SmartFcatory

PHP Library

Small and simple library

With many useful tools and functions

Without overhead

Designed based on IoC

PHP has bad reputation. Why?

- PHP is often blamed and criticized. It is often not treated as serious development platform.
- However, PHP is widespread and alive since many years, whereas many “progressive” and “perspective” technologies are obsolete or dead.
- PHP is supported by almost all providers, its hosting is not expensive, the development is easier, faster and less expensive.
- PHP can also be scaled and, precompiled and optimized.
- PHP offers all possibilities to code precisely and professionally.

Reasons for the bad PHP image

1. PHP is very easy to learn. A newcomer can start code with PHP after a week. The easiness stops many people from evolving in professional PHP programming. They think they can already program in PHP and there is nothing more necessary. That why there are tons on horrible unprofessional PHP code in the world.

2. PHP applications runs as a rule on a cheap hosting with very limited resources. It make the impression that PHP is slow. Whereas Java and .NET Applications runs usually on performant expensive hardware. A professionally written PHP application on the same performant hardware would also show good results.

Reasons for the bad PHP image (part 2)

3. Database bottleneck. Many slow operations are caused not through the PHP, but through the long running SQL queries due to not optimal queries or due to blocking, but PHP is blamed for that.

4. Usage of not optimal frameworks. Frameworks accelerate the development process, but as a rule, they come along with much overhead caused by unnecessary abstractions and intermediate elements.

Framework or not framework - advantages

Main advantages:

- Framework accelerates the implementation of the standard operations.
- Unification of the approaches, e.g. working with databases and form handling. Framework offers standard code patterns for many areas of the programming.
- Some frameworks offer automatic code generation for some standard tasks.
- Unit testing and IoC (Inversion of Control) support.

Framework or not framework - disadvantages

Main disadvantages:

- Too much abstractions. Too much intermediate constructions causing functional overhead.
- Low performance by high load due to the functional overhead of the framework.
- Low flexibility for handling special cases that differ from the standard approach.
- Internal code of the framework is hard to understand and to maintain.
- Internal code of the framework should not be changed. It might cause unexpected errors and make the framework hard to update. Your manual changes might be overwritten by the next update. You have to take care about reintegration of your changes in the updated version. Your changes might become incompatible with the new version of the framework.
- Older versions of the framework might be incompatible with the new versions of PHP. You will be not able to run your software on the newer versions of PHP. Your software written on an older version of the framework might be incompatible with the new version of the framework. Large reengineering and retesting of your software might be required to port your software to the new version of the framework in order to run it on the newer versions of PHP.

Framework or not framework – golden middle

Even if we are going to avoid disadvantages of the frameworks, it is still possible to create a library that covers, simplifies and accelerates many issues of the programming without having the disadvantages.

A golden middle
between a framework with overhead and
native programming
is possible.

Let's describe what issues can and should be covered by a library without having disadvantages listed above.

What a Library should cover

The library should be based on the IoC (Inversion of Control) principle.

The library should offer own solution of session management that wraps the standard solution. If the standard solution turns out to be inapplicable in some cases, you can rewrite the session implementation without touching the business logic code on many places.

The library should offer an universal solution for tracing, debugging, logging and profiling. When running in a production environment, it is not always easy to use advanced external debugging and profiling tools.

What a Library should cover (part 2)

The library should offer an universal message handling for reporting errors, warnings and information messages also over multiple requests and over asynchronous requests.

The library should offer an event management solution what corresponds to the principal of the IoC – lose coupling of the program modules and functions.

The library should offer a solution for rapid implementation of own API interfaces and for implementation of the interaction with other API interfaces – on JSON and on XML format.

What a Library should cover (part 3)

The library should offer a solution for interaction with databases. Even if the PHP offers own universal solutions like PDO, it makes sense to wrap their implementation. If in the future, there will be a better solution, or the current solution turns out to be inefficient in a new version of PHP, we can easily reimplement the DB wrapper without touching the business logic code.

Beside the accessing of the database, the library should also offer tools for simplifying of the standard DB operations like loading and saving records and datasets to/from arrays.

The library should offer a solution for localization and management of the translation texts.

What a Library should cover (part 4)

The library should offer an universal mechanism for settings management. Almost each application has some settings. Usually, these are:

Configuration settings

The configuration settings are stored in a file and define the startup parameters of the application, the database connection settings, the master admin password etc.

Runtime settings

The runtime settings are usually stored in the database and can be changed while running the application. They define some general aspects of the application behavior.

User settings

The user settings are related to the user and are kept in the user session as long as the session remains valid. They define some aspects of the application behavior specific for the current user.

What a Library should cover (part 5)

The library should offer a wrapper for placement of the form elements with automatic bounding them to the request variables with the corresponding name. But the user should not be limited in ability to define usual HTML properties of the form elements.

The library should offer a flexible tool for rendering of the HTML tables based on arrays.

Inversion of Control principle

The main goal and paradigm of the IoC is writing an efficient flexible program code which can be easily changed and automatically tested. This goal can be reached when programming is done based on the following principles:

- Modularity and lose coupling of the modules and functions. Behavior of a module can be easily changed and improved without or with low affecting other modules and functions.
- Wrapping the standard system functions. If some standard functions turn out to be inefficient or buggy and must be replaced, there will be no need to replace them in thousand places of the program code. You will have just to reimplement the wrappers.
- Design the common interfaces and then implement them in the classes. Refer to the interfaces in your program code instead of referring to the classes.
- No direct object creation in the business logic or in the class method implementation. All objects should be created over requests to a factory. The factory is the single point of control. If you have to reimplement the logic of an object, you have just to change the factory so that it provides the new object with the new logic. You will not have to touch the code in many places where this object is requested.
- Thanks to using the factory as the single point of control, you can flexibly implement automated testing procedures by implementing of a special factory that creates MOK and STUB objects instead of real objects for testing purposes.

SmartFcatory

PHP Library

Small and simple library

With many useful tools and functions

Without overhead

Designed based on IoC

Directory Structure

```
docs
src
  SmartFactory
    Interfaces
    DatabaseWorkers
```

Directory Structure (part 2)

docs

This directory contains the documentation about classes, interfaces and functions of the library SmartFactory.

src

This is the root directory for all classes and interfaces. The class loader is implemented based on PSR4 approach. You have no need to add additional class loader function for your classes.

src/SmartFactory

This directory contains the core classes and interfaces of the library SmartFactory.

src/SmartFactory/Interfaces

This directory contains the core interfaces of the library SmartFactory.

src/SmartFactory/DatabaseWorkers

This directory contains the core classes of the library SmartFactory for working with databases.

Factory methods

The *SmartFactory* has the following factory methods:

singleton(Interface);
The method *singleton* creates an object that support the interface *Interface* and ensures that only one instance of this object exists. The singleton is a usual patter for the action objects like *SessionManager*, *EventManager*, *DBWorker* etc. It makes no sense to produce many instances of such classes, it wastes the computer resources and might cause errors.

instance(Interface);

The method *instance* creates an object that support the interface *Interface*. By each request, a new object is created. If you request data objects like User, a separate instance must be created for each item.

Factory methods (part 2)

dbworker($parameters = null);

The method *dbworker* provides the *DBWorker* object for working with the database (currently supported MySQL und MS SQL). If the *parameters* are omitted, the system takes the parameters from the configuration settings and reuses the single instance of the *DBWorker* for all requests.

If the user passes the parameters explicitly, a new instance of the *DBWorker* is created upon each new request.

Factory methods (part 3)

Class *ObjectFactory*

The class *ObjectFactory* is an auxiliary class that empowers the factory methods. It provides the methods for binding of class implementations to the interfaces and for creation of objects for the requested interface, e.g.:

```
ObjectFactory::bindClass(ISessionManager::class, SessionManager::class);
```

If you implement another class of *ISessionManager* and want that the factory creates the objects of this class, you have just to say:

```
ObjectFactory::bindClass(ISessionManager::class, MySessionManager::class);
```

Request of the instances is done over the method:

```
function getInstance($interface_or_class, $singleton);
```

The method looks which class is bound to the *interface_or_class* and creates and instance of it. If the parameter *singleton* is true, it ensures that only single instance is used.

Factory methods (part 4)

Class *ObjectFactory*

If you have a standalone class or you want explicitly say that you need objects of this class, you can bind the class to itself:

```
ObjectFactory::bindClass(JsonApiRequestHandler::class, JsonApiRequestHandler::class);
```

The method *bindClass* supports also the initialization method, where you can initialize the created instance:

```
ObjectFactory::bindClass(ConfigSettingsManager::class, ConfigSettingsManager::class, function($instance)
{
  $instance->init(["save_path" => "../config/settings.json",
                   "config_file_must_exist" => false
                  ]);

  $instance->loadSettings();

  $instance->setValidator(new ConfigSettingsValidator());

});
```

Factory methods (part 5)

File *initialization_inc.php*

Binding of the basic classes is done in the file

src/intialization_inc.php

Use *test_intialization_inc.php* in your test procedures.

ISessionManager

This is a wrapper around the standard functionality of the PHP session management. If the standard solution turns out to be inapplicable in some cases, you can rewrite the session implementation without touching the business logic code on many places.

This solution offers the following two advantages:

- Non-blocking read-only sessions. This feature is essential for the asynchronous requests. Per default, starting a session blocks the session data file until the request is finished. That means, that the asynchronous session aware requests can be run only serialized, what neglect the sense of the of asynchronous requests. Starting asynchronous request in read-only modus allows using the session data, whereby the execution is not blocked.
- Session context. If many instances of the application should run in parallel subfolders, and all subfolders are within the same session, and the provider does not let you to change the session path, then you can use different context in each instance to ensure that the session data of these instances does not mix.

Thanks to the PHP 7.x feature (returning reference), you can use with the session variables in a comfortable way:

```
session()->vars()["user"]["name"] = "Alex";
session()->vars()["user"]["age"] = "22";

$sessionvars = &session()->vars();
$sessionvars["user"]["sex"] = "M";
```

IDebugProfiler

This is build-in debugging, logging and tracing solution. It offers the methods for generation of the logs in the folder *logs*.

You can use it for debugging the code and measuring of the execution time of some critical code parts.

IErrorHandler

This is build-in solution of the error handling. All errors, warnings and notices are caught up and traced with details to the trace file *trace.log* in the folder *logs*.

Tracing can be turned on/off.

You can add an event handler to the event *php_error* to do additional error handling.

IEventManager

This is the solution for event management. You can register your own handling functions to definite events and, then, fire these events.

```
event()->addHandler("event1", function($event, $params)
{
  echo "Event handler called: $event<br>";
  echo "<pre>";
  print_r($params);
  echo "</pre>";
});

$params = ["p1" => 100, "p2" => 200];
event()->fireEvent("event1", $params);
```

You can also suspend and resume events.

ILanguageManager

This is the solution for localization. The main idea is using a text code as a reference and provide the translation depending on the current language:

```
text('DatabaseName');
```

Additionally, the solution offers the list of language names and country names. They can be also accessed by a code and the translations are provided based on the current languages.

Furthermore, there is a function for getting the language and country ISO code from the name, what might be useful if the language or country is entered by name, but you have to store the ISO code.

The library *SmartFactory* offers a user friendly comfortable GUI interface for adding new languages and new text entries. The translations are saved in the JSON file *localization/texts.json*.

ILanguageManager (part 2)

Detecting the current language

The library offers ready to use solution for choosing the default language. If is done in the method:

```
detectLanguage($context = "default");
```

Priority of choosing

1. explicitly set by the request parameter *language*
2. last language in the session
3. last language in the cookie
4. browser default language
5. the first one from the supported list
6. English

Context

Some applications may consist of two parts - administration console and public site. A usual example is a CMS system. For example, you are using administration console in English and editing the public site for German and French. When you open the public site for preview in German or French, you want it to be open in the corresponding language, but the administration console should remain in English. With the help of context, you are able to maintain different languages for different parts of your application. If you do not need the context, just do not specify it.

DBWorker

This is a wrapper around the database connectivity. It offers an universal common way for working with databases of different types. Currently, MySQL and MS SQL are supported.

If in the future, there will be a better solution, or the current solution turns out to be inefficient in a new version of PHP, we can easily re-implement the DB wrapper without touching the business logic code. Adding support for new database types is also much easier with this wrapping approach.

The main paradigm concerning the work with the databases is the full control over SQL query code. Queries can be directly edited. NO abstract constructions, where the columns, conditions, joins are added over the method calls instead of writing the clear query text. The typical code is:

```
if(!dbworker()->execute_query("SELECT FIRST_NAME, LAST_NAME FROM USERS"))
{
  return sql_error(dbworker());
}

while(dbworker()->fetch_row())
{
  echo dbworker()->field_by_name("FIRST_NAME") . " " . dbworker()->field_by_name("LAST_NAME") . "<br>";
}

dbworker()->free_result();
```

DBWorker (part 2)

DBWorker provides also many additional useful functions:

- Checking whether the corresponding extension is installed.
- Escaping the strings for the current database type.
- Formatting date and time to compatible format for the current database type.
- Streaming large data.
- Execution of the prepared queries with binding variables.
- Execution of the stored procedures.
- Fetching the result set into an array.

IRecordsetManager

This is the solution for efficient work with records and record sets. It is based on the *DBWorker* functionality. It saves your time by working with typical tasks like loading and saving a record or a set of records.

The usage is:

- Define the table and column mapping.
- Use an array as record with the keys named identically to the column names.
- Use an array as record set where the dimensions are the foreign keys to the parents records.

IShardManager

This is the solution for managing database shards. The shard manager allows registering multiple shards and request connections to them. It ensures that only one connection to a shard is used within a request. Moreover, you can add some number of shards to a load balancing group, and then randomly request a shard from this group. This ensures more or less equal load balancing between the shards of the group.

Usage:

1) Register shards:

```
$shardmanager = singleton(IShardManager::class);

$shardmanager->registerShard("myshard", ["db_type" => "MySQL",
                                         "db_server" => "localhost",
                                         "db_name" => "framework_demo",
                                         "db_user" => "root",
                                         "db_password" => "root",
                                         "autoconnect" => true,
                                         "read_only" => false]
                            );
```

2) Request the dbworker for a shard:

```
$dbw = dbshard("myshard");
```

ISettingsManager

This is the solution for the settings management. There are 3 implementations for different types of settings.

ConfigSettingsManager
The configuration settings are stored in a file and define the startup parameters of the application, the database connection settings, the master admin password etc.

RuntimeSettingsManager
The runtime settings are usually stored in the database and can be changed while running the application. They define some general aspects of the application behavior and they can be adjusted on the fly.

UserSettingsManager
The user settings are related to the user. They define some aspects of the application behavior specific for the current user.

ISettingsManager supports *ISettingsValidator* that is used for the validation of the settings. You can implement your validation logic in a class implementing the interface *ISettingsValidator* and inject it into the *ISettingsManager*.

All implementations support context – e.g. general settings, database settings etc. You can implement different interfaces for editing different types of settings and validate and save only a subset of settings. E.g. the database server is a required field, but the validation of the database server applies only if the database connection is edited. The context is essential only for validation and saving, but for the retrieval, any settings is available independent of the context.

ConfigSettingsManager

This settings manager handles the basic configuration settings of the application like the startup parameters of the application, the database connection settings, the master admin password etc. The settings are stored in the file *config/settings.json*. *ConfigSettingsManager* is initialized as follows:

```
ObjectFactory::bindClass(ConfigSettingsManager::class, ConfigSettingsManager::class, function($instance) {

  $instance->init(["save_path" => "../config/settings.json",
                   "config_file_must_exist" => false,
                   "save_encrypted" => true,
                   "salt_key" => "some key"
                  ]);

  $instance->setValidator(new ConfigSettingsValidator());
});
```

Optionally, you can save the settings encrypted.

You do not need any preliminary special settings name definitions. When you introduce a new setting, just start saving and getting it.

RuntimeSettingsManager

This settings manager handles the runtime configuration settings of the application. The settings are stored in the database in a column as JSON text. *RuntimeSettingsManager* is initialized as follows:

```
ObjectFactory::bindClass(RuntimeSettingsManager::class, RuntimeSettingsManager::class, function($instance) {
  $instance->init(["dbworker" => dbworker(),
                   "settings_table" => "SETTINGS",
                   "settings_column" => "DATA"
                  ]);

  $instance->setValidator(new RuntimeSettingsValidator());
});
```

You do not need any preliminary special settings name definitions. When you introduce a new setting, just start saving and getting it. But you need provide a settings table with a column where the settings data will be saved.

UserSettingsManager

This settings manager handles the user specific settings of the application. The settings are stored in the database with relation to the user – either in the user table or an additional table related to the user. Each single setting can be a subject of SQL where clause, thus, each single setting is saved to a separate column. *UserSettingsManager* is initialized as follows:

```
ObjectFactory::bindClass(UserSettingsManager::class, UserSettingsManager::class, function($instance) {
  $instance->init([
        "dbworker" => app_dbworker(),

        "settings_tables" => [
            "USERS" => [
                "ID" => DBWorker::DB_NUMBER,
                "LANGUAGE" => DBWorker::DB_STRING,
                "TIME_ZONE" => DBWorker::DB_STRING
            ],
            "USER_FORUM_SETTINGS" => [
                "USER_ID" => DBWorker::DB_NUMBER,
                "SIGNATURE" => DBWorker::DB_STRING,
                "STATUS" => DBWorker::DB_STRING,
                "HIDE_PICTURES" => DBWorker::DB_NUMBER,
                "HIDE_SIGNATURES" => DBWorker::DB_NUMBER
            ]
        ],
        "multichoice_tables" => [
            "USER_COLORS" => [
                "USER_ID" => DBWorker::DB_NUMBER,
                "COLOR" => DBWorker::DB_STRING
            ]
        ]
    ]);


   $instance->setValidator(new UserSettingsValidator());
});
```

Furthermore, multichoice settings can also be handled over the child table, e.g.:

```
"USER_COLORS" => ["USER_ID", "COLOR", DBWorker::DB_STRING]
```

When you introduce a new user setting, you have to create a column for it in the corresponding table add it to the initialization and specify the data type.

JsonApiRequestManager

This class offers the solution for the development of the API endpoints. There is a folder *api* with the file *index.php* which handles all API requests. For handling the requests you have to create a class implementing the interface *IJsonApiRequestHandler* and register it to the request name:

```
class LoginHandler implements IJsonApiRequestHandler
{
  public function handle($rmanager, $api_request, &$response_data, &$additional_headers)
  {
    if(empty($_REQUEST["user"]) || empty($_REQUEST["password"]))
    {
      $response_data["result"] = "error";

      $response_data["errors"][] = ["error_code" => "login_failed", "error_text" => "Wrong login or password!"];

      $additional_headers[] = 'HTTP/1.1 401 Unauthorized';

      return;
    }

    $response_data["result"] = "success";

    $response_data["user"] = [
      "first_name"  => "John",
      "Last_name"  => "Smith‟
    ];
  }
} // LoginHandler
```

The *JsonApiRequestHandler* uses the map for setting the handlers to the API endpoints. No expensive regular expressions are used. Therefore, the *JsonApiRequestHandler* is fast in comparison to other route solutions. However, you can still handle complex API endpoints be setting the default handler.

JsonApiRequestManager (part 2)

Registration (api/index.php):

```
singleton(ISessionManager::class)->startSession();

$rmanager = singleton(JsonApiRequestManager::class);

$rmanager->registerApiRequestHandler("login", "MyApplication\\Handlers\\LoginHandler");

$rmanager->handleApiRequest();
```

Additionally, you can register:

Default Handler

It will be called if no handler mapping is found for a route.

Pre process handler

If it is defined, it will be called before each processing. You can setup whether standard processing should be done after. It is useful for setting the maintenance mode.

Post process handler

If it is defined, it will be called after each processing (also after errors).

XmlApiRequestManager

This class offers the solution for the development of the API based exchange of the XML data packets. There is a folder *xmlapi* with the file *index.php* which handles XML API requests. For handling the requests you have to create a class implementing the interface *IXmlApiRequestHandler* and register it to the request name:

```
class RoomHandler implements IXmlApiRequestHandler
{
  public function handle($rmanager, $api_request, $xmldoc)
  {
    $xsdpath = new \DOMXPath($xmldoc);
    $nodes = $xsdpath->evaluate("/Request/City");

    if($nodes->length == 0)
    {
      $response_data["errors"] = [
        ["error_code" => "no_city", "error_text" => "City is undefined!"]
      ];

      $rmanager->reportErrors($response_data);

      return false;
    }

    $city = $nodes->item(0)->nodeValue;

    ... continued on the next page
```

XmlApiRequestManager (part 2)

```
    ... continued from the previous page

    $outxmldoc = new \DOMDocument("1.0", "UTF-8");
    $outxmldoc->formatOutput = true;

    $response = $outxmldoc->createElement("Response");
    $outxmldoc->appendChild($response);

    $node = $outxmldoc->createElement("City");
    $response->appendChild($node);
    $text = $outxmldoc->createTextNode($city);
    $node->appendChild($text);

    $rooms = $outxmldoc->createElement("Rooms");
    $response->appendChild($rooms);

    $node = $outxmldoc->createElement("Room");
    $node->setAttribute("Price", 100);
    $rooms->appendChild($node);
    $text = $outxmldoc->createTextNode("Single");
    $node->appendChild($text);

    $node = $outxmldoc->createElement("Room");
    $node->setAttribute("Price", 200);
    $rooms->appendChild($node);
    $text = $outxmldoc->createTextNode("Double");
    $node->appendChild($text);

    $rmanager->sendXMLResponse($outxmldoc);

    return true;
  }
} // RoomHandler
```

XmlApiRequestManager (part 3)

Registration:

```
$rmanager = singleton(HotelXmlApiRequestManager::class);

$rmanager->registerApiRequestHandler("GetRooms", "MyApplication\\Hotel\\RoomHandler");

$rmanager->handleApiRequest();
```

PS

The class *XmlApiRequestManager* is abstract. You have to derive your class form it and implement the following methods:

```
protected abstract function parseXML(&$api_request, &$xmldoc);

public abstract function reportErrors($response_data, $headers);
```

The method *parseXML* should parse the incoming XML data and define which *api_request* should be called. The *api_request* is usually defined by the checking of a Tag in the XML data package. The parameter *xmldoc* should be assigned with the XML document. In this method, you can also handle the way how the data is coming – as RAWPOSTDATA or as a value of a post variable.

The method *reportErrors* is called in the case of any error and should respond with an XML data package containing the information of the error. The structure of this XML data package depends on the concrete implementation. The parameter *response_data* contains the array of errors and might contain additional information that is placed there in the user handle function.

Form fields functions

The *SmartFactory* provides a set of functions for placement of the form fields values of which can be automatically bound to the request variables with the corresponding name. These are:

```
input_text($params, $echo = true);
input_password($params, $echo = true);
input_hidden($params, $echo = true);
textarea($params, $echo = true);
select($params, $echo = true);
checkbox($params, $echo = true);
radiobutton($params, $echo = true);
```

The *params* are the array of the attributes in the form key = value. Example:

```
<?php input_text(["id" => "data_input_text",
                  "name" => "data[input_text]",
                  "value" => "value",
                  "class" => "my_class",
                  "style" => "width: 300px",
                  "placeholder" => "enter the data",
                  "title" => "enter the data",
                  "data-prop" => "some-prop",
                  "onblur" => "alert('Hello & Buy!')"
                  ]); ?>
```

If the *value* attribute is omitted, than the element value is set to that of the corresponding request variable.

Per default, the code is echoed directly but you can get it also as a string, e.g. to pass it as the HTML piece to another function.

Form fields functions (part 2)

Some elements have special parameters. There are:

select

```
<?php
$options = [
   "yellow" => "Yellow",
   "blue" => "Blue",
   "red" => "Red"
];
?>

<?php select(["id" => "data_multiselect",
              "name" => "data[multiselect][]",
              "multiple" => "multiple",
              "options" => $options
             ]);
?>
```

checkbox

```
<?php checkbox(["name" => "data[checkbox]",
                "checked" => true
               ]); ?>
```

radiobutton

```
<?php radiobutton(["name" => "data[radiocolor]",
                   "checked" => true
                  ]); ?>
```

Table generator

The *SmartFactory* provides a useful function for creating a table from an array:

```
table(&$array, $params = [], $echo = true);
```

The parameter *array* should contain the set of rows. It can be either directly echoed or returned as HTML string. The *params* give additional abilities to control the table rendering.

```
$captions = [
  "name" => "Task name",
  "employee" => "Employee",
  "time_estimation" => "Time estimation",
  "deadline" => "Deadline",
  "comments" => "Conmments“
];

$formatter = function ($rownum, $colnum, $colname, $val) {
  if($colname == "time_estimation") return format_number($val, 2);
  if($colname == "deadline") return date(text("DateTimeFormat"), $val);

  return $val;
}?>

<?php table($rows,
            ["captions" => $captions,
             "class" => "my_table",
             "style" => "background-color: #dddddd",
             "col_class_from_keys" => true,
             "no_escape_html" => false,
             "formatter" => $formatter
            ]); ?>
```

Short functions

To avoid large code constructions, some frequently used ones are packed into short functions. These are:

```
function text($text_id, $lng = "", $warn_missing = true, $default_text = "")
{
  return singleton(ILanguageManager::class)->text($text_id, $lng, $warn_missing, $default_text);
}

function messenger()
{
  return singleton(IMessageManager::class);
}

function session()
{
  return singleton(ISessionManager::class);
}

function debugger()
{
  return singleton(IDebugProfiler::class);
}

function debug_message($msg)
{
  return singleton(IDebugProfiler::class)->debugMessage($msg);
}


function event()
{
  return singleton(IEventManager::class);
}
```

Short functions (part 2)

```
function config_settings()
{
  return singleton(ConfigSettingsManager::class);
}

function runtime_settings()
{
  return singleton(RuntimeSettingsManager::class);
}

function user_settings()
{
  return singleton(UserSettingsManager::class);
}

function sql_error($dbw)
{
  messenger()->setError(text("ErrQueryFailed"),
                        $dbw->get_last_error() . "\n\n" .
                        $dbw->get_last_query()
                       );
  return false;
}
```

Next steps

One of the goals of this presentation was to give you an overview over the possibilities and functions of the library *SmartFactory*.

To get familiar with the *SmartFactory* do the following:

- View and study the API documentation in the folder docs or here API documentation.
- Study the core code of the library *SmartFactory*.
- Study the Demo Application that shows usage of the *SmartFactory*.

Have fun with using of the *SmartFactory*!

How to start

1) Git-clone the demo application *SmartFactoryDemo* and run 'composer update'.
2) Study the directory structure of the demo application and the code.
3) Implement your classes and functions.
4) Bind you classes to the interfaces in the file *intialization_inc.php* to be able to use the IoC approach for creating objects offered by the library *SmartFactory*.
5) Implement you business logic in the root directory or any subdirectory.
6) Implement the API request handlers for JSON or XML requests if necessary.
7) Add translation texts for your application over the *localization/edit.php* or directly into the XML file *localization/texts.json*. Use the script *localization/check.php* to check your localization texts for missing translations.

Directory structure of the application using SmartFactory

```
config
logs
application
   api
   localization
   src
   tests
   xmlapi
database
```

Recommended directory structure of the application using SmartFactory

config
This directory contains the configuration files. This folder is outside of the access per http(s).

logs
This directory is used for logging, debugging and tracing. This folder is outside of the access per http(s).

application
This is the root directory of the application.

application/api
This directory contains the processor *index.php* of the JSON API requests.

application/localization
This directory contains the translation file *texts.json* and the editor *edit.php* for user friendly editing of the translation texts, and the file *check.php* for checking the localization texts for missing translations.

application/src
This is the root directory for all code sources.

application/tests
This directory contains the test units.

application/xmlapi
This directory contains the processor *index.php* of the XML API requests.

database
This directory contains the SQL scripts for creation of the database for the demo application.