CORE - A PRIMER

The core system explained.

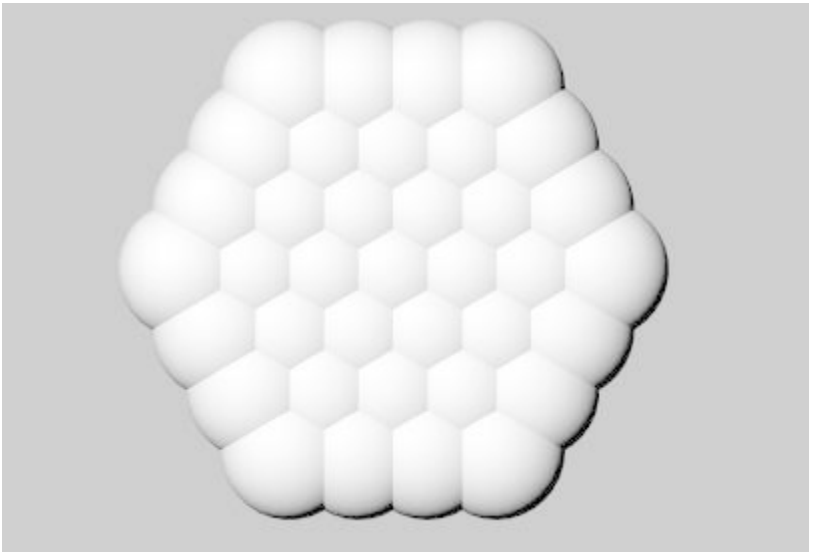


image: Circle of Life
http://i98.photobucket.com/albums/l280/kachina2012/circleoflife3d_1.jpg

By: Willem van Heemstra (willem@vanheemstrapictures.com)
Modified: 08/07/2012

Table of content

CORE

1. CLIENT SIDE

1. User Interface (UI)

Follows the ExtJS[1] javascript architecture and uses its libraries to request services (i.e. data) from the server.

...

Folder hierarchy:

core			
	app		
		controller	
		model	
		store	
		view	
			organisation
			party
			person

The core / **app_embed.js** file (example):

```
Ext.Loader.setConfig({ enabled: true });
Ext.application({
        name: 'core',
        appFolder: 'core/components/core/apps/core/app',
        autoCreateViewport: true,
        models: ['Station', 'Song'],
        stores: ['Stations', 'RecentSongs', 'SearchResults'],
        controllers: [
        //      'Parties',
        //      'Organisations',
        //      'Persons',
                'Station',
                'Song'
        ],
});
```

The core / app / view / **Viewport.js** file (example):

```
Ext.define('core.view.Viewport', {
    extend: 'Ext.container.Viewport',
        requires: [
                'core.view.NewStation',
```

[1] http://docs.sencha.com/ext-js

```
        'core.view.SongControls',
        'core.view.StationsList',
        'core.view.RecentlyPlayedScroller',
        'core.view.SongInfo'
       ],
       // By not defining properties like flex, width, height in the views,
       // we can easily adjust the application's overall layout in one single place (this
Viewport),
       // adding to the maintainability and flexibility of our architecture.
       initComponent: function() {
              var panel = new Ext.Panel({
           renderTo: 'content',
           layout: 'fit',
                     dockedItems: [{
                            dock: 'top',
                            xtype: 'toolbar',
                            height: 80,
                            items: [{
                                   xtype: 'newstation',
                                   width:150
                            }, {
                                   xtype: 'songcontrols',
                                   flex: 1
                            }, {
                                   xtype: 'component',
                                   hmtl: 'Core<br>Internet Radio'
                            }]
                     }],
                     layout: {
                            type: 'hbox',
                            align: 'stretch'
                     },
                     items: [{
                            width: 250,
                            xtype: 'panel',
                            layout: {
                                   type: 'vbox',
                                   align: 'stretch'
                            },
                            items: [{
                                   xtype: 'stationslist',
                                   flex: 1
                            }, {
                                   html: 'Ad',
                                   height: 50,
                                   xtype: 'panel'
                            }]
                     }, {
                            xtype: 'container',
                            flex: 1,
                            layout: {
                                   type: 'vbox',
                                   align: 'stretch'
                            },
                            items: [{
                                   xtype: 'recentlyplayedscroller',
                                   height: 250
                            }, {
                                   xtype: 'songinfo',
                                   height: 250
                            }]
                     }]
       });
              //pass along browser window resize events to the panel
       Ext.EventManager.onWindowResize(panel.doLayout, panel);

              this.callParent();
    }
});
```

Write-Store - Making the ExtJS Store transaction enabled

The **write-store**[2] is an extension to the excellent ExtJS library by Jack Slocum. It answers several shortcoming of ExtJS data Stores, which are used to present data:

- ExtJS stores are primarily read-only with respect to the server, rather than read-write. You can change data in the Store, but cannot transmit it back to the server.
- ExtJS stores do not have a full transaction support. If you make several changes and then decide you want to undo them, modern RDBMS systems will let you reject or roll back the transaction; ExtJS stores do not support this.
- ExtJS stores cannot use other Stores as their data source. This becomes important to reduce round-trip requests to/from the server in a time-sensitive Ajax environment. For example, if a single query returns a list of persons, and each person has several telephone numbers, it makes sense from a presentation to the user perspective to have the persons in one Store, perhaps displayed by one grid, and the telephone numbers in another Store, perhaps displayed by a separate grid. ExtJS currently requires you to retrieve the persons in one request and each set of telephone numbers, when desired, in separate requests.

write-store extends ExtJS to resolve these shortcomings.
It is **strongly** recommended that you understand how to use ExtJS before using write-store.

JSormDB - Database at the client with support of the Server

jsormdb[3] is a full-featured stand-alone database written entirely in JavaScript. jsormdb can be used to store data within a client-side or server-side JavaScript application. jsormdb supports full transaction semantics including commit and partial/full rollback, and queries.
jsormdb also supports loading information from and sending information to a remote data source. In this, jsormdb acts as a local representation of remote data, and can fully interact with the data. Commits are sent to the remote data source in one of multiple methods, and the returned data can be entered in the jsormdb in one of multiple methods.
jsormdb was inspired by the write-store extension to ExtJS. The author of write-store saw a general need for a proper data store and built jsormdb from scratch to meet that need.
jsormdb currently supports only one table per instance. A direct result of this limitation is that a join query is not possible. Multi-table is scheduled for a future release.

WARNING: An error will be raised if the following has not been fixed:
Inside the javascript file **jsormdb.js** and **jsormdb-src.js** one has to put quotes around the version number, like so: var JSORM={version:**"**1.3b**"**}

[2] http://jsorm.com/wiki/Write-store
[3] http://jsorm.com/wiki/Jsormdb

1. Persistence ---- NO LONGER REQUIRED, WE NOW USE JSormDB !!!

Persistence[4] is a JavaScript framework to persist (i.e. temporarily store) objects in a browser and/or server environment. We use it to store objects in the browser in its database or local storage.

persistence.js utilizes the browser's SQLite database to store and query data.

Browser support

- Modern webkit browsers (Google Chrome and Safari)
- Firefox (through Google Gears) NOTE: Google Gears is deprecated.
- Opera (10+)
- Android browser (tested on 1.6 and 2.x)
- iPhone browser (iPhone OS 3+)
- Palm WebOS (tested on 1.4.0)

The in-memory store can be used in other browser (such as Firefox) as a fall-back option. The in-memory store has the ability to write and load the entire database to the browser's window.localStorage.

So, we will detect the web browser and if the web browser does not support Web SQL Database (for example, Firefox and IE), we fall back to using the in-memory store.
See http://persistencejs.org/stores/memory

```
if (/Firefox[\/\s](\d+\.\d+)/.test(navigator.userAgent)){
  // FireFox detected

}
else if (/MSIE (\d+\.\d+);/.test(navigator.userAgent)){
  // IE detected

}
else {
  // a WebSQL database compatible browser

}
```

HTML 5 Client-side DatabaseStorage API demo

See http://www.webkit.org/demos/sticky-notes/index.html

To 'play' with all features of the browser Web SQL Database, go here:
http://playground.html5rocks.com/#async_transactions

////////////////////////////////////// STAY WITH PERSISTENCE.JS //

-- NO NEED TO DIVERT TO OTHER SOLUTIONS AS PERSISTENCE.JS HAS FALL BACK --

[4] http://persistencejs.org

For compatibility reasons, we should abstract from the Web SQL Database (as neither IE nor FireFox support it) and use an abstraction layer, such as **JStore**[5] (i.e. JQuery Store).

Or stick with **HTML 5 Local Storage** as it supported in almost all browsers natively:
See here for more info on the use of HTML 5 Local Storage (which uses key/value pairs):
http://diveintohtml5.info/storage.html

Or use **PersistJS**[6], which handles client-side storage seamlessly and transparently to your code. You use a single API and get support for the following backends:

- flash: Flash 8 persistent storage.
- gears: Google Gears-based persistent storage.
- localstorage: HTML5 draft storage.
- whatwg_db: HTML5 draft database storage.
- globalstorage: HTML5 draft storage (old spec).
- ie: Internet Explorer userdata behaviors.
- cookie: Cookie-based persistent storage.

Any of those can be disabled—if, for example, you don't want to use cookies. With this library, you'll get native client-side storage support in IE 5.5+, Firefox 2.0+, Safari 3.1+, and Chrome; and plugin-assisted support if the browser has Flash or Gears. If you enable cookies, it will work in everything (but will be limited to 4 kB).

Each backend exposes the exact same interface, which means you don't have to know or care which backend is being used.

//
///

Folder Hierarchy for Persistence for Client (i.e. web pages) usage:

assets					
	templates				
		core			
			javascripts		
				persistence	
					<the persistence javascript files>

Addition to all web pages that use persistence:

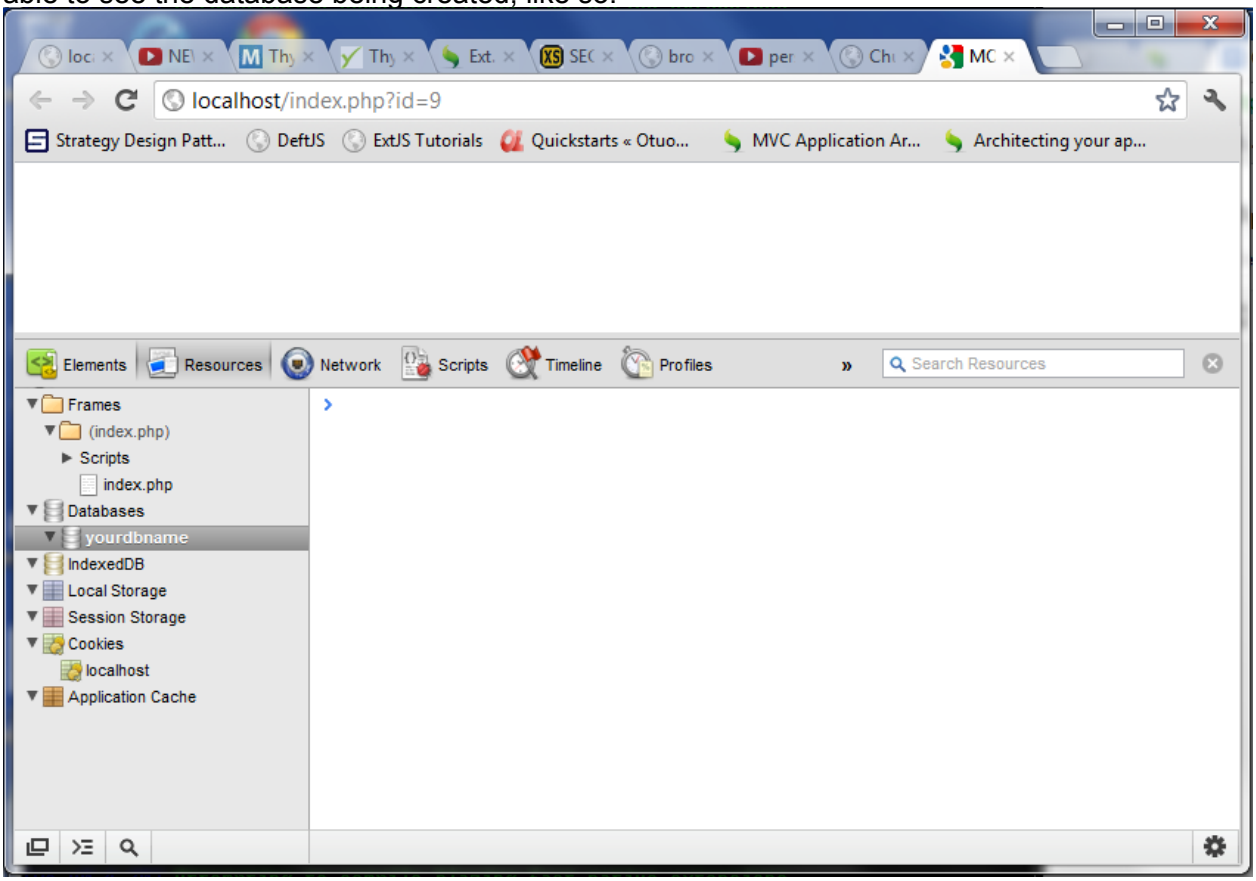
```
<html>
...

<script src="assets/templates/core/javascripts/persistence/persistence.js" type="application/
javascript"></script>
<script src="assets/templates/core/javascripts/persistence/persistence.store.sql.js"
type="application/javascript"></script>
```

[5] http://code.google.com/p/jquery-jstore/
[6] http://pablotron.org/?cid=1557

```
<script src="assets/templates/core/javascripts/persistence/persistence.store.websql.js"
type="application/javascript"></script>
<script type="application/javascript">
  persistence.store.websql.config(persistence, 'yourdbname','A database description', 5 * 1024
* 1024);
</script>

...
</html>
```

NOTE: The above example includes a WebSQL store (which includes Google Gears support). The first argument is always supposed to be persistence. The second in your database name (it will create it if it does not already exist, the third is a description for your database, the last argument is the maximum size of your database in bytes (5MB in this example).

When this web page is loaded in a web browser, using Chrome's Developer Tools you will be able to see the database being created, like so:



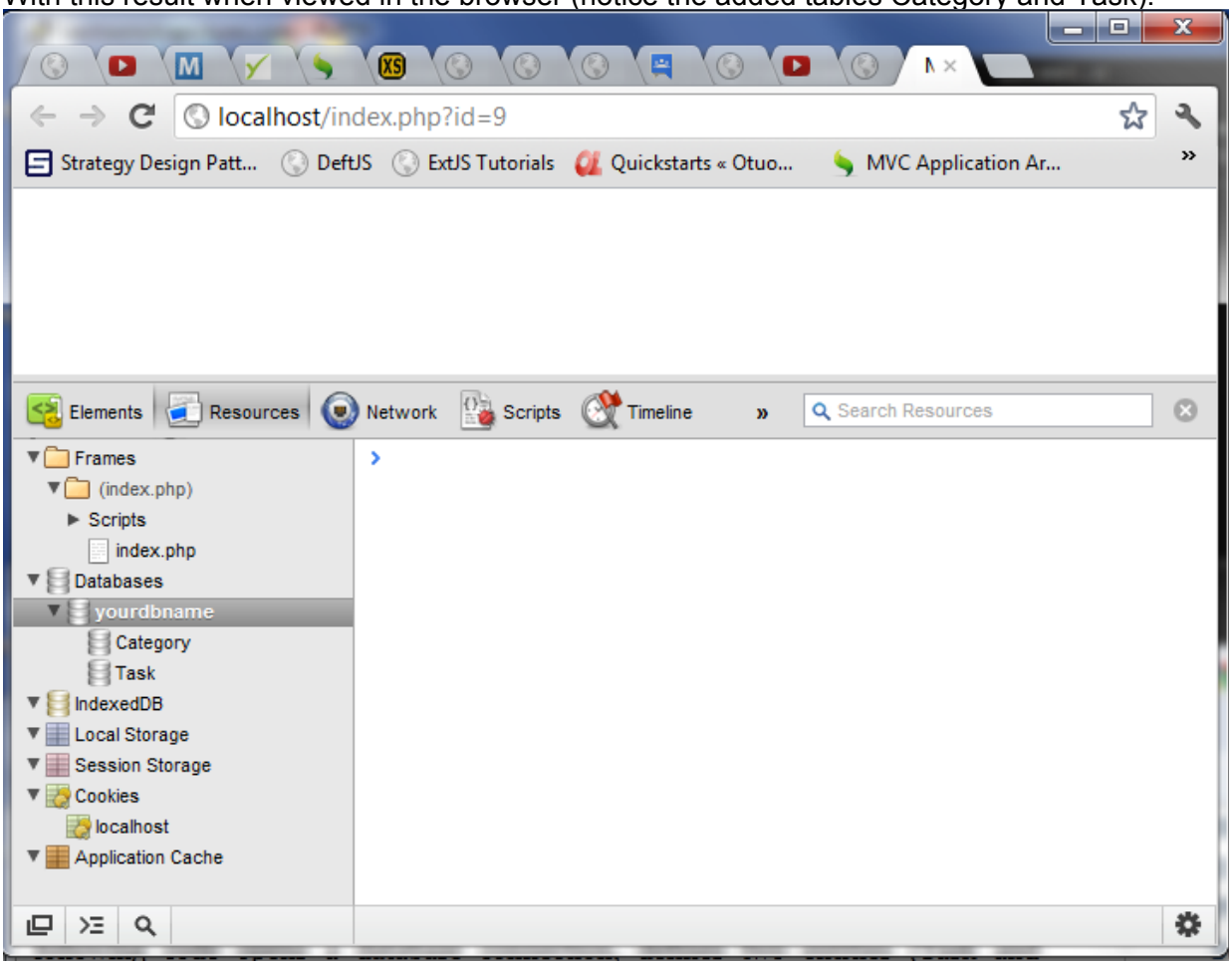
We can add tables and relationships to the database on the client like so:

```
<html>
...

<script src="assets/templates/core/javascripts/persistence/persistence.js" type="application/
javascript"></script>
<script src="assets/templates/core/javascripts/persistence/persistence.store.sql.js"
type="application/javascript"></script>
<script src="assets/templates/core/javascripts/persistence/persistence.store.websql.js"
type="application/javascript"></script>
<script type="application/javascript">
```

```
  persistence.store.websql.config(persistence, 'yourdbname','A database description', 5 * 1024
* 1024);

var Task = persistence.define('Task', {
    name: "TEXT",
    description: "TEXT",
    done: "BOOL"
});

var Category = persistence.define('Category', {
    name: "TEXT"
});

Category.hasMany('tasks', Task, 'category');

persistence.schemaSync(null, function(tx) {
  alert('Successfully synchronized the schema!');
});
</script>

...
</html>
```

NOTE: Schema property types are based on **SQLite** types. Options are: TEXT, INT, BOOL, DATE and JSON.

With this result when viewed in the browser (notice the added tables Category and Task):



Instances of the defined entities can then be created in a natural way, and subsequently marked to be persisted (note: below is javascript in the web page):

```
var task = new Task();
task.name = "My new task";
var category =
    new Category({name: "My category"});
persistence.add(task);
persistence.add(category);
```

NOTE: "Important: Changes and new objects will not be persisted until you explicitly call flush()"

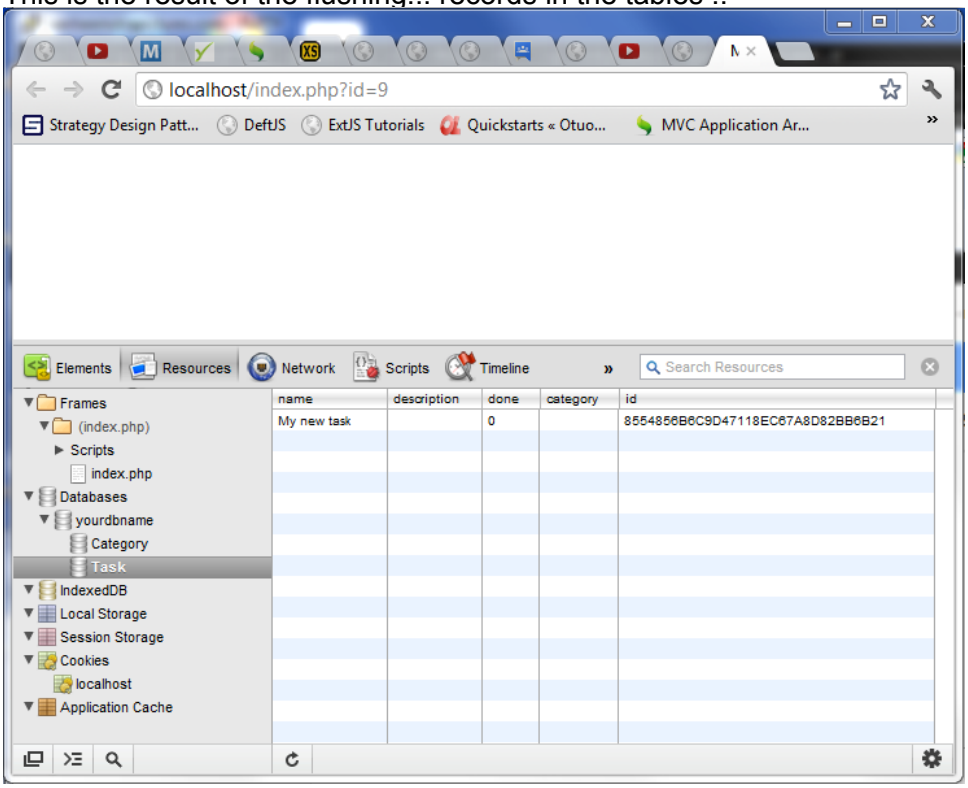
One-to-many and many-to-many relationships can be used as collections (note: below is javascript in the web page):

```
category.tasks.add(task);
```

When modifications to objects have been made, these have to be **flushed** to the database:

```
persistence.flush(null, function() {
  alert('All objects flushed!');
});
```

This is the result of the flushing... records in the tables !!



A nice feature of persistence.js is QueryCollections, which are virtual collections that can prefetch relations, can easily be filtered and sorted (and in the future paginated):

```
Task.all().filter("done", '=', true)
          .prefetch("category")
          .order("name", false)
          .list(null, function (results) {
    results.forEach(function (t) {
        console.log('[' + t.category.name + '] '
                    + t.name);
```

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

The creator (Zef Hemel) has add synchronization support to it so that the local database can transparently be synchronized with a (view on) a remote database, which is a typical use case of applications like these.

The Client side of sync-ing using persistence.js

Persistence.js is not limited to the client side, and this is one of its true strengths. On the server side, in a node.js environment, Persistence.js can make use of the node-mysql module to store and retrieve data from a MySQL database. The persistence.sync module allows for synchronization of tables between a browser database and server side MySQL.

See here: http://persistencejs.org/plugin/sync

Adding sync capability, requires the web page to contain these script tags:

```
<html>
...
<script src="assets/templates/core/javascripts/persistence/persistence.js" type="application/
javascript"/></script>
<script src="assets/templates/core/javascripts/persistence/persistence.sync.js"
type="application/javascript"/></script>
…
</html>
```

After including both persistence.js and persistence.sync.js in your page, you can enable syncing on entities individually:

```
var Task = persistence.define("Task", {
  name: "TEXT",
  done: "BOOL"
});

Task.enableSync('/taskChanges');
```

The argument passed to enableSync is the URI of the sync server component.

To initiate a sync, the EntityName.**syncAll(..)** method is used (e.g. Task.syncAll(...)):

```
function conflictHandler(conflicts, updatesToPush, callback) {
  // Decide what to do with the conflicts here, possibly add to updatesToPush
  callback();
}

EntityName.syncAll(conflictHandler, function() {
  alert('Done!');
});
```

There are two sample conflict handlers:

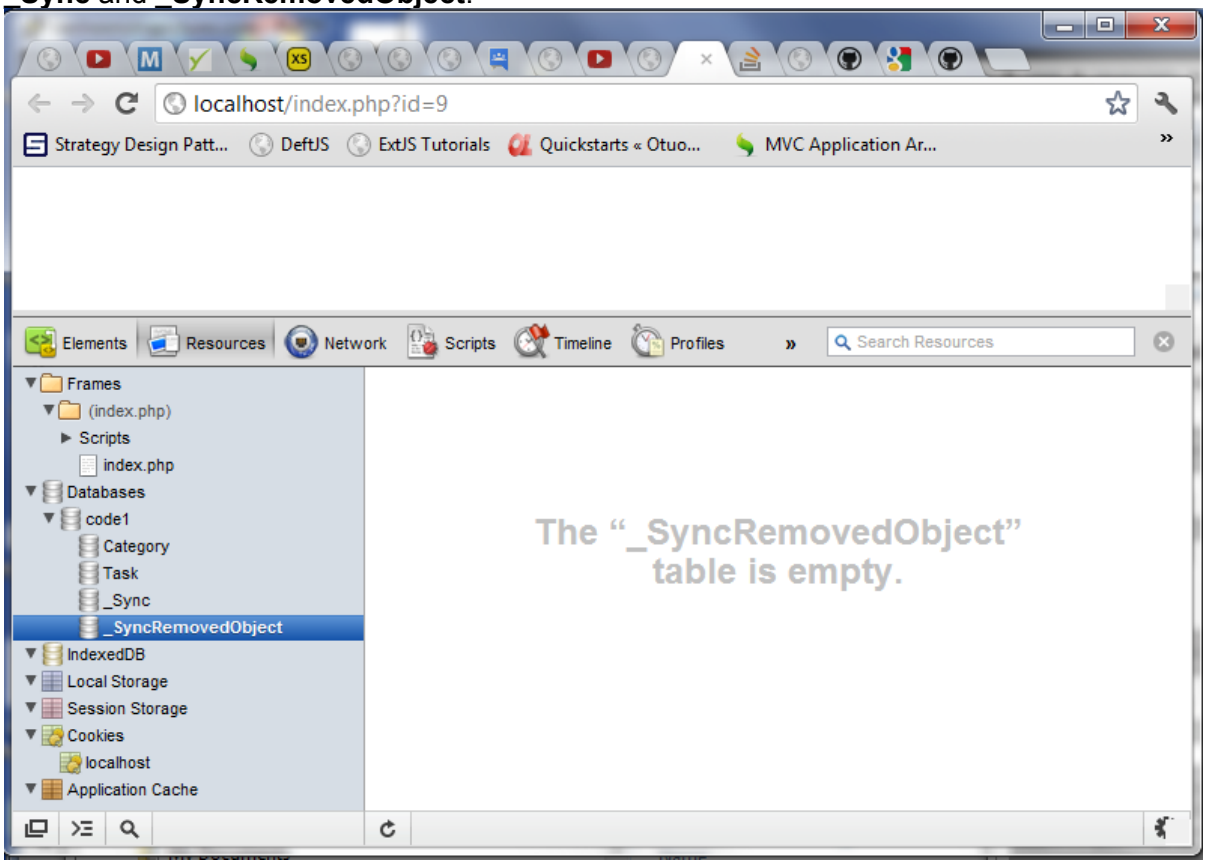
1. persistence.sync.preferLocalConflictHandler, which in case of a data conflict will always pick the local changes.
2. persistence.sync.preferRemoteConflictHandler, which in case of a data conflict will always pick the remote changes.

For instance:

```
EntityName.syncAll(persistence.sync.preferLocalConflictHandler, function() {
  alert('Done!');
});
```

Note that you are responsible for syncing all entities and that there are no database consistencies after a sync, e.g. if you only sync Tasks that refer to a Project object and that Project object has not (yet) been synced, the database will be (temporarily) inconsistent.

The addition of the **persistence.sync.js** code to the **persistence.store.websql.js** code in the web page will automatically create two more tables inside the database in the browser, called **_Sync** and **_SyncRemovedObject**:



NOTE: We have renamed the database to 'code1' in above example.

This is the WebSQL that is executed by `persistence.store.websql.js:78`:

```
CREATE TABLE IF NOT EXISTS `Category` (`name` TEXT, `id` VARCHAR(32) PRIMARY KEY) null

CREATE TABLE IF NOT EXISTS `Task` (`name` TEXT, `description` TEXT, `done` INT, `category`
VARCHAR(32), `id` VARCHAR(32) PRIMARY KEY) null

CREATE INDEX IF NOT EXISTS `Task__category` ON `Task` (`category`) null

CREATE TABLE IF NOT EXISTS `_SyncRemovedObject` (`entity` VARCHAR(255), `objectId` VARCHAR(32),
`id` VARCHAR(32) PRIMARY KEY) null

CREATE TABLE IF NOT EXISTS `_Sync` (`entity` VARCHAR(255), `localDate` BIGINT, `serverDate`
BIGINT, `serverPushDate` BIGINT, `id` VARCHAR(32) PRIMARY KEY) null
```

```
INSERT INTO `Task` (`name`, `description`, `done`, id) VALUES (?, ?, ?, ?) ["My new task", "",
0, "6A38B884845E43B8B15D447D1F468023"]

INSERT INTO `Category` (`name`, id) VALUES (?, ?) ["My
category", "C88CFB677D404FC185301770160EFDD4"]
```

NOTE: It creates an '**id**' field by default and creates unique values for each record for this field (e.g. `6A38B884845E43B8B15D447D1F468023`).

Here is another code example (with a database called code2), which shows the **creation** (Task 1 through Task 10), **retrieval** (All Tasks), **updating** (by randomly setting 'done' from '0' to '1') and **deletion** of records (whose done field was set to '1' where 1 means true):

```
<script type="application/javascript">
//establish local database
persistence.store.websql.config(persistence, 'code2', 'A database description', 5 * 1024 *
1024);

//define Entity
var Task = persistence.define('Task', {
  name: "TEXT",
  done: "BOOL"
});

//wipe local database clean
persistence.reset(function(){

 //write schema
 persistence.schemaSync(function(){

  //create ten dummy tasks
  for(var i=1;i<=10;i++){
   var task = new Task();
   task.name = "Task " + i;
   task.done = false;
   persistence.add(task);
  }

  //commit dummy tasks to database
  persistence.flush(function(){

   //retrieve all tasks from database
   Task.all().list(function(tasks){
    //callback counter
    var taskCounter = tasks.length;

    //asynchronously loop through items
    tasks.forEach(function(task){
      //randomly set items to done
      task.done = Math.round(Math.random());

      //decrement callback counter
      //check if this is last run
      if(--taskCounter == 0){

       //write all changes back to database
       persistence.flush(function(){

       //call destroyAll on a collection with the filter done=true
       Task.all().filter('done','=',true).destroyAll();
      });
     }
    });
   });
  });
```

```
  });
});
</script>
```

Note: example taken from http://jacobmumm.com/demos/persistencetask/index.html

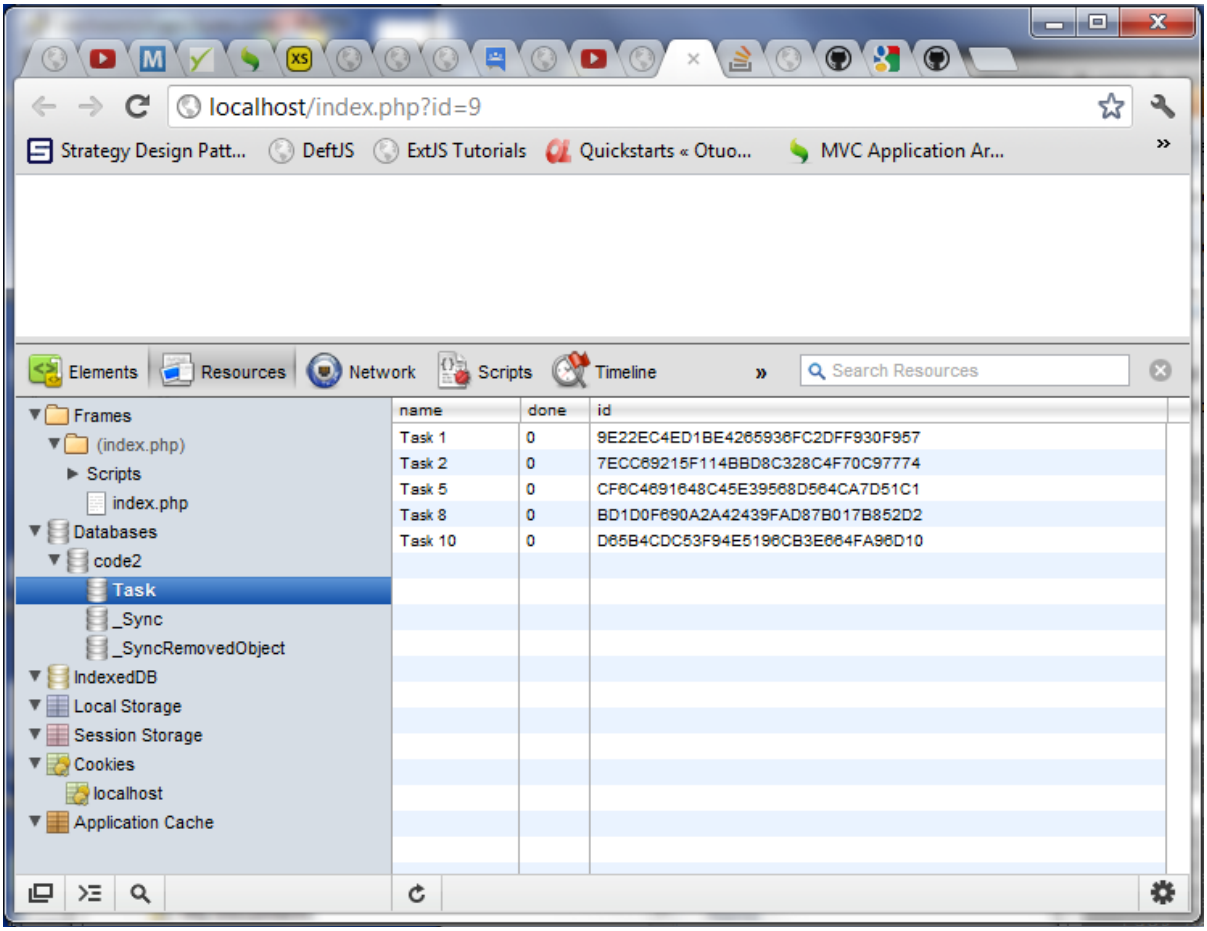
This is the executed WebSQL in the browser by persistence.store.websql.js:78:

```
    DROP TABLE IF EXISTS `Task` null
    DROP TABLE IF EXISTS `_SyncRemovedObject` null
    DROP TABLE IF EXISTS `_Sync` null
    CREATE TABLE IF NOT EXISTS `Task` (`name` TEXT, `done` INT, `id` VARCHAR(32) PRIMARY
    KEY) null
    CREATE TABLE IF NOT EXISTS `_SyncRemovedObject` (`entity` VARCHAR(255), `objectId`
    VARCHAR(32), `id` VARCHAR(32) PRIMARY KEY) null
    CREATE TABLE IF NOT EXISTS `_Sync` (`entity` VARCHAR(255), `localDate` BIGINT,
    `serverDate` BIGINT, `serverPushDate` BIGINT, `id` VARCHAR(32) PRIMARY KEY) null
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 1",
    0, "9E22EC4ED1BE4265936FC2DFF930F957"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 2",
    0, "7ECC69215F114BBD8C328C4F70C97774"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 3",
    0, "A1C34523D1CF4CB5AB4429B60210A463"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 4",
    0, "B981EF0315BF480290E6A1A52F0C8E01"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 5",
    0, "CF6C4691648C45E39568D564CA7D51C1"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 6",
    0, "D34FDB5497C74412ABCD03D87EC42AB4"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 7",
    0, "B78A3C7F491A45569471F8AB056AFBCC"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 8",
    0, "BD1D0F690A2A42439FAD87B017B852D2"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 9",
    0, "639574F31E86496EA022C95A8FA69DCD"]
    INSERT INTO `Task` (`name`, `done`, id) VALUES (?, ?, ?) ["Task 10",
    0, "D65B4CDC53F94E5196CB3E664FA96D10"]
    SELECT `root`.id AS Task_id, `root`.`name` AS `Task_name`, `root`.`done` AS `Task_done`
    FROM `Task` AS `root`  WHERE 1=1 []
    UPDATE `Task` SET `done` = ? WHERE id = '9E22EC4ED1BE4265936FC2DFF930F957' [0]
    UPDATE `Task` SET `done` = ? WHERE id = '7ECC69215F114BBD8C328C4F70C97774' [0]
    UPDATE `Task` SET `done` = ? WHERE id = 'A1C34523D1CF4CB5AB4429B60210A463' [1]
    UPDATE `Task` SET `done` = ? WHERE id = 'B981EF0315BF480290E6A1A52F0C8E01' [1] UPDATE
    `Task` SET `done` = ? WHERE id = 'CF6C4691648C45E39568D564CA7D51C1' [0]
    UPDATE `Task` SET `done` = ? WHERE id = 'D34FDB5497C74412ABCD03D87EC42AB4' [1]
    UPDATE `Task` SET `done` = ? WHERE id = 'B78A3C7F491A45569471F8AB056AFBCC' [1]
    UPDATE `Task` SET `done` = ? WHERE id = 'BD1D0F690A2A42439FAD87B017B852D2' [0]
    UPDATE `Task` SET `done` = ? WHERE id = '639574F31E86496EA022C95A8FA69DCD' [1]
    UPDATE `Task` SET `done` = ? WHERE id = 'D65B4CDC53F94E5196CB3E664FA96D10' [0]
    SELECT id FROM `Task`  WHERE (1=1 AND `done` = ?) [1]
    DELETE FROM `Task`  WHERE (1=1 AND `done` = ?) [1]
```

See above WebSQL making use of **prepared statements**, where the ? is set through the value in the index [0].

Note how by setting '**WHERE 1=1**' all records are retrieved.

And this is the outcome as shown in the browser's database table 'Task':



NOTE: We have renamed the database to 'code2' in above example.

Date handling can be deceptive. Make sure that any fields defined as date objects are being sent as an epoch rather than any other format. This can silently fail, or succeed in certain Web SQL implementations and not others.

- This synchronization library synchronizes on a per-object granularity. It does not keep exact changes on a per-property basis, therefore conflicts may be introduced that need to be resolved.
- It does not synchronize many-to-many relationships at this point
- Error handling is not really implemented, e.g. there's no way to deal with a return from the server other than "status: ok" at this point.

Querying with persistence

Here is an example of an efficient way to query for a Person object and its related Person object (i.e. the father). It will print the person's name and the name of his/her father from 1 query !!!

```
Person.all().prefetch("father").each(tx, function(p) {
  println(p.name);
  println(p.father.name);
});
```

Adding full text search capability, requires the web page to contain these script tags:

```
<html>
...
<script src="assets/templates/core/javascripts/persistence/persistence.js" type="application/
javascript"/></script>
<script src="assets/templates/core/javascripts/persistence/persistence.search.js"
type="application/javascript"/></script>
…
</html>
```

Declare the data model (example):

```
var Task = persistence.define('Task', {
    name: "TEXT",
    description: "TEXT",
    done: "BOOL"
});

var Category = persistence.define('Category', {
    name: "TEXT"
});
```

Define which columns should be indexed:

```
Task.textIndex('name');
Task.textIndex('description');
Category.textIndex('name');
```

Indexing will be enabled and any new entity instances and changes to them will automatically be (re)indexed. You can then search as follows:

```
Task.search('important').list(null, function(results) {
   console.log("All tasks including the word 'important'");
   console.log(results);
});
```

Note that Task.search(...) returns a query collection, so the usual things such as filter, limit and skip (but not order, because results are always sorted by number of occurrences) can be used, e.g.:

```
Task.search('important')
 .limit(10)
 .skip(currentPage * 10)
 .list(null, function(results) {
     ...
  });
```

ORDER in PERSISTENCE: schemaSync() first, then add entities, then persistence.flush();

2. SERVER SIDE

2. Content Management System (CMS)

Uses the MODx[7] content management system.

2. Request Handling (NodeJS)

Introduction

Follows the NodeJS[8] javascript architecture and uses its libraries for handling requests from the client.

Installing on Windows

Download the binary from nodejs.org and follow the installation instructions.

- Windows (64 bit) will install Node.exe in C:\Program Files (x86)

and in C:\Users\{user_name}\AppData\Roaming\ it will place npm (within which the node_modules are stored) and npm-cache.

- Make sure to run all commands from the Console as root/Administrator by starting the Console with CTRL+SHIFT

Installing on Mac OS

Download the binary from nodejs.org and follow the installation instructions.

- Mac OS X will install in /usr/local/bin/node and /usr/local/bin/npm
- Make sure that /usr/local/bin is in your $PATH
- Make sure to run all command from the Terminal as root/Administrator by starting them with (you will be asked to provide the root password the first time):

```
sudo
```

Installing on Linux (e.g. BlueHost)

NOTE: to be able to access the port that is set by the NodeJS server(s), when using a hosted solution like Bluehost you will have to have extended your subscription to a **Dedicated IP Address**. See the Control Panel on the Bluehost site to order a Dedicated IP Address (ca $2.50 per month extra).

http://www.vanheemstrapictures.com Dedicated IP address is:

```
http://69.195.104.126
```

Inside the $HOME/downloads/git directory type:

```
git clone http://github.com/joyent/node.git
```

A new directory will be created by git called ‘node’.

Inside this node directory type (so it installs it in the node directory of our home directory):

[7] http://www.modxcms.com
[8] http://nodejs.org

```
./configure --prefix=$HOME/node
```

Next, type:

```
make
```

Followed by:

```
make install
```

Now we need to make the node command usable so do the following commands:

Go back to the HOME directory like so:

```
cd $HOME
```

Followed by:

```
nano .bashrc
```

Now add the following line to the bottom of the file.

```
export PATH=$HOME/node/bin:$PATH
```

Save the edited file, by using this combination of keys:

ctrl+x

and confirm by Y, then ENTER and ENTER again.

You will need to reload bash, like so:

```
source .bashrc
```

Enter into the directory where nodejs has been installed and then into the bin directory to type:

```
node --version
```

This should echo the version of the newly installed nodejs (here v0.8.2).

It is recommended to also **update** NodeJS with all its dependencies, by going into the directory where nodejs has been installed and type:

```
npm update
```

Nodejs Files Location within our Core

Folder hierarchy within our MODx installation under public_html/core/components/core/apps/

core				
	app			
		controller	< all entity controller js	

			files >	
		model	< all entity model js files >	
		store	< all entity store js files >	
		view		
			< all entity sub-folders >	
				< all entity view js files >
			< all viewport js files >	
	data	< all static JSON files >		
	lib			
		Ext	< all Ext src files for server use, see N-Ext >	
	persistence			
		< all the persistence javascript files >		
< all app_embed.js, app.js, server.js, database.js, etc files >				

The core / **server3.js** file (example):

```
// see http://www.youtube.com/watch?v=qws6LOvDQRE
var sys = require('sys'),
    http = require('http');

/* call this file as a URL
/with adding the options below.
/e.g. localhost:3000/add/2/2
/will return 4
/add/2/2 => 4
/sub/103/42 => 61
/mul/3/2 => 6
/div/100/25 =>4
*/
var operations = {
    add: function(a,b){return a + b},
    sub: function(a,b){return a - b},
    mul: function(a,b){return a * b},
    div: function(a,b){return a / b}
}

http.createServer(function(req, res) {
```

```
    var parts = req.url.split("/"),
        op = operations[parts[1]],
        a = parseInt(parts[2], 10),
        b = parseInt(parts[3], 10);
    //sys.puts(sys.inspect(parts));
    var result = op ? op(a,b) : "Error";
    res.writeHead(200, {
        'Content-type': 'text/plain'
    });
    res.end("" + result);
}).listen(3000, "127.0.0.1");

sys.puts('Server running at http://127.0.0.1:3000/');
```

To start this server, type the command from within its directory:

```
node server3.js
```

Opening a browser and following the following URL should allow you to interact with this server:

```
http://localhost:3000
```

To stop (i.e. kill) the server, push this combination of keys in the active console window from where you started the server:

```
CTRL+C
```

Uses Node-MySQL[9] for connecting to the MySQL database. Use Node Package Manager (NPM)[10] to install, which comes with NodeJS.

NOTE: On Windows you want to be running the console as Administrator. To do so, type this into the Run box from the Start menu:

```
cmd -d
```

Now instead of hitting the Enter key, use **Ctrl+Shift + Enter**. You will be prompted with the obnoxious User Account Control dialog… but it will then open up a command prompt in Administrator mode.

NOTE: On Windows npm by default places the node_modules folder in C:\Users\<myname>\AppData\Roaming\npm\node_modules

On Windows this means adding the mysql folder to the node_modules folder.
If the module(s) cannot be found, add this to the Environment Variables:

`NODE_PATH=/path/to/node_modules`
e.g.
NODE_PATH=C:\Users\Willem van Heemstra\AppData\Roaming\npm\node_modules

A quick check to see which modules have been installed for nodejs is by executing this command within the nodejs program folder:

```
npm ls
```

[9] https://github.com/felixge/node-mysql
[10] http://npmjs.org

For node-mysql this should list:

```
…mysql (followed by its version, e.g. @2.0.0-alpha3)
```

To install modules (e.g. forever) using the g for global reference, type this:

```
npm install forever -g
```

To update npm use this command from within the specific module dir inside the node_modules:

```
npm -g update npm
```

To find outdated modules use this command from within the specific module dir inside the node_modules:

```
npm outdated
```

See http://npmjs.org/doc/install.html for its use.

To install the latest version of mysql type:

```
npm install mysql@2.0.0-alpha3 -g
```

The core / **database1.js** file (example)

```
var mysql      = require('mysql');
var connection = mysql.createConnection({
  host  : 'localhost',
  user  : 'root',
  password : '',
  database : 'core'
});

connection.connect();

connection.query('SELECT * from PERSON', function(err, rows, fields) {
  if (err) throw err;

  console.log('Query result: ', rows);
});

connection.end();

// Output is returned in JSON by default.
```

This requires that you have installed the mysql for node.

```
npm install mysql@  -g
```

To do a general test after having installed mysql for node, from within the 'test' directory under node_modules / mysql type this command (note: mysql should be running):

```
npm run.js
```

Connection options

When establishing a connection, you can set the following options:

host: The hostname of the database you are connecting to. (Default: localhost)
port: The port number to connect to. (Default: 3306)
socketPath: The path to a unix domain socket to connect to. When used host and port are ignored.
user: The MySQL user to authenticate as.
password: The password of that MySQL user.
database: Name of the database to use for this connection (Optional).
charset: The charset for the connection. (Default: 'UTF8_GENERAL_CI')
insecureAuth: Allow connecting to MySQL instances that ask for the old (insecure) authentication method. (Default: false)
typeCast: Determines if column values should be converted to native JavaScript types. (Default: true)
debug: Prints protocol details to stdout. (Default: false)
multipleStatements: Allow multiple mysql statements per query. Be careful with this, it exposes you to SQL injection attacks. (Default: `false)

***JSormDB** - Object Relational Mapper for JavaScript & Database*

Install JSormDB:

Get a clone from the repository:

```
git://github.com/deitch/jsormdb.git
```

Or by typing inside the directory where you want the clone (e.g $HOME/downloads/git/):

```
git clone https://github.com/deitch/jsormdb.git
```

Inside the downloaded project directory (here: jsormdb) type:

```
npm install -g
```

Install JSorm Utilities:

Get a clone from the repository:

```
git://github.com/deitch/jsorm-utilities.git
```

Or by typing inside the directory where you want the clone (e.g $HOME/downloads/git/):

```
git clone https://github.com/deitch/jsorm-utilities.git
```

Inside the downloaded project directory (here: jsorm-utilities) type:

```
npm install -g
```

Sample HTML page code:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 TRANSITIONAL//EN">
<html>
	<!-- Copyright (c) Atomic Inc. 2009 http://jsorm.com -->
	<head>
		<title>jsormdb Sample</title>
		<style>
			#content {
				position: absolute;
				left: 5%;
				top: 10%;
				width: 75%;
				height: 300px;
			}
			.grid-cell {
				float: left;
				width: 45%;
```

```
                        height: 100%;
                        border: 1px solid;
                        margin: 5px;
                        padding: 5px;
                }
                .output {
                        padding: 5px;
                }
                .output-data {
                        display:inline;
                }

        </style>
        <script type="text/javascript" src="assets/templates/core/javascripts/jquery/
jquery-1.3.2.min.js"></script>
        <script type="text/javascript" src="assets/templates/core/javascripts/json2/
json2.js"></script>
        <script type="text/javascript" src="assets/templates/core/javascripts/jsorm-
utilities/jsorm.js"></script>
        <script type="text/javascript" src="assets/templates/core/javascripts/jsormdb/
jsormdb-src.js"></script>
        <script type="text/javascript" src="assets/templates/core/javascripts/jsormdb/
jsormdb-sample.js"></script>
        <script type="text/javascript">
                $(document).ready(function(){
                        var buttons = ['load','change','commit','reject'];
                        var i,len;
                        var enable = function(db,enabled) {
                                var i,len;
                                for (i=0,len=buttons.length;i<len;i++) {
                                        if (enabled[buttons[i]]) {
                                                $('#'+buttons[i]
+db).removeAttr('disabled');
                                                $('#'+buttons[i]+db).show();
                                        } else {
                                                $('#'+buttons[i]
+db).attr('disabled',true);
                                                $('#'+buttons[i]+db).hide();
                                        }
                                }
                        };
                        var display = function(id) {
                                var db = SAMPLE["getDb"+id]();
                                $("#db"+id+ " .output-
data").html(JSON.stringify(db.find()));
                        };

                        for (i=1;i<=2;i++) {
                                (function(db){
                                        // handlers for the buttons
                                        $("#load"+db).click(function(event)
{SAMPLE["loadDb"+db]();event.preventDefault();});
```

```
                                        $("#change"+db).click(function(event){
                                                SAMPLE["changeDb"+db]();
                                                enable(db,{'commit':true,'reject':true});
                                                display(db);
                                                event.preventDefault();
                                        });
                                        $("#commit"+db).click(function(event){
                                                SAMPLE["commitDb"+db]();
                                                enable(db,{'load':true});
                                                display(db);
                                                event.preventDefault();
                                        });
                                        $("#reject"+db).click(function(event){
                                                SAMPLE["rejectDb"+db]();
                                                enable(db,{'change':true});
                                                display(db);
                                                event.preventDefault();
                                        });

                                        // handlers for load functions
                                        SAMPLE["getDb"+db]().on('load',function(){
                                                enable(db,{'change':true});
                                                display(db);
                                        });

                                        // start with only load
                                        enable(db,{load: true});
                                })(i);
                        }

                });
        </script>
</head>
<body>
        <div id="content">
                <div id="instructions">
                        This is the sample page for jsormdb. Be sure to examine the
source code, which is unminified. All JavaScript is in
                        <a href="jsormdb-sample.js">jsormdb-sample.js</a>, except for
UI controls, which are in this file. <p/>

                        Note that this sample page includes jquery. This is needed for
manipulating this sample page <u>only</u>
                        and has nothing to do with jsormdb itself. jsormdb has <b>no</
b> dependencies (other than JavaScript, of
                        course).<p/>

                        In this sample page, two databases are created. The first is
loaded from raw data.
                        The second is loaded from a remote server,
                        using http, and processed via a JsonParser.
                        To each database, data is then added, removed and changed, and
```

```
a
                        commit or reject can be performed.
                    </div>
                    <div id="db1" class="grid-cell">
                        <h2>Raw database</h2>
                        <div class="controls">
                            <input type="button" id="load1" value="Load"></
input><br/>
                            <input type="button" id="change1" value="Change"></
input><br/>
                            <input type="button" id="commit1" value="Commit"></
input>
                            <input type="button" id="reject1" value="Reject"></
input>
                        </div>
                        <div class="output">
                            <h4>Database Contents</h4>
                            <div class="output-data"></div>
                        </div>
                    </div>
                    <div id="db2" class="grid-cell">
                        <h2>Remote database</h2>
                        <div class="controls">
                            <input type="button" id="load2" value="Load"></
input><br/>
                            <input type="button" id="change2" value="Change"></
input><br/>
                            <input type="button" id="commit2" value="Commit"></
input>
                            <input type="button" id="reject2" value="Reject"></
input>
                        </div>
                        <div class="output">
                            <h4>Database Contents</h4>
                            <div class="output-data"></div>
                        </div>
                    </div>
                </div>
                <div>
        </body>
</html>
```

Persistence A storage solution for Node.JS---- NO LONGER REQUIRED, WE NOW USE JSormDB !!!

Install **Persistence**[11] by running this command (**TIP**: install from GIT as this prevents a deprecated node-waf dependency):

```
npm install https://github.com/zefhemel/persistencejs.git -g
```

On Linux (e.g. Bluehost):

Inside the $HOME/downloads/git directory type:

```
git clone https://github.com/zefhemel/persistencejs.git
```

Inside the $HOME/downloads/git/persistencejs directory type:

```
npm install -g
```

Now you will find that npm has successfully installed persistencejs in the following directory: $HOME/node/lib/node_modules/persistencejs as it shows you this response.

```
persistencejs@0.2.5 /home2/vanheems/node/lib/node_modules/persistencejs
```

Here 0.2.5 is the version of persistence that it has installed. And /home2/vanheems is the equivalent of $HOME.

To use the **latest version** of any modules, from now on you just go inside the specific git sub-directory (e.g. $HOME/downloads/git/persistencejs/) and type:

```
git pull
```

Followed by this:

```
npm install -g
```

That's all there is to it!!

NOTE: Persistence requires that MySQL (or if used SQLite) for Node.JS has is installed.

SQLite[12] should thus be asynchronous to work with Persistence. Hence - if you will be needing SQLite - install SQLite for Node.JS like so:

```
npm install node-sqlite -g
```

NOTE: This is different from the default sqlite, which is synchronous.

The Server side of sync-ing with persistence using Node.JS

Persistence will try to create tables in the database that it synchronizes to on the server (i.e. MySQL):

_Sync table:

```
CREATE TABLE IF NOT EXISTS `_Sync` (`entity` VARCHAR(255), `localDate` BIGINT, `serverDate`
```

[11] http://persistencejs.org

[12] https://github.com/orlandov/node-sqlite

```
BIGINT, `serverPushDate` BIGINT ENGINE=InnoDB DEFAULT CHARSET=utf8
```

This above SQL is related to the below code from persistence.sync.js:

```
…
   persistence.sync.Sync = persistence.define('_Sync', {
        entity: "VARCHAR(255)",
        localDate: "BIGINT",
        serverDate: "BIGINT",
        serverPushDate: "BIGINT"
      });
...
```

_SyncRemovedObject table:

```
CREATE TABLE IF NOT EXISTS `_SyncRemovedObject` (`entity` VARCHAR(255), `objectId` VARCHAR(32),
`date` BIGINT, `id` VARCHAR(32) PRIMARY KEY) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

This above SQL is related to the below code from persistence.sync.js:

```
...
   persistence.sync.RemovedObject = persistence.define('_SyncRemovedObject', {
        entity: "VARCHAR(255)",
        objectId: "VARCHAR(32)"
      });
...
```

The way persistence sync's objects (persistence.sync.js) at the web browser end:

Step 1: Look at local versions of remotely updated entities.
Step 2: Remove all remotely removed objects.
Step 3: Store new remote items locally.
Step 4: Find local new/updated/removed items (not part of the remote change set).

Task table (per example, not part of the default tables):

```
CREATE TABLE IF NOT EXISTS 'Task' ('name' TEXT, 'done' INT, 'id' VARCHAR(32) PRIMARY KEY)
ENGINE=InnoDB DEFAULT CHARSET=utf8
```

The server must expose a resource located at the given URI that responds to:

GET	requests with a since=<UNIX MS TIMESTAMP> GET parameter that will return a JSON object with two properties: ● now, the timestamp of the current time at the server (in ms since 1/1/1970) ● updates, an array of objects updated since the timestamp since. Each object has at least an id and_lastChange field (in the same timestamp format). For instance: /taskChanges?since=1279888110373 Which returns:

	{"now":1279888110421, "updates": [{"id": "F89F99F7B887423FB4B9C961C3883C0A", "name": "Main project", "_lastChange": 1279888110370 }] }
POST	requests with as its body a JSON array of new/updated objects. Every object needs to have at least an id property. Example, posting to: /taskChanges with body: [{"id":"BDDF85807155497490C12D6DA3A833F1", "name":"Locally created project"}] The server is supposed to persist these changes (if valid). Internally the items must be assigned a _lastChange timestamp TS. If OK, the server will return a JSON object with "ok" as status and TS as now. *Note:* it is important that the timestamp of all items and the one returned are the same. {"status": "ok", "now": 1279888110797}

Sequelize *A multi-dialect Object-Relational-Mapper for Node.JS*

The Sequelize[13] library provides easy access to a MySQL database by mapping database entries to objects and vice versa. To put it in a nutshell... it's an ORM (Object-Relational-Mapper). The library is written entirely in JavaScript and can be used in the Node.JS environment.

Keeping NodeJS servers running with Forever

The purpose of **Forever**[14] is to keep a child process (such as your node.js web server) running continuously and automatically restart it when it exits unexpectedly.

Install Forever by running this command:

```
npm install forever -g
```

See for an explanation of setting up Forever with NodeJS:
http://www.exratione.com/2011/07/running-a-nodejs-server-as-a-service-using-forever/

On Windows, make sure that there is a directory C:\root and that this directory is writeable. Forever uses this directory to write its log files to.

To **start** a server through forever (so it stays alive at all times), type this command in the directory of the server file (e.g. server4.js):

```
forever start server4.js
```

[13] http://sequelizejs.com/
[14] http://blog.nodejitsu.com/keep-a-nodejs-server-up-with-forever

You will notice that now when you open a browser window, the server can be contacted on its port. It has been started implicitly through forever.

To **list** all processes (i.e. node files) that forever is keeping alive, type this command:

```
forever list
```

This should return a list of each node files that is running prepended with an index number (e.g. 0 server4.js [24611, 24596]

To **stop** a server through forever, type this command (using the process index number, here 0):

```
forever stop 0
```

For **help** with forever, simply type this command:

```
forever --help
```

See for all commands with forever, http://blog.nodejitsu.com/keep-a-nodejs-server-up-with-forever

To see the **logs** of forever, on Windows look inside C:\root\.forever directory.

Also in the C:\root\.forever\pids directory you find the process identifier (pid) of each node file that is running.

To **kill** a process/processes on Windows (using the pid, for example 6568 and 5759) type:

```
Taskkill /PID 6568 5759 /F
```

Forever and Apache

The way it's usually setup is setting up a proxy in apache to redirect traffic to whatever port node is running on. There are tons of examples online. It's useful when you are running other apps (rails for example) on the same box.

Connect: the Middleware that works with Node

Connect adds one new unique aspect to node's HTTP server and that's the idea of layers. An app is structured like an onion. Every request enters the onion at the outside and traverses layer by layer till it hits something that handles it and generates a response. In Connect terms, these are called filters and providers. Once a layer provides a response, the path happens in reverse.

The Connect framework simply takes the initial request and response objects that come from node's http callback and pass them layer by layer to the configured middleware modules in an application.

Install Connect by running this command from the Node program directory:

```
npm install connect -g
```

The core / **connect-http.js** file

```
var Connect = require('connect');

Connect.createServer(function (req, res, next) {
  // Every request gets the same "Hello Connect" response.
  res.simpleBody(200, "Hello Connect");
}).listen(8080);
```

Express: built on top of Connect

Express[15] contains Connect and has added features to it.

A tip for Express: when you create a project, you may be shown some errors about missing folders, re-execute the command again to create the project successfully.

Install a clone from the Github for express like so:

```
git clone https://github.com/visionmedia/express.git
```

Enter into the newly created directory 'express' and type following command:

```
npm install -g
```

TIP: To switch between releases of (here as an example, express) type:

```
$ npm uninstall express
$ npm install express@3.0.0beta4
```

This will have installed express for Node.js. Check it with the following command:

```
express --version
```

This should return the version of the newly installed module.

To have Express as a service for our core app, we have to step inside the core app's directory:

```
cd $HOME/public_html/core/components/core/apps/core
```

or if we are on Windows and use wamp server do this:

```
cd C:\wamp\www\core\components\core\apps\core
```

Now that we are in the core app's directory use the call for express, here with support of sessions and using *less* for styling:

```
express --sessions --css less
```

It will ask for confirmation if (and as) the directory is not empty. You can accept with Y for yes as the directories and files it will create are not effecting our directories and files already in there.

apps			
	core		
		app	
		data	
		lib	
		persistence	

[15] http://expressjs.com

		app1_embed.js	
		public	
		routes	
		views	
		app.js	
		package.json	

NOTE: the directories and files newly created by express are in **bold**

After express has created its directories and files in this directory, it will request that you install dependencies, as follows:

```
cd . && npm install
```

That will install a bunch of modules used by the app, which are stored in the newly created directory *node_modules*:

node_modules	
	.bin
	express
	jade
	less-middleware

With that the base of the app is ready. It is already a working app. Let's see what it outputs.

Start the app:

```
node app
```

It will prompt you with something like:

```
Express server listening on port 3000 in development mode
```

Stop the app with the combined keys: CTRL +C

To assist in not having to restart this server every time we edit the file(s), this is a good moment to install **nodemon**, which will reload our application each time it changes so you don't need to restart it:

```
npm install nodemon -g
```

From now on we will be using nodemon instead of node as a command.

Start the app again using nodemon:

```
nodemon app
```

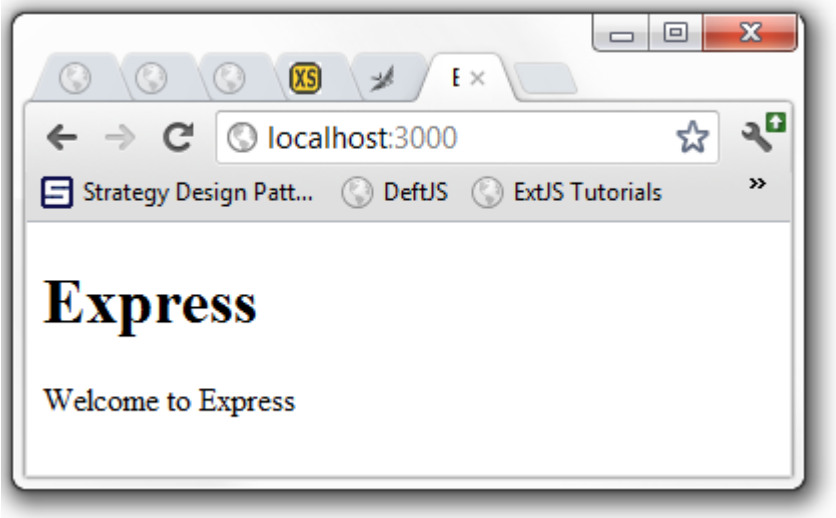
NOTE: you may be better of typing the file's full name, like so: `nodemon app.js`

It will prompt you with something like:

```
16 Jul 21:20:25 - [nodemon] v0.6.23
16 Jul 21:20:25 - [nodemon] watching: C:\wamp\www\core\components\core\apps\core
16 Jul 21:20:25 - [nodemon] starting 'node app'
Express server listening on port 3000 in development mode
```

Great!!

Then load http://localhost:3000/ in your browser. You will see a simple webpage with the title "Express", and the webpage says:



NOTE: This page is being loaded and a reference to the stylesheet '**style.css**' is made which resides in the folder:

public\stylesheets

You will find a '**style.less**' file there as well, as we had indicated to prefer less as our style code language.

So looks like the app is working. Time to find out how it works.

Request flow in Express
This is how a request to an Express server flows:

Route → Route Handler → Template → HTML

The route defines the URL schema. It captures the matching request and passed on control to the corresponding route handler.

The route handler processes the request and passes the control to a template. The template constructs the HTML for the response and sends it to the browser.

The route handler need not always pass the control to a template, it can optionally send the response to a request directly.

Templates here are made with the use of **jade**. These jade files are inside the *views* folder:

core		
	routes	index.js
	views	index.jade layout.jade

NOTE: To quickly convert HTML to Jade use this tool:
http://html2jade.aaron-powell.com/

Routes are URL schema for a website. In Express you define them using app.get(), app.post() ,app.delete() etc. The get, post, delete methods are derived from their corresponding HTTP verbs.

The routes directory is a convention, not a compulsion. In the routes directory we create appropriately named files which will handle the routes we define in the app.js file. We will import these files into our app and assign the functions defined in them to handle various routes.
The imported file becomes sort of like an instance of the class of the route handler file.

Again as a recommended convention, we create an appropriately named variable for the imported file from the routes directory (here:index). Then we pass one of its functions as the second parameter for the route (here: index) from the views directory.

For example:

```
app.get('/', index.index);
```

Rendering Views
You have seen res.render() and res.send() in action already and probably have a fair idea about what they do. res.send() will directly send a string of text to the browser and res.render()will render a Jade template.

While it is possible to create a website entirely using res.send(), it is certainly not recommended to do so, because you will only end up with a complex and dirty looking codebase. Jade is the default templating engine for Express. Let's find out the basics of res.render() and the Jade templates.

Open the file named layout.jade in the views directory. Let's examine its content:

```
!!!
html
 head
   title= title
   link(rel='stylesheet', href='/stylesheets/style.css')
 body!= body
```

This is Jade code.

The very basics of Jade is this:
i. HTML tag names to create tags, but without the angular brackets

ii. Spaces or tabs for indentation, but **don't mix them**
iii. # to assign an id to a tag
iv. . to assign a class to a tag, or create a div with a class if not already created
v. (property='value') to create attributes and assign values to a tag

The !!! you see in layout.jade is doing a doctype declaration of HTML5. Also notice the relatively 'cryptic' title= title and body!= body. This is what they are doing:

The following is an explanation of two pieces of code that might intrigue you.

title= title: The template expects a variable called title from the route handler. The variable will be set as the content of the title tag AFTER escaping special characters.

body!= body: The template expects a variable called body from the route handler which called the red.render() method. The variable will be set as the content of the body tag WITHOUT escaping special characters. If special characters like < and > are escaped, we won't have any HTML code within the body tag, hence we don't want the contents of the body variable to be escaped.

Our renderer code looks for a Jade template named index.jade in the views folder and passes an object to it. In our case it is { title: 'Express' }. The properties of the object will be available as variable names in the template with their respective name. In our case, we will have a variable named title in index.jade.

Invisible to the naked eye, another property is 'sent' to the template - layout with a default value of layout.jade. So the 'actual' renderer code can be understood as res.render('index', { layout: 'layout', title: 'Express' }).
When a template gets the layout variable. It makes all its variables available to the layout with their respective names, and itself in a variable named body.

Since layout has a default value, probably it might make you wonder if you could specify a layout of your own choice, or maybe even not use one. Your guesses are right.

To specify a custom layout do this:

```
{ layout: 'layout-ie', title: 'Express' }
```

If you don't want to use a layout do this:

```
{ layout: false, title: 'Express' }
```

By default res.send() and res.render() send the HTTP status code of 200. You can specify your own HTTP status code.

Custom status code example for res.send():

```
res.send('File not Found', 404);
```

Custom status code example for res.render() (make sure you have created a file named 404.jade in the views directory):

```
res.render('404', { title: 'File not Found', status:404 });
```

Open the file named 404.jade in the views directory. Let's examine its content:

```
!!!
html
 head
   title= title
   link(rel='stylesheet', href='/stylesheets/style.css')
 body!= body
```

This is Jade code.

Stylesheets
Even though we code our CSS in the .less file, we always include the .css file in the Jade template:

```
link(rel='stylesheet', href='/stylesheets/style.css')
```

Changes made to the views and everything in the public directory will be updated immediately when you refresh the browser. Refresh the browser and see what you get. In rare cases, the view may not be update - feel free to restart the server.

HTTP 403 Forbidden Error
Occasionally, you might see that (for example when trying to load images) you get a HTTP 403 Forbidden error message.

The HTTP 403 Forbidden error is caused by limited access to a web domain or directory. Administrators can limit access to a web domain's directory by removing the anonymous user or requiring a password to view the content of the location. To remove or fix the error requires added permissions on the Windows host machine using the Internet Information Services (IIS) console. Once permissions are set, browsers to the domain will be able to access and read files normally again.

Read more: How to Fix the HTTP 403 Forbidden Error | eHow.com http://www.ehow.com/how_5180610_fix-http-forbidden-error.html#ixzz21XLYw2fj

….. MORE HERE TO FOLLOW ...

***Locomotive**: brings additional MVC-based structure, for architecting larger applications, while leveraging the power of Express and Connect middleware.*

Locomotive[16] is a web framework for Node that makes building web applications easier and more efficient. This is accomplished by architecting an application around REST principles, the familiar MVC design pattern, and establishing a set of conventions for organizing the source code of an application.

Locomotive closely follows the expectations popularized by Ruby on Rails, so developers familiar with that environment can adapt quickly, while enjoying the benefits of idiomatic JavaScript and the power of Node's asynchronous I/O.

Locomotive is built entirely on top of Express, a minimalist and high-performance web framework. Locomotive brings additional MVC-based structure, for architecting larger applications, while leveraging the power of Express and Connect middleware.

While adopting the best ideas from existing web frameworks, Locomotive has its own set of opinions about the needs of modern applications in the era of big data and real-time events. In particular, these challenges demand polyglot persistence. As such, Locomotive places no restrictions on the model layer, allowing developers to choose the database (or databases) and object mapping layer that best fits their needs.

The end result is an application comprised of beautifully written code, that is both familiar, yet able to meet the demands of the modern web.

In order to follow Locomotive's directory structure, we have to extend ours as follows:

`apps`				
	`core`			
		`app`		
			controllers	
				`<namespace> e.g.` **admin**
			models	
			views	
		config		
			environments	
				all.js
			initializers	
			routes.js	

[16] http://locomotivejs.org

		`data`		
		`lib`		
		`persistence`		
		`app1_embed.js`		
		`public`		
		`routes`		
		`views`		
		`app.js`		
		`package.json`		

NOTE: The directories and files in **bold** are the added directories and files for Locomotive.

Directory Structure

Locomotive favors convention over configuration. Part of these conventions is a well defined directory structure.

`app/controllers`
Contains the controllers that handle requests sent to an application.

`app/models`
Contains the models for accessing and storing data in a database.

`app/views`
Contains the views and layouts that are rendered by an application and sent as a response to the client.

`config`
Configuration for the application, including routes, databases, etc.

`config/environments`
Environment-specific configuration. For example, *development* and *production* are two environments that require different configuration.

`config/initializers`
Initialization code that is `require()`'d before the applications starts.

`public`
Static files and compiled assets served by the application.

Routing

Locomotive's router is responsible for receiving requests to your application and dispatching each request to a controller's action. Dispatching is based on the request's URL and HTTP method.

The router also dynamically generates routing helper functions that build paths and URLs. These helpers are available in controllers and views, eliminating the need to hardcode strings in views.

Routes are configured in `config/routes.js`

Resource Routing

Resource routing declares routes to a resourceful controller. These routes map HTTP methods to controller's actions, according to REST conventions. Resource routing is the preferred routing mechanism in a Locomotive application.

For example, declaring this route:

```
this.resources('photos');
```

will result in the following routes being mapped to PhotosController:

Method	Path	Action
GET	/photos	index
GET	/photos/new	new
POST	/photos	create
GET	/photos/:id	show
GET	/photos/:id/edit	edit
PUT	/photos/:id	update
DELETE	/photos/:id	destroy

Additionally, the following routing helpers will be declared and available in controllers and views:

Helper	Returns
photosPath	/photos
photoPath(id)	/photos/123
newPhotoPath	/photos/new
editPhotoPath(id)	/photos/123/edit

Each of these path routing helpers has a corresponding URL routing helper (such as photosURL), that returns an absolute URL, including scheme and host. Using these helpers, as opposed to hard-coding paths, makes an application easier to maintain as routes change.

Singleton Resources

An application may contain singleton resources that are not referenced using an ID. For example,/account is often used to show account details for the logged in user (rather than /account/:id).

In this case, a singleton resource route can be declared:

```
this.resource('account');
```

resulting in the following routes being mapped to AccountController:

Method	Path	Action
GET	/account/new	new
POST	/account	create

GET	/account	show
GET	/account/edit	edit
PUT	/account	update
DELETE	/account	destroy

Additionally, the following routing helpers will be declared and available in controllers and views:

Helper	Returns
accountPath	/account
newAccountPath	/account/new
editAccountPath	/account/edit

Each of these path routing helpers has a corresponding URL routing helper.

Namespaces

You may want to organize an application by grouping controllers under a namespace. Most commonly, administrative functionality is grouped under an Admin:: namespace by placing controllers in the app/controllers/admin directory.

This namespace is reflected in the routes:

```
this.namespace('admin', function() {
  this.resources('posts');
});
```

resulting in the following URLs and routing helpers being mapped to Admin::PostsController:

Method	Path	Action	Helper
GET	/admin/posts	index	adminPostsPath
GET	/admin/posts/new	new	newAdminPostPath
POST	/admin/posts	create	
GET	/admin/posts/:id	show	adminPostPath(id)
GET	/admin/posts/:id/edit	edit	editAdminPostPath(id)
PUT	/admin/posts/:id	update	
DELETE	/admin/posts/:id	destroy	

Each of the above path routing helpers has a corresponding URL routing helper.

Nested Routes

It is common to have resources that are logically children of other resources. These relationships can be captured in your routing:

```
this.resources('bands', function() {
  this.resources('albums');
});
```

In addition to the routes for bands, the following URLs and routing helpers are mapped to AlbumsController

Method	Path	Action	Helper
GET	/bands/:band_id/albums	index	bandAlbumsPath(bandId)
GET	/bands/:band_id/albums/new	new	newBandAlbumPath(bandId)
POST	/bands/:band_id/albums	create	
GET	/bands/:band_id/albums/:id	show	bandAlbumPath(bandId,id)
GET	/bands/:band_id/albums/:id/edit	edit	editBandAlbumPath(bandId,id)
PUT	/bands/:band_id/albums/:id	update	
DELETE	/bands/:band_id/albums/:id	destroy	

Each of the above path routing helpers has a corresponding URL routing helper.

While no limit is placed on the number of levels to which resources can be nested, applications are encouraged not to nest more than one level deep, as doing otherwise quickly becomes cumbersome.

Match Routes

In addition to resource routing, Locomotive can match arbitrary URL patterns to a controller's action.

For example, declaring this match route:

```
this.match('songs/:title', { controller: 'songs', action: 'show' });
```

will cause SongsController's show() action to handle requests for URLs which match the pattern.

```
/songs/like-a-rolling-stone
/songs/all-along-the-watchtower
```

Shorthand notation, in the form of *controller#action* is also available to target a controller's action.

For example, the following route is equivalent to the one declared above:

```
this.match('songs/:title', 'songs#show');
```

The values associated with named placeholders, in this case :title, will be available in the controller via the param() function.

```
this.param('title');  // returns 'like-a-rolling-stone'
```

Routing to Middleware

Instead of routing to a controller, you can route directly to any Connect-compatible middleware.

For example, the following will route login requests to Passport for authentication.

```
this.match('login', passport.authenticate('local', { successRedirect: '/',
                                                      failureRedirect: '/login' })
                    { via: 'post' });
```

Routes can be sent to a middleware chain by specifying each function as an element within an array.

```
this.match('somewhere', [ middlewareOne(),
                          middlewareTwo() ]);
```

Root Route

The root route, `/`, can be declared with the root() function:

```
this.root('pages#main');
```

Controllers

Controllers consist of a set of functions known as *actions*. Routing will dispatch a request to a controller's action, which does the work of loading data and rendering views. The rendered view will be sent in the response to the client.

Defining Controllers

In Locomotive, controllers are instances of Controller. Defining a new controller is as simple as creating an instance and exporting it via the module.

```
var PhotosController = new Controller();

module.exports = PhotosController;
```

Action Functions

Any functions attached to the controller are available as actions. When Locomotive receives a request, it will create a new instance of the controller and call the appropriate action function.

For example, if a request is received for /photos/123, Locomotive will call the show() action of PhotosController.

```
PhotosController.show = function() {
  this.render();
}
```

Controllers are responsible for sending a response to the request. This is typically accomplished by render()ing a view or issuing a redirect().

Parameters

A controller will often inspect data sent in the request, in order to construct an appropriate response. Use the param() function to access data contained in route, query, or body parameters.

```
PhotosController.show = function() {
  this.title = this.param('title');
  this.render();
}
```

The value returned by param() will be found by checking parameters in the following order:

- Checks route params (req.params), ex: /photos/:id
- Checks query string params (req.query), ex: ?id=12
- Checks urlencoded body params (req.body), ex: id=12

Rendering Views

Rendering a view is accomplished by calling render(). Any instance variables attached to the controller will be made available to the view. By convention, variables named with a leading underscore are considered private and will not be made available to the view.

```
PhotosController.show = function() {
  // this._photo is "private", and not available in the view
  this.title = this._photo.title;
  this.description = this._photo.description;
  this.render();
}
```

The view found in views/photos/show.html.ejs will be rendered.

```
<h2><%= title %></h2>
<p><%= description %></p>
```

NOTE: This is EJS code

Alternatively:

The view found in views/photos/show.jade will be rendered.

```
extends layout

block content
  h1= title
  p #{description}
```

NOTE: This is Jade code

Render an Action's View

By default, the view corresponding to the invoked action will be rendered, in this case show. A different action's view can be rendered by specifying it (here for example ‘index’):

```
this.render('index');
```

Render a Controller's View

If you want to render a template corresponding to a different *controller*'s action, that can be accomplished as follows (so here instead of using the photos controller’s action ‘show’, we use the albums controller’s action ‘show’):

```
this.render('albums/show');
```

Options

Options can be used to render a different format (here: xml) or use a different template engine (here: jade).

```
this.render({ format: 'xml' });
```

```
this.render({ engine: 'jade' });
```

Redirecting

In some situations, instead of rendering a view, a controller may want to redirect the request.

```
this.redirect('/login');
```

Filters

Filters are functions that are run before or after the action function. Each function is executed sequentially, one after the other, when each filter calls next. This is especially useful for loading data from a database and avoiding nested blocks of async code.

```
PhotosController.before('show', function(next) {
  var self = this;
  Photo.findOne(this.param('id'), function(err, photo) {
    if (err) { return next(err) }
    self._photo = photo;
    next();
  });
});
```

Connect-compatible middleware can also be used directly as a filter.

```
PhotosController.before('create', connect.limit('10mb'));
```

After filters are identical to *before* filters, with the exception that they are executed after the action has completed.

```
PhotosController.after('show', function(next) {
  // update view counter
  next();
});
```

Request and Response

If an application needs access to the raw request and response, each is available as instance variable within the controller.

```
PhotosController.show = function() {
  var req = this.req;  // aka this.request
  var res = this.res;  // aka this.response
  this.render();
}
```

Views

Locomotive inherits its view layer from Express, so any compatible template engine can be used, including EJS, Jade, Haml, and others.

Configuration

By default, Locomotive uses EJS as its template engine. That is easily changed by setting the option in config/environments/all.js:

```
this.set('view engine', 'jade');
```

Selecting an Engine at Render Time

Controllers can selectively override the application's default template engine in the call to render:

```
PhotosController.show = function() {
  this.render({ engine: 'haml' });
}
```

Datastores

Locomotive recognizes that the demands of modern web applications entail the use of a variety of datastores. Choosing the correct datastore is an important decision, and Locomotive does not impose on that choice. Locomotive is fully functional, independent of the choice of database or object mapping layer used by an application.

Model Object Introspection

Locomotive includes a urlFor helper that can be used to build URLs to a controller's action. For example:

```
urlFor({ controller: 'animals', action: 'show', id: '123' });
// => http://www.example.com/animals/123
```

This helper is also model aware. It can be handed a model object (here: Animal), which will be introspected in order to determine the type of record and build the corresponding URL.

```
var animal = new Animal();
animal.id = '123';
urlFor(animal);
// => http://www.example.com/animals/123
```

By default, model awareness works for any datastore that represent each distinct type of record with a unique class. In cases where this does not apply, other datastores can be made model aware through the use of adapters.

QUESTION: how would this apply to MySQL ???

Mongoose

To use the Mongoose adapter, register it in config/environments/all.js:

```
this.datastore(require('locomotive-mongoose'));
```

Mongoose documents can then be passed directly to urlFor():

```
Animal.findById(this.param('id'), function(err, animal) {
  if (err) { return next(err); }
  self.url = self.urlFor(animal);
  next();
});
```

THIS COMPLETES THE GUIDE ON LOCOMOTIVE (http://locomotivejs.org/guide/).

***Sequelize**: a well-documented ORM project that supports MySQL*

The **Sequelize**[17] project is a well documented ORM project that supports MySQL. It supports a variety of ORM features that includes models, validation and relationship handling.[18]

The Sequelize library provides easy access to a MySQL database by mapping database entries to objects and vice versa. To put it in a nutshell... it's an ORM (Object-Relational-Mapper). The library is written entirely in JavaScript and can be used in the Node.JS environment.

Installation

You can install Sequelize via NPM:

```
npm install sequelize -g
```

Or just download the code from the git repository DIRECTLY INTO THE LIB FOLDER so with a simple 'git pull' we will have the latest release (!):
NOTE: Use this approach for ALL MODULES !!

```
cd %HOME/public_html/core/components/core/apps/core/lib
git clone git://github.com/sdepold/sequelize.git
```

Enter the newly created directory, install globally, and automatically get the latest dependencies:

```
cd sequelize
npm install -g
```

And require its entry file index.js in your Node file (for example: app.js):

```
var Sequelize = require(__dirname + "/lib/sequelize/index")
```

This will make the class Sequelize available.

Usage

.. see http://sequelizejs.com/

We follow advice coming from this tutorial:

http://www.ziggytech.net/technology/web-development/experiences-with-node-js-porting-a-restful-service-written-in-java/

A more advanced app.js

The app.js file contains the core instructions for the application. It defines the application, sets global variables, initializes the ORM, routes files, and starts our HTTP server. There are many good resources available on the web for creating this core file. This example uses a combination of many different techniques from multiple sources. It also includes added support for pulling in command line parameters. There are modules available that add helper methods

[17] http://sequelizejs.com/
[18] http://www.ziggytech.net/technology/web-development/experiences-with-nodejs-researching-node/

for this, but it is pretty simple to handle without them.

Example of the more advanced **app_sequelize.js**:

```
/**
 * Module dependencies.
 */
var express = require('express');
//NO LONGER IN USE var store = require('./routes/store');
//NO LONGER IN USE var http = require('http');
//NO LONGER IN USE var app = express();

/**
 * Create Application Server.
 */
var app = module.exports = express.createServer();

/**
 * Set Route Init Path.
 */
var routes = "./routes.js";

/**
 * Set Database Init Path.
 */
var database = "./database.js";

/**
 * app
 * @type {Express}
 *
 * The Singleton of Express app instance
 */
GLOBAL.app = app;

/**
 * Retrieve Command Line Arguments
 * [0] process : String 'node'
 * [1] app : void
 * [2] port : Number 3000
 */
var args = process.argv;

/**
 * port
 * @type {Number}
 *
 * HTTP Server Port
 */
var port = args[2] ? args[2]: 3000;

/**
 * Database Connections
 */
var database_options = {
        schema: "core",
        user: "root",
        password: "",  // High Security FTW.
        host: "localhost",
        port: "3306"
};
```

```
/**
 * Configuration
 */
app.configure(function(){
//NO LONGER IN USE  app.set('port', process.env.PORT || 3000);
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser('your secret here'));
  app.use(express.session());
  app.use(app.router);
  app.use(require('less-middleware')({ src: __dirname + '/public' }));
  app.use(express.static(__dirname + '/public'));
});

app.configure('development', function(){
  app.use(express.errorHandler({dumpExceptions: true, showStack: true}));
  database_options.logging = true;
});

app.configure('production', function(){
  app.use(express.errorHandler());
  database_options.logging = false;
});

/**
 * db (database)
 * @type {Object}
 * @param Object [database_options] the database options
 */
GLOBAL.db = require(database) (database_options);

/**
 * Routes
 */
var routes = require(routes);

//NO LONGER USED app.get('/', store.home);

/**
 * HTTP Server
 */
app.listen(port);
console.log("Express server listening on port %d in %s mode using Sequelize ORM.",
        app.address().port,
        app.settings.env);
//NO LONGER USED http.createServer(app).listen(app.get('port'), function(){
//NO LONGER USED  console.log("Express server listening on port " + app.get('port'));
//NO LONGER USED});
```

Routing file

This is the routing file that defines what URL's will use which resource. Using the express-resource module, simple routes were created for the RESTful resources with very minimal lines of code. In this example, the URL is passed in and the Person resource module to the resource

to define that relationship.

routes.js

```
/**
 * Resource Routes
 */
module.exports = function() {
    var Resource = require('express-resource');
    app.resource('services/persons', require('./resource/Person'));
};
```

Database file

In the database.js file, the database object is created and a utility method for pushing the data from the model to an object is appended. After the ORM has created its database connection, the models can be built.

database.js

```
/**
 * Database Connection
 */
module.exports = function(options) {
    var database = {
        options: options
    };

    var Sequelize = require('sequelize');
    database.module = Sequelize;
    database.client = new Sequelize(options.schema, options.user, options.password, {
            host: options.host,
            port: options.port,
            logging: options.logging,
            dialect: 'mysql',
            maxConcurrentQueries: 100
        });

    /**
     * @type {Object}
     * Map all attributes of the registry
     * (Instance method useful to every sequelize Table)
     * @this {SequelizeRegistry}
     * @return {Object} All attributes in an Object.
     */
    database.map = function() {
        var obj = new Object(),
            ctx = this;
        ctx.attributes.forEach(function(attr) {
            obj[attr] = ctx[attr];
        });

        return obj;
    };

    database.models = require('./models.js')(database);

    return database;
};
```

Models file

Using the Sequelize ORM model structure, the individual model Java classes are converted to a singular JSON object that represent all the database tables with validations, constraints and

relationships. In this models.js example, there is a mapAttributes linked to the db.map function to return only the data in a specific model. For this example, the Person model is shown.

models.js

```
/**
 * Sequelize ORM Models
 */
module.exports = function(db) {
    /**
     * @type {Object}
     * All models we have defined over Sequelize, plus the db instance itself
     */
    var self = {
        Person: db.define('PERSON', {
            KP_PERSONID: { type: Sequelize.INTEGER, allowNull: false, primaryKey: true},
            PERSON_FIRSTNAME: { type: Sequelize.STRING, allowNull: true, validate: { max: 225 }
 },
            PERSON_LASTNAME: { type: Sequelize.STRING, allowNull: true, validate: { max: 225 }
 }
        },
        {
            timestamps: false,
            freezeTableName: true,
            instanceMethods: {
                mapAttributes: db.map
            }
        }),
    };
    return self;
};
```

Resources files

A basic set of CRUD functions are constructed for the model and replicated for each resource. The Sequelize ORM made this task very simple and allowed for querying the database with a few very simple function calls and callbacks. After querying the ORM, the success callback will call the mapAttributes function and return a JSON object to the response object. In the case of an error, the error callback function fires returning the error to the response object. For this example, the List and the Show methods are shown. Other methods can be created for a RESTful implementation. Documentation for the Express Resource module can be found on the project site.

resources/Person.js

```
var model = models.Person;

/*
 * List
 * @GET /model/
 * @returns JSON list of records
 */
exports.index = function(request, response) {
    model.findAll()
        .success(function(models) {
            response.json(models.map(function(record) {
                return record.mapAttributes();
            }));
        })
        .failure(function(error) {
            response.send(error);
        });
};
```

```
/*
 * Show by Id
 * @GET /model/:id
 * @returns JSON requested record
 */
exports.show = function(request, response) {
    model.find(request.params.person)
        .success(function(record) {
            response.json(record.mapAttributes());
        })
        .failure(function(error) {
            response.send(error);
        });
};
```

========= OUR OWN APPROACH - BUILDING THE FILES STEP BY STEP ===========

STEP 1) Creating the app file, called: core_server.js

We start off with the minimalist code, that works (this is in line with ExpressJS as of version 3.0[19]):

```
/**
 * Create Application Server.
 */
var express = require('express');
var http = require('http');
var server = http.createServer(express);

/**
 * HTTP Server
 */
server.listen(3000);
console.log("Express server listening on port %d.",
        server.address().port);
```

core_server.js

This file sits here in our directory structure:

core						
	components					
		core				
			apps			
				core		
					core_server.js	

If not already started, start this server from its directory with the use of nodemon, like so:

```
nodemon core_server.js
```

Once the file is saved, nodemon will notice a file change and will restart this server.

This will output:

```
Express server listening on port 3000.
```

STEP 2) Modify the file core_server.js for the express dependency

We change the core_server.js file to:

```
/**
 * Module dependencies.
```

[19] https://github.com/visionmedia/express/wiki/New-features-in-3.x

```
 */
var express = require('express');
var http = require('http');

/**
 * Create Application Server.
 */
var server =  http.createServer(express);

/**
 * HTTP Server
 */
server.listen(3000);
console.log("Express server listening on port %d.",
        server.address().port);
```

core_server.js

Now, the module dependencies are nicely listed on their own.

STEP 3) Modify the file core_server.js for the globalization of the Singleton of Express express instance.

We change the core_server.js file to:

```
/**
 * Module dependencies.
 */
var express = require('express');
var http = require('http');

/**
 * Create Application Server.
 */
var server = http.createServer(express);

/**
 * express
 * @type {Express}
 *
 * The Singleton of Express express instance
 */
GLOBAL.express = express;

/**
 * HTTP Server
 */
server.listen(3000);
console.log("Express server listening on port %d.",
        server.address().port);
```

core_server.js

Now the Singleton of Express express instance is globally accessible.

STEP 4) Modify the file core_server.js to allow for Command Line Arguments

We change the core_server.js file to:

```
/**
 * Module dependencies.
```

```
 */
var express = require('express');
var http = require('http');

/**
 * Create Application Server.
 */
var server = http.createServer(express);

/**
 * express
 * @type {Express}
 *
 * The Singleton of Express express instance
 */
GLOBAL.express = express;

/**
 * Retrieve Command Line Arguments
 * [0] process : String 'node'
 * [1] app : void
 * [2] port : Number 3000
 */
var args = process.argv;

/**
 * HTTP Server
 */
server.listen(3000);
console.log("Express server listening on port %d.",
        server.address().port);
```

core_server.js

Now we have prepared an args variable that can accept command line arguments.

STEP 5) Modify the file core_server.js by adding a port variable that can be set through Command Line Arguments

We change the core_server.js file to:

```
/**
 * Module dependencies.
 */
var express = require('express');
var http = require('http');

/**
 * Create Application Server.
 */
var server = http.createServer(express);

/**
 * express
 * @type {Express}
 *
 * The Singleton of Express express instance
 */
GLOBAL.express = express;

/**
 * Retrieve Command Line Arguments
 * [0] process : String 'node'
 * [1] app : void
 * [2] port : Number 3000
```

```
 */
var args = process.argv;

/**
 * port
 * @type {Number}
 *
 * HTTP Server Port
 */
var port = args[2] ? args[2]: 3000;

/**
 * HTTP Server
 */
server.listen();
console.log("Express server listening on port %d.",
        server.address().port);
```

core_server.js

Now we have added a port variable that can be set through Command Line Arguments. For example if we would like to set the port to a number other than 3000 (the default) we would only need to start the core_server like so (e.g. for port 3010):

```
nodemon core_server.js -process -app -port=3010
```

MORE AND NEW APPROACH TO SEQUELIZE[20]

Sequelize is an object relational mapper (ORM) that takes much of the repetition out of the tasks in the preceding sections.

You can use Sequelize to define objects shared between the database and your program, then pass data to and from the database using those object rather than writing a query for every operation.

This becomes a major time saver when it comes time to perform maintenance or add a new column, and makes overall data management less error-prone.

If a database by the name of ‘core’ is already created, for user ‘root’ with password ‘<blank>’ it's time to create a Person entity inside the database. Sequelize handles the creation for you, so you don't have to take care of any manual SQL at this point:

```
var Sequelize = require('sequelize');

var db = new Sequelize('core', 'root', '', {
  host: 'localhost'
});

var Person = db.define('person', {
  kp_PersonID: Sequelize.INTEGER,
  Person_FirstName: Sequelize.STRING,
  Person_LastName: Sequelize.STRING
});

Person.sync().on('success', function() {
  console.log('person table was created.');
}).on('failure', function(error) {
  console.log('Unable to create person table');
});
```

The output is:

```
Executing: CREATE TABLE IF NOT EXISTS `person` (`kp_PersonID` INT, `Person_FirstName`
VARCHAR(255), `Person_LastName` VARCHAR(255), `id` INT NOT NULL auto_increment ,
`createdAt` DATETIME NOT NULL, `updatedAt` DATETIME NOT NULL, PRIMARY KEY (`id`))
 ENGINE=InnoDB;
person table was created.
```

In this example, a Person was defined as an entity containing a first and last name field and a kp (=key primary) id field. As you can see in the output, Sequelize added an auto-incremented primary key column (here: id), a createdAt column, and an updatedAt column. This is typical of many ORM solutions, and provides standard hooks by which Sequelize is able to reference and interact with your data.

Sequelize differs from other libraries in being based on a **listener-driven architecture**, rather than the callback-driven architecture used elsewhere. This means that you have to listen for both success and failure events after each operation, rather than having errors and success indicators returned with the operation's results.

[20] http://ofps.oreilly.com/titles/9781449398583/chapter_8.html

The following example creates two tables with a many-to-many relationship. The order of operation is:

1. Set up the entity schemas.
2. Synchronize the schemas with the actual database.
3. Create and save a Book object.
4. Create and save a Person object.
5. Establish a relationship between the Person and the book.

```
var Sequelize = require('sequelize');

var db = new Sequelize('core', 'root', 'dev', {
  host: 'localhost'
});

var Person = db.define('person', {
  kp_PersonId: Sequelize.INTEGER,
  Person_FirstName: Sequelize.STRING,
  Person_LastName: Sequelize.STRING
});

var Book = db.define('book', {
  name: Sequelize.STRING
});

Person.hasMany(Book);
Book.hasMany(Person);

db.sync().on('success', function() {
  Book.build({
    name: 'Through the Storm'
  }).save().on('success', function(book) {
    console.log('Book saved');
    Person.build({
      kp_PersonID: '1',
      Person_FirstName: 'David',
      Person_LastName: 'Beckham'
    }).save().on('success', function(record) {
      console.log('Person saved.');
      record.setBooks([book]);
      record.save().on('success', function() {
        console.log('Person & Book Relation created');
      });
    });
  }).on('failure', function(error) {
    console.log('Could not save book');
  });
}).on('failure', function(error) {
  console.log('Failed to sync database');
});
```

About Express-Resource

express-resource[21] provides resourceful routing to express.

Installation

Download the code from the git repository DIRECTLY INTO THE LIB FOLDER so with a simple 'git pull' we will have the latest release (!):

```
cd %HOME/public_html/core/components/core/apps/core/lib
git clone git://github.com/visionmedia/express-resource.git
```

Enter the newly created directory, install globally, and automatically get the latest dependencies:

```
cd express-resource
npm install -g
```

Usage

To get started simply require('express-resource'), and this module will monkey-patch Express, enabling resourceful routing by providing the app.resource() method. A "resource" (here: person) is simply an object, which defines one or more of the supported "actions" listed below:

```
exports.index = function(req, res){
  res.send('person index');
};

exports.new = function(req, res){
  res.send('new person');
};

exports.create = function(req, res){
  res.send('create person');
};

exports.show = function(req, res){
  res.send('show person ' + req.params.person);
};

exports.edit = function(req, res){
  res.send('edit person ' + req.params.person);
};

exports.update = function(req, res){
  res.send('update person ' + req.params.person);
};

exports.destroy = function(req, res){
  res.send('destroy person ' + req.params.person);
};
```

The app.resource() method returns a new Resource object (here: person), which can be used to further map pathnames, nest resources, and more.

```
var express = require('express')
  , Resource = require('express-resource')
  , app = express.createServer();
```

[21] https://github.com/visionmedia/express-resource#readme

```
app.resource('persons', require('./person'));
```

Default Action Mapping

Actions are then mapped as follows (by default), providing `req.params.forum` which contains the substring where ":person" is shown below:

GET	/persons	-> index
GET	/persons/new	-> new
POST	/persons	-> create
GET	/persons/:person	-> show
GET	/persons/:person/edit	-> edit
PUT	/persons/:person	-> update
DELETE	/persons/:person	-> destroy

__NowJS__: Middleware for Real-Time connection between Client and Server

NowJS[22] maintains a shared namespace (called “now”) between javascript code on the client (i.e. in the web page) and the server. This allows for direct calling of remote javascript functions (e.g. for a chat application).

On Windows: download the binary from here: https://github.com/Flotype/now/zipball/windows
Unzip and place it inside the node_modules directory.

See a nice tutorial here:
http://nodegames.blogspot.co.uk/2011/12/install-nowjs-in-windows.html

Others:

NOTE: NowJS requires **Socket.IO**. (get it here https://github.com/LearnBoost/socket.io) and **Node-Gyp** and **Node-Proxy** and **Hiredis-Node**.

```
git clone https://github.com/LearnBoost/socket.io.git
```

```
git clone https://github.com/TooTallNate/node-gyp.git
```

```
git clone https://github.com/samshull/node-proxy.git
```

```
git clone https://github.com/pietern/hiredis-node.git
```

If any of the above install give an error on redis, see below warning:

WARNING: Install this version of redis (0.7.1), not the newer one (0.7.2).

```
npm install redis@0.7.1
```

Get a clone from the repository:

```
git://github.com/Flotype/now.git
```

Or by typing inside the directory where you want the clone (e.g $HOME/downloads/git/):

```
git clone https://github.com/Flotype/now.git
```

Inside the downloaded project directory (here: now) type:

```
npm install -g
```

[22] http://nowjs.com

***N-Ext** - Use the non-DOM parts of ExtJS (Ext Core + Ext.data currently) server-side with Node.js*

N-Ext[23] is the compression of *Node* and *Ext*. Its purpose is to allow developers to use the ExtJS 4 Javascript framework in a server-side environment using Node.js.

Install N-Ext:

Get a clone from the repository:

```
git://github.com/xcambar/n-ext.git
```

Or by typing inside the directory where you want the clone (e.g $HOME/downloads/git/):

```
git clone https://github.com/xcambar/n-ext.git
```

Inside the downloaded project directory (here: n-ext) type:

```
npm install -g
```

Set up N-Ext:

1. Download the latest version of ExtJS 4
2. Unzip the `src` folder in `%PROJECT_ROOT%/lib/Ext` (this can be changed with a single line of code)
3. In the main file (it can be any file) of your application, type the following:
4.
```
var sencha = require('n-ext');
    sencha.setExtPath('./lib/Ext/');
    sencha.bootstrapCore();
```

   The variable `sencha` above is an instance of the `nExtLoader` prototype, which API can be found here. The 'setExtPath' method can take either an absolute or relative path. Write it as you would declare a path for node modules, as it is Node's `require()` that runs under the hood.

And that's it! You now have the beloved `Ext` namespace available application-wide.
….
n-ext_server1.js (for example):

```
var sencha = require('n-ext');
sencha.setExtPath('./lib/Ext/');
sencha.bootstrapCore();
```

More info in the wiki at https://github.com/xcambar/n-ext/wiki

Note on Ext.Loader
All the namespaces which roots are located on the same folder that the Ext namespace are available directly to the Dynamic class loading package of ExtJS 4.

Example: Consider the following file structure

apps				

[23] https://github.com/xcambar/n-ext#readme

	core			
		app		
		data		
		lib		
			Ext	
			Package1	
			Package2	

If you bootstrap your application as above, you can write the following code without having to configure Ext.Loader:

```
Ext.require('Package1.model.Awesome');
```

Declare another location for custom namespaces ?

ExtJS requires that you declare each and every namespace wherein you want to use the autoloader they provide. So, if you want to use a namespace `Foo`, which contains classes you want to have dynamically loaded in your app, you have to do the following:

```
Ext.Loader.setPath('Foo', './lib/Foo');
```

With *N-Ext*, you can use another mechanism, taking advantages of the capabilities of NodeJS' `require` function. Let's say you have a bunch of namespaces (NS1, NS2 and so on) located in a single folder (a priori another folder that the one containing the Ext library) called `vendor`. Using the line of code below, you won't be required to run `Ext.Loader.setPath(..)` anymore for those namespaces.

```
var sencha = require('n-ext');
sencha.addLibPath('./vendor/'); // <-- This one!
sencha.bootstrapCore();

Ext.require('NS1.model.Foo'); // It Works!
```

Of course, you can add as many paths as you want.

3. Database(s)

Relational Databases
There are still many good reasons to use a traditional database with SQL, and Node interfaces with popular open source choices.[24]

3. MySQL

Uses MySQL[25] relational database.

3.1.1 Using **NodeDB**

Installation
The MySQL client development libraries are a prerequisite for the Node database module. On Ubuntu, the libraries can be installed using apt:

```
sudo apt-get install libmysqlclient-dev
```

Using npm, install a package named db-mysql:

```
npm install -g db-mysql
```

In order to run the examples in this section you will need to have a database called 'core' with a user 'root' having password '<blank>'. The following script will create the database table and basic schema:

```
DROP DATABASE IF EXISTS core;

CREATE DATABASE core;

GRANT ALL PRIVILEGES ON core.* TO 'root'@'%' IDENTIFIED BY '';

USE core;

CREATE TABLE person( kp_PersonId int auto_increment primary key, Person_FirstName varchar(255),
Person_LastName varchar(255) );
```

Selection
The following example selects all kp_PersonId and Person_FirstName columns from a person table:

```
var mysql = require( 'db-mysql' );

var connectParams = {
  'hostname': 'localhost',
  'user': 'root',
  'password': '',
  'database': 'core'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");
```

[24] http://ofps.oreilly.com/titles/9781449398583/chapter_8.html
[25] http://www.mysql.com

```
  this.query()
    .select(['kp_PersonId', 'Person_FirstName'])
    .from('person')
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
      } else {
        console.log(rows);
      }
    });
});
```

As you can probably guess, this executes the equivalent of the SQL command SELECT kp_PersonId, Person_FirstName FROM person.

The output is for example:

```
{ kp_PersonId: 1, Person_FirstName: 'David' }
```

Insertion

Inserting data is very similar to selection because commands are chained in the same way. This is how to generate the equivalent to INSERT INTO person (Person_FirstName) VALUES ('Victoria');

```
var mysql = require( 'db-mysql' );

var connectParams = {
  'hostname': 'localhost',
  'user': 'root',
  'password': '',
  'database': 'core'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");

  this.query()
    .insert('person', ['Person_FirstName'], ['Victoria'])
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
        console.log(error);
      }
      else console.log(rows);
    });
});
```

The output is:

```
{ id: 2, affected: 1, warning: 0 }
```

The .insert command takes 3 parameters:

1. The table name
2. The column names being inserted

3. The values to insert in each column

NOTE: The database drivers take care of escaping and converting the data types in your column values, so you don't have to worry about SQL Injection attacks through code passing through this module.

Updating
Like selection and insertion, updates rely on chained functions to generate equivalent SQL queries. This example demonstrates the use of a query parameter to filter the update, rather than performing it across all records in the database table.

```
var mysql = require( 'db-mysql' );

var connectParams = {
  'hostname': 'localhost',
  'user': 'root',
  'password': '',
  'database': 'core'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");

  this.query()
         .update('person')
         .set({'Person_LastName': 'Beckham' })
         .where('Person_FirstName = ?', [ 'Victoria' ])
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
        console.log(error);
      }
      else console.log(rows);
  });
});
```

The output is:

```
{ id: 0, affected: 1, warning: 0 }
```

Updating a row consists of three steps:

1. The .update command, which takes the table name (*person* in this case) as a parameter.
2. The .set command, which uses a key-value object pair to identify the column names to update and their values.
3. The .where command, which tells MySQL how to filter the rows that will be updated.

Deletion
Deletion is very similar to updates, except in the case of a delete there are no columns to update. If no where conditions are specified, all records in the table will be deleted.

```
var mysql = require( 'db-mysql' );

var connectParams = {
  'hostname': 'localhost',
```

```
  'user': 'root',
  'password': '',
  'database': 'core'
}

var db = new mysql.Database( connectParams );

db.connect(function(error) {
  if ( error ) return console.log("Failed to connect");

  this.query()
    .delete()
    .from('person')
    .where('Person_FirstName = ?', [ 'David' ])
    .execute(function(error, rows, columns) {
      if ( error ) {
        console.log("Error on query");
        console.log(error);
      }
      else console.log(rows);
    });
});
```

The output is:

```
{ id: 0, affected: 1, warning: 0 }
```

The .delete command is similar to the .update command, except it does not take any column names or data values. In this example, wildcard parameters are demonstrated in the "where" clause: 'Person_FirstName = ?'. The question mark is replaced by the Person_FirstName parameter in this code before execution. The second parameter is an array, because if multiple question marks are used, the database driver will take the values in order from this parameter.

3.1.2 Using Sequelize

See separate section about Sequelize[26].

MySQLDumper - Backup for MySQL Database

MySQLDumper[27] is a PHP and Perl based tool for backing up MySQL databases. You can easily dump your data into a backup file and - if needed - restore it. It is especially suited for shared hosting web spaces, where you don't have shell access. MySQLDumper is an open source project and released under the GNU-license.

Notes about MySQLDumper (see also the FAQ here http://www.mysqldumper.net/faq/)

- MySQLDumper is running under any operating system. Just start the script in your browser under "http://www.yourDomain.com/path_to_mysqlumper/"
- Your server is running in Safe-Mode and so mysqldumper is not allowed to create directories. You need to create the following directories manually (Don't forget to chmod all directories to 0777.):
    - work
    - work/backup

[26] http://ofps.oreilly.com/titles/9781449398583/chapter_8.html
[27] http://www.mysqldumper.net/

- work/config
- work/log
- work/structure

Folder hierarchy with MySQLDumper:

public_html			
	mysqldumper		
		< all folders other than work >	
		work	
			backup
			config
			log
			structure

3. Filemaker Pro

Uses FileMaker Pro[28] relational database.

[28] http://www.filemaker.com

Background

Asynchronous Programming

In browsers, Javascript and the web page's rendering engine share a single thread. The result of this is that only one thing can happen at a time. If a database query would be performed *synchronously*, like in many other programming environments like Java and PHP the browser would freeze from the moment the query was issued until the results came back. Therefore, many APIs in Javascript are defined as *asynchronous* APIs, which mean that they do not block when an "expensive" computation is performed, but instead provide the call with a function that will be invoked once the result is known. In the meantime, the browser can perform other duties.
(Source: http://persistencejs.org/async)

The big thing to be aware of is that much of persistence.js is **async** and you need to handle that. So you want code like:

```
persistence.schemaSync( function( err, tx ) {
    if ( err ){
        console.log( "schemaSync: err:", err );
    }
    else{
        var item = new Item();
        item.ItemId = "123";
        item.Name = "Test";
        persistence.add(item);
        persistence.flush( function( err ) {
            if ( err ){
                console.log( "flush err:", err );
            }
            else{
                //... ok
            }
        });

    }
});
```

NOTE: This snippet uses the code at: https://github.com/zefhemel/persistencejs/tree/new

Resources

Ext & ExtJS

- http://prezi.com/odwelgo7wdue/t3con11-sfo-typo3-and-extdirect/
- http://prezi.com/akbjh19yylqy/data-modeling-and-extdirect/

Node.JS

- NodeManual.org
- http://nodebits.org/
- http://howtonode.org/
- http://www.w3resource.com/node.js/installing-node.js-windows-and-linux.php
- http://ninjadeveloper.net/blog/2011/12/12/the-road-to-node-setting-up-node-js-on-bluehost/
- http://raquelvelez.com/blog/2012/02/basic-functionality-complete/
- http://www.nodebeginner.org/
- http://net.tutsplus.com/tutorials/javascript-ajax/this-time-youll-learn-node-js/
- http://net.tutsplus.com/tutorials/javascript-ajax/node-js-step-by-step-blogging-application/
- http://www.ziggytech.net/technology/web-development/experiences-with-nodejs-researching-node/
- http://www.ziggytech.net/technology/web-development/experiences-with-node-js-porting-a-restful-service-written-in-java/
- http://www.ziggytech.net/technology/web-development/experiences-with-node-js-final-thoughts/

NVM

- http://www.backdrifter.com/2011/02/18/using-nvm-and-npm-to-manage-node-js/#more-396

NPM

- http://blog.nodejitsu.com/npm-cheatsheet
- http://www.backdrifter.com/2011/02/18/using-nvm-and-npm-to-manage-node-js/#more-396

Jade

- http://nodejsrocks.blogspot.com/2012/04/need-reasons-to-love-jade-template.html
- http://html2jade.aaron-powell.com/

Connect

- http://project70.com/nodejs/understanding-connect-and-middleware/
- http://stephensugden.com/middleware_guide/
- http://howtonode.org/connect-it
- https://github.com/senchalabs/connect

Express.JS

- http://expressjs.com/guide.html#Middleware
- http://www.hacksparrow.com/express-js-tutorial.html
- http://www.hacksparrow.com/running-express-js-in-production-mode.html
- https://github.com/visionmedia/express/wiki/New-features-in-3.x

Express-Resource
- https://github.com/visionmedia/express-resource

Locomotive
- http://www.backdrifter.com/
- http://locomotivejs.org/
- http://jaredhanson.net/
- http://www.youtube.com/jaredhanson
- http://raquelvelez.com/blog/2012/03/mvc-ftw/

N-Ext
-

Sequelize
-

JSormDB
- http://jsorm.com/wiki/Jsormdb

Persistence.JS --- NO LONGER REQUIRED, WE USE JSormDB
- http://zef.me/2774/persistence-js-an-asynchronous-javascript-orm-for-html5gears
- http://zef.me/tag/persistence-js
- http://jacobmumm.com/2011/09/20/asynchronous-javascript-with-persistencejs/
- http://jacobmumm.com/demos/persistencetask/index.html
- http://persistencejs.org/objects?do=index

Persist.JS--- NO LONGER REQUIRED, WE USE JSormDB
- https://github.com/jeremydurham/persist-js

MySQL
- http://nodejsrocks.blogspot.com/2012/04/nodejs-expressjs-mysql.html

MySQLDumper
- http://sourceforge.net/projects/mysqldumper/
- http://www.mysqldumper.net/