CORE - A PRIMER

The core system explained.

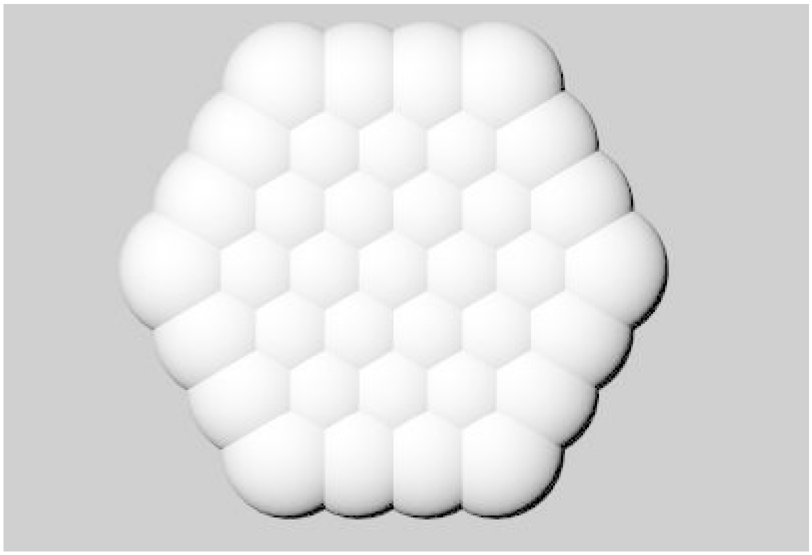


image: Circle of Life
http://i98.photobucket.com/albums/l280/kachina2012/circleoflife3d_1.jpg

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();
    }
});
```

1. Persistence

Persistence[2] 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.

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

assets					
	templates				
		core			
			javascripts		
				persistence	
					<the persistence javascript files>

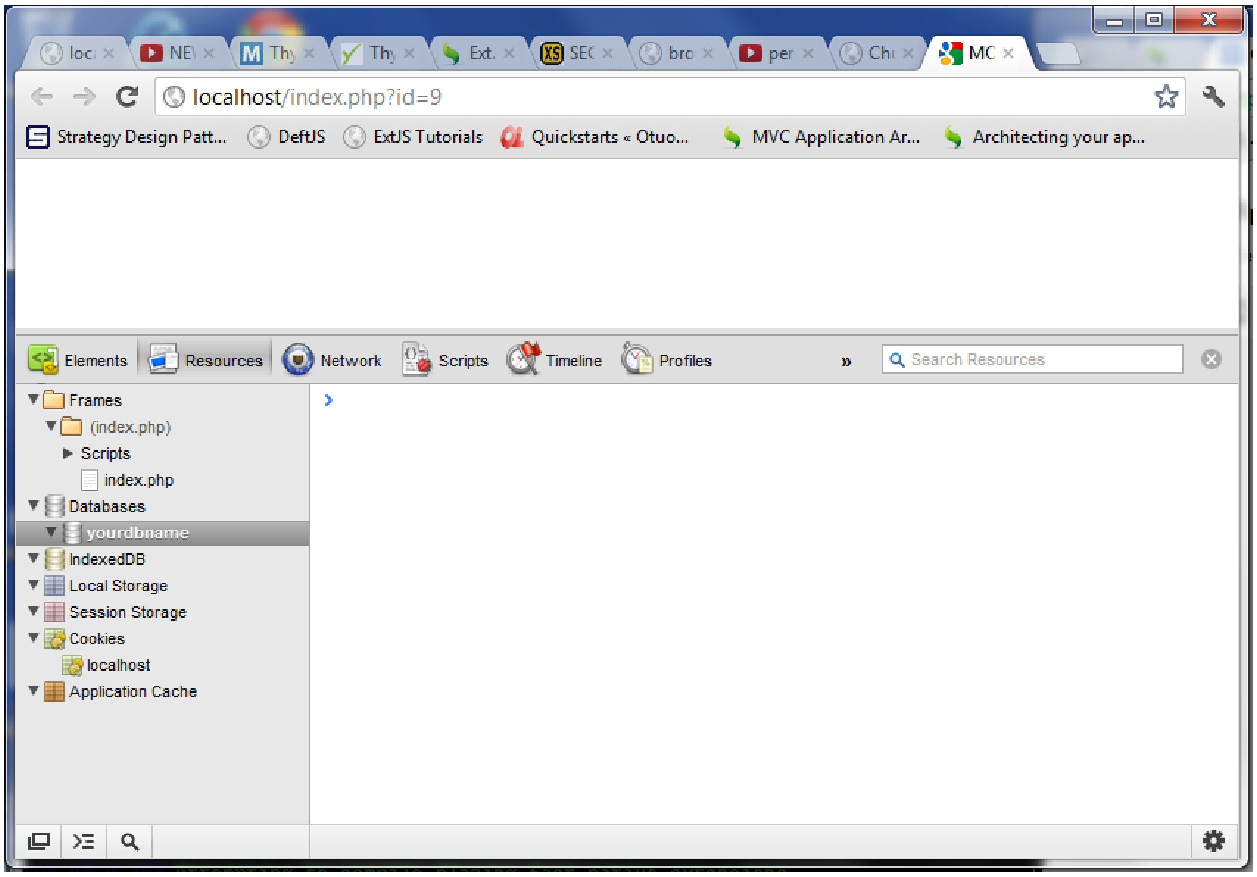
Addition to all web pages that use persistence:

```
<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>
<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:

[2] http://persistencejs.org



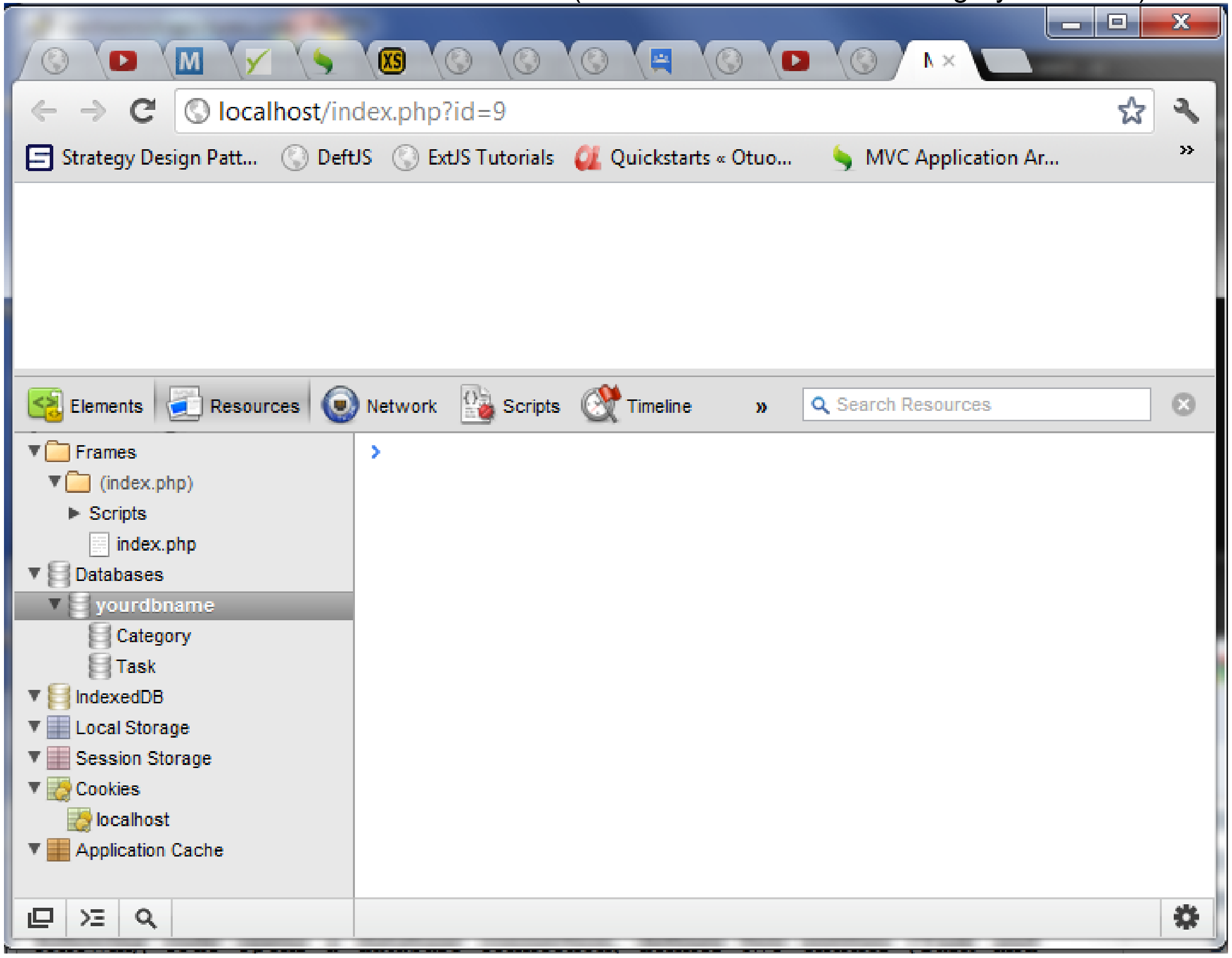
We can add tables and relationships to the database on the client like so:

```
<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()"

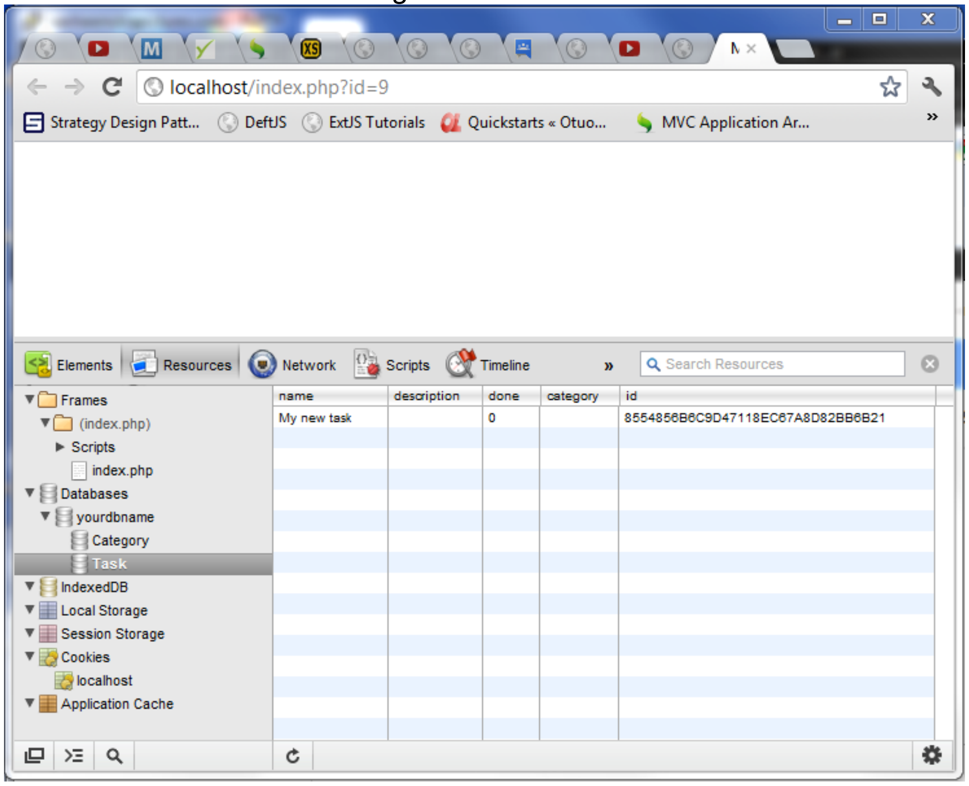
One-to-many and many-to-many relationships can be used as collections (note: below is javascript in the web page):

```
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 !!



name	description	done	category	id
My new task		0		8554856B6C9D47118EC67A8D82BB6B21

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:

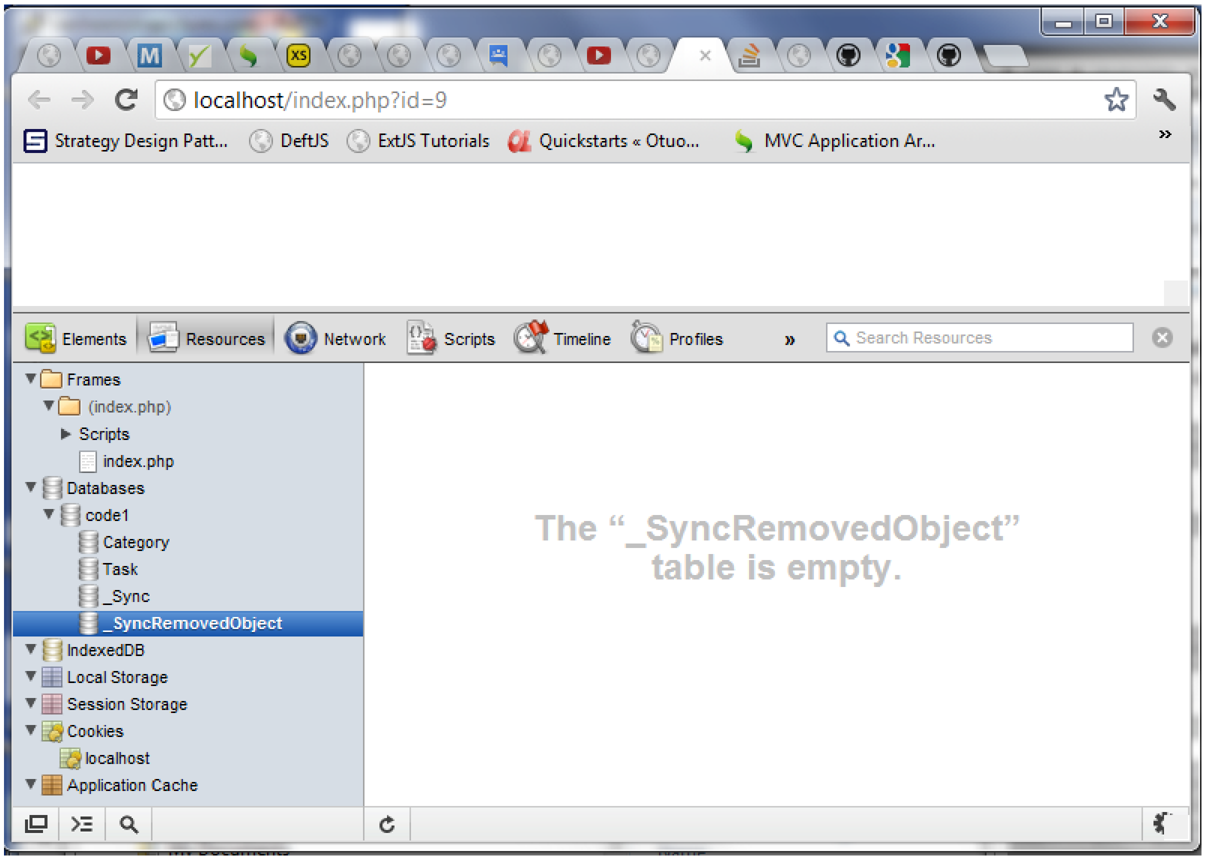
1. persistence.sync.preferLocalConflictHandler, which in case of a data conflict will always pick the local changes.
2. persistence.sync.preferRemoteConflictHandler, which in case of a data conflict will always pick the remote changes.

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:

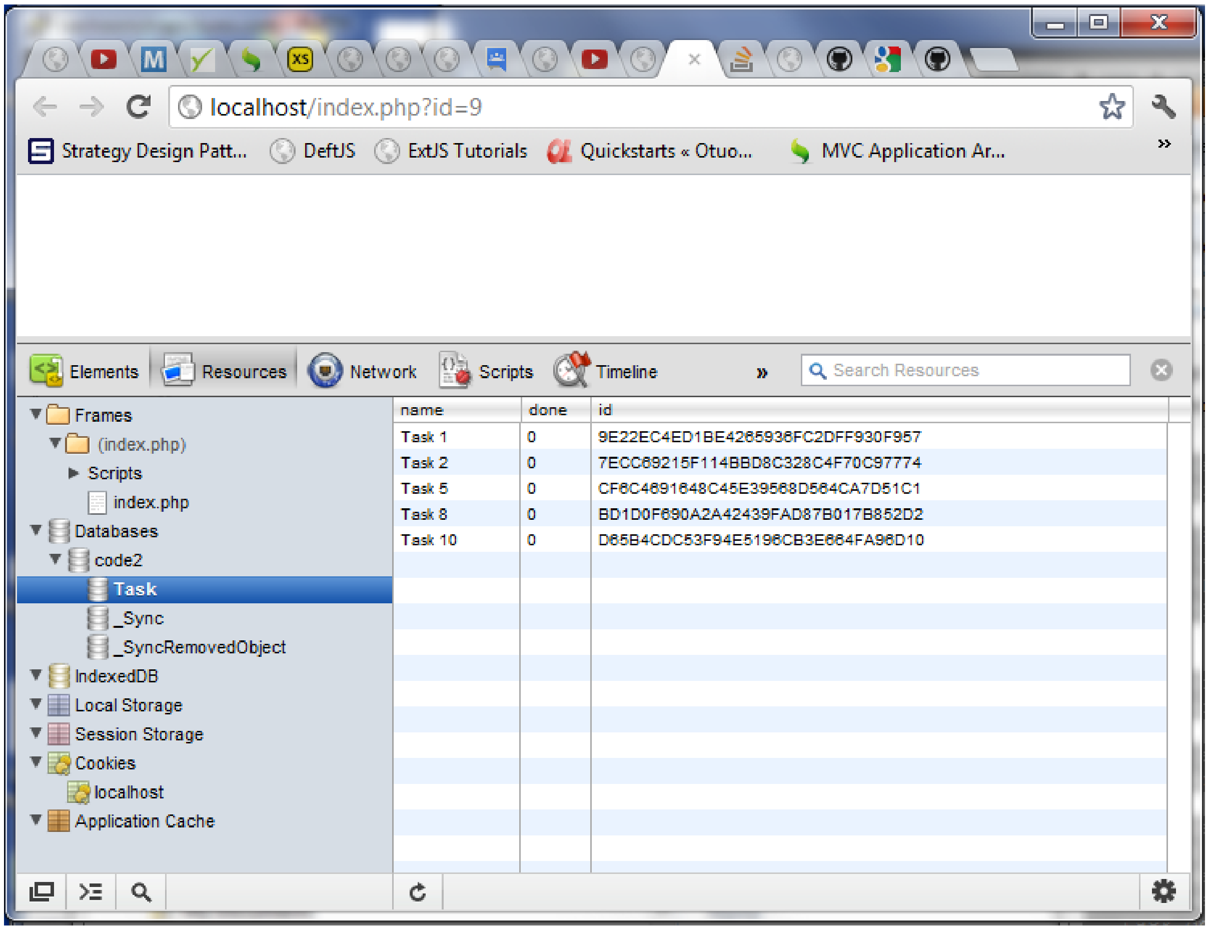
```
    DROP TABLE IF EXISTS `Task` null
    DROP TABLE IF EXISTS `_SyncRemovedObject` null
    DROP TABLE IF EXISTS `_Sync` null
    CREATE TABLE IF NOT EXISTS `Task` (`name` TEXT, `done` INT, `id` VARCHAR(32) PRIMARY
    KEY) null
    CREATE TABLE IF NOT EXISTS `_SyncRemovedObject` (`entity` VARCHAR(255), `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’:



name	done	id
Task 1	0	9E22EC4ED1BE4265936FC2DFF930F957
Task 2	0	7ECC69215F114BBD8C328C4F70C97774
Task 5	0	CF6C4691648C45E39568D564CA7D51C1
Task 8	0	BD1D0F690A2A42439FAD87B017B852D2
Task 10	0	D65B4CDC53F94E5196CB3E664FA96D10

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[3] content management system.

2. Request Handling (NodeJS)

Introduction

Follows the NodeJS[4] 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):

[3] http://www.modxcms.com
[4] 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 >		
	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[5] for connecting to the MySQL database. Use Node Package Manager (NPM)[6] 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
```

For node-mysql this should list:

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

[5] https://github.com/felixge/node-mysql
[6] http://npmjs.org

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)

Persistence *A storage solution for Node.JS*

Install **Persistence**[7] 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[8] 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.

[7] http://persistencejs.org
[8] https://github.com/orlandov/node-sqlite

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`
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[9] 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**[10] 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.

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

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
```

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[11] 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
```

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.

***NowJS**: Middleware for Real-Time connection between Client and Server*

NowJS[12] 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:

[11] http://expressjs.com
[12] http://nowjs.com

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
```

3. Database(s)

Uses MySQL[13] relational database.

Uses FileMaker Pro[14] relational database.

[13] http://www.mysql.com
[14] 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

NodeJS

- 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/

NPM

- http://blog.nodejitsu.com/npm-cheatsheet

Connect

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

ExpressJS

- http://expressjs.com/guide.html#Middleware

PersistenceJS

- 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