Booosta Framework 3.0

Documentation

Peter Buzanits

Version 0.1

Inhaltsverzeichnis

The Booosta PHP Framework

What is Booosta?

Booosta is a Framework for the fast development of PHP web applications. It provides powerful classes and functions that make life for web developers easier. It is licensed under the GPL v3 which means it is free software. Some external parts may be licensed under GPL v2 or other open source licenses.

Booosta is an ERAD (extreme rapid application development) environment, which provides basic MVC functionality (Model, View, Controller).

Booosta provides modules for database abstraction, database access through data objects, a web application infrastructure, a template system, an Ajax helper class, sending emails or an image gallery and many others.

It contains several external open source PHP and JavaScript projects like modules from the PEAR repository or jquery. These are provided with the Booosta release.

How does Booosta compare to the big PHP frameworks like Symfonie or Zend?

Those frameworks are very large ones that provide functionality for developing huge web applications. If you want to implement your own social network you should consider using one of these instead of Booosta. Booosta has been designed to develop small and mid-sized web applications very fast, easy and efficient.

KISS (keep it simple, stupid) has been a major design goal of Booosta. The time you use to set up one of the large frameworks for use in your application is the time you need to implement your whole application with Booosta – of course if it is really a very, very simple application that does nothing but displaying, inserting, editing and deleting database contents. More complex applications still need time to be developed ;-).

Getting started with Booosta

For using Booosta you have to download the release and copy it into the root directory of your web app. You have to include `lib/framework.incl.php` in your PHP files. All modules and other include files you need are automatically included in the `framework.incl.php`.

Requirements

PHP 5.4 is required for using Booosta since version 3.0.

Installation

1. Download and unpack the framework in your webspace.
   `tar xzf booosta-3.x.x.tgz`
2. Create a database user and a database
3. Start your favourite web browser and point it to the address of your webspace to start the installer.
4. Fill in all the informations the installer asks from you.

That are basically all steps you have to do. The installer sets up your installation with a working user system including user administration and the ability to log in and out with valid users. Further information can be found in the chapter about the module usersystem.

Structure of Booosta 3

The Booosta Framework is a complete object oriented PHP framework. If you are not familar with writing object oriented code, don't be afraid. You do not need to know very much about object orientation if you use Booosta.

The basic Booosta Framework only consists of a small amount of code and a basic class named Base. Almost every functionality of Booosta is provided by modules. Booosta comes with a variety of pre-installed modules. Additional modules can be easily installed by just copying the new module into a specific directory.

After installation your webspace has the following directory structure:

`lib/`	all Booosta code and modules reside here
`local/`	local configuration of your installation
`tpl/`	your templates are going here
`systpl/`	Booosta system templates
`run`	a wrapper script to run module code on the commandline
`lang.de`	an example language file for german language
`index.php`	start file – you can replace it by your own version

You can maintain one central installation of Booosta for easier updates. In this case, you have to symlink the directories lib and systpl and the file run into the webspaces. The remaining directories and files have to be copied or created in every webspace that uses Booosta.

Database structure

Booosta installs some tables in the database. Obviously you cannot use these table names for your own tables. These tables are:

```
`adminuser` , `adminuser_privilege` , `adminuser_role` , `cache` ,
`form_token` , `privilege` , `role` , `role_privilege` , `role_role` ,
`settings` , `user` , `user_privilege` , `user_role`
```

These tables are basically used by the modules usersystem, cache and webapp. See the documentation of these modules which tables are used and which you can delete if you do not use it.

For your application you will need you own database tables. To get the maximum benefit out of Booosta your tables should meet some guidelines:

- a primary key
- an autoincrement or similar technique on the primary key
- a foreign key for all fields pointing to records in other tables
- fields holding boolean values should be of type tinyint in mysql

In Mysql foreign keys can only be realized using the Innodb engine. Tables in the MYISAM engine cannot have foreign keys. So it is suggested to always use the Innodb database engine.

Booosta programming basics

The programming of Booosta is done by extending predefined classes and „starting“ those classes afterwards. Here a simple example that uses the module Webapp:

```
<?php
include_once 'lib/framework.incl.php';

class myApp extends \booosta\webapp\Webapp {
  protected function action_default() {
     $this->TPL['firstname'] = 'Claire';
     $this->maintpl = 'tpl/hello.tpl';
  }
}

$app = new myApp();
$app();
?>
```

The Booosta Webapp module works with templates. All output is rendered with templates, that contain markers which will be replaced by the application. In the above example you could have a template named `hello.tpl` in your tpl-directory looking like that:

```
<h1>Greeting</h1>
Hello, {%firstname}!
```

Try this with your brand new Booosta installation. Save the above script as `greeting.php` and the template as `tpl/hello.tpl` . When you now call `http://`*`yourwebserver`*`/greeting.php` you should see `Hello, Claire!` in the main area of the web page.

When you create new objects from Booosta classes, you should use the `makeInstance` method instead of the PHP new keyword:

```
$parser = $this->makeInstance('templateparser');
```

This has several advantages. For example, inside the new object you can access the object that created it whith `$this->parentobj` . You also can do this with your self defined classes, if they are derived from the Booosta Base class (or from a child class of it).

Booosta modules

As stated above, nearly all Booosta functionality comes from its modules. There are modules providing database access, modules doing encryption, modules for fancy HTML/Javascript-Elements. The creation of a web app is done by a module and the templates this app uses are also parsed by a module.

Here we introduce and describe the modules delivered with Booosta.

Webapp

As the creation of web apps is the main purpose of the Booosta Framework, this module can be considered as the most important one. The Webapp module provides the functionality for creating, editing and deleting records of a database table, so you do not need to implement those functions yourself.

Usersystem

The module usersystem extends the functionality of webapp and therefore requires that this module is also installed. Usersystem is a powerful module that provides two types of user accounts: admin accounts and normal user accounts. It also provides mechanisms to manage those accounts including user self registration, assign roles and privileges to them and check these privileges in the application.

Templateparser

The module templateparser provides a powerful engine to parse and display web content based on templates. Templates are basically HTML files with special markups in it that are replaced by content from the application at runtime.

Formelements

This module provides several classes that represent HTML form elements or a combination of form elements. You can work with objects of these classes in your code and then output the generated HTML code of the element.

Translator

The translator module provides mechanisms to make your webapp multi lingual.

Database

The database module is a generic class that provides basic components for other database modules. It provides the method init_db() and the variable $this->DB for every class that is derived from the Booosta base class. Connections to a database and queries on it must be provided by other modules that are designed for special database systems like MySQL or Postgresql.

Mysqli

Mysqli is a database module that uses the mysqli PHP extension. This module provides methods to access a MySQL database. This module requires the module database.

Dataobjects

Dataobjects is a module that provides classes that represent a database record. Each object from these classes holds the data of one row of the according database table and has the methods to manipulate them in the database.

mkfiles

The module mkfiles provides command line scripts to automatically create necessary PHP and template files for manipulating data in an existing database table.

Webapp

As the creation of web apps is the main purpose of the Booosta Framework, this module can be considered as the most important one. The Webapp module provides the functionality for creating, editing and deleting records of a database table, so you do not need to implement those functions yourself.

All of those standard functions provide hooks, where you can alter the behavior of these functions. Hooks are implemented by just defining the hook function in your class which you have derived from `\booosta\webapp\Webapp`.

For the MVC fans among you this classes provides the controller part.

A very quick start to set up an application with Webapp is to use another Booosta module: mkfiles

mkfiles provides a commandline script that creates a simple web app based on an existing database table including the PHP script and all necessary templates. Let's say you have a database table „customer“. To create an application where you can create, edit and delete records in this table, you go to your webspace main directory and type:

```
./run mkfiles customer
```

As you will see, you have got some new files in your webspace! There is a `customer.php` and in the tpl directory there are the templates `customer_default.tpl`, `customer_new.tpl` and `customer_edit.tpl`. When you now call `http://`*`yourserver`*`/customer.php` you can immediately start adding, editing and deleting records. If your web application is ment to do nothing but this, it is already finished ;-)

Let's take a closer look into the files that have been created by the mkfiles module…

customer.php:

```
<?php
include_once 'lib/framework.incl.php';

class App extends booosta\usersystem\Webappadmin
{
  #protected $fields = 'name,edit,delete';
  #protected $header = 'Name,Edit,Delete';
  #protected $checkbox_fields = '';
}

$app = new App('customer');
$app->auth_user();
$app();
?>
```

This script extends the class Webappadmin from the Usersystem module. More information about this module can be found in the documentation of this module. Basically it is a web app that is run by an admin user.

Inside the class there are some commented out lines, that can be manipulated and activated by removing the # at the beginning of the line. It tells the Webapp module which fields of the database table to display and which headers to use in the HTML table for them. If you have large tables you probably do not want to display them all in the list of records.

The third line is a list of database fields that are represented by a checkbox field in the HTML templates. Since the early days of the WWW there is a problem that browsers have with HTML checkboxes: If the checkbox is unchecked when submitting the form, the corresponding variable is not sent to the server with „0“ or „no“ or „off“. It is just omitted. So the application does not get any values. This is a problem when automatically processing all submitted variables. So you have to tell the Webapp module which fields are checkbox fields.

The first line after the class definition creates an instance of this class and provides the name of the database table to work on as parameter. The next line tells the object to check, if a valid user ist logged in to the web application at that time. As the object is of class Webappadmin, this has to be an admin user.

If you do not want to use any user authentication, you can delete this line. Then you also could derive your class from `\booosta\webapp\Webapp`. The last line executes the run method of the object. `$app()` is just short for `$app->run()` here.

If you play around a little bit with this app in your browser, you will notice in the address line that it passes the variable `action`, if you go to the pages to create, edit or delete records. This parameter determines, which internal method is called. This method is `action_`*`actionname`*. For example if you pass `action=edit`, the run method calls the internal method `action_edit`. If there is no action parameter, the method `action_default` is called.

This is important to know, because you can override every function in your class. If you want a different behavior of your app when the action parameter ist set to edit, just define the method action_edit in your class:

```
protected function action_edit() { … }
```

Of course you can define your own actions and define the corresponding action method. If you pass `action=statistics`, then define the method `action_statistics`!

As you probably know, creating and editing records always have two steps: First show a form to fill in the desired data and second to save this data in the database. So additionally to `action_new` and `action_edit` we have two other methods: `action_newdo` and `action_editdo`.

For deletion, there are also two steps: First ask the user to confirm the deletion and second remove the record from the database. These two steps are done in the methods action_delete and `action_deleteyes`. The latter one is called when the web user clicked on „Yes“ at the confirmation page and the action parameter is `deleteyes`.

Hooks

Often you do not want to completely change the behavior of the web app but add only some additional functionality to it. For example send an email, when an new record is created. Then it would be a pain to override the method action_newdo and implement the whole logic to save the record to the database yourself.

This is where hooks come into the game. Hooks are methods doing nothing and are called at the beginning and the end of all predefined methods. Of course they do nothing as long as you have not overridden them. When you override them, they do exactly what you told them to do in your code.

Here is a list of the available hook functions:

`before_default()`	`runs at begin of action_default()`
`after_default()`	`runs at end of action_default()`
`in_default_makelist($list)`	`Runs after creation of the record list in action_default()`
`before_action_new()`	`Runs at begin of action_new()`
`after_action_new()`	`Runs at end of action_new()`
`before_action_newdo()`	`runs at begin of action_newdo()`
`after_action_newdo()`	`runs at end of action_newdo()`
`before_action_edit()`	`runs at begin of action_edit()`
`after_action_edit()`	`runs at end of action_edit()`

`before_action_editdo()`	`runs at begin of action_editdo()`
`after_action_editdo()`	`runs at end of action_editdo()`
`before_action_delete()`	`runs at begin of action_delete()`
`after_action_delete()`	`runs at end of action_delete()`
`before_action_deleteyes()`	`runs at begin of action_deleteyes()`
`after_action_deleteyes()`	`runs at end of action_deleteyes()`
`before_add_($data, $obj)`	`runs before record is added to the DB`
`after_add_($data)`	`runs after record has been added to the DB`
`before_edit_($id, $data, $obj)`	`runs before record is edited in the DB`
`after_edit_($id, $data)`	`runs after the record has been edited in the DB`
`before_delete_($id)`	`runs before record is deleted from the DB`
`after_delete_($id)`	`runs after record has been deleted from the DB`

You might be confused about the hook function `in_default_makelist($list)`. The method `action_default()` generates an object of type `\booosta\tablelister\Tablelister`. In that hook you can modify this object available in the `$list` parameter. See the documentation of the corresponding module tablelister for details about this class.

As stated, these hook functions do nothing by default. You need to override them in your class. This can be accomplished by just defining a method with the corresponding name in you class:

```
…
class myApp extend \booosta\webapp\Webapp {
  protected function before_add_($data, $obj) {
    // do something like manipulating $obj or so...
…
```

The parameters have the same meaning in every hook function: `$id` holds the content of the ID field that is worked on. `$data` is an array holding the data transmitted by the web form and `$obj` is an object of type `\booosta\dataobject\Dataobject` which you can manipulate. All the data that `$obj` holds after the end of the hook function ist stored in the database.

An example for manipulating the `$obj` object would be:

```
protected function before_add_($data, $obj) {
  if($obj->get('price') < 0) $obj->set('price', 0);
}
```

In this example the database field `price` of a new inserted record is set to 0 if a negative value is submitted by the web form.

Foreign keys – sub data

Often you have database tables that contain records with a relation to records in another table. For example you have departments of a company and employees in this company. Every employee works in one department. So every record in the employee table has a relation to a record of the department table.

Database designers call this a foreign key. The employee table has a field which contains the ID of the corresponding department in the department table. In most database systems like MySQL or Postgresql you can define such foreign keys in the database. See the manuals of these databases for further information.

Booosta uses those foreign keys in several ways. First it provides a select field in the new- and edit-templates created with the mkfiles module. In the above example the templates for creating and editing employees would have a select with all departments in the database.

Second the edit page of the supertable (in our example department) will contain a list of all corresponding sub records. In our example a list of all employees of that department. This list is at the end of the page.

If you use the mkfiles module for the generation of your script and templates, you must tell the mkfiles script which sub and super tables belong to the table you work on. There are two possible syntaxes:

```
./run mkfiles tablename subtablename supertablename
./run mkfiles table=tablename subtable=subtablename supertable=supertablename
```

Applied to our previous example the webapp consisting of department and employee would be built with:

```
./run mkfiles table=department subtable=employee
./run mkfiles table=employee supertable=department
```

Data in the Webapp class
There are several data fields in the webapp class. Inside your class derived from Webapp you can access them with `$this->`*`name`* `or you can set them in the definition of the class variables.`

`$VAR`	An array with all the POST and GET variables that have been passed to the PHP script.
`$TPL`	An array that contains all the variables that are available to the template engine.
`$maintpl`	The name of the main template file. Just set this variable to load the according file as main template in the template engine.
`$idfield`	This variable contains the name of the primary key column in the database. It defaults to '`id`'.
`$id`	This variable contains the content of the parameter in `VAR` with the primary key column.
`$self`	This variable contains the name of the current PHP script.
`$goback`	If this variable is set to true and the system template `systpl/feedback.tpl` is loaded than the users browser is redirected to the page set in `$this->backpage`. Otherwise it displays the

	message in `$this->TPL['output']` and a link to `$this->backpage` is displayed.
`$backpage`	see description at `$this->goback`
`$name`	Holding the name of the database table the class is working on
`$supername`	The name of the database table where a foreign key points to. This can be an array with several names.
`$subname`	The name of the database table having a foreign key to the current table. This can be an array with several names.
`$superscript`	Name of the script that manages the supertable. Defaults to `$supername.php`
`$subscript`	Name of the script that manages the subtable. Defaults to `$subname.php`
`$tpldir`	The directory where the template folder `tpl` is found. Defaults to .
`$extra_templates`	Extratemplates to pass to the template parser. See docu of the templateparser module for further information.
`$preincludes`, `$includes`	Code that is included in your template where the {%_includes} tag is.
`$keyfilter` `$fkeyfilter`	Filter strings for the datatable object created in the action_default method. See the docu of the class datatable for further information
`$fields`	List of database fields to be displayed by action_default. Can be a comma delimited string or an array.
`$nfields`	List of database fields not to be displayed by action_default. If $fields is not set and $nfields is set, all other fields are displayed. Can be a comma delimited string or an array.
`$default_clause`	SQL where clause that limits the displayed records in action_default.
`$default_order`	SQL order clause that determines the sort order of the displayed records in action_default.
`$sub_default_clause`	SQL where clause that limits the displayed records of the subtable in action_edit.
`$sub_default_order`	SQL order clause that determines the sort order of the displayed records of the subtable in action_edit.
`$header`	Headers of the HTML table that displays the records in action_default. Can be a comma delimited string or an array.
`$condition`	Condition for the display of fields in the record list of action_default. See the documentation of the module tablelister for further info.
`$autoheader`	Flag that determines if the datatable in the record list should have an autogenerated header. Only useful with $use_datatable=true.
`$sub_keyfilter` `$sub_fkeyfilter` `$sub_fields`	These have the same meaning as the variables without the sub_ prefix, but are valid for the sub table list(s). All can be comma delimited strings or an array.

`$sub_nfields`
`$sub_header`
`$sub_condition`	

`$use_datatable`	Display the list of records with the module datatable instead of tablelister. See further information in the documentation of these modules.
`$editvars` `$addvars`	When editing or adding a record, do not process all passed variables, but only those in $editvars and $addvars. Can be a comma delimited string or an array.
`$lang`	Language symbol (two characters, e. g. en)
`$no_output`	Boolean value. When set to true, the class does not generate any output. Useful if you just want to output plain values with print and do not want to have any templates parsed.
`$use_form_token`	Use CSRF protection with form tokens
`$form_token_table`	Database table where the form tokens are stored. Defaults to form_token
`$form_token_time`	Timeout in seconds when a form token becomes invalid. Defaults to 1800.
`$pass_vars_to_template`	A list of variables passed to the script that are available in the template. Can be a comma delimited string or an array. If this variable is set to the boolean value true, all variables are passed to the template. Do this only if you really know what you do, because this can be a major security risk! You have been warned!
`$encode` `$decode`	Array which determines, which fields should be encoded before storage in the database and decoded after reading it from the database. The keys of this array are the field names and the value a function name. This is useful when you encrypt passwords in the database for example.
`$error`	If an error occurred, this holds the error message.

`$cancel_insert`
`$cancel_update`
`$cancel_delete`
	In the hook functions that run before the physical insert, update or delete of a database record, this values can be set to true. Then the insert, update or delete will not be performed.

`$foreign_keys`	A list of foreign keys of the current database table. Can be a comma delimited string or an array. If it is an array, the keys of the array are the field that is the foreign key and the value is the table that is referenced by this foreign key.
`$sub_foreign_keys`	Same as $foreign_keys, but for the subtable(s)

Methods in the Webapp class
The set_* functions set the according data variable in the object. For the meaning of the variable please see the docu of the class data above…

`set_keyfilter($filter)`	Sets the variable `$keyfilter`

set_fkeyfilter($filter)	Sets the variable $fkeyfilter
set_default_clause($c)	Sets the variable $default_clause
set_default_order($o)	Sets the variable $default_order
show_fields($fields)	Sets the variable $fields
hide_fields($fields)	Sets the variable $nfields
set_header($header)	Sets the variable $header
set_conditions($cond)	Sets the variable $condition
add_condition($cond)	Adds a value to $condition
set_sub_key_filter($f, $i)	Sets the variable $sub_key_filter for the subtable $i. $i is the zero based number of the subtables defined in the class.
set_sub_fkey_filter($f, $i)	Sets the variable $sub_fkey_filter for the subtable $i. $i is the zero based number of the subtables defined in the class.
set_sub_default_clause ($clause, $i)	Sets the variable $sub_default_clause for the subtable $i. $i is the zero based number of the subtables defined in the class.
set_sub_default_order ($clause, $i)	Sets the variable $sub_default_order for the subtable $i. $i is the zero based number of the subtables defined in the class.
show_sub_fields($fields,$i)	Sets the variable $sub_fields
hide_sub_fields($fields,$i)	Sets the variable $sub_nfields
set_sub_header($header, $i)	Sets the variable $sub_header
set_sub_conditions ($conditions, $i)	Sets the variable $sub_condition
add_sub_condition($key, $condition, $i)	Adds a condition to the variable $sub_condition
set_idfield($field)	Sets the variable $idfield
set_checkbox_fields($field)	Sets the variable $checkbox_fields
set_blank_fields($fields)	Sets the variable $blank_fields
set_lang($lang)	Sets the variable $lang
get_lang()	Gets the value of the variable $lang
set_extra_templates($tpl)	Sets the variable $extra_templates
pass_vars_to_template($var)	Sets the variable $pass_vars_to_template
set_foreign_keys($fk)	Sets the variable $foreign_keys

`set_supername($name)`	Sets the variable `$supername`
`set_superscript($script)`	Sets the variable `$superscript`
`set_sub_foreign_keys($f,$i)`	Sets the variable `$sub_foreign_keys`
`set_subname($name, $i)`	Sets the variable `$subname`
`set_subscript($script, $i)`	Sets the variable `$subscript`
`add_foreign_key($key, $val)`	Adds a value to the variable `$foreign_key`
`add_sub_foreign_key($key, $value, $i)`	Adds a value to the variable `$sub_foreign_key`
`add_encode($field, $func)`	Sets an encode function for the field `$field`
`add_decode($field, $func)`	Sets an decode function for the field `$field`
`run()`	Starts the execution of the webapp code
`get_dbobject($id)`	Gets an object of class dataobject with the content of the current record.
`translate_list($list)`	Translates all values in the array `$list`
`redirect($url)`	Sends a redirection header to the browser
`raise_error($message, $bp)`	Puts out the error message `$message` in the main area of the template and stops execution of the code. The optional parameter `$bp` gives the backpage to jump to when clicked on „Back“
`add_includes($code)`	Sets the variable `$includes`
`add_preincludes($code)`	Sets the variable `$preincludes`
`add_javascript($obj, $tags)`	Adds the javascript put out by `$obj` to the variable `$includes`.
`generate_form_token()`	Generates a form token for CSRF protection
`check_form_token($token)`	Checks a form token for validity
`clear_form_tokens()`	Deletes all old form tokens
`set_backpage($backpage, $action, $name)`	Sets the backpage for return after the action `$action` on table `$name` is performed
`set_editvars($vars)`	Sets the variable `$editvars`
`set_addvars($vars)`	Sets the variable `$addvars`

Configuration of webapp

The following setting can be set in `local/config.incl.php`:

`language` : default language

Usersystem

The module usersystem extends the functionality of webapp and therefore requires that this module is also installed. Usersystem is a powerful module that provides two types of user accounts: admin accounts and normal user accounts. It also provides mechanisms to manage those accounts including user self registration, assign roles and privileges to them and check these privileges in the application.

In the installer of Booosta you can choose if the web browser is redirected to the admin login or the user login from the main `index.php`. These login pages are in `lib/modules/usersystem/admin.php` and `lib/modules/usersystem/user.php`.

In the config file `local/config.incl.php` you can set if users can register themselves or not. If this is set to true, on the user login page appears a link to the user registration. In the config file you also define, if users have to confirm their email address by clicking on a link sent to their address.

There are several scripts dealing with user administration. In the default installation they are all linked in the admin users menu:

Manage privileges
`lib/modules/usersystem/admin_privilege.php`
Here you can create, edit and delete privileges. Some privileges are already predefined. You should alter or delete them only, if you really know what you do.

Manage roles
`lib/modules/usersystem/admin_role.php`
Roles are a powerful method for gathering several privileges together and assign them to a user. Here you can create, edit and delete roles. When editing a role, you can assign privileges and even other roles to them.

Manage admin users
`lib/modules/usersystem/admin_adminuser.php`
Here you can create, edit and delete admin users. When editing an admin user you can set the roles and privileges of this user.

Manage users
`lib/modules/usersystem/admin_privilege.php`
Here you can create, edit and delete users. When editing an user you can set the roles and privileges of this user.

Manage own settings
`lib/modules/usersystem/admin_self.php`
Here you can reset your own password.

This module adds the following classes to the module webapp:

`class Webappadmin extends \booosta\webapp\Webapp`
This is the class where all apps run by an adminuser should inherit from.

`class Webappuser extends \booosta\webapp\Webapp`
This is the class where all apps run by an user should inherit from.

`class adminuser extends \booosta\genericuser\Genericuser`
This is the user object that holds the currently logged in admin user.

`class user extends \booosta\genericuser\Genericuser`
This is the user object that holds the currently logged in user.

`class Admin_Authenticator extends \booosta\db_authenticator\DB_Authenticator`
The authenticator object for admin users

`class User_Authenticator extends \booosta\db_authenticator\DB_Authenticator`
The authenticator object for users

This module adds the following values to the module webapp:

`$user`	The user object of the currently logged in user. Is of class adminuser or user.
`$loginscript`	The script that is used for login.
`$auth_actions`	A boolean value, that determines if the actions view, new, edit and delete are only available to users with the according privileges. Does not apply to admin users.
`$privs`	Array with the necessary privilege for each action. The keys of the array are the actions (eg. Edit, delete) and the value is the name of the privilege a user has to have to perform this action.

This module adds the following methods to the module webapp:

`set_priv($action, $prv)`	Sets a privilege in `$privs`. `$action` is the action for that the privilege is set and `$prv` the name of the privilege.
`auth_user($user_class)`	Authenticates the logged in user. If no valid user is logged in, the browser jumps to `$loginscript`. `$user_class` is one of user or adminuser. In a webappadmin object it defaults to adminuser, in a webappuser object it defaults to user.
`auth($action)`	Checks, if the logged in user has the necessary privilege for the action `$action`.
`before_auth($action)`	Hook function that is executed before the check in auth()

Templateparser

The module templateparser provides a powerful engine to parse and display web content based on templates. Templates are basically HTML files with special markups in it that are replaced by content from the application at runtime. For those who love the MVC model this is the view part of it.

These markups („tags“) can be special tags that expand to HTML-Elements like `{LINK …}` or variables in the form of `{%varname}`. These variables can be set in the application (or the controller for the MVC freaks among you) with the array `$TPL`:

In app: `$this->TPL['message'] = 'hello';`
In template: `The message is {%message}`

This module defines the following classes:

```
class \booosta\templateparser\Templateparser extends \booosta\base\Module
class \booosta\templateparser\Tags extends \booosta\base\Base
```

\booosta\templateparser\Templateparser provides the following methods:

`set_tags($tags)`	Sets the name of the uses tags. There has to be a class with the name `$tags` that is derived from the class Tags. Valid names are `DefaultTags` and `HTML5Tags`.
`parse_template($tpl, $subtpls = null, $tplvars = null)`	Does the acutally parsing. `$tpl` is the template to use. It can be a filename or a string with the content of the template. `$subtpls` are the sub templates which are included in the main template with `##INCLUDE name##`. `$tplvars` is an array of variables, that are used as `{%varname}` in the template.

Valid Tags from DefaultTags are:

`{FORMSTART action.php}`	marks the beginning of the form. `action.php` is the file that is called by submitting the form.
`{FORMEND}`	marks the end of a form.
`{FORMSTARTN action.php name::myname}`	marks the beginning of the form with a particular name
`{FORMSTARTM action.php}`	like `FORMSTARTN`, but gives the form the enctype multipart/form-data
`{TEXT name value}`	Input element of type 'text'. It gets the name and the prefilled value provided.
`{TEXTM name value}`	Text-field that is mandatory.

`{HIDDEN name value}`	Input element of type 'hidden'.
`{PASSWORD name value}`	Input element of type 'password'.
`{PASSWORDM name value}`	Password input that is mandatory.
`{CHECKBOX name checked}`	Input element of type 'checkbox'. If the value of checked evaluates to true in PHP, the checkbox will be checked.
`{RADIO name thisvalue checkvalue}`	Input element of type 'radiobutton'. If the value of checkvalue is the same as of thisvalue, the radio button will be selected.
`{FILE name}`	Input element of type 'file'. You need a form with `enctype multipart/form-data` to use this for file uploads.
`{DATE name value}`	Displays a date selector field.
`{DATEINIT}`	For using the `DATE` element, you must place this element on your result page. This contains code to initialize the date selector.

`{TEXTAREA name cols rows`
`Default Text here}`
	Input element of type '`textarea`'. Number of columns and rows can be defined (optional). The default text that should be filled in has to start in the second line!

`{WYSIWYG name cols rows`
`Default Text here`
`}`
	Displays a WYSIWYG-Editor with the given text in it.

`{SELECT name defaultvalue size`
`[key1]value1`
`[key2]value2`
`…}`
	Input element of type 'select'. defaultvalue is the value that is selected by default. size is the optional parameter for the size of the select field. From the second line the content of the select starts.
	
	The `value`*`x`* tells the text that should be displayed in the select and den `key`*`x`* the value that will be sent by the form.

`{TABLESELECT name table default size showfield idfield}`	Shows a Select field that gets its data from the table `table`. `showfield` is the database field that is shown in the select. `idfield` the fields that is returned by the select after submit.
`{TABLESELECTN name table default size showfield idfield}`	Like `TABLESELECT`, but adds an empty (null) entry to the top of the select.
`{TABLESELECT0 name table default size showfield idfield}`	Like `TABLESELECT`, but adds an empty entry with value 0 to the top of the select.
`{NUMBERSEL name default x y}`	A select with numbers from `x` to `y` (including)
`{BUTTON name value}`	Shows a button with the specified value
`{FORMSUBMIT}`	Submit button for the form.
`{FORMSUBMITVAL value}`	Submit button that submits a special value.

`{FORMSUBMITPIC image}`	A form submitter picture.

`{IMG imagename}`	Image
`{LINK linktext target}`	Link with given text that links to target.
`{PICLINK imagefile target}`	A linked picture.
`{REDIRECT location}`	Javascript redirection.
`{LIST datavar}`	Shows a html table with the data found in an template variable named `datavar`. This variable has to contain an array.

HTML5Tags adds the following tags to DefautTags:

Those are HTML 5 form tags. Not every web browser supports every of these inputs.

`{COLOR name}`	Color selector in HTML 5
`{DATETIME name}`	Datetime input element
`{DATETIMEL name}`	Datetime-local input element
`{EMAIL name value size}`	Input field for e-mails
`{MONTH name}`	Input field for months
`{NUMBER name value min max}`	Input field for numbers with default value and minimal and maximal allowed values
`{RANGE name value min max}`	Input field for ranges with default value and minimal and maximal allowed values
`{TEL name}`	Input field for telephone numbers
`{TIME name}`	Input field for time
`{URL name}`	Input field for URLs
`{WEEK name}`	Input fields for weeks

Template File

The template file is a simple HTML file. Everything in this file is sent to the browser as it is. As our web applications are not only static HTML pages, there are variables that can be used inside this files. A variable is noted in the form:

```
{%varname}
```

The template engine parses this and replaces this with the content of the according variable passed to the engine (see later here how to do that).

You can use `if`, `for` or `while` with a leading `%` in the line (must be the first character of the line):

```
%if({%age} < 18):
  {%firstname} {%lastname} is an underager!<br>
%else:
  {%firstname} {%lastname} is {%age} years old and therefore adult<br>
%endif;
```

You can use local variables with `%%` prefix:

```
%for({%%i}=1; {%%i} <= 10; {%%i}++):
  Line number {%%i}<br>
%endfor;
```

Include Subtemplates

Often it is useful to use subtemplates inside a main template. For example you can have a basic layout for your web application that is the same in all of your pages and load the output of your application into a designated area of the browser window. You can define the basic layout in the main template and load subtemplates into a `<div>` inside this main template, which can be positioned with CSS.

You define the place where the output of the subtemplate should be put by the code:

```
##INCLUDE TPLNAME##
```

where `TPLNAME` is the name of the subtemplate.

To invokes the template engine and you have to call the following function in your webapp:

```
$output = $parser->parse_template($maintemplate, $subtemplates, $variables);
```

`$maintemplate` is the template file for the main template. `$subtemplates` is an array of subtemplates. The indexes of this array are the names of the subtemplates, that is used in the `##INCLUDE` statement. `$variables` is an array with the variables that are used in the template with the `{%varname}` notation. The indexes of this array are the variable names and the values the according values.

Here is a simple example of a template file and the usage of `parse_template()`:

maintemplate.tpl:

```
<html>
<body>
<h1>Welcome to my web application!<h1>

<div id='content'>
```

```
##INCLUDE MYSUBTPL##
</div>

</body>
</html>
```

subtemplate.tpl:

```
Hello, my name is {%firstname} {%lastname} and I am {%age} years old!
```

```
index.php:

$data = C_user::get_object(1);
$TPL['firstname'] = $data->get('firstname');
$TPL['lastname'] = $data->get('lastname');
$TPL['age'] = $data->get('age');

print parse_template('maintemplate.tpl', array('MYSUBTPL'=>'subtemplate.tpl'),
$TPL);
```

Formelements

This module provides several classes that represent HTML form elements or a combination of form elements. You can work with objects of these classes in your code and then output the generated HTML code of the element.

The following classes are currently available:

\booosta\formelements\Select

This is a class that creates a HTML form select element.

```
$select = $this->makeInstance('Select', $name, $options, $selected);
```

`$name` is the name of the element. This is also the name of the POST variable that will be sent to the server after the according form is submitted. `$options` is an array of the available options in the select. The keys of the array are the values that will be sent after a form submit. The values of the array are the values displayed in the select. `$selected` is the key of the element of `$options` that will be preselected when the select is shown.

Example:

```
$options = array('A' => 'apple', 'B' => 'banana', 'C' => 'lemon');
$select = $this->makeInstance('Select', 'fruit', $options, 'B');
$this->TPL['fruits'] = $select->get_html();
```

In the above example the HTML select can be inserted in the template with `{%fruits}`. It shows a select with the three visible entries apple, banana, lemon where banana is preselected. If the form is submitted the character A, B or C are submitted in the variable 'fruit'.

This class provides the following methods:

`set_size($size)`	Sets the size of the select. Default is 1
`set_multiple($flag)`	Sets a boolean that determines if multiple options can be selected.
`get_multiple()`	Gets a boolean that tells if multiple options can be selected.
`set_formsubmitter($flag)`	Sets a boolean that determines if the select should submit its form when changed
`set_formsubmitcode($code)`	Changes the code that submits the form via javascript
`set_extra_attr($attr)`	Sets extra attributes for the HTML select tag
`set_onchange($code)`	Sets the javascript code that is executed when changing the select
`add_top($elements)`	Adds elements on the top of the select. `$elements` can be a scalar value or an array for adding several elements
`add_bottom($elements)`	Adds elements on the bottom of the select. `$elements` can be a scalar value or an array for adding several elements
`get_html()`	Gets the HTML code of the select

There are three classes that provide special extensions to the Select:

NumberSelect

```
makeInstance('NumberSelect', $name = null, $maxval = 10, $minval = 0, $default =
null, $size = null, $multiple = null)
```

This creates a Select object, that shows numbers between $minval and $maxval.

YesNoSelect

```
makeInstance('YesNoSelect', $name = null, $default = null, $lang = null)
```

This creates a Select object with the values Yes and No in the given language `$lang`. For using different languages please see the docu of the module language. When the form is submitted, the value 1 (yes) or 0 (no) is sent to the server.

JSelect

This creates a Select that works like the normal Select. Additionally you can add or remove elements to/from the Select and select elements in it via javascript.

This class provides the following additional methods:

`set_onchange($code)`	Sets the javascript code that is executed when changing the select.
`set_id($id)`	Sets the ID of the select. You need this ID to call the according javascript functions.
`get_id()`	Gets the ID of the select. If you do not use `set_id()` the ID is generated dynamically. You can get it with this method.
`get_html()`	Gets the HTML code of the select

These javascript functions are provided:

`ID``_addoption(caption, value)`	Adds an option to the select *ID*
`ID``_rmoption(num)`	Removes the option number num from the select *ID*
`ID``_select_key(key)`	Selects an option via key key in the select *ID*
`ID``_select_text(text)`	Selects an option via text text in the select *ID*

\booosta\formelements\Shiftbox

A `Shiftbox` is a combination of two multi row selects with some buttons inbetween to shift elements from one to the other. It is used to select several options out of some available ones. In the left select there are the selected options, in the right one the unselected.

When the form is submitted a variable named `shiftbox_`*`name`*`_sellist`. *name* is the name of the shiftbox as declared in the constructor. The key values of the array that are selected are then filled into this variable seperated with a hyphen (-). You need to avoid the hyphen in the keys of course for not breaking this.

```
$box = $this->makeInstance('shiftbox', $name, $options_selected, $all_options);
```

$name is the name of the element and determines the name of the POST variable sent to the server after the form is submitted which will be `shiftbox_`*`name`*`_sellist`. `$options_selected` is an array that contains the options that will be selected (which means in the left select) when the shiftbox is displayed. `$all_options` is an array with all available options. This must contain all options from `$options_selected`!

Example:

```
$all_options = array('A' => 'apple', 'B' => 'banana', 'C' => 'lemon', 'D' =>
'pear');
$options_selected = array('B' => 'banana', 'D' => 'pear');
$box = $this->makeInstance('shiftbox', 'fruits', $options_selected,
$all_options);
$this->TPL['fruits'] = $box->get_html();
```

In the PHP script that receives the submit request:

```
$all_options = array('A' => 'apple', 'B' => 'banana', 'C' => 'lemon', 'D' =>
'pear');
$selected = $this->VAR['fruits'];
$options = explode('-', $selected);
foreach($options as $option)
  print 'You selected ' . $all_options[$option] . '<br>';
```

This class provides the following methods:

`set_selsize($size)`	Sets the size of the selects of the shiftbox
`set_multiple($flag = true)`	Sets a boolean that determines if multiple options can be selected.
`set_headlines($h, $h_not)`	Sets the headlines of the two selects.
`set_onchange($code)`	Sets the javascript code that is executed when changing the select
`get_javascript()`	Gets the complete javascript of the shiftbox
`get_html()`	Gets the HTML code of the select

Translator

The translator module provides mechanisms to make your webapp multi lingual. Different languages are realized in Booosta in two ways: templates are not translated on runtime. There must be a template file for every language you want to support. Template files with other languages get the extension `.xx` added, where `xx` is the two letter language code. If you have a template file named test.tpl you call the german translation test.tpl.de

For translating strings that are generated by the application, you need to provide a file with a translation map. This map is an array indexed with key strings and the translation of this strings as value. An example map file would be:

```
<?php
$this->map = [
'edit' => 'bearbeiten',
'delete' => 'l&ouml;schen',
'Search' => 'suchen',
'new ' => 'Neue ',
];
?>
```

Per default the language file `lang.xx` is used (eg. `lang.de`). You can provide a different language file in the constructor. Translations can be got by calling the method `t()`

```
$tr = makeInstance('Translator', $lang = 'en', $map_path = '', $map = null);
print $tr->t('edit');
```

If the key is not found in the translation map, then the key is returned by `t()` without modification.

The webapp module loads the translator module automatically, if it is installed. It provides a `t()` method itself:

```
print $this->t('edit');
```

Database

The database module is a generic class that provides basic components for other database modules. It provides the method `init_db()` and the variable `$this->DB` for every class that is derived from the Booosta base class. Connections to a database and queries on it must be provided by other modules that are designed for special database systems like MySQL or Postgresql.

To actually use a database, you must install and this other module. See the according documentation. In the Booosta configuration file `local/config.incl.php` you have to tell Booosta which database module to use. For using the MySQLi module you set it to

```
'db_module' => 'mysqli',
```

After that, the variable `$DB` is available in every class that is based on `\booosta\base\Base`, i. e. every Booosta class. You can access it with

```
$object->DB
```

or within the class (object) with

```
$this->DB
```

This variable points to an object where you can call all the database methods that are available. For example

```
$this->DB->query(„delete from customer where id=1");
```

This module provides the method init_db for every Booosta class. Usually you do not need to call this method, if you have configured the database credentials in the `local/config.incl.php`. It is automatically called. Under special circumstances, like you make a class that accesses the database in the constructor before the parent constructor is called, you might need to call this method prior accessing `$this->DB`.

Mysqli

Mysqli is a database module that uses the mysqli PHP extension. This module provides methods to access a MySQL database. This module requires the module database.

To use this module you need to configure your Booosta application with `'db_module' => 'mysqli'` in local/config.incl.php. From inside a Booosta class you can access the provided methods with

```
$this->DB->methodname($parameters);
```

For example:

```
$this->DB->query('delete from customer where id=1');
```

The available methods are:

`query($sql)`	Performs the query `$sql` on the database
`multi_query($sql)`	Performs multiple querys in the string `$sql` that are separated by `;`
`prepare($sql)`	Prepares a query for execution
`execute($statement)`	Executes a statement that has been prepared by `prepare()`
`get_error()`	Gets the error string of the last query
`query_list($sql, $num)`	Gets an array of all columns from the first result record of `$sql`. If `$num` is set to true, the resulting array holds each value twice. Once with the column name as index and once indexed with 0, 1,… If `$num` is `false` (which is default) the array only is indexed with the column names. Example: `list($firstname, $name) = $this->DB->query_list('select firstname, name from customer where id=1');`
`last_insert_id()`	Returns the last auto increment value inserted
`query_value_set($sql)`	Returns an array representing a set of values returned by the query `$sql`. It gets all the first colums returned by `$sql`.

	Example: `$lastnames = $this->DB->query_value_set('select lastname from customer');`
`query_arrays($sql)`	Returns all records returned by the query `$sql` in a two dimensional array. Example: `$result = $this->DB->query_arrays('select * from customer');`
`query_value($sql)`	Returns the first column of the first row of the query `$sql` Example: `$result = $this->DB->query_value('select name from customer where id=1');`
`query_index_array($sql)`	`$sql` should return two values: The index and the value of the resulting array. Example: `$result = $this->DB->query_index_array('select id, name from customer');` this returns an array where the names are in the value and the database id in the index.
`query_index_valueset($sql, $value, $index)`	Works like query_index_array(), but uses the columns `$value` and `$index`. Example: `$result = $this->DB->query_index_valueset('select * from customer', 'name', 'id');` has the same result as above
`escape($string)`	Escapes the string so that all special mysql characters are escaped.

Dataobjects

Dataobjects is a module that provides classes that represent a database record. Each object from these classes holds the data of one row of the according database table and has the methods to manipulate them in the database.

Dataobjects can be instanciated inside of Booosta object with

```
$this->makeDataobject($class);
```

where `$class` is the name of the according database table. To make a new object for a record in the customer table you use

```
$this->makeDataobject('customer');
```

To get an object with the data of a record you use

```
$obj = $this->getDataobject($class, $whereclause, $create);
```

If `$create` is set to true (default is false), a new empty object is created if no matching record is found.

For example:

```
$obj = $this->getDataobject('customer', 'id=1');
```

If you want to get your record with the primary key you can simply provide the key as number as second parameter:

```
$obj = $this->getDataobject('customer', 1);
```

You can get an array of Dataobjects with

```
$this->getDataobjects($class, $whereclause);
```

For example:

```
$customers = $this->getDataobjects('customer', „lastname='Smith'“);
```

The following methods are available:

`insert()`	Inserts the data of the object as new record in the database
`update()`	Updates the data in the database with current data in the object
`save()`	If the record already exists in the database it is updated. If it is a new record (i. e. The object has been created with `makeDataobject()` and has not yet been inserted in the database table) it is inserted as new database table row
`get($field)`	Returns the value of the field `$field` of the current record
`set($field, $value)`	Sets the value of the field `$field` to `$value`
`delete()`	Deletes the current record from the database
`get_data()`	Returns an array with all the fields of the current record
`set_data($data)`	Sets the fields of the current record to the values of the array `$data`. This array has to be indexed with the names of the fields.

mkfiles

The module mkfiles provides the possibility to create necessary PHP and template files for manipulating data in an existing database table via the admin web interface.

As the creation of those files can be a security risk during production use, the file creation only works, when a file named ENABLE_MKFILES is present in the web root directory. This file exists after the installation of Booosta. After you created all PHP and template files we strongly recommend, that you delete this file!

The web creation of the files should be accessible via the admins main menu if ENABLE_MKFILES exists. In the form there are four selects. The „mode“ select lets you choose if the files are created for admin users or ordinary users. The „table“ select lets you choose the database table for which the files should be created. You must have created this table in the database before you do this.

The „supertable“ select lets you choose a table, where the actual table has a foreign key to. For example if you work on the employee table and every employee belongs to a department, department would be the supertable of employee.

The „subtable“ is the opposite of the supertable. In out above example employee would be the subtable of department.

We again remind you to delete the ENABLE_MKFILES files in your web root if you are done. You can recreate it every time you want to use mkfiles again. Remember that if you run it again, it overwrites all existing files!

The module mkfiles also provides command line scripts to automatically create necessary PHP and template files for manipulating data in an existing database table.

The scripts are called via the wrapper script run in the Booosta base directory:

```
./run mkfiles table=tablename subtable=subtablename supertable=supertablename
```

this command creates the files for the table tablename that are used by the admin users. The options subtable and supertable are optional. A subtable is a table that has a foreign key to the current table. A supertable is a table where the current table has a foreign key to. See the docu for the module Webapp for examples.

```
./run mkuserfiles table=tablename subtable=subtablename
supertable=supertablename
```

this does the same like mkfiles, but for the ordinary users (not admin users).

The wrapper script run relys on the command line PHP interpreter to be found at `/usr/bin/php`. If this is not the case, you need to edit the file run and put the correct path in the first line.