Table of Contents

1. Introduction

OneDB is a software that puts together all the logic in order to enable site editors, publishers, and programmers in order to easily administrate and build websites.
Documentation is split into two sections, one for User Guide, and the other for Programmer Guide.

2. Backend Guide

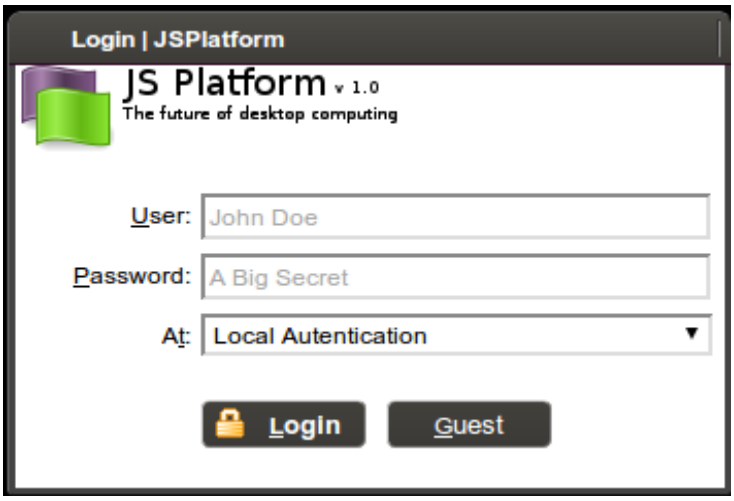
It is very easy to work with OneDB as a user. It's interface is very intuitive, and even if it seems a little bit complicated on first usage, all options are logically grouped for allowing user to work very comfortable.

2.1. Starting the application

2.1.1. LogIn

OneDB backend runs smoothly only on webkit-based browsers, so it was designed to work on Google Chrome (Safari might work too but not tested) web browser. Other browsers might not function properly, so please open the link to OneDB backend

What you should do first it would be to Log-In to OneDB backend:



In this box you should input the **Username** and **Password** that your web-admin administrator provided.

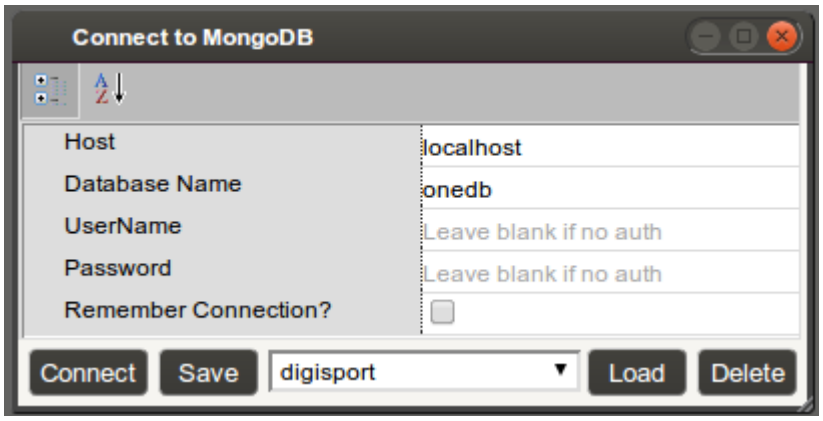
Depending on your company infrastructure, you might have to choose the authentication method, from the At: drop down.

After you input your user and password, click the Login button.

The JSPlatform log-In screen

2.1.2. Starting the application and connecting to database

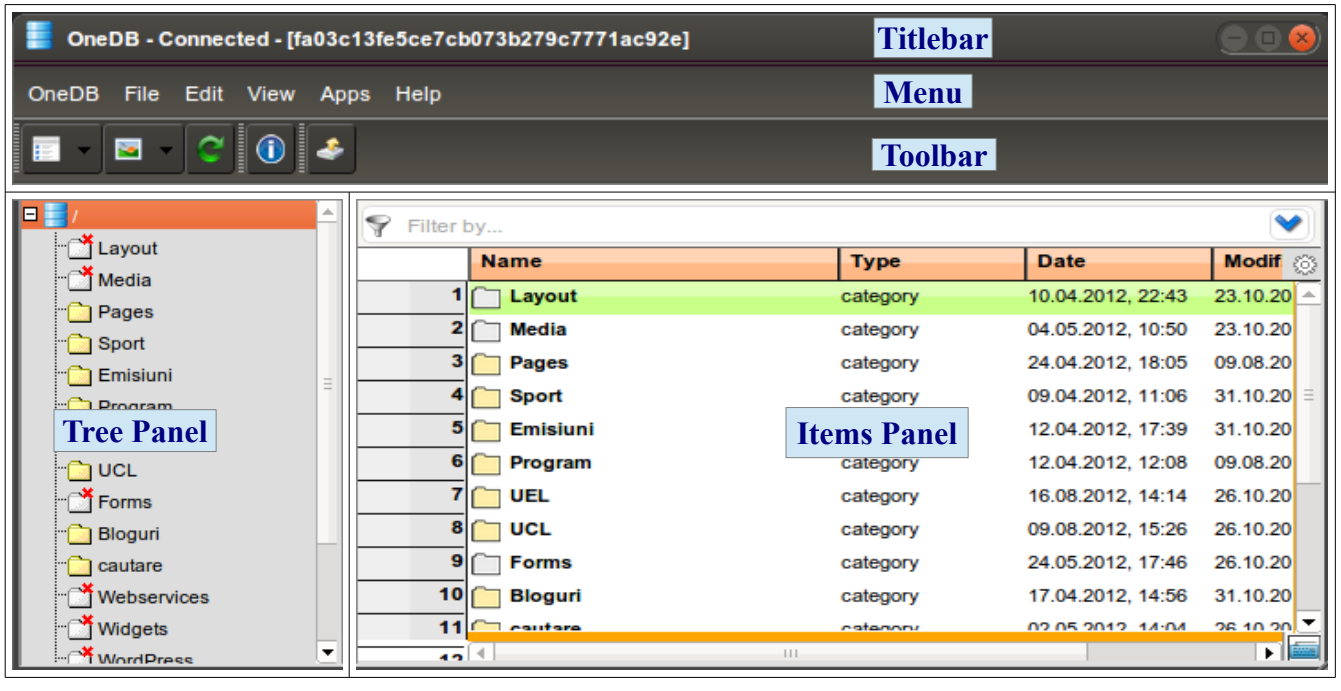
After you Log-In to JSPlatform screen, you will see on your web-Desktop the OneDB application Icon. After you double-click it, a dialog prompting you to input the name of the MongoDB database, a user-name and a password:



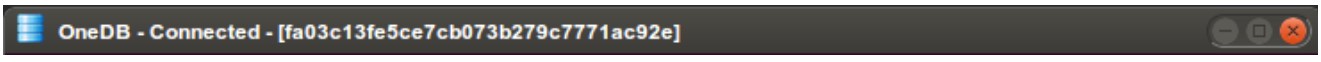
OneDB web-Desktop icon | OneDB connection dialog

Your administrator will provide you with the values you should enter in the "**Host**", **"Database Name"**, "**UserName"** and "**Password**". After you complete those fields, you can optionally save those fields into your server session, by clicking the "Save" button. After you provide a connection name, your connection will be remembered next time you open your application. All you have to do is to click the "Load" button, and the fields Host, Database Name, UserName and Password will be populated automatically with values you previously saved.

2.1.3. Application window



2.1.3.1. Titlebar



The titlebar is similar like any regular window titlebar. You can drag the window up / down / left / right

with the help of the mouse, you can minimize, maximize, restore or close the window with the help of the upper-right buttons. Nothing more to be said here.

2.1.3.2. Menu

Here we have the application menu. Items are grouped logically, in order to increase the productivity and to make the interface more intuitive.

Menu Options

Menu	Submenu	Shortcut Key	What it does
OneDB	Opens the submenu **OneDB**. Here you can find applications specific to database filesystem and settings.		
	Connect to Server	-	Opens the dialog from where you can connect to a MongoDB Database
	Disconnect from Server	-	Disconnect from currently connected MongoDB Database
	Database administration	-	Opens a dialog from where you can see statistics and do administrative related stuffs in the MongoDB database
	Prepare server for first usage	-	Prepares OneDB environment. This option should be used only by administrators
	MongoDB console	-	Opens a console from where you can issue commands and view results from the MongoDB database.
	MongoSphinx	-	Opens the dialog from where administrators can setup a MongoSphinx server connection, which helps OneDB to do text searches across documents.
	Registry Editor	-	Opens a dialog from where you can modify or alter OneDB registry settings, which impacts the way this software behaves.
	Plugin Manager	-	Opens the Plugin Manager, an application from where you can load and unload plugins.
	Exit	-	Exits software
File	Opens the submenu **File**. Here you can find file operations commands that you can do on articles and categories.		
	New … Category Folder	F7	Creates a category which will be a child of the focused category from the Tree Panel
	New … Search Category Folder	-	Creates a search category which will be a child of the focused category from the Tree Panel
	New … JSON Webservice Category Folder	-	Creates a JSON webservice category, which will be a child of the focused category from the Tree Panel
	New … Article Document	Ctrl + Alt + N	Creates a new Article Document in the current location.
	New … Widget Extension	-	Creates a new Widget in the current location
	New … Layout Selection	-	Creates a new Layout in the current location
	New … <other item >	-	Creates other item type, implemented by a plugin.
	Upload File(s) Here	Ctrl + U	Opens a dialog from where you can upload files into database

	Open in JSON Inspector	-	Opens a dialog which display the selected items from the Items Panel in JSON format.
	Properties	Alt + Enter	Opens the properties dialog for selected items from the Items Panel.
Edit	Opens the submenu **Edit**. Here you can do editing operations with Categories or Articles from the Items Panel		
	Search	Ctrl + F	If a MongoSphinx connection is made by a platform administrator, it will open the Search Dialog, from where you can do text-searches in database
	Rename	F2	Renames current selected item from the Items Panel
	Copy	Ctrl + Shift + C	Copies current selected items in the Items Panel. You can paste those items in other location by using the Edit / Paste command.
	Cut	Ctrl + Shift + X	Cuts current selected items in the Items Panel.
	Paste	Ctrl + Shift + V	Paste previously copied / cutted items in current location
	Delete	F8	Deletes current selected items from the Items Panel.
View	Opens the **View** submenu. From here you can do tunings on how to display items in OneDB, and other options that can be grouped in this category.		
	Icons … Tinny	-	Display the items from the Items Panel as icons of 16x16 pixels
	Icons … Large	-	Display the items from the Items Panel as icons of 48x48 pixels
	Toolbar	F10	Toggles toolbar visibility
	Refresh	Ctrl + Enter	Reload items from the Items Panel
	Show hidden objects	-	Toggles weather hidden objects will be displayed or not
	View Thumbnails	-	Tries to load a thumbnail with a preview image instead of generic item icon in the Items Panel
	Load Maximum … x articles	-	Loads maximum X articles in the Items Panel
	Plugins Panel	-	Splits the Items panel so that it make room for a secret panel, where plugins of the software are placing aditional interface components.
Apps	Opens the **Apps** submenu. Here you have a list of handful applications that you can use		
	Form Handlers	-	Opens an application from where programmers can easily design forms for your website.
	User Accounts	-	Opens an application from where programmers can easily implement user accounts for your website.
	Text Editor	-	Opens a text editor with full syntax highlighting.
Help	Opens the **Help** menu.		

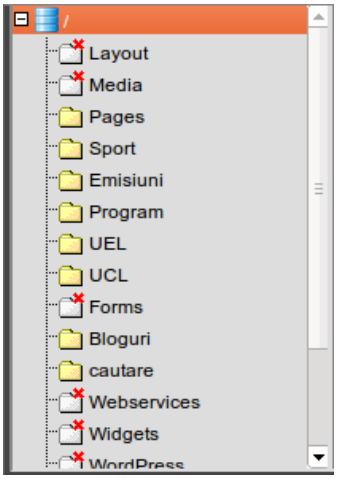
2.1.3.3. Toolbar

The toolbar is a region where we placed the most common tasks in the application. By right-clicking the toolbar anywhere, you can see more options, like for example customize your icon size, or show labels under the icons.

Toolbar buttons list

Button	Label	What it does
	Large Icons	Displays the items from the Items panel as 48x48 icons view
	Small Icons	Displays the items from the Items panel as 16x16 icons view
	Details	Display the items from the Items panel as a grid (Default View)
	Show Thumbnails	Tries to show thumbnails instead of generic item icons in the Items Panel
	Show Icons	Shows generic item icons instead of displaying thumbnails, making the interface to load faster
	Refresh	Refreshes the contents of the current Tree Panel location
	View as JSON	Opens JSON Inspector, an application which displays the selected items properties from the Items Panel
	Upload Files Here	Opens an upload files dialog box, alowing the user to upload files from it's computer or from a list of url's.

2.1.3.4. Tree Panel



The tree panel is useful to show you the hierarchy of all categories from your database.
Categories that are displayed with multiple icons:

-  - Standard categories
-  - WebService categories
-  - Search categories

Categories that are grayed are offline, and categories that are colored are online.

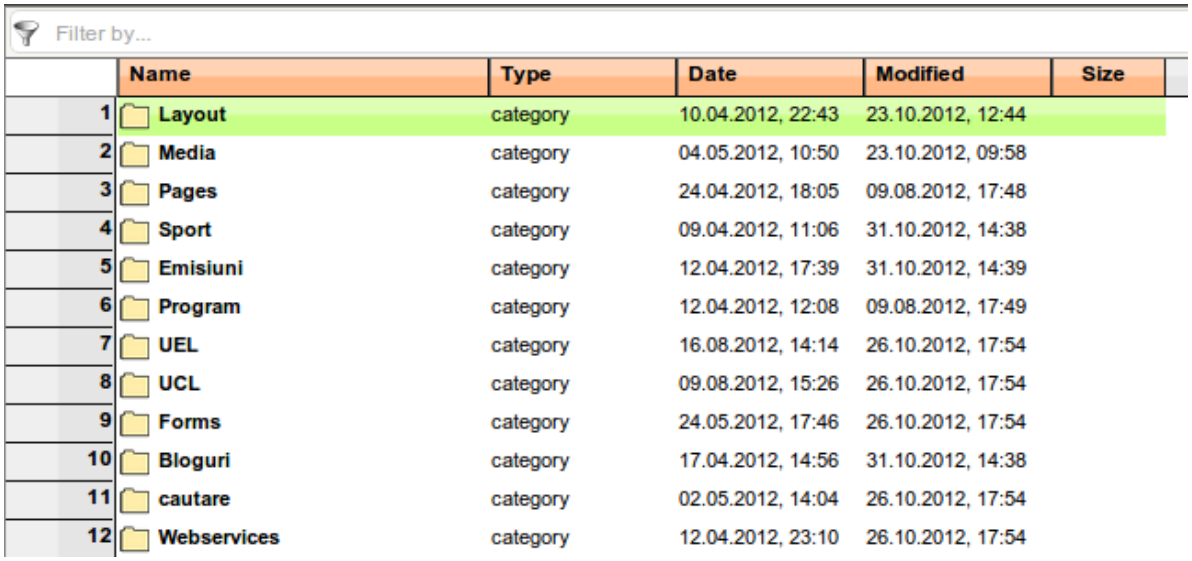
The tree panel is drag and drop capable, so you can drag items inside from the Items Panel, or sort categories inside the Tree Panel with a simple drag and drop operation.

By pressing **F2** on a focused category from the Tree Panel, you can easily rename that category.

2.1.3.5. Items Panel

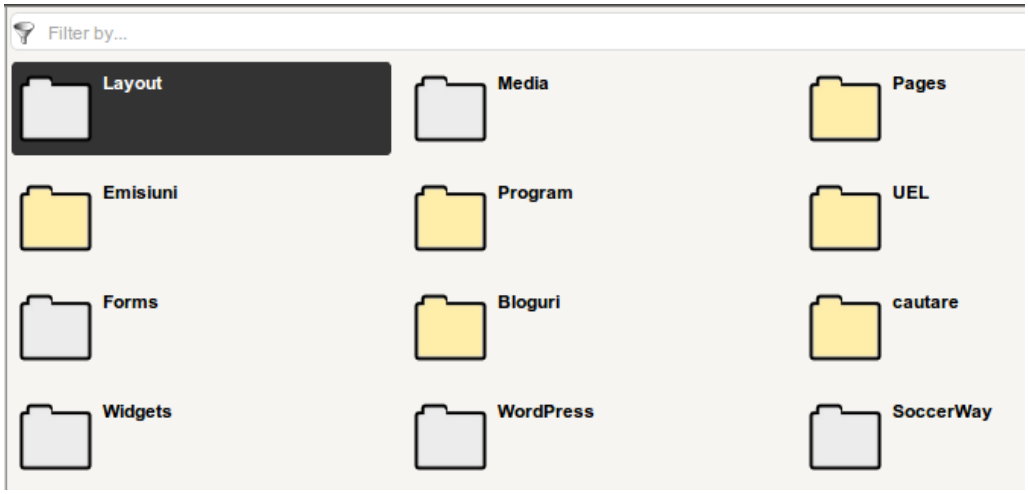
Depending on what view settings you made, the items panel can look in 3 modes:

Default view: Detailed



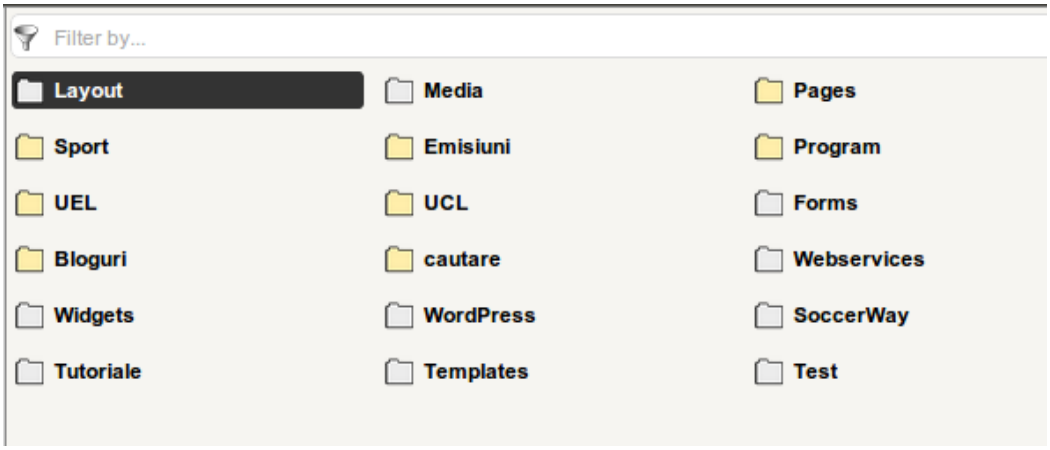
This view is the most flexible one. It supports multiple selections, sorting on custom fields, column resizing, etc.

The large Icons View



While the detailed items view might show too much information, a user can choose to see the items as icons. This view is showing only the name of the items. Grayed icons means that the item is offline.

The Small Icons View



This view is rendering icons in a more compact mode, showing 16x16 icons.

2.1.3.5.1. Basic Operations you can do in the View Panel

Operation	Effect
Double-click an item	Opens item. If item is a category, it will navigate into that category.
Hit Enter on an item	Opens item. If item is a category, it will navigate into that category.
F2 on a focused item	Renames Item
F8 on a selection of items	Deletes selected items

2.1.3.5.2. Other operations that you can do in the backend interface

2.1.3.5.2.1. Drag and drop

You can drag a selection of items from the Items Panel, and drop them into the Tree Panel.

2.2. Categories

2.2.1. JSON Webservice categories

2.2.2. Search categories

2.3. Files and operations you can do with files

2.3.1. Uploading Files in OneDB

You can upload files in OneDB in two ways:

- From your computer
- From an URL (list)

To start uploading files in the current Items Panel, all you have to do is to click the **File … Upload files here**, or click the  icon from the toolbar, or hit **Ctrl + U while the items panel is focused**.

A dialog window will open, and depending weather you want to upload files from your computer or to upload files from an URL list, you can pick the corresponding tab (From Your Computer or From URL List) into that dialog.

Tip: You can do drag-and-drop of files from your desktop into the tab “From Your Computer”.

2.3.2. Main types of files

Although any type of files can be uploaded into OneDB database, some files are treated specially into the platform:

File Type	Extension	Supports Viewing Inline	Supports Editing Inline	Other features
Images	*.jpg, *.jpeg, *.png, *.gif	Yes	Yes	Dynamic resizing
Video files	*.mp4, *.flv	Yes	No	Transcoding Video snapshot
Video files	*.avi, *.mpg, *.mkv, other extensions	Maybe, depending on storage engine implementation	No	Transcoding might be supported, Video snapshot
Text files	*.txt	Yes	Yes	
Cascading style sheets	*.css	Yes	Yes	
Hyper Text Markup Language Files	*.htm, *.html	Yes	Yes	

JavaScript Files	*.js	Yes	Yes
PHP Files	*.php	Yes	Yes

2.3.3. File Storage Engines

OneDB is supporting multiple file storage engines. Depending on your network configuration, currently it can store files either directly into MongoDB database, via the GridFS storage engine, either it can store files into a cloud system.

There is also a built-in plugin, called "storage", which can help you to migrate files from one storage to another.

Default OneDB file storage engine is into MongoDB gridFS, but this behaviour might be changed later, depending on what storage engines will be implemented in the future.

2.3.3.1. Migrating files from one storage type to another

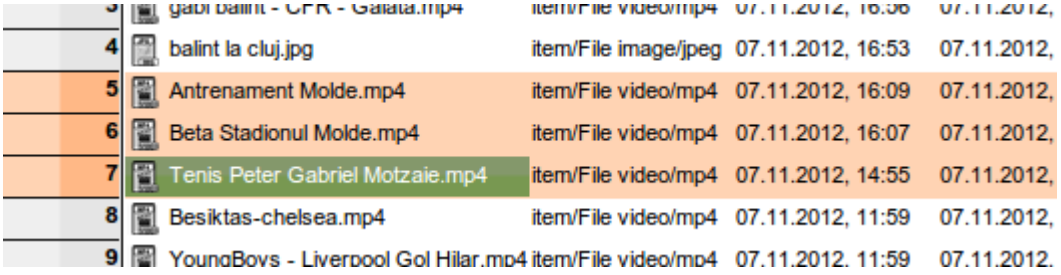
By loading the plugin "`%plugins%/storage`", you can migrate a file or a selection of files, in batch mode, from a storage-type to another.

Please consult the section "Plugins" in order to learn how to load a plugin on demand.

2.3.4. GeoBlocking

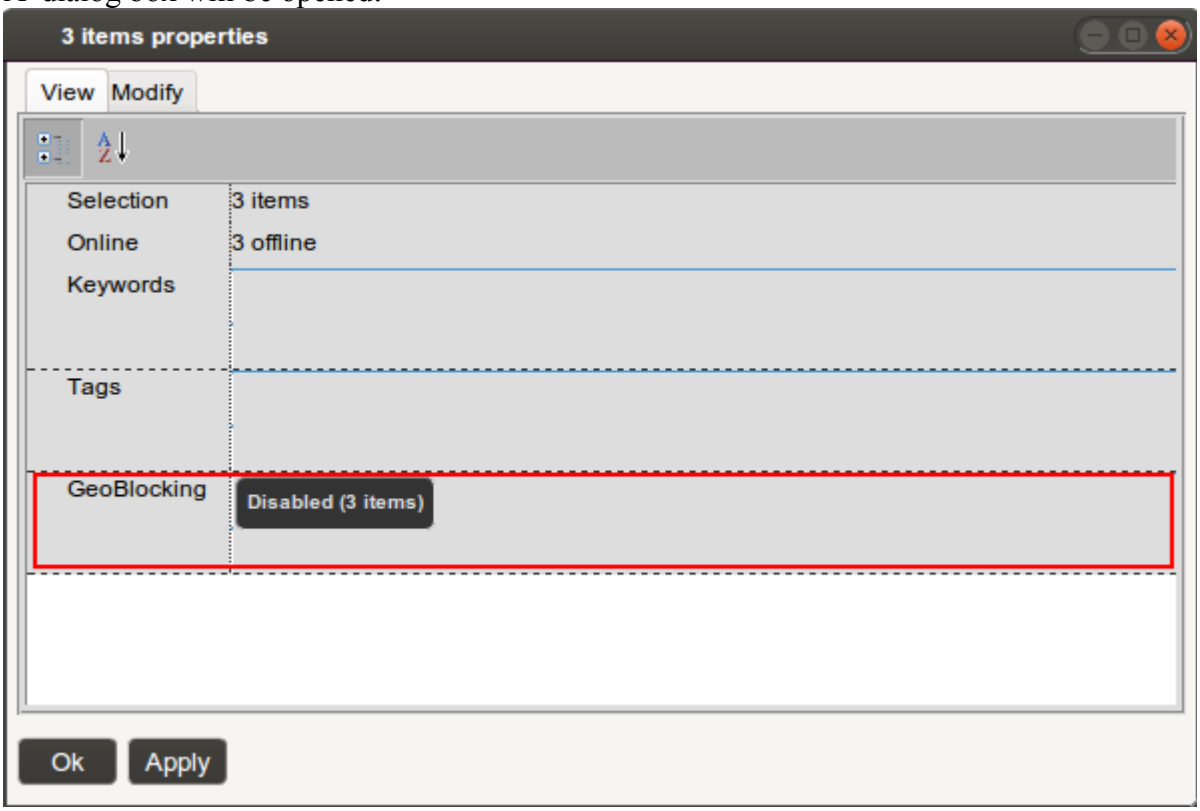
Definition: GeoBlocking is a feature that is allowing files to be served only to clients from specific countries. The country from which the file is accessed by, is determined by client's **ip** address.

2.3.4.1. GeoBlocking a file or a list of files

1	Select a file or a group of files in the Items Panel:  **TIP:** You can select multiple file by clicking on the numbered edge while holding the Ctrl + Shift keys.
2	Hit "Ctrl + Enter" or use the program menu: File / Properties

A dialog box will be opened:

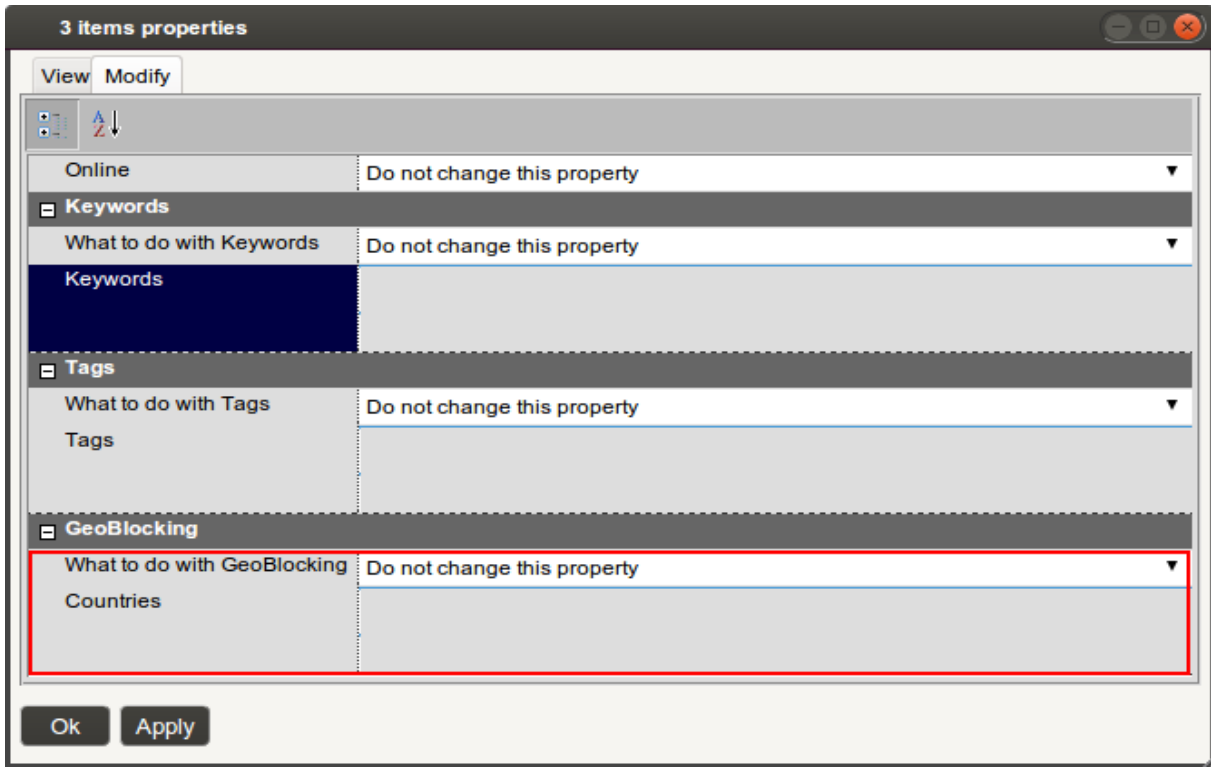
3



You can see if files are GeoBlocked by inspecting the “GeoBlocking” section in the “View” Tab.

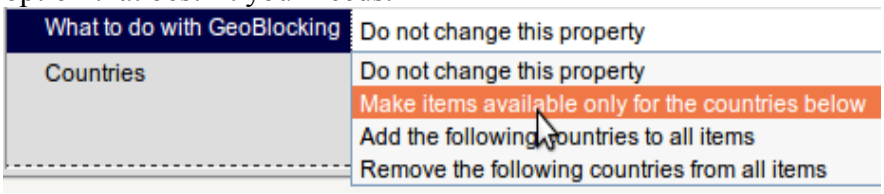
If you want to modify the geoBlocking setting, click the “Modify” tab:

4

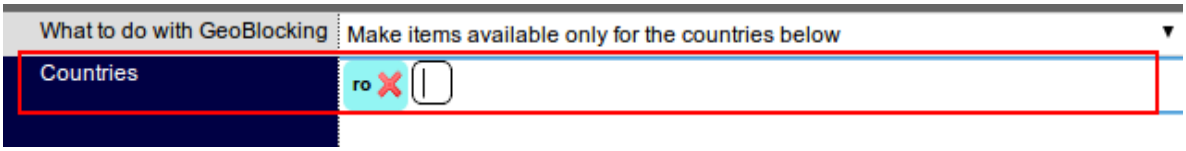


In the dropdown: What to do with GeoBlockging, depending on what you want, select an option that best fit your needs:

5



Input or set the country codes in the “Countries” field:



6 **Important:** You must input the country code in 2 characters format.

Example of country codes:

`uk` United Kingdom (England)
`ro` Romania
`it` Italy
`es` Spain

7 **Click on Ok or Apply**

2.3.4.2. Setting up GeoBlocking

In order to properly setup GeoBlocking, you should meet the following prerequisites:

- Decide what ip2country webservice provider you would use

  OneDB has a built-in ip2country provider, which is provided by it's internals. However, if your assets are accessed frequently and most of them are geo-blocked, you might want to implement an ip2country webservice in your local network, so that you ensure a minimum latency to your files.

  The ip2country webservice should be implemented as a global function:

```
function Any_Function_Name ( <string> $ip ) { … }
      @return: {
            <str[2] country code lowercased> if ip is a wan IP
            <ascii “-” sign> on error or non-country ip's (e.g. 127.0.0.1)
      }
```

  and loaded into OneDB as a database plugin.

- If you decide to use your own ip2country webservice, you should make that plugin database loadable, and specify the name of your ip2coutry function into OneDB Registry, under the key name: “OneDB.GeoBlocking.FunctionName”

2.4. Documents

2.4.1. Creating a new document

2.4.2. The “Document Editor” application

2.5. Widgets and views

2.5.1. Creating and running a Widget

2.5.2. Assigning a widget as a view for a category or an article

2.6. Json Inspector

2.7. Forms

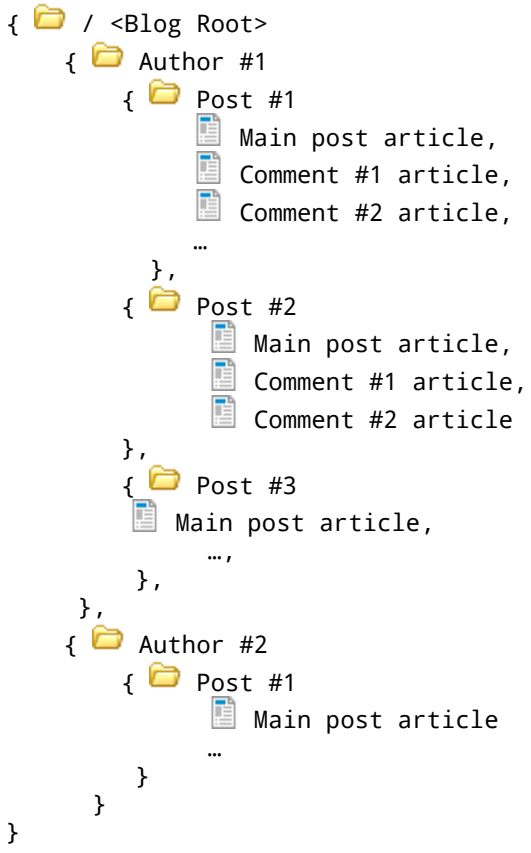
2.7.1. Setting up a form

2.7.2. Creating a form view and viewing results

3. Programmer's guide

3.1. General overview

By analyzing sites in the past, we reached the conclusion that 99% of those sites can be represented in a filesystem-way. For example, a Blog site can be represented like this:



So, in this structure, we can observe the following:

- Items that have the icon can be called "**Categories**"
- Items that have the icon can be called "**Articles**"
- **All items have a unique "name" inside their parent namespace.** For example, inside a category we cannot have two items called "Comment #2 article".
- All items can be accessed through a **path structure**, for example when we're referring to the second post of the Author #1 – which is a category – we can say: "**/Author #1/Post #2/**", and when we're referring to the comment #2 from that post, we can address it as: "**/Author #1/Post #2/Comment #2 article**"
- **Paths that are pointing to a categories are terminated all the time with a "/" character** (VERY IMPORTANT!), and **paths that are pointing to articles** are never terminated with a "/" character. This is very important, and we can quickly figure that "/path/to/foo/" is pointing to a category called "foo", which is a child category of "to", which is a child category of "path", and a "/path/to/bar" is pointing to an article called "bar", which is a child of a category called "to", which is a child category of "path", which of course, is a child of the root category.

Now that we had a little chat about what is a "**Category**" (in the shortest terms: *a category is something that can hold items inside, and can be seen as a folder*), and what is an "**Article**" (in the shortest terms: *an article is something that holds information, and can be seen as a file*), and what is a "*Path*" (in the shortest terms: *a path is a property of each item, weather it's a category or an article, which makes it's unique and accessible*), we can have an overview of the storage concept of OneDB. From a programmer's point of view, OneDB is a database filesystem, which can hold various items called articles with various property in a grouped hierarchy, called categories. Categories can contain

sub-categories, and articles, and so on.

An article can be seen as a generic naming of a “File”, “Document”, “Picture”, “Video”, or any other structure that you can imagine and that can be indivisible.

A category can be seen as a generic naming of a “Collection”, “Folder”, “Group of objects”, etc.

So, if OneDB is a database filesystem (which at this point might be nothing worthy than a hard disk partition containing folders and files), what's the thing that makes it so special?

Well, let me tell you what's that thing. That thing is called “querying”. You can query for categories and articles in OneDB based on items attributes, obtain a dataset of objects, manipulate them, apply a filter for example, sor them again, and dump their data (optionally through a templating system) into a browser window, and – voila! That is a website :)

So for being able to successfully use and implement OneDB as a solution for any-website, you should start viewing any data structure of that any-website as a filesystem. You can be as creative as you want, of course, and that is the beauty of OneDB, it let you be as flexible as you want.

3.2. Main classes overview

These are the main classes that are founding OneDB.

`OneDB`
`OneDB_MongoObject`
`OneDB_Article`
`OneDB_Category`
`OneDB_ResultsNavigator`

Now, transposing those classes name to the small talk (I hope) from the sub-chapter 3.1., here is the deal:

OneDB is the main class. It instantiates all sub-classes, it implements the filesystem, everything. All classes are created by OneDB.

When we query through filesystem, we're expecting to obtain objects (categories and articles) which are of types: `OneDB_Category`, and `OneDB_Article`. Both `OneDB_Article` and `OneDB_Category` are implementing a more generic object, called `OneDB_MongoObject`.

We can query OneDB, but we always manipulate articles and categories from the query result-sets with the `OneDB_ResultsNavigator`. ResultsNavigator is – as it's name saying – a mechanism we're using to navigate, sort, filter, limit, reverse, etc. the results we're obtaining after we're querying the database filesystem. So, here is a new concept: Navigator.

For example, take a look at this chain commands:

```
$myDB->articles(
      //we're doing a search in database for items of article type
      array(
            'online' => true,
            'owner' => 'bill',
            'keywords' => 'important'
      )
)->filter( function( $article ) {
      //we're applying a filter to our result-set
      return $article->numberOfPosts > 0;
} )->sort( function( $articleA, $articleB ) {
      //we're sorting the result-set based on article's names
      return strcmp( $articleA->name, $articleB->name );
} )->each( function( $article ) {
      //we're dumping some html code to the browser
      echo '<tr><td>'
          . htmlentities( $article->name ) . '</td><td>'
            . htmlentities( $article->owner ) . '</td><td>'
            . '<a href="' . $article->getPath() . '">view item</a>'
            . '</td></tr>';
} );
```

What is that? The part highlighted with the red color is the OneDB class instance. The yellow color is the query part. That part is telling OneDB to fetch all articles from database for example, which are in an online state, have their owner named “bill”, and have a keyword “important” in their keywords list. The blue part is called the navigation part, where we're doing two things (we can do those two things in any order we want, by the way): **1)** we're applying an additional filter on the result set, modifying the result set to contain only the articles who have posts inside, and **2)** we're sorting the results of articles ascending, on their name. After we have data prepared for our output purpose, comes the green part, which is taking each article in the order from it's result-set, and do something with it. In the above code, we're dumping a HTML code representing a table row which has the name and the owner of articles listed.

Of course, the green part could be something for example like:

```
->each( function( $article ) {
      $article->numberOfViews++;
} );
```

or,

```
->each( function( $article ) {
      $article->delete();
} );
```

So, stuff you can do in the red-color and the yellow color part you can do with the:

```
class OneDB,
```

stuff you can do in the blue color part you will do with the:

```
class OneDB_ResultsNavigator
```

and the rest of the stuff, you can do with the:

```
class OneDB_Article
class OneDB_Category
```

3.3. class OneDB

This is the main class of OneDB. In fact, this is the only class that the programmer is explicitly instantiating in it's code.

This class is extended through decorating pattern classes located in `OneDB/plugins/core.OneDB.class/OneDB_plugin_<method_name>.class.php` plugins.

For example, a method called "`foo`" would be dynamically loaded from: `OneDB_plugins/core.OneDB.class/OneDB_plugin_foo.class.php`

constructor	`function __construct (`
	`[ <array> $config ]`
	`) @return <void>`
	
	Class constructor
	
	`$config` – an array in the following format:
	
	`array(`
	`'db.host' => '<path_to_mongodb_server>',`
	`'db.database' => '<database_name>',`
	`'db.user' => '<user_name_for_database>',`
	`'db.pass' => '<password_for_database>',`
	`'cache.enabled' => <boolean>`
	`)`
	
	Tips: `$config` parameter is optional. If not present, OneDB will try to fetch it's configuration in this order, from:
	• Global variable `$_ONEDB_CFG_`
	• `$_SESSION['onedb_connection']`

Plugin Method

```
function articles (
      <array> $query,
      [ <array> $orderBy = NULL, ]
      [ <int>   $limit = NULL ]
) @return <OneDB_ResultsNavigator.Article>
```

Fetches articles from database, according to **$query**, IF **$orderBy** it does an internal MongoDB order, IF **$limit** it applies a limit to the result-set.

$query: An array with query condition. See MongoCollection::find

$orderBy: An array of fields by which to sort. Each element in the array has as key the field name, and as value either 1 for ascending sort, or -1 for descending sort. Each result is first sorted on the first field in the array, then (if it exists) on the second field in the array, etc. This means that the order of the fields in the fields array is important. See MongoCursor::sort for more information

$limit: Limit to maximum $limit number of results

Example

```
$my->articles (
      array(
            '_childOf' => '/foo/bar/',
            'online' => true
      ),
      array(
            'date' => -1,
            'name' => 1
      ),
      3
)[ ➜ … ]

// Note: the _childOf clause is a magic query field, please
// read the section “Magic Query Fields” first
```

Plugin Method

```
function categories (
      <array> $query,
      [ <array> $orderBy = NULL ]
) @return <new OneDB_ResultsNavigator.Category>
```

Fetches categories from database, according to **$query**, IF **$orderBy** it does an internal MongoDB order,

$query: An array with query condition. See **MongoCollection::find**

$orderBy: An array of fields by which to sort. Each element in the array has as key the field name, and as value either 1 for ascending sort, or -1 for descending sort. Each result is first sorted on the first field in the array, then (if it exists) on the second field in the array, etc. This means that the order of the fields in the fields array is important. See **MongoCursor::sort** for more information

Example:

```
$onedb->categories(
      array(
            '_childOf' => '/foo/bar/',
            'online' => true
      ),
      array(
            'name' => 1,
            'date' => -1
      )
)[ →… ]

// Note: the _childOf clause is a magic query field, please
// read the section “Magic Query Fields” first
```

Magic query fields

Magic query fields for methods articles and categories

Although the parameter $query is used to do a MongoCollection::find() call into database, some fields from it are treated in a different way by OneDB:

Field	Scope Method	Description

`$query['_childOf']`
`@type <string>`
	`->categories($query)`
	
	`->articles($query)`
		Returns all items that are a child of subject

`$query['selector']`
`@type <string>`
	`->categories($query)`	Selector is in format of a CSS rule:
		
		Example:
		
		`$onedb->categories(`
		`  array(`
		`  'selector' => '/path/to/foo/ *'`
		`  ) )`
		
		`// Returns all items that`
		`// are children to`
		`// category /path/to/foo/`
		
		`$my->categories(`
		`  array(`
		`  'selector' => '/path/to/foo/ >`
		`  *'`
		`  ) )`
		
		`// Returns all items that`
		`// are direct children of`
		`// “/path/to/foo/”`
		
		`$my->categories(`
		`  array(`
		`  'selector' => '/path/to/foo/'`
		`  ) )`
		
		`// Return category located`
		`// at “/path/to/foo”`

Plugin method	`function getElementByPath(`
	`<string> $elementPath`
	`) @return <OneDB_Class or OneDB_Article>`
	
	Use this function to obtain a single element, by a given path.
	
	Example:
	
	`echo $my_db->getElementByPath('/site/widgets/banner-968')->run();`

Plugin Method	`function sphinxSearch (`
	`) @return <new OneDB_TextSearch_Server>`
	
	Return a sphinxSearch textSearch instance. For this method to work, you must have the following checklist solved before:
	
	• On the backend, setup a correct path to a MongoSphinx server in OneDB Registry:
	Entry name: sphinxSearch
	Entry value: url to mongoSphinx server.
	
	Example:
	
	`$documentIDList = $onedb`
	`->sphinxSearch()`
	`->search( "free text to search into database", 200 );`
	
	would return a list with maximum 200 articles from database, which match free-text with string "free text to search...". This method is great for implementing a

3.4. class OneDB_MongoObject

A little chat about the **OneDB_MongoObject** first. **OneDB_MongoObject** is the core component between **MongoDB** and **OneDB**. He manages automatic object saving in database, and is adding event-based functionality to objects. Weather we're speaking about a category or an article, we're speaking about a **OneDB_MongoObject**.

When we're setting a property to a **OneDB_MongoObject**, this proerty is automatically saved on object destructor into database, unless we're setting the property **_autoSave** to **FALSE**, in this case the programmer need to issue a manual "**save()**" to the **OneDB_MongoObject**.

OneDB_MongoObject is a great starting point to extend and add features to all objects from **MongoDB** database.

MongoObject supports triggers and events in a similar way with JavaScript eventListeners, and with MySQL trigger functionality.

Constructor

```
function __construct(
     <MongoCollection> &$collection,
     <MongoID|NULL>     $objectID,
     <array>            $firstLoadDataIfObjectIDWasSet = NULL
) @return <void>
```

This is the object constructor. You won't have to implicit call this.

Properties

Property	Type	Description
property “_id”	MongoId DEFAULT NULL	The MongoDB _id of this object. If **NULL**, it means that the object hasn't been saved yet to database.
property “_parent”	MongoId DEFAULT NULL	The _id of container category. If **NULL**, the item is located straight in the root of the tree path.
property “name”	string	The name of the object. This name can contain only: a-z, A-Z, space, _, -, and '.' sign. Names should be unique in combination with _parent.
property “type”	string	The type of the item.
property “online”	bool	Weather this item is into an online state or not.
property “_autoCommit”	bool DEFAULT NULL	If this is manually set to **TRUE**, modifications on properties are automatically saved into database at destruct time. Otherwise, you'll have to invoke a **save()** manually for saving the object into it's collection.
property “_collection”	Mongo Collection	current object MongoCollection where $this object resides into MongoDB database
property “icon”	string DEFAULT NULL	Returns current object icon assigned from backend. If you set in backend an icon to this object, you can retrieve that property from here as a string(e.g. **“onedb/129031890289afadb”**). You can place that into a **** tag in your website, or use the OneDB_picture helper in order to set the size of the icon when dumping data to browser.

	property “keywords”	array DEFAULT NULL	Keywords assigned to this object (if they were set) or NULL otherwise
	property “tags”	array DEFAULT NULL	Tags that were assigned to this object (if they were set) or NULL otherwise
	property “owner”	string DEFAULT NULL	The name of the user that created this object
	property “date”	unix timestamp DEFAULT NULL	The date represented as a Unix_Timestamp when this object was first saved into database
	property “modifier”	string DEFAULT NULL	The name of the user that last saved this object
	property “modified”	unix timestamp DEFAULT NULL	The date represented as Unix_Timestamp when this object was last saved into database
	property “description”	string DEFAULT NULL	A short description of this object (UTF-8 String) which was set through the backend interface

Method

```
function addTrigger(
      <string>   $propertyName,
      <string>   $timing,
      <callable(
         <mixed>$oldValue,
         <mixed>&$newValue,
         <mixed>$self)>
                  $trigger
) @return <void>
```

Adds a trigger to a property. Triger can be executed “before” or “after” the property is set. If trigger throws exeption, the property is not modified.

`$propertyName` – The name of the property who will fire the trigger

`$timing` – When the trigger to be fired – `'before'` or `'after'`

`$callable` – A function that will be called with three parameters:

- `$oldValue`,
- `$newValue`,
- and an instance of `$this` object (`$self` in `$callable` function body)

Example:

```
$this->addTrigger(
      'propertyName',
      'before',
      function( $old, &$new, $self ) {
            echo “Modifying 'propertyName' “
                  . “( old = '$old', new = '$new' ) “
                  . “ of object with _id: $self->_id\n”;
      }
);

$this->propertyName = 23; //The trigger is executed.
// If trigger will modify the $new value, $this->propertyName will
// have that value.
// If trigger will throw an exception, propertyName will remain to
// it's actual value before assigning it the value of 23
```

Method	`function setReadOnly(`
	`  <string> $propertyName`
	`) @return <void>`
	
	Marks a property as Read-Only. Any attempts to write that property later, will generate an Exception
	
	Example:
	
	`$this->foo = 23; // Ok`
	
	`$this->setReadOnly( 'foo' );`
	
	`$this->foo = 24; // Throws exception`

Method	`function _addGetter(`
	`  <string> $propertyName,`
	`  <callable($self)> $func`
	`) @return <void>`
	
	Adds a getter for a custom property. Getters are usefull because their result can be automatically generated, but their values are not stored into database.
	
	Example:
	
	`echo $this->fooBar; // NULL`
	
	`$this->_addGetter('fooBar', function( $self ) {`
	`  return "Object fooBar: $self->_id";`
	`});`
	
	`echo $this->fooBar; //Object fooBar: af223addd23ffd12`

Method	`function save(`
	`) @return <void>`
	
	Manually saves the object in it's MongoCollection

Method	`function deleteProperty(`
	`  <string> $propertyName`
	`) @return <void>`
	
	Deletes a property from object.
	
	Example:
	
	`$this->foo = 45;`
	
	`echo $this->foo; // 45`
	
	`$this->deleteProperty( 'foo' );`
	
	`echo $this->foo; // NULL, because $this->foo became undefined`

Method	`function deleteDependencies(`
	`) @return <void>`
	
	If other objects are stored into MongoDB filesystem, e.g. video-snapshots or other types, this function deletes them from their collection, as they are not needed anymore into database. This function is automatically called by the delete() method

Method	`function delete(`
	`) @return <void>`
	
	Deletes the object from it's MongoDB Collection

Method	`function isChildOf(`
	`  <MongoId|string> $categoryID`
	`) @return <bool>`
	
	If object is a child of a category with MongoDB _id `$categoryID`, this function will return **`TRUE`**, otherwise will return **`FALSE`**
	
	Example:
	
	Assuming that we have a valid path: '`/path/to/foo/`', (we have 3 categories if you observed in this path):
	
	`// TRUE:`
	`$foo->isChildOf( $bar->_id );`
	
	`//TRUE, because NULL is the ID of the root category:`
	`$foo->isChildOf( NULL );`
	
	`// FALSE:`
	`$bar->isChildOf( $foo->_id );`

Magic Method	`function __toString(`
	`) @return <string>`
	
	Returns the object as a string, in JSON format.
	
	Example:
	
	`echo "$foo";`
	
	`// {"_id": …, …}`

Method	`function toArray(`
	`) @return <array>`
	
	Return the object as an array

Method	`function extend(`
	`<array> $object`
	`) @return <void>`
	
	Extends properties of `$this` object.
	
	`$object` is an array, in format:
	
	`array(`
	`'property_name' => <value>,`
	`'property_name' => <value>`
	`…`
	`).`
	
	Example:
	
	`echo $this->a_custom_property; //NULL`
	
	`$this->extend( array(`
	`'a_custom_property' => 23,`
	`'another_property' => 'yes'`
	`) );`
	
	`echo $this->a_custom_property; //23`

Method

```
function import(
      <string> $requiredClassPath
) @return <void>
```

Adds the class `$object` as a decorator class for this object. When calling a method that doesn't exists in `$this` namespace, if that method exists in the `$object` class instance, it will return `call_user_func_array( array( $object, $mehodName ), $arguments )` instead.

Example:

```
// Not working
echo $this->aCustomMethod( 1, 2 );


// WORKING:
// create a Foo.class.php file:
class Foo {
      protected $_self;

      public function __construct( &$thatInstance ) {
            $this->_self = $thatInstance;
      }

      public function aCustomMethod( $a, $b ) {
            return $a + $b;
      }
}

$this->import( 'Foo' );

echo $this->aCustomMethod( 1, 2 ); //returns 3
```

Method

```
function addEventListener(
      <string>           $eventName,
      <callable($self)> $func
) @return <void>
```

Class is emitting events based on different actions. Events attached to an action by calling `addEventListener` method.

`$eventName` parameter can be any string, but some events are predefined in `OneDB_MongoObject`:

`'save'`	event that is emitted before the object is saved
`'delete'`	event that is emitted before the object is deleted

`$callable` is a closure function which will be called with (`$this`) as argument.

Example:

```
$this->addEventListener('save', function( $that ) {
      throw
      new Exception("Object with _id $that->_id cannot be saved!");
});

$this->save(); //throws above exception

$this->addEventListener('my-custom-event', function( $that ) {
      echo "My cyustom event listener function!";
});

$this->on('my-custom-event');
//My custom event listener function attached before
```

Method

```
function on(
      <string> $eventName
) @return <void>
```

Triggers an event with name `$eventName`

Events are added to specific actions with method `addEventListener`

Method	`function views(`
	`) @return <class OneDB_ObjectView($self)>`
	
	Returns the corresponding views class for current OneDB_MongoObject. For more information, consult the Views section of documentation.
	
	Example:
	
	`echo $foo->views()->{"category.index"}->run();`
	
	`// Returns the view called 'index' and type category for`
	`// the foo mongoObject, that will run with the default`
	`// $argument = $foo parameter.`

Method	`function getServer(`
	`) @return <class OneDB>`
	
	Returns OneDB class owner of this object.
	
	Example:
	
	`$my_onedb = $foo->getServer();`

3.5. class OneDB_Category

This is the class that is representing a "category" in OneDB. A category, as we explained into the introduction, is a "container" or a "collection" which can hold articles or other categories.

Categories are of three types in OneDB:

- Category
- Category.JSONWebserviceCategory (decorator) – Category whose items are obtained through a JSON webservice from a remote URL
- Category.SearchCategory (decorator) - Category whose items are obtained through a search.

The dotted notation is coming from the reason that the Category.JSONWebserviceCategory and Category.SearchCategory are in fact categories decorated with plugins called `OneDB_Category_plugin_JSONWebserviceCategory.class.php` and `OneDB_Category_plugin_SearchCategory.class.php`, both decorators located into `plugins/core.OneDB_Category.class` folder of OneDB.

When we're using a "." (dot) inside of a class name notation inside this documentation, we're referring to the base class (part from the left side of the dot) decorated with class from the right of the dot. Example: Foo.Bar is referring to the class Foo, which is decorated with class Bar.

Note: Although we're presenting most common methods and properties, these should be for reference purpose only, because 99.99% of the time we would manage operations from the backend interface.

TIP: Most common methods of this class that a programmer will use are: `getPath()`, `getParent()`, `getChildren()`, and `articles()`.

<table>
<tr><td>Prototype</td><td>

```
class OneDB_Category extends OneDB_MongoObject
```

</td></tr>
<tr><td>Constructor</td><td>

```
function __construct(
      <MongoCollection> $collection,
      <MongoID | NULL>  $objectID,
      <array | NULL>    $firstLoadDataIfObjectIDWasSet
) @return <void>
```

This is the class constructor. Please note that you, as a programmer won't have to instantiate categories, as OneDB is instantiating them during the query process.</td></tr>
<tr><td>Properties</td><td>

Name	Type	Description
All properties of OneDB_MongoObject are present to OneDB_Category, plus these ones:		
`property "isVirtual"`	`Boolean DEFAULT NULL`	Weather this is a virtual category or not. A virtual category could for example receive it's items from a JSON Webservice from your network, or could be a Search category, whose items would be obtained after a search operation in other physical categories from our database.
`+ All properties of OneDB_MongoObject`		

</td></tr>
</table>

Method	`function getPath(`
	`  <boolean> $escape = TRUE`
	`)`
	
	Returns the full category path.
	
	`$escape` → Weather the parts of the path to be escaped with urlencode function (convert spaces to "+" characters for example)
	
	Example:
	
	Given the path "/foo object/bar/", and assuming that $bar is an already instantiated instantiated OneDB_Category object:
	
	`echo $bar->getPath()`
	`// "/foo+object/bar/"`
	
	`echo $bar->getPath( FALSE )`
	`// "/foo object/bar/"`

Method	`function getParent(`
	`) @return <OneDB_Category>`
	
	Returns the parent category that is containing this category
	
	Example:
	
	Given the path "`/foo object/bar/`", and assuming that `$bar` is an already instantiated instantiated `OneDB_Category` object:
	
	`$bar = $foo->getParent();`
	`echo $foo->name;`
	`// "foo object"`
	`if ($foo->isChildOf( $bar ) ) echo "TRUE";`
	`// TRUE`

Method	function getChildren(
	  <array> $orderBy = NULL
	) @return <array of OneDB_Category>
	
	Return all direct categories of this category.
	
	$orderBy → Weather to do MongoCursor::sort($orderBy) before returning the results.
	
	Example:
	
	Assuming that $foo is an instantiated OneDB_Category:
	
	$children = $foo->getChildren();
	
	foreach ($children as $child)
	  echo $child->name, "\n";
	
	/* will produce:
	  bar
	  zebra
	  army
	*/
	
	$children = $foo->getChildren(
	  array('name' => 1)
	);
	
	foreach ($children as $child)
	  echo $child->name, "\n";
	
	/* will produce:
	  army
	  bar
	  zebra
	*/

Method	`function createCategory(`
	`) @return <new OneDB_Category>`
	
	Creates a category inside this category and returns it. Note that you'll have to setup name for that category and do an implicit save after that.
	
	Example:
	
	`// - Create a new category called 'foo' inside an existing`
	`// category called 'bar'.`
	`// - Assuming that the 'bar' category is instantiated`
	`// as $bar`
	`try {`
	`  $foo = $bar->createCategory();`
	`  $foo->name = "foo";`
	`  $foo->save(); // we're doing an explicit save`
	`} catch (Exception $e) {`
	`  echo "Could not create new category: ", $e->getMessage();`
	`}`

Method	`function createArticle(`
	`  <string> $type = NULL`
	`) @return <<OneDB_Category>[.$type]>`
	
	Creates an article inside this category. If argument $type is specified, resulting category will be decorated with OneDB_Category_plugin_$type.class.php
	
	Example:
	
	`// Assuming that $foo is an instantiated category`
	
	`// creating a new article of type Document inside $foo category`
	`try {`
	`  $bar = $foo->createArticle( 'Document' );`
	`  $bar->name = "New Document";`
	`  $bar->owner = "Matthew";`
	`  $bar->save();`
	`} catch (Exception $e) {`
	`  echo "Could not create a new document: ",`
	`  $e->getMessage();`
	`}`

Method	`function articles(`
	`  <Array> $filter  = array(),`
	`  <Array> $orderBy  = array().`
	`  <int>  $limit  = NULL`
	`) @return <OneDB_ResultsNavigator.Article>`
	
	Returns all direct articles for this category.
	
	Example:
	
	`// Listing all online articles that are direct`
	`// child of category $foo:`
	
	`$foo->articles(`
	`  array(`
	`  'online' => 1`
	`  )`
	`)->each( function( $article ) {`
	`  echo $article->name, "\n";`
	`});`

3.6. class OneDB_Article

The OneDB_Article class is implementing an article on the OneDB database filesystem. With the help of the decorator plugins, we then decorate the OneDB_Article class to serve more roles: Files, Documents, JSON Objects, etc.

Articles, like categories, are have a common ancestor: OneDB_MongoObject, so they don't have too many built-in methods or properties in the non-decorated version. However, a lot of specific methods have been added to decorated OneDB_Article, so we will display them here.

Prototype	`class OneDB_Article extends OneDB_MongoObject`

Method	`function getParent(`
	`) @return <OneDB_Category>`
	
	Returns the parent OneDB_Category which holds this article

Method	function getPath(<boolean> $escapePath = TRUE) @return <string> Returns the path to this article. If $escapePath is set to TRUE, the components of the path will be url-escaped (meaning that spaces for example will be transformed into "+" signs)

3.6.1. Class OneDB_Article.Document

This decorator is used to extend the OneDB_Article in order to encapsulate a HTML document that is edited by the publishers via the backend interface.

Prototype	**class OneDB_Article.Document decorates OneDB_Article**
Properties	**Property** — **Type** — **Comment** property "title" — UTF-8 String property "document" — UTF-8 String property "textContent" — UTF-8 String property "revision" — int + All properties of OneDB_MongoObject class.
Method	function relatedDocs(<int> $maxDocs) @return <OneDB_ResultsNavigator.Article> Returns a navigator with $maxDocs related articles with this one. Related documents are set through backend interface by editors, in the Save dialog box of the DocumentEditor.

Method	`function html(`
	`) @return <UTF-8 String>`
	
	Returns the body of the article, enclosed into a tag, with embedded article meta properties.
	TIP: This is the recommended way of embedding articles into websites, not dumping directly the "`document`" property.

3.6.3. Class OneDB_Article.File

This decorator is used in order to extend the generic OneDB_Article to a File Article. It supports multiple storage engines through the OneDB_Storage api, storage migration from one engine to another, video transcoding, etc.

Prototype	`class OneDB_Article.File decorates OneDB_Article`
Properties	**Property** / **Type** / **Comment** `property "mime"` — `ASCII String` — The **mime-type** of the file `property "size"` — `Int` — The **size** in bytes of the file + All properties of OneDB_MongoObject class.

Method	`function _getStorage(`
	`) @return <new OneDB_Storage($self)>`
	
	Return the OneDB_Storage.* api class used for storage for this file.
	
	Additional file operations are implemented in the OneDB_Storage.* class, which are storage-dependent, like for example moving the file from a storage type to another storage type.

Method	`function _getStorageType(`
	`) @return <string>`
	
	Returns the name of the OneDB_Storage.* api used to store this file.

Method	`function setStorageType(`
	`  <string> $storageTypeName = 'database'`
	`) @return <void>`
	
	Sets the storage api type which will be used to store the file.

Method	`function storeFile(`
	`  <string> $filePath`
	`) @return <void>`
	
	Stores a file from a disk location, to current storage type.

Method	`function storeURL(`
	`  <string> $urlPath`
	`) @return <void>`
	
	Stores a file from an URL location, to current storage type.

Method	`function setContent(`
	`  <string> $content  = NULL,`
	`  <string> $mimeType = NULL`
	`) @return <void>`
	
	Sets the content of the file from a string. Additionally, a different mime-type can be specified for the new file content.

Method	`function getFile(`
	`) @return <MongoGridFS_File or compatible class>`
	
	Returns a Mongo GridFS File object, or a compatible class that is implementing it's methods. This is used by file wrappers, so fortunately you won't have to deal with this method at a general level.

3.6.4. class OneDB_Article.Layout

This decorator is used to extend the OneDB_Article in order to encapsulate a list of OneDB objects into a single container, while maintaining a special order, specified by the backend editors.

Prototype	`class OneDB_Article.Layout decorates OneDB_Article`

Properties	**Property**	**Type**	**Comment**
	`property "acceptItemTypes"`	Regular Expression String	What kind of items can be stored into this Layout Object.
	`property "maxItems"`	Integer	What is the maximum number of items that this layout object will return when calling method items()
	`All properties of the OneDB_MongoObject class are supported.`		

Method	`function items(`
	`) @return <OneDB_ResultsNavigator.Generic>`
	
	This function returns a navigator with the items supplied by the editors via the backend interface.		

3.6.5. Class OneDB_Article.Widget

A widget is a block of code stored into database, which can be executed by OneDB in context of an object view, or by the programmer by using the method run() explicitly. Widgets are made of four programming language sections: PHP, HTML, Css, and JavaScript via the backend interface, and are of 3 types:

- Php Widgets
- Xtemplate Widgets
- HTML Widgets

Widgets can also receive arguments, and execute themselves according to input arguments provided by the programmer or OneDB by itself. The most common argument name that a widget is receiving as input from the OneDB is called $argument, and is of type either OneDB_Category, either OneDB_Article.

Prototype	`class OneDB_Article.Widget decorates OneDB_Article`		
Properties	**Property**	**Type**	**Description**
	`All properties of the OneDB_MongoObject class are supported.`		

Method	`function setEnv(`
	`  <array> $env`
	`) @return <$self>`
	
	Sets the environment variables of the widget. Returns $this, in order to allow method chaining like jQuery.		

Method	`function run(`
	`  <array> $env = array()`
	`) @return <UTF-8 String>`
	
	Executes the widget, and returns it's output.

Method	`function dependencies(`
	`) @return <array>`
	
	This method is used to retrieve all depedencies of type css or javascript that the developer mentioned in the backend interface via the advanced tab of the WidgetEditor application.
	
	`@return: array(`
	`  'css' => array(`
	`  'inline' => '',`
	`  'files' => array(`
	`  ...`
	`  )`
	`  ),`
	`  'javascript' => array(`
	`  'inline' => '',`
	`  'files' => array(`
	`  ...`
	`  )`
	`  )`
	`);`

3.7. class OneDB_ResultsNavigator

A result navigator is a mechanism we're using in OneDB to manipulate query result-sets. Result navigators can be decorated via plugins, in order to support commands on specific sets of data of specific types. For example, a OneDB_ResultsNavigator.Category has some methods that have sense only on sets of categories, and a OneDB_ResultsNavigator.Article has some methods that have sense only on sets of articles obtained via queries through OneDB.

Prototype	`class OneDB_ResultsNavigator`

Constructor	`function __construct(`
	`  <array> $items,`
	`  <OneDB> &$server,`
	`  [ <string> $navigatorType = 'generic' ]`
	`)`
	
	Constructor for the class. You won't initialize this class by yourself, OneDB will do that for you.

Properties

Property	Type	Description
`property "length"`	int	Returns the number of items from current data-set.

Method

```
function flatten(
) @return <OneDB_ResultsNavigator.{selfType}>
```

When items from this result navigator are represented as a multi-dimensional tree structure, this function will convert that structure to a bi-dimensional structure.

`@return:` A results navigator with a bi-dimensional structure.

Method

```
function here(
      <callable($this)> $func
) @return <$this>
```

This method is useful to inject some code in the chain of commands, and continue the chaining.

`$callable` is a closure function which will be called with `$this` parameter as 1'st argument.

Method

```
function each(
      <callable($item[,$index])> $func
) @return <$this>
```

Calls the callable function `$func($item,$itemIndex)` for each element from the result navigator, and return the same navigator

Method

```
function filter(
      <bool or callable( $item )> $func
) @return <OneDB_ResultsNavigator.{selfType}>
```

Filters the contents of the data-set, with a user defined function `$func` which will be called on each items from the dataset. Function `$func` will be evaluated as boolean.

Method	`function sort(`
	`  <callable($itemA,$itemB )> $func`
	`) @return <OneDB_ResultsNavigator.{selfType}>`
	
	Returns a results navigator with items sorted by the callable user defined function `$func`.
	
	`$func` – comparision function, which will receive 2 parameters: `$itemA`, and `$itemB`. Function should return a negative integer (-1 or less) if `$itemA` < `$itemB`, 0 if `$itemA` = `$itemB`, and positive values (1 or more) if `$itemB` < `$itemA`.

Method	`function reverse(`
	`) @return <OneDB_ResultsNavigator.{selfType}>`
	
	Returns another navigator with items in reverse order

Method	`function skip(`
	`  <int> $howMany`
	`) @return <OneDB_ResultsNavigator.{selfType}>`
	
	Returns another result navigator, omitting the first `$howMany` items from this one

Method	`function limit(`
	`  <int> $howMany`
	`) @return <OneDB_ResultsNavigator.{selfType}>`
	
	Returns another result navigator with a maximum `$howMany` from this one

Method	`function unique(`
	`  <string> $propertyName`
	`) @return <OneDB_ResultsNavigator.{selfType}>`
	
	Returns another result navigator, with unique-by-property `$propertyName` values (acts similar with a group-by in sql clauses)

Method	`function get(`
	`  <int> $index`
	`) @return <mixed>`
	
	Returns the element located at the `$index` position from results. Throws exception if `$index` is out of range.

Method	`function join(`
	`  <OneDB_ResultsNavigator> $resultSet`
	`) @return <OneDB_ResultsNavigator.Generic>`
	
	Returns current items + items from the `$resultSet` OneDB_ResultsNavigator as a new navigator of type Generic.

Method	`function continueIf(`
	`  <boolean or $callable($this)> $boolOrCallable`
	`) @return <OneDB_ResultsNavigator or OneDB_DummyClass>`
	
	if parameter `$boolOrCallable` is evaluated as false, breaks the chain-commands by returning a OneDB_DummyClass. Otherwise, returns `$this`, in order to continue the chain of commands.

Method	`function applySortOrder(`
	`  <array> $sortOrder`
	`) @return <OneDB_ResultsNavigator.{selfType}>`
	
	This method is used by the `"%plugins%/sort"` plugin, in order to apply an editor custom sort order on items inside a category.

3.7.1. Class OneDB_ResultsNavigator.Article

This decorator is dedicated on working with sets of articles.

Prototype	`class OneDB_ResultsNavigator.Article decorates` `  OneDB_ResultsNavigator`

Method	`function getParent(`
	`) @return <OneDB_ResultsNavigator.Category>`
	
	Returns a navigator which contains the parents of articles from current navigator

3.7.2. Class OneDB_ResultsNavigator.Category

This decorator is dedicated on working with sets of categories.

Prototype	`class OneDB_ResultsNavigator.Category decorates` `  OneDB_ResultsNavigator`

Method	`function articles(`
	`  [ <array> $filters = array(), ]`
	`  [ <array> $orderBy = NULL,  ]`
	`  [ <int>  $limit = NULL  ]`
	`) @return <OneDB_ResultsNavigator.Article>`
	
	Returns all articles in categories from current data-set, based by an optional `$filter`, and doing an `MongoCursor::sort()` specified in `$orderBy`. Also, if `$limit` parameter is present, the destination `OneDB_ResultsNavigator.Article` will contain maximum `$limit` items.

Method	`Function getParent(`
	`  [ <array> $orderBy = NULL ]`
	`) @return <OneDB_ResultsNavigator.Category>`
	
	Returns a navigator with parent categories from the current data-set, applying a `MongoCursor::sort()` on results if the parameter `$orderBy` is specified.

3.7.3. Class OneDB_ResultsNavigator.Generic

This decorator is a generic one, it has no additional features from the OneDB_ResultsNavigator one.

3.8. Other OneDB classes

3.8.1. class OneDB_DatabaseExtender

3.8.2. class OneDB_DataParser

3.8.3. class OneDB_DummyClass

3.8.4. class OneDB_Form

3.8.5. class OneDB_JSONCollection

3.8.6. class OneDB_ObjectView

3.8.7. class OneDB_Registry

3.8.8. class OneDB_RootCategory

3.8.9. class OneDB_Security

3.8.10. class OneDB_SiteCache

3.8.11. class OneDB_Storage

3.8.12. class OneDB_TextSearch_Server

3.8.13. class OneDB_Tree

3.8.14. class OneDB_URLFile

3.8.15. class OneDB_WidgetCache

3.9. Helper functions

4. Extending OneDB

OneDB can extended by either:

- Creating an on-the-fly loadable a plugin
- Extending the core classes

On-the-fly loadable plugins are placed usually into a folder, and copied into the OneDB/plugins/ folder. After that, they are loaded either every time, via the %database%/plugins collection, either on-demand, via the backend.

4.1. Creating Plugins

A plugin has the following anatomy:

%plugin_folder%/
    plugin.json
    plugin.php
    [handler.php]

4.1.1. %plugin_folder%/plugin.json

This file is the plugin configuration file. It is written in JSON format, because it is both loaded by the backend (in JavaScript) and by the frontend (in PHP). The file should contain a JSON parseable object in the following format (without comments):

```
{
    "plugin": {
        "hasPanel":   <bool>,
        "panelName":  <string>,
        "panelClass": <string:PanelClassName>,
        "main":       <string:MainPluginBackendClass>
    }
}
```

The plugin.hasPanel is specifying the OneDB Backend that the plugin, when loaded, will have a tab inside the Plugins Panel zone.

The plugin.panelName will be the name of the panel.

The plugin.panelClass variable is specifying what class name to be loaded on the backend interface for initializing the panel content in the Plugins Panel zone. The class (better said function in JavaScript) prototype should be like this:

```
window.<myPluginPanelClass> = function( TabPanel ) {
```

```
  /* Example of a panel interface 'foo' plugin class
  // insert the plugin interface inside the panel...
  TabPanel.insert(
    document.createElement('p')
  ).setHTML( 'This is my plugin panel interface' );
  // End of example
  */
}
```

For being able to write a panel plugin, knowing the JSPlatform javascript api would be a plus.

The window.<myPluginPanelClass> function should be placed inside the plugin folder inside a JS file (file name doesn't matter, but the file should have a .js extension), for example: <myPluginPanelClass>.class.js.

The plugin.main variable from the plugin.json file is specifying another name of a JavaScript class, which will be loaded in the same manner, from a JavaScript file loaded from the root of the %your_plugin% folder.

Example of such class would be:

```
/* plugin.json */

{
      “plugin”: {
            …
            “main”: “myPluginMainClass”
            …
      }
}

/* myPluginMainClass.class.js */
window.myPluginClass = function( oneDB_window ) {
      …
      // add plugin functionality to your JSPlatform OneDB_Window object
      …
}
```

For a backend plugin demo loadable through the OneDB backend, see the OneDB/%plugins%/foo/ folder. You can load that plugin via the OneDB … Plugin Manager menu option, and when clicking the “Load Plugin” button, in the field “Input plugin name:” enter: “%plugins%/foo” and click the “Ok” button.

4.2. Loading Plugins

4.3. Extending core classes

5. Building, Developing and Debugging Websites in OneDB