a Global Storage database. Therefore if you use it within an application that is supporting the simultaneous activity of multiple users, you must use a module such as ewd-qoper8 to isolate such synchronous access from the main Node.js process. See https://robtweed.wordpress.com/2016/03/03/higher-level-database-operations-with-node-js/ for more information on this.

Installing ewd-document-store

ewd-document-store can be installed on its own using:

npm install ewd-document-store

However, if you are using ewd-qoper8, when you install either ewd-qoper8-cache or ewd-qoper8-gtm (the ewd-qoper8 modules used for interfacing Cache and GT.M respectively within an ewd-qoper8 worker process), then ewd-document-store is installed for you automatically as a dependency.

If you are using InterSystems Cache, then you must make sure that the Node.js interface file - cache.node - is installed. Make sure you're using the version of cache.node appropriate to your operating system and version of Node.js. Versions of cache.node are included in all recent distributions of Cache.

If you are using the Open Source GT.M database, then you must install the Node.js interface module for GT.M, known as nodem:

npm install nodem

Using ewd-document-store

Standalone

To use ewd-document-store standalone, for example within a test-harness script, you must first load the relevant database interface module, open a connection to the database, and then invoke the ewd-document-store abstraction.

Here's an example using Cache:

```
var DocumentStore = require('ewd-document-store');
var interface = require('cache');
var db = new interface.Cache();

// Change these parameters to match your GlobalsDB or Cache system:

var ok = db.open({
  path: '/opt/cache/mgr',
  username: '_SYSTEM',
  password: 'SYS',
  namespace: 'USER'
});

var documentStore = new DocumentStore(db);
```

and using GT.M:

```
var DocumentStore = require('ewd-document-store');
var interface = require('nodem');
var db = new interface.Gtm();

var ok = db.open();
var documentStore = new DocumentStore(db);
```

With ewd-qoper8

ewd-document-store is automatically loaded and invoked when you use ewd-qoper8-cache or ewd-qoper8-gtm within your ewd-qoper8 worker module's on('start') event handler.

For example, to use it with Cache:

```
this.on('start', function() {

  var connectCacheTo = require('ewd-qoper8-cache');
  connectCacheTo(this);

 // the ewd-document-store abstraction of the Cache database Global Storage
 // is now available via this.documentStore

});
```

For more information see: https://github.com/robtweed/ewd-qoper8-cache

To use it with GT.M:

```
this.on('start', function() {

  var connectGTMTo = require('ewd-qoper8-gtm');
  connectGTMTo(this);

 // the ewd-document-store abstraction of the GT.M database is now available via
this.documentStore

});
```

For more information, see: https://github.com/robtweed/ewd-qoper8-gtm

The DocumentStore Object

ewd-document-store makes the database available as a **DocumentStore** object. You can see that all the examples above instantiate an instance of a DocumentStore object.

A DocumentStore object provides access to the Documents stored within it. A DocumentStore represents an entire Global Storage database (in Cache it actually represents a single Namespace), and is a collection of Documents. Each Document has a name, the value of which is up to you, but must start with an upper- or lower-case alphabetic character, followed by up to 30 alphanumeric characters

Methods

The DocumentStore object has a single method - list(). This returns an array containing the document names within the DocumentStore, eg:

```
var documentList = documentStore.list();
```

Constructors

The DocumentStore object provides you with a single Constructor - DocumentNode(). This allows you to create instances of DocumentNode objects that represent any node within a named Document, e.g.:

```
var theNodeIWant = new documentStore.DocumentNode('myDocument', ['a']);
```

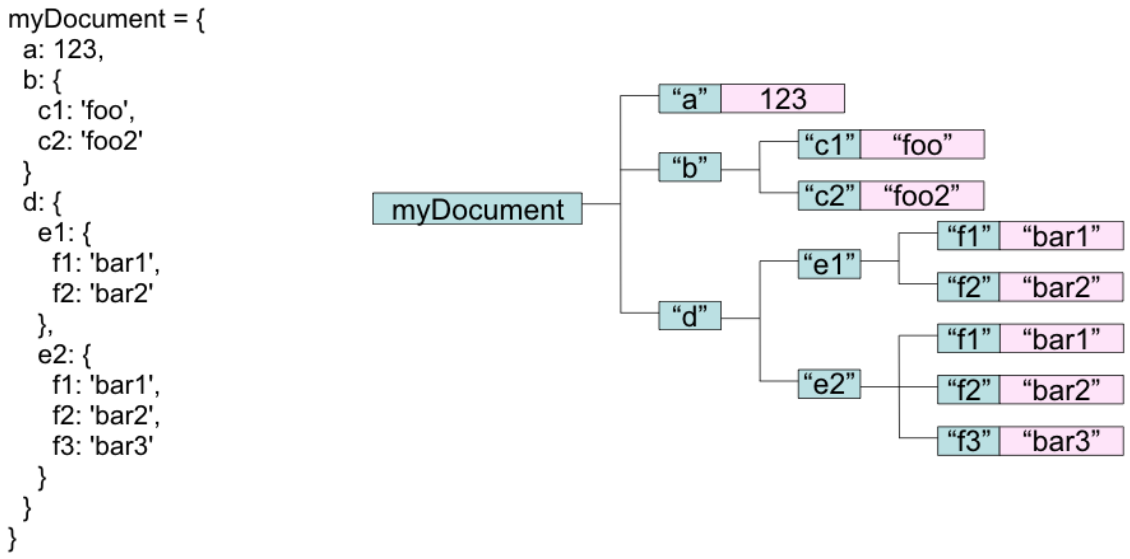
The arguments are:

- **documentName**: the name of the Document within which you want to access a node
- p**ath**: an array of names, representing the specific path you want to access within the Document's hierarchy of nodes.

To understand how ewd-document-store works, you need to understand the concept of DocumentNode objects. This is covered in the next section.

DocumentNode Objects

The diagram below represents how a JSON document (on the left) is stored in the Document Store (on the right):



As you can see, it's represented as a hierarchy of nodes (coloured blue), each of which may either have:

- a value (represented in pink); or
- one or more child nodes

Each node has a name which can be alphanumeric and up to 31 characters in length

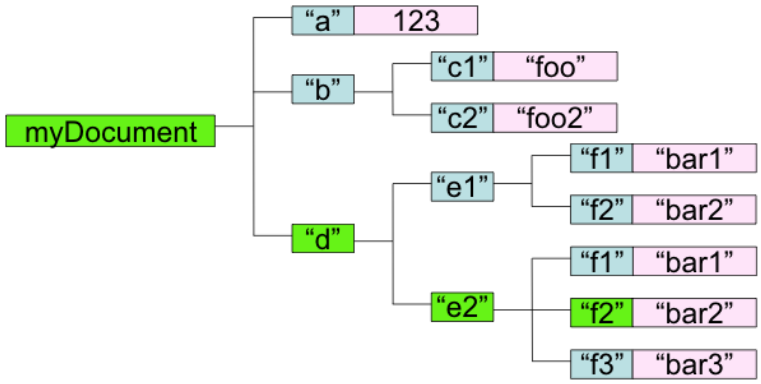
ewd-document-store can represent each and any of these nodes within a Document as a DocumentNode object.

Notice that even the top-level document name is represented by a node which is made accessible by using the Constructor:

var topNode= new documentStore.DocumentNode('myDocument');

Suppose, then, that we wanted the node in the Document named *myDocument* that represents *myDocument.d.e2.f2* The path of nodes needed to access this node within the database is shown in green below:

```
myDocument = {
  a: 123,
  b: {
    c1: 'foo',
    c2: 'foo2'
  }
  d: {
    e1: {
      f1: 'bar1',
      f2: 'bar2'
    },
    e2: {
      f1: 'bar1',
      f2: 'bar2',
      f3: 'bar3'
    }
  }
}
```

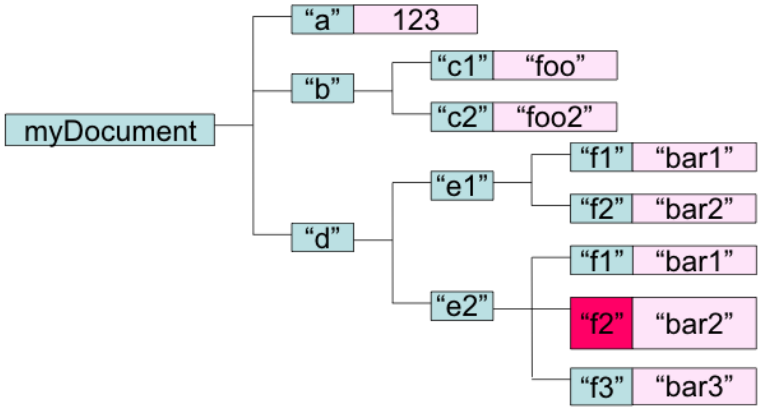


To create an instance of a DocumentNode object that represents this node, we would use the following Constructor:

var theNodeIWant= new documentStore.DocumentNode('myDocument', ['d', 'e2', 'f2']);

So *theNodeIWant* would actually represent the actual node shown below in red:

```
myDocument = {
  a: 123,
  b: {
    c1: 'foo',
    c2: 'foo2'
  }
  d: {
    e1: {
      f1: 'bar1',
      f2: 'bar2'
    },
    e2: {
      f1: 'bar1',
      f2: 'bar2',
      f3: 'bar3'
    }
  }
}
```



There's one vitally important thing to understand about a DocumentNode object - it doesn't actually have to physically exist in the database when you instantiate the object. Indeed, the document itself doesn't need to physically exist when you instantiate a DocumentNode object. At this stage, it's representing a potential physical database storage node that we're going to decide what to do with. There's a number of things we can do with a DocumentNode object, including:

- physically creating it on disk by giving it a value (if it doesn't already exist);
- getting its value if it does already exist and has a value;
- changing its value if it already exists;
- deleting it (which also deletes any sub-nodes that might exist beneath it);
- retrieving the sub-tree of nodes beneath and returning them as a JavaScript object
- appending a JavaScript document as a sub-tree of nodes beneath it
- using it as a starting point for traversal and/or exploration of the Document's hierarchy of nodes

For example, if a document named *myDocument* does not yet exist in your Document Store and you do the following:

```
var firstNode= new documentStore.DocumentNode('myDocument', ['d', 'e2', 'f2']);
firstNode.value = 'bar2';
```

then the following will be created on disk:

In other words, this one action to create the leaf 'f2' node has automatically created a 'myDocument', 'a' and 'e2' node as well.

We could now add another node to this document - for example myDocument.b.c2. To do so, we could do this:

```
var secondNode= new documentStore.DocumentNode('myDocument', ['b', 'c2']);
second.value = 'foo2';
```

But we could alternatively do this:

```
firstNode.parent.parent.parent.$('b').$('c2').value = 'bar2';
```

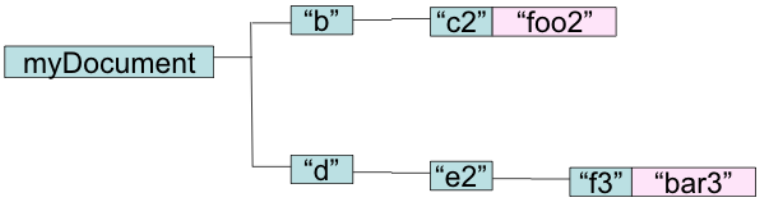
Or this:

```
var myDocument = new documentStore.DocumentNode('myDocument');
myDocument.$('b').$('c2').value = 'bar2';
```

Or even this:

```
var myDocument = new documentStore.DocumentNode('myDocument');
var obj = {
  b: {
       c2: 'bar2'
     }
  }
};
myDocument.setDocument(obj);
```

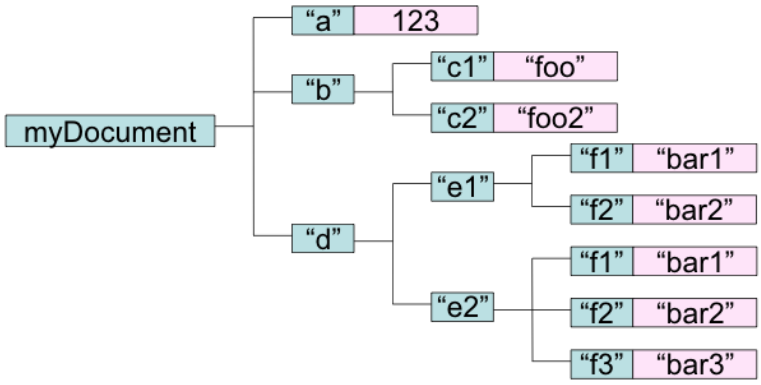
In all three cases, the outcome would be the same. The physical Document would now look like this:



You'll notice that a DocumentNode object's setDocument() method allows you to insert sub-documents into an existing document, and this happens directly to disk.

You'll also see from the diagram that the the immediate child nodes of the top-level myDocument node are automatically sorted based on the alphanumeric collating sequence of the node names. This is a really important and powerful feature of the underlying Global Storage database engine.

We can save subdocuments and/or individual nodes randomly anywhere into this document, and the child nodes of any particular parent node will always be stored on disk in alphanumeric order:



So if we retrieve the child nodes of the top-level parent node using its forEachChild() method, they will be retrieved in that collated pre-sorted order:

```
var e2 = new documentStore.DocumentNode('myDocument', ['d', 'e2']);
e2.forEachChildfunction(nodeName) {
  console.log(nodeName);
});
```

Will display:

```
f1
f2
f3
```

Remember that the ewd-document-store APIs access the underlying database synchronously, so the callback function of each iteration of the forEachChild() method will be triggered in nodeName collating sequence.

You don't have to work directly to disk with a Document in your Global Store: you can retrieve all or parts of the document as JavaScript objects, eg: using the Document depicted above:

```
var e2 = new documentStore.DocumentNode('myDocument', ['d']);
var obj = e2.getDocument();  // {e2: {f1: 'bar1', f2: 'bar2', f3: 'bar3'}}
```

You could modify this in-memory object and then replace the original in the on-disk Document:

```
var dNode= e2.parent;
e2.delete(); // delete the e2 node from the Document and all its sub-nodes
dNode.setDocument(newObj); // attach the new object to its parent
```

And of course all the newly added nodes would be automatically collated / sorted on disk.

This description has introduced you to a number of the DocumentNode's properties and methods. The following describes the complete set that is available:

Properties

value	read/write	The value of the Document Node, if any. Returns an empty string if the node doesn't exist Values can be text and numeric. On-disk text values of "true" and "false" are mapped to and from JavaScript true and false booleans
name	read	The Document Node's name
exists	read	Returns true if the Document Node exists on disk and has either a value or sub-nodes
hasValue	read	Returns true if the Document Node exists on disk and has a value
hasChildren	read	Returns true if the Document Node exists on disk and has one or more child Nodes
parent	read	Returns a DocumentNode object representing the node's parent node
firstChild	read	If the Document Node has child nodes, this returns the first child node as a DocumentNode object. Otherwise returns undefined
lastChild	read	If the Document Node has child nodes, this returns the last child node as a DocumentNode object. Otherwise returns undefined
nextSibling	read	If the Document Node has a next sibling Node (following it in DocumentNode.name collating sequence), it is returned as a DocumentNode Object. Otherwise returns undefined
previousSibling	read	If the DocumentNode has a previous sibling Document Node (preceding it in DocumentNode.name collating sequence), it is returned as a DocumentNode Object. Otherwise returns undefined

Methods

Method Name	Arguments	Description
$	nodeName	Returns the child Node with the specified Node Name as a DocumentNode object Additionally, the DocumentNode object that invoked this function has the returned DocumentNode added to it as a new property - the name of this property is '$' followed by the specified node name. For example if you invoke: myNode.$('abd').value = 'hello world'; then myNode will now have a property named $abd that can be used to access this new Document Node So if you want to access the same node again, you can do this: var value = myNode.$abd.value;
delete	n/a	Deletes the current GlobalNode from disk, along with the entire tree of sub-nodes beneath it (if it has any child nodes)
increment	n/a	Increments the current Global Node's on-disk value by 1. If the Global Node doesn't exist, it is created on-disk with a value of 1
countChildren	n/a	Returns the number of child nodes for the current Global Node. Note: this should be used with care: a Global Node could potentially have huge numbers of child nodes, and they have to be physically navigated in order to return the count

Method Name	Arguments	Description

forEachChild	optional object:
	
	{
	direction: 'forwards' ll 'reverse',
	prefix:
	range: {
	from:
	to:
	}
	}
	
	Callback function, with the following arguments:
	
	- nodeName (the child node's name)
	- childNode DocumentNode object
	
	Note that within the callback function, *this* represents the DocumentNode object that invoked the forEachChild method
		Iterates through the current Document Node's child nodes. The direction and range of the iteration can be modified by an optional first argument.
		
		By default, all child nodes will be accessed in forward alphanumeric collating sequence.
		
		direction defaults to forward
		
		range.from defaults to first child
		
		range.to defaults to last child
		
		prefix limits the iteration to just those child nodes whose names start with the specified prefix string
		
		range.from and *range.to* specify the child node name start and end prefixes within which you want to limit your search
		
		For example:
		
		var params = {
		direction: 'reverse',
		prefix: 'rob'
		};
		node.forEachChild(params, function(name, childNode) {
		// process each child whose name starts rob in
		// reverse order
		});

getDocument	n/a	Returns the current DocumentNode's sub-tree of on-disk nodes as a JavaScript object
setDocument	object	Appends the specified JavaScript object as a sub-tree of on-disk Document Nodes to the current Document Node
forEachLeafNode	Callback function, with the following arguments: - value (the sub-node's value) - sub-node DocumentNode object Note that within the callback function, *this* represents the DocumentNode object that invoked the forEachLeafNode method	Specialised but powerful iterator method. This will iterate through just those sub-nodes that have an on-disk value. The iteration is not therefore limited to the current Document Node's immediate child nodes, but all sub-nodes at any level beneath it that have a value. This is a very fast way to access all the leaf nodes within a Document (or part of a Document)

Events

DocumentStore events are emitted when setting or changing a Document Node's value, and when a DocumentNode is deleted. These events can be used for your own custom indexing logic. Note that the Document Store has no built-in indexing - it is up to you to implement your own indexing, if required, using Documents (or sub-documents).

Event name	Arguments	Notes

beforeSet	Object:
	
	{
	documentName: {string},
	path: {array of node names}
	}
	
	This provides the parameters that can identify the Document Node whose value is about to change
		Fires immediately before a Document Node's value is set or changed by the following properties/ methods:
		
		- value
		- increment
		- setDocument

afterSet	Object:
	
	{
	documentName: {string},
	path: {array of node names}
	before: {
	value: {previous value of node, if any},
	exists: {boolean - did node already exist?}
	},
	value: {the node's new value}
	}
	
	This provides the parameters that can identify the Document Node whose value has just been changed, and its previous status and value if any
		Fires immediately after a Document Node's value is set or changed by the following properties/ methods
		
		- value
		- increment
		- setDocument

beforeDelete	Object:
	
	{
	documentName: {string},
	path: {array of node names}
	before: {
	value: {previous value of node, if any},
	exists: {boolean - did node already exist?}
	}
	}
	
	This provides the parameters that can identify the Document Node that is about to be deleted
		Fires immediately before a Document Node is deleted by the delete() method

Event name	Arguments	Notes

afterDelete	Object:
	
	{
	  documentName: {string},
	  path: {array of node names}
	  before: {
	  value: {previous value of node, if any},
	  exists: {boolean - did node already exist?}
	  }
	}
	
	This provides the parameters that can identify the Document Node that has just been deleted, along with its previous status and value if any
		Fires immediately after a Document Node is deleted by the delete() method

These events are available to the DocumentStore object, eg:

```
documentStore.on('afterSet', function(obj) {
  // set / reset your index here
});
```

or, if using ewd-document-store with ewd-qoper8 in your worker module:

```
this.documentStore.on('afterSet', function(obj) {
  // set / reset your index here
});
```

Conclusions

ewd-document-store is a very powerful Document database whose "unit of storage" is not limited to being an entire JSON object. Instead, it provides a very flexible and fine-grained Document data store, allowing direct on-disk read/write access to entire Documents, sub-documents within a Document, or even right down to an individual name/value pair anywhere within a Document.

Iteration through a Document Node's child nodes and to its next and previous siblings is exceptionally fast - optimised by the underlying Global Storage database engine which also automatically sorts and collates node names.