easyObject

modern web applications made easy

User Manual

version 1.0 - December 2012

by Cedric Françoys

http://www.cedricfrancoys.be/easyobject

Table of contents

1. How is it working ?

BSUR: Browse, Search, Update, Remove

There are only four main methods to handle objects, and they are the same for all objects classes:

- *browse* (to obtain the value of one or more fields from one or more objects) ;
- *search* (too look for identifiers of objects matching one or more criteria) ;
- *update* (to create an object or modify one or more fields of one or more objects) ;
- *remove* (to delete one or more objects).

Examples :

```
// create a new object
$ids = update('core\User', array(0));

// update an object
update('core\User', $ids, array(
                                        'firstname' => 'John',
                                        'lastname' => 'Doe',
                                        'password' => 'secret'
                                        ));

// create and update 3 objects at once
$ids = update('core\User', array(0, 0, 0), array(
                                        'firstname' => 'John',
                                        'lastname' => 'Doe',
                                        'password' => 'secret'
                                        ));

// update several objects
update('core\User', $ids, array('password'=>md5('test')));

// search for one or more objects
$ids = search('core\User', array(
                                 array('lastname', 'like', '%oe%'),
                                 array('id', 'in', range(1, 100)
                                 ));

// browse one or more objects (obtain firstnames of users whose id is in the list)
$values = &browse('core\User', $ids, array('firstname'));
```

For a complete list of methods and their signatures, see description of the main methods.

Tree structure description

Location	Description
`Root`	Directory where the easyObject application is located (ex. : *home/easyobject/www*)
`files`	
`.htaccess`	Apache configuration file used to prevent directory listing and handle 404 errors
`fc.lib.php`	Classes and files inclusion library
`index.php`	This script is also referred to as the *dispatcher* : its task is to include required libraries and to set the context. **This is the unique entry point.**
`rpc_server.php`	Server script for *client-server mode*
`url_resolver.php`	Script invoked when a 404 error is raised, in charge of url rewriting
`folders`	
`actions`	Folder containing scripts that do some operations on objects and that might return JSON data (?do=…)
`apps`	Folder containing applications scripts that output HTML pages (?show=…)
`data`	Folder containing scripts that give information about objects by returning JSON answers (?get=…)
`library`	Folder containing every file that might be included by PHP scripts (files and classes)
`classes`	dossier contenant les classes php
`objects`	Folder containing objects definitions (files *.class.php*), so that the edition templates (views) and translation (i18n) files.
`orm`	Applicative core: contains classes object.class.php, objectManager.class.php, IdentificationManager.class.php, I18n.class.php and ErrorHandler.class.php
`db`	Dossier contenant les classes associées à l'interface avec la base de données. DBConnection.class.php : factory pattern DBManipulator.cass.php et classes héritées : adaptation des méthodes avec les commandes propres au DBMS utilisé
`utils`	Classes utilitaires utilisées par le cœur applicatif
`files`	contains *config.inc.php*
`[Zend]`	If required, this is where the Zend framework must be placed Note : it's not required by easyObject
`html`	
`js`	The javascript libraries used by applications scripts (/apps)
`css`	html stylesheets required by application scripts (/apps)

Object definition

The structure of every object class is defined in a *.class.php* placed into: *library/classes/objects/[package]*.

Every class inherits from a common ancestor: the *Object* class declared in the *core* namespace and defined in *library/classes/orm/Object.class.php*

A class is always referred to with the package name to which it belongs. The syntax is : 'package_name\class_name' (ex. :'school\Teacher').

A class consists of several fields, each of them having a name and a type, and a list of methods.
Some methods are system (their name is standard and used by the ORM), and others are specific to a class and defined by the user (see below).

Consistency between a .class.php definition and database schema

In parallel, a table must be defined in the database that has a structure matching the related class definition. The main constraint being that the types must be compatibles (ex.: a *varchar(255)* column in the database may represent a *string*, as well as a *short_text* or a *text* in the related class).

Consistency between type definition and related table structure in DB must be permanently maintained. This is left to the responsibility of the developer: no process neither checks nor fixes potential errors (seeUtility and stand-alone scripts).

Fields types

There are three kinds of fields:

1) Direct types
   boolean, integer, string, short_text, text, date, time, datetime, timestamp, selection, binary
2) Relational types
   one2many, many2many, many2one, related
3) Functional types
   function

or two categories:

1) Basic fields: the value of those fields is stored directly in the SQL table related to the object and don't need to be processed
   boolean, integer, string, short_text, text, date, time, datetime, timestamp, selection, binary, many2one
2) Complex fields: those fields require more work to be retrieved
   one2many, many2many, related, function

System fields

Some fields are common to all objects:

Field	Description
id	unique identifier of the instance
created	date and time of the object creation
modified	date and time of the last modification of the object
creator	identifier of the user that created the object
modifier	identifier of the user that made the last changes
published	boolean telling if the object has been published
deleted	boolean telling if the object has been deleted (object in the trash bin)

Common methods

Some methods, defined in the *core\Object* class, are common to all objects:

Method	Description
__construct	the constructor, invoked when a new instance is created
getSpecialFields	returns the names and types of the system fields (described above)
getColumns	returns the description of the user fields
getSchema	returns the full schema (system fields + fields defined in *getColumns*)
gettable	returns the name of the related SQL table
getId	return the identifier of the current object
getModifiedFields	returns the list of loaded fields
getModifiedFields	returns the list of fields that have been modified since the object has been loaded
resetLoadedFields	reset load flags
resetModifiedFields	reset modification flags
getUsedLangs	returns the list of the languages for which at least one field is defined
getFieldsNames	returns the names of the fields which type is mentioned in the specified list
getValues	returns an associative arrau containing the names and values of the specified fields
setValues	assign the specified values to the fields of the object

getColumns() method

The definition of this method is mandatory for every object. It returns the structure of the object (list of fields and their attributes) under the form of an associative array.

Example from the *school\Course* class:

```
public static function getColumns() {
    return array(
       'label'             => array(
                                 'type' => 'string'),
       'school_id'         => array(
                                 'type'                => 'many2one',
                                 'foreign_object'      => 'school\School'),
       'teachers_ids'      => array(
                                 'type'                => 'many2many',
                                 'foreign_object'      => 'school\Teacher',
                                 'foreign_field'       => 'courses_ids',
                                 'rel_table'           => 'school_rel_course_teacher',
                                 'rel_foreign_key'     => 'teacher_id',
                                 'rel_local_key'       => 'course_id'),
       'classes_ids'       => array(
                                 'type'                => 'one2many',
                                 'foreign_object'      => 'school\_Class',
                                 'foreign_field'       => 'course_id'),
    );
}
```

Common base

Here is the list of the attributes common to all fields:

Attribute	Description
type	the field type (a valid *easyObject* type)
[*label*]	the label of the field (that may appear in the edition forms) default value = name of the field
[*help*]	a tip to provide the final user about the role of the field or the way to fill it in
[*multilang*]	boolean telling if this field can be translated dafult value = false note : only basic fields can be translated (that excludes relational and functional fields)
[*search*]	boolean telling if the UI have to present this field as a search criteria default value = false notes : - a search query is possible on complex fields that have the store attribute set (see below) - search queries have better performances if related fields are indexed in database
[domain]	attribute for many2many and one2many fields allows to limit the list of related objects to those that match a specific criteria (same format as the one used in the *search* method)

Available types

boolean

Used for fields holding a numeric value of Boolean type (true or false).
notes : we use the PHP built-in constant : ***true*** and ***false***

integer

Used for fields holding a signed numeric value (negative or positive).
notes: with PHP it depends on the platform (generally 32 bits signed), with SQL it depends on the chosen type and size

string

Used for fields holding a short string with no formatting nor carriage returns (ex.: lastname, place, …)

short_text

Used for fields holding a string of characters that may contain carriage returns but no formatting (ex.: a short description, an address, …).

text

Used for fields holding a text that might be long and include formatting.
notes : with SQL, the types TEXT, BLOB or MEDIUMTEXT, MEDIUMBLOB are recommended

binary

Used for fields holding any binary value (ex. : a picture, a document, …)
notes : with SQL, the MEDIUMBLOB type is recommended

selection

Used for fields holding a value selected from a pre-defined list.

Example from the *core\Log* class:

```
'action'      => array(
                    'type'        => 'selection',
                    'selection'   => array(
                                        'R_CREATE'    => R_CREATE,
                                        'R_READ'      => R_READ,
                                        'R_WRITE'     => R_WRITE,
                                        'R_DELETE'    => R_DELETE,
                                        'R_MANAGE'    => R_MANAGE)),
```

Note : the part at the left side of the arrow is the one that will be outputted in the selection box, the right part is the one stored in DB.

date

Used for fields holding a date with the following format: YYYY-mm-dd

time

Used for fields holding a time with the following format: HH:mm:ss

datetime
Used for fields holding a date with the following format: YYYY-mm-dd HH:mm:ss

timestamp
Equivalent to datetime, but measured in number of seconds since the Unix Epoch (1st January 1970 00:00:00 GMT).

many2one
Used for fields holding a N-1 relation, that is to say a numeric value that represents the identifier of the pointed object.

Attribute	Description
foreign_object	the class toward which is pointing the current field

Example from the *school\Course* class:

```
'school_id'  => array(
                    'type'              => 'many2one',
                    'foreign_object'    => 'school\School'),
```

one2many
Used for a field holding à 1-N relation.

Attribute	Description
foreign_object	the class toward which is pointing the current field
foreign_field	the name of the field of the pointed clqss that is pointing back toward the current class
[*foreign_key*]	the field that serves as identifier for objects pointed by the relation (by default, *id* field)

Example from the *school\Teacher* class:

```
'classes_ids'=> array(
                    'type'           => 'one2many',
                    'foreign_object' => 'school\_Class',
                    'foreign_field'  => 'teacher_id'),
```

many2many
Used for a field holding a M-N relation.

Attribute	Description
foreign_object	the class toward which is pointing the current field
foreign_field	the name of the field of the pointed clqss that is pointing back toward the current class
rel_table	the name of the SQL table dedicated to the m2m relation (recommended syntax: *package*_rel_*class1_class2*)
rel_local_key	name of the column in *rel_table* holding the identifier of the current object
rel_foreign_key	name of the column in de *rel_table* holding the identifier of the pointed object

Example from the *school\Teacher* class:

```
'courses_ids' => array(
                    'type'             => 'many2many',
                    'foreign_object'   => 'school\Course',
                    'foreign_field'    => 'teachers_ids',
                    'rel_table'        => 'school_rel_course_teacher',
                    'rel_foreign_key'  => 'course_id',
                    'rel_local_key'    => 'teacher_id'
                    ),
```

related

This kind of field allows to specify an indirect relation of type *many2one* or *one2many*.
The principle is to indicate the field of another object, accessible by successive relations.

Attribute	Description
result_type	the type resulting from the indirect relation (i.e. type of the final pointed field)
foreign_object	name of the final pointed field
path	array holding the names of the cascade fields to consult in order to get to the final field (note : the first field must belong to the current class)

Example of the *school_id* field from the *school\Lesson* class:

```
'school_id'  => array(
                    'type'            => 'related',
                    'result_type'     => 'many2one',
                    'foreign_object'  => 'school\School',
                    'path'            => array('class_id','course_id','school_id')),
```

function

A functional field (or calculated field) allows to provide a class with fields which value depend on one or more other fields from the current object or any other object.

Notes : the use of the *store* attribute require, most of the time, that the fields on which depends the value of the functional field have an *onchange* event triggering the update of the calculated field (see example).

Attribute	Description
result_type	type of the value resulting from the invoked function
store	boolean telling if the function result must be stored in database (in that case, the related table must contain a column for the field)
function	string holding the name of the method to invoke, with format : `package\Class::method` (note : this method will be called with PHP function *call_user_func*)

Example of the field *rights_txt* from the *core\Permission* class:

Extrait de la méthode *getColumns()* :
```
'rights'      => array(
                        'type' => 'integer',
                        'onchange' => 'core\Permission::onchange_Rights'),
'rights_txt' => array(
                        'type' => 'function',
                        'store' => true,
                        'result_type' => 'string',
                        'function' => 'core\Permission::callable_getRightsTxt'),
```

Méthodes additionnelles :
```
public static function callable_getRightsTxt($om, $uid, $oid, $lang) {
    $rights_txt = array();
    $res = $om->browse($uid, 'core\Permission', array($oid), array('rights'), $lang);
    $rights = $res[$oid]['rights'];
    if($rights & R_CREATE)    $rights_txt[] = 'create';
    if($rights & R_READ)      $rights_txt[] = 'read';
    if($rights & R_WRITE)     $rights_txt[] = 'write';
    if($rights & R_DELETE)    $rights_txt[] = 'delete';
    if($rights & R_MANAGE)    $rights_txt[] = 'manage';
    return implode(',', $rights_txt);
}

public static function onchange_Rights($om, $uid, $oid, $lang) {
    $om->update($uid,
                'core\Permission',
                array($oid),
                array('rights_txt' =>
                       Permission::callable_getRightsTxt($om, $uid,$oid, $lang)),
                $lang);
}
```

getDefaults() method

The returned values are either the result of a closure, or a static method (from the current class or any other class).

Example from the *school\Student* class:

Extrait de la méthode *getColumns()* :
```
       'birthdate'          => array('type' => 'date'),
       'subscription'       => array('type' => 'date'),
```

Méthodes additionnelles :
```
public static function getDefaults() {
       return array(
              'subscription'       => 'school\Student::default_subscription',
              'birthdate'          => function() { return '2000-01-01'; }
       );
}

public static function default_subscription() {
       return date("Y-m-d");
}
```

getTable() method

It is also possible to rewrite the *getTable()* method. This can be useful when the name by default is not a good choice or if, for any reason, it does not fit the developer needs.

Example : *school_Class*

```
public function getTable() { return 'school_class'; }
```

Inheritance

It is possible to rewrite a class (in some situations, the overwriting is even mandatory).
To set up an inheritance mechanism:

- the class must not extend the *\core\Object* class, but the class from which it inherits;
- the *getTable()* method must be used to define the name of the related table in database (by default the name of the related table will be the one of the parent class) ;
- the *getColumns()* class must return an associative array holding every field of the new class (in order not to rewrite the fields that are common with the parent class, one may use the PHP function *array_merge* with *parent::getColumns()* as first parameter).

Example : *knine\User*

```
namespace knine {
      class User extends \core\User {

             public function getTable() { return 'core_user'; }

             public static function getColumns() {
                   return array_merge(parent::getColumns(),
                          array(
                          'articles_ids' => array(
                                'type'              => 'many2many',
                                'foreign_object'    => 'knine\Article',
                                'foreign_field'     => 'authors_ids',
                                'rel_table'         => 'knine_rel_article_user',
                                'rel_foreign_key'   => 'article_id',
                                'rel_local_key'     => 'user_id')
                   ));
             }
      }
}
```

Permissions management system

The ‘Permission’ class is dedicated to the rights management: for each object class (including the ‘Permission’ class itself), rights can be defined for one or more user groups.

Involved classes are:

core\User (library/classes/objects/core/User.class.php)

```
public static function getColumns() {
       return array(
              'firstname'         => array('type' => 'string'),
              'lastname'          => array('type' => 'string'),
              'login'             => array('type' => 'string', 'label' => 'Username'),
              'password'          => array('type' => 'string', 'label' => 'Password'),
              'language'          => array('type' => 'string'),
              'groups_ids'        => array('type' => 'many2many',
                                          'foreign_object'  => 'core\Group',
                                          'foreign_field'   => 'users_ids',
                                          'rel_table'       => 'core_rel_group_user',
                                          'rel_foreign_key' => 'group_id',
                                          'rel_local_key'   => 'user_id')
       );
}
```

core\Group (library/classes/objects/core/Group.class.php)

```
public static function getColumns() {
       return array(
              'name'              => array('type' => 'string'),
              'users_ids'         => array('type' => 'many2many',
                                          'foreign_object'  => 'core\User',
                                          'foreign_field'   => 'groups_ids',
                                          'rel_table'       => 'core_rel_group_user',
                                          'rel_foreign_key' => 'user_id',
                                          'rel_local_key'   => 'group_id'),
              'permissions_ids'   => array('type' => 'one2many',
                                        'foreign_object'    => 'core\Permission',
                                        'foreign_field'     => 'group_id')
       );
}
```

core\Permission (library/classes/objects/core/Permission.class.php)

```
public static function getColumns() {
       return array(
              'class_name'        => array('type' => 'string'),
              'group_id'          => array(
                                        'type'              => 'many2one',
                                        'foreign_object'    => 'core\Group',
                                        'foreign_field'     => 'permissions_ids'
                                     ),
              'rights'            => array('type' => 'integer')
       );
}
```

The possible rights that can be assigned are defined in the file *config.inc.php* :

```
define('R_CREATE',    1);
define('R_READ',      2);
define('R_WRITE',     4);
define('R_DELETE',    8);
define('R_MANAGE',    16);    // autorisation to manage the rights
```

The field ‘rights’ of the *Permission* class is a binary mask (logical OR) of the rights given to the related group.

If, for some class, no permission is defined for none of the groups to which belongs a user, this one receives the default permissions, defined in the configuration file (*DEFAULT_RIGHTS* constant).

Translation mechanism

Two distinct mechanisms are used depending on what we are handling:

1) translation of the terms related to a class;
2) translation of object fields values (instance).

Translation of the terms related to a class

Each package has a folder named ‘i18n’ containing, for each language, a subfolder which name matches the IS0 639 code of the language (ex. : fr_BE ou zh_CN). Inside those folders, for each class, is a *.json* file which prefix is identical to the class it refers to (ex. : Student.json).
Those translation files are in UTF-8 and in JSON format, and contain the translation terms of all items that might be translated (attributes ‘label’ et ‘help’).
This system is not the lighter in terms of overhead but offers the advantage of being usable as is by the UI (no processing server-side).

Indeed, the translation file is directly accessible via HTTP request:
'library/classes/objects/'+package_name+'/i18n/'+lang+'/'+object_name+'.json',

On the other hand, to obtain the file via script, you have to invoke *?get=core_i18n_lang*.
Location : `data/core/i18n/lang.php`
URL example: http://localhost/easyobject/?get=core_i18n_lang&class=school\Student&lang=fr

Translation of objects fields values

All basic fields can be translaterd (see Fields types).
To allow this, a table in DB is dedicated to the translation terms.
When a field is marked as ‘multilang’, the values of its translations can be retrieved with a SQL query.

'Translation' object
fields:

string lang (code ISO 639-1)
string object_class
string object_field
integer object_id
mediumblob value

For the value field, the SQL MEDIUMBLOB type is used (overhead of 3 bytes, max size of 16,7 Mo).
Indeed, the size of the 'value' column may vary greatly from one type to another (the *binary* type may represent a document, a picture, a video, …). And, most of the time, fields that must be translated are

texts (string, short_text ou text). In any case, a 3 characters overhead is acceptable (and set a 16 Mo limit should not be a problem).
Note : as the only condition for the SQL type is to be compatible with the associated easyObject type, one must pay attention, in order for a binary field to be translated (for instance a PDF doc available in different languages), that the value of this field never has a greater size to the one of the SQL MEDIUMBLOB type (by example : BLOB, LONGBLOB, …).

2. How to use it?

Main methods

The following methods can be invoked as is at any time.
Note : these methods are defined in PHP (easyobject.api.php) as well as in Javascript (easyobject.api.js), and their signatures are strictly identical in both languages.

If an error is raised, most of these methods return an integer holding one or more error codes (that can be isolated with a binary mask), that are defined in the *config.inc.php* file:

Constant	Value	Description
`UNKNOWN_ERROR`	0	something went wrong (that requires to check the logs)
`INVALID_PARAM`	1	one or more parameters have invalid or incompatible value
`SQL_ERROR`	2	error while building SQL query or processing it
`UNKNOWN_OBJECT`	3	unknown class or object
`NOT_ALLOWED`	4	action violates some rule or user don't have required permissions

user_id() method

integer **user_id**()

Returns the identifier of the current user (based on the current PHP session id).

user_lang() method

string **user_lang**()

Returns the language (ISO 639 format) of the current user (based on the current PHP session id).

login() method

boolean **login**(string *$login*, string *$password*)

This method tries to validate the identification of a user. If the identification succeeds, the method returns TRUE and the current session is then bound to the id of the related User object.

validate() method

mixed **validate**(string *$object_class* , array *$values*)

Checks whether the values of given object fields are valid or not.
The returned value is either FALSE or an array associating, for each invalid field, the field name and the associated error message id.

browse() method

mixed &**browse**(string *$object_class* [, *$ids*=null [, *$fields*=null [, *$lang*=DEFAULT_LANG]]])

On success this method returns, for the specified class, an array associating the list of the values of each given field for each given object id.

If an error occurs, an integer holding one or more error codes is returned.

Note: This method may generate a SQL query for each of the specified ids (for objects that have not yet been loaded into the manager). A good practice then consists of not calling the method for each field separately, but by grouping all required fields in the *$fields* array.
Example:

```
$values = &browse('core\User', array(3), array('login', 'firstname', 'lastname'));
echo "{$values[3]['login']}, {$values[3]['firstname']}, {$values[3]['lastname']}" ;
```

search() method

mixed **search**(string *$object_class*, array *$domain*=null, string *$order*='', string *$sort*='asc', integer *$start*=0, *$limit*='', *$lang*=DEFAULT_LANG)

Returns the list of the objects identifiers matching a given domain, sorted on a given field, possibly limited to a specific section.

update() method

mixed **update**(string *$object_class*, array *$ids*, array *$values* [, *$lang*=DEFAULT_LANG])

This method serves for creating new objects as well as for modifying them.
When an item of the *ids* list is set to zero, a new object is created and its identifier placed in the array returned by the method. For examples, see BSUR: Browse, Search, Update, Remove.

remove() method

mixed **remove**(string *$object_class*, array *$ids*, boolean *$permanent*=false)

This method allows to remove one or more objects. If the parameter 'permanent' is left to FALSE, the field *delete* of the object is set to TRUE and the object does not appear in the lists but the object may be retrieved and the removal canceled. If the parameter 'permanent' is set to TRUE, then is object is removed from the database (and can no longer be retrieve unless the database has been backed up).

get() method

```
mixed &get(string $object_class , integer $object_id)
```

It is also possible to get the instance of an object.
To retrieve or change the value of the fields declared in the *getColumns* method, it is then possible to use *getters* and *setters* using standard OO syntax (implemented with the PHP magic method __*call*).
Example :

```
$User = &get('core\User', 3);
$first_name = $User->getFirstname();
$User->setFirstname('Lulu');
```

Notes: This method implies the loading of all fields (that may be numerous and which value may require a lot of processing to be obtained). So, it is recommended to use it only when the complete instance of the object is actually needed and not in order to retrieve only one specific field (in which case, it is preferable to use the *browse* method).

getStatic() method

```
mixed &getStatic(string $object_class)
```

In some cases, it’s necessary to get an empty object for a specific class (for instance, while validating the schema of a class).

Objects edition

To define the layout, the list of the fields and the possible interactions between the fields of an edition form, we use a system of views similar to templates.

Each package has a folder names ‘views’ that contains, for each class, one or more HTML file.
These files are written in HTML5, and contain information about the fields and labels to display and their positioning, for the edition of the related class.
Again, this system is not the lightest in terms of overhead but has the advantage of being usable as is by the Javascript client (no server-side processing).

Their name format is: object_name.(list | form).view_name.html

A series of HTML5 tags has been chosen in order to be used in views. Those tags were chosen because of the usage that is generally done of them:

- reserved to inputs (as this part is generated automatically)
- having, by default, no visual impact

Tags and attributs for the views:

Tag	Attribute	Description
form	*action*	the action to be executed when form is submitted (button 'save')
section	*name*	used to group several items in sections accessible by selecting tabs
fieldset	*title*	displays a frame that allows to group several items
span	*width*	width of the *span*, in % of the parent width
div	*width*	width of the du *div*, in % of the parent width
label	*[for]*	the name of the field the label is related to, if any
	[name]	the identifier of the independent label, if any
var	*id*	name of the field
	[*onchange*]	action to be executed in case the field is modified by the user (written in javascript, using jQuery syntax)
	[*readonly*]	boolean telling if the UI must allow the modification of the field (default value = false) note : this mechanism is limited to the presentation layer and must not be used as right management
	[*required*]	boolean telling if the field is mandatory (default value = false)
	[*domain*]	string containing the domain to which we want to limit the items to display (applicable with m2m or o2m lists)
	[*view*]	the name of the view to be used (if this attribute is not specified, the related **.list.default.html* file will be used)

Example of a form for the *school\Student* class (Student.form.default.html):

```
<form action="core_objects_update">
    <section name="identification">
        <fieldset title="identification">
            <span width="50%">
                <label for="firstname"></label>
                <var id="firstname" required="true"></var>
                <br />
                <label for="lastname"></label>
                <var id="lastname" required="true"></var>
            </span>
            <span width="50%">
                <label for="birthdate"></label>
                <var id="birthdate" required="true"></var>
                <br />
                <label for="subscription"></label>
                <var id="subscription" required="false"></var>
            </span>
        </fieldset>
    </section>
    <section name="data">
        <fieldset title="data">
            <span width="100%">
                <label name="classes_ids"></label>
                <br />
                <var id="classes_ids" view="list.default"></var>
            </span>
        </fieldset>
    </section>
</form>
```

Example of a list for the *school\Student* class (Student.list.default.html):

```
<ul>
      <li id="id" width="10%"></li>
      <li id="firstname" width="23%"></li>
      <li id="lastname" width="23%"></li>
      <li id="birthdate" width="22%"></li>
      <li id="subscription" width="22%"></li>
</ul>
```

Basic applications

Some applications (?show queries) are included in the core to ease management and development of new packages:

- *core_manage* : management of existing objects (list, creation, edition, removal) by package (URL example: http://localhost/easyobject/?show=core_manage)
- *core_utils* : (see Utility and stand-alone scripts) (URL example: http://localhost/easyobject/?show=core_utils)
- *core_setup* : to validate a fresh install of easyObject.

Utility and stand-alone scripts

Utility scripts allow to achieve some tasks that are not handled by the core, for instance checking the consistency of a new package.
They act like some sort of plugins, written in PHP and located in the folder: *data/utils/*

Some scripts are planned, in progress or already available:
Available (beta):

- core validation
- package validation: consistency checks between DB and class as well as syntax validation for classes (PHP), views (HTML) and translation files (json)

Planned :

- Create a compatible database based on a SQL schema
- Generate a PHP class from an existing table
- Generate files for default views (list.default.html and form.default.html)
- Data import / export

To consult the list of the plugins and apply them to one or more package, you may use the *core_utils* application (URL example: http://localhost/easyobject/index.php?show=core_utils).

Note: Of course, those scripts may be written by the user-developer or adapted to any particular purposes.