Application Configuration

Immediately after initialization of the framework the adjustment of your application based on the config file occurs. By default the config file is placed in directory **Application/_config**, but you can place it where you
want. All you need to do this is to change path to the config file in **connect.php**

The configuration file is not a necessary element for the functioning of the framework. You can not use it at all. Вы можете не использовать его. In this case, the values of all configuration variables take default values (see below).

Config files can be of two types:

1. PHP-files. In these files an associative array of the application configuration data is determined and returned. Each key of the array is the name of some configuration property, and each value is the appropriate initial value of this property. Contents of the typical config php-file can be so:

   ```
   <?php

   return ['property1' => ...,
           'property2' => ...,
           'property3' => ...,
           ...];
   ```

2. INI-files. In these files names and values of parameters correspond names and values of configuration properties. If value of a config variable is an associative array then such variable is represented as a section in ini-file and its name corresponds the variable name and section parameters correspond values of the associative array. If you want to use multidimensional or mixed array as value of a configuration variable then you should use json representation of such array.

General configuration properties

These properties allow to manage analysis of errors and exceptions, logging and displaying error information. There are only 6 such properties:

1. **debugging** - boolean parameter, which responsible for the debug mode in which all available information about catched error or exception is displayed. The debug mode should be turned on during development stage and turned off for production version of your application.
2. **logging** - boolean parameter, which responsible for the error logging. Error logs are created as text files that contains serialized array of error data.
3. **templateDebug** - file path to the template which used for displaying of error information.
4. **templateBug** - file path to the template which used for displaying of error message in the production mode (when **debugging** is FALSE).
5. **customDebugMethod** - a delegate that defines a custom method of error analysis, and displaying information about them.
6. **customLogMethod** - a delegate that defines a custom method of logging errors.

There are other configuration properties that describe the behavior of the various modules of the framework. They will be discussed in the appropriate chapters.

Directory aliases

The framework allows to set aliases for one or more directories of your application and get paths to these directories by their aliases. To store aliases the configuration property **dirs** is used. Value of this property is an associative array. Keys of this array are directory aliases and values are directory full paths relative to the site root.

The framework uses some aliases (cache, logs and etc.) to obtain paths to the determined directories. All these aliases are specified in the config file bt default.

Example of alias section of configuration INI-file:

```
...
[dirs]
application = "app"
framework   = "lib"
logs        = "app/tmp/logs"
cache       = "app/tmp/cache"
```

```
temp        = "app/tmp/null"
ar          = "app/core/model/ar"
...
```

Example of aliases in configuration PHP-file:

```
<?php

return ['dirs' => ['cache' => 'app/tmp/cache',
                   'logs'  => 'app/tmp/logs',
                   'temp'  => 'app/tmp/null'],
...
```

To get full directory path by its alias the static method **dir()** of class **CB** is used:

```
...
echo CB::dir('cache');
...
```

If the given alias is not correspond to any directory then method **dir()** will assume that this alias is a directory relative to the site root directory.

Also, besides of the method for getting full directory path there is the static method **url()** of class **CB** allows to get relative path (URL) of a directory by its alias:

```
...
echo CB::url('temp');
...
```

Also as in the case of method **dir()**, if alias is not correspond to any directory of the site then the method **url()** will consider it as a directory.

Configuration file loading

After creation of config file you can load it via method **setConfig()** of class **CB**. It is enough to pass the path to the configuration file as the first parameter of the method:

```
...
$cb = \B::getInstance();
$cb->setConfig('/path/to/my/config.php');
...
```

In order to get an array of configuration data, you need to call method **getConfig()** with no parameters:

```
...
$cb = CB::getInstance();
// Loads data.
$cb->setConfig('/path/to/my/config.php');
// Displays config data in the browser.
print_r($cb->getConfig());
...
```

You can also load data from multiple configuration files, and the data will merge with each other:

```
...
$cb = CB::getInstance();
// Loads data from the first file.
$cb->setConfig('/path/to/my/config.php');
// Loads data from the second file.
$cb->setConfig('/path/to/my/config.ini');
...
```

In this example the data from the second file overrides configuration data from the first one.

The second parameter of the methods **setConfig()** and **getConfig()** determines the configuration section in which the configuration data will be

loaded. Example:

```
...
$cb = \CB::getInstance();
// loads config data from the first file.
$cb->setConfig('/path/to/config/file1.php');
// loads config data to the section "foo" from the second file.
$cb->setConfig('/path/to/config/file2.php', 'foo');
// displays data from the section "foo".
print_r($cb->getConfig('foo'));
...
```

If we want to replace the existing configuration data to others, then we need to pass TRUE as the third parameter of the method **setConfig()** (the default value is FALSE):

```
...
$cb = CB::getInstance();
// Loads data from the first file.
$cb->setConfig('/path/to/my/config.php');
// Replaces loaded data by the data from the second file.
$cb->setConfig('/path/to/my/config.ini', null, true);
...
```

It is possible to download the configuration data directly from an associative array. As in the case of files, if the second parameter of FALSE, it will merge with the new data with existing, otherwise data will be replaced with new ones:

```
...
$cb = CB::getInstance();
// Loads data from the first file.
$cb->setConfig('/path/to/my/config.php');
// Replaces data of section "foo" by the data from the array.
$cb->setConfig(['dirs' => [], 'debugging' => false], 'foo', true);
...
```

Work with configuration data

Once the configuration data is loaded, you can read the configuration variables, their values change, and delete. This is accomplished through the implementation of the interface **ArrayAccess** class **CB**. In other words, you can refer to an object **CB** as an associative array, whose elements correspond to the configuration variables:

```
...
// Gets CB object.
$cb = CB::getInstance();
// Reads value of the property "foo".
echo $cb['foo'];
// Writes to "foo" a value.
$cb['foo'] = 123;
// Removes property "foo".
unset($cb['foo']);
// If you want to get the value of a specific attribute,
// you can use a more compact notation:
echo CB::getInstance()['foo'];
...
```