The Hive

Encoding & Conversion

Localisation

Routing

File System

Misc

Base

The Base class represents the framework core. It contains everything you need to run a simple application. The file `base.php` also includes the essential Cache (cache), Prefab (prefab-registry), View (view), ISO (iso) and Registry (prefab-registry#registry) classes to reduce unnecessary disk I/O for optimal performance.

Feel free to remove all other files in the `lib/` directory, if all you need are the basic features provided by this package.

Namespace: `\`
File location: `lib/base.php`

The Hive

The hive is a memory array to hold your framework variables in the form of key / value pairs. Storing a value in the hive ensures it is globaly available to all classes and methods in your application.

set

Bind value to hive key

```
mixed set ( string $key, mixed $val [, int $ttl = 0 ] )
```

Examples of setting framework variables:

```
$f3->set('a',123); // a=123, integer
$f3->set('b','c'); // b='c', string
$f3->set('c','whatever'); // c='whatever', string
$f3->set('d',TRUE); // d=TRUE, boolean
```

Setting arrays:

```
$f3->set('hash',array( 'x'=>1,'y'=>2,'z'=>3 ) );
// dot notation is also possible:
$f3->set('hash.x',1);
$f3->set('hash.y',2);
$f3->set('hash.z',3);
```

Setting object properties:

```
$f3->set('a',new \stdClass);
$f3->set('a->hello','world');
echo $f3->get('a')->hello; // world
```

shorter ArrayAccess Syntax, since v3.4.0

```
$f3->LANGUAGE = 'en';
$f3->foo = 1234;
$f3['bar'] = 'buzzword';
```

Caching properties

If the `$ttl` parameter is > 0, and the framework cache engine (cache) is enabled, the specified variable will be cached for `$ttl` seconds. Already cached vars will be updated by reusing the old expiration time.

If you need to cache vars for an infinite time, check the Cache->set (cache#set) method.

You can cache strings, arrays and all other types - even complete objects. `get()` will load them automatically from the cache.

Examples of caching framework variables:

```
// cache string
$f3->set('simplevar','foo bar 1337', 3600); // cache for 1 hour
//cache big computed arrays
$f3->set('fruits',array(
    'apple',
    'banana',
    'peach',
), 3600);
// cache objects
$f3->set('myClass1', new myClass('arg1'), 3600);
// change expire time for a single cookie var
$f3->set('COOKIE.foo', 123, 3600);  // 1 hour
$f3->set('COOKIE.bar', 456, 86400); // 1 day
```

System variables

The framework has its own system variables (quick-reference#system-variables). You can change them to modify a framework behaviour, for example:

```
$f3->set('CACHE', TRUE);
$f3->set('HALT', FALSE);
$f3->set('CASELESS', FALSE);
```

It is also possible to set the PHP globals through F3's COOKIE, GET, POST, REQUEST, SESSION, FILES, SERVER, ENV system variables (quick-reference#cookie,-get,-post,-request,-session,-files,-server,-env). These 8 variables are automatically in synch with the underlying PHP globals.

> **Notice:** If you set or access a key of SESSION, the session gets started automatically. There's no need for you to do it by yourself.

Remember: Hive keys are *case-sensitive*.
Furthermore, root hive keys are checked for validity against these allowed chars: [a-z A-Z 0-9 _]

get

Retrieve contents of hive key

```
mixed get ( string $key [, string|array $args = NULL ] )
```

To get the value of a previously saved framework var, use:

```
$f3->set('myVar','hello world');
echo $f3->get('myVar'); // outputs the string 'hello world'
$local_var = $f3->get('myVar'); // $local_var holds the string 'hello world'
```

If the returned value is a string containing one or more format placeholders (base#format), arguments can be passed directly after the key's name:

```
$f3->set('var1','Current date: {0,date} - Current time: {0,time}');
$f3->set('var2','Departure: {0,date} - Arrival: {1,date}');
echo $f3->get('var1',time());//shorthand for $f3->format($f3->get('var1'),time());
echo $f3->get('var2',array($timestamp1,$timestamp2));//shorthand for $f3->format($f3->get('var2'),$timestamp1,$timestamp2);
```

> **Notice:** When caching is enabled and the var hasn't been defined at runtime before, F3 tries to load the var from Cache when using get().

Accessing arrays is easy. You can also use the JS dot notation `'myarray.bar'`, which makes it much easier to read and write.

```
$f3->set('myarray',
           array(
                    0 => 'value_0',
                    1 => 'value_1',
                'bar' => 123,
                'foo' => 'we like candy',
                'baz' => 4.56,
                )
         );

echo $f3->get('myarray[0]'); // value_0
echo $f3->get('myarray.1'); // value_1
echo $f3->get('myarray.bar'); // 123
echo $f3->get('myarray["foo"]'); // we like candy
echo $f3->get('myarray[baz]'); // 4.56, notice alternate use of single, double and no
quotes
// a new ArrayAccess syntax is also possible since v3.4.0
echo $f3->myarray['foo'];
echo $f3['myarray']['baz'];
```

sync

Sync PHP global variable with corresponding hive key

```
array sync ( string $key )
```

Usage:

```
$f3->sync('SESSION'); // ensures PHP global var SESSION is the same as F3 variable SE
SSION
```

F3 will automatically sync the following PHP globals: **GET**, **POST**, **COOKIE**, **REQUEST**, **SESSION**, **FILES**, **SERVER**, **ENV**

ref

Get reference to hive key and its contents

```
mixed &ref ( string $key [, bool $add = true, mixed $var = null ] )
```

Usage:

```
$f3->set('name','John');
$b = &$f3->ref('name'); // $b is a reference to framework variable 'name' , not a cop
y
$b = 'Chuck'; // modifiying the reference updates the framework variable 'name'
echo $f3->get('name'); // Chuck
```

You can also add non-existent hive keys, array elements, and object properties, when 2nd argument is TRUE by default.

```
$new = &$f3->ref('newVar'); // creates new framework hive var 'newVar' and returns re
ference to it
$new = 'hello world'; // set value of php variable, also updates reference
echo $f3->get('newVar'); // hello world

$new = &$f3->ref('newObj->name');
$new = 'Sheldon';
echo $f3->get('newObj')->name; // Sheldon
echo $f3->get('newObj->name'); // Sheldon
echo $f3->get('newObj.name'); // Sheldon

$a = &$f3->ref('hero.name');
$a = 'SpongeBob';
// or
$b = &$f3->ref('hero'); // variable
$b['name'] = 'SpongeBob';  // becomes array with key 'name'
$my_array = $f3->get('hero');
echo $my_array['name']; // 'SpongeBob'
```

If the 2nd argument `$add` is `false`, it just returns the read-only hive key content. This behaviour is used by get(). If the hive key does not exist, it returns NULL.

Use the 3rd argument when you want to find a reference in your own array/object instead of from the hive.

```
$fruitQty = ["Bananas"=>5, "Oranges"=>2, "Apples"=>42, "Mangos"=>1];
$apples = &$f3->ref("Apples", true, $fruitQty); // References $fruitQty["Apples"]
$apples = 10;

echo $fruitQty["Apples"]; // 10
```

This way you can also use dot notation with your own objects.

exists

Return TRUE if the hive key is set (or return timestamp and TTL if cached)

```
bool exists ( string $key [, mixed &$val=NULL] )
```

The `exists` function also checks the Cache backend storage when the key is not found in the hive. If the key is found in cache, it then returns `array ( $timestamp, $ttl )`.

> **Notice:** exists uses PHP's `isset()` function to determine if the hive key is set and is not NULL.

Usage:

```
$f3->set('foo','value');

$f3->exists('foo'); // true
$f3->exists('bar'); // false, was not set above
```

`exists` is especially useful with PHP global variables automatically synched by F3 (base#sync):

```
// Synched hive keys with PHP global variables
$f3->exists('COOKIE.userid');
$f3->exists('SESSION.login');
$f3->exists('POST.submit');
```

Notice: If you check the existence of a SESSION key, the session get started automatically.

You can use the `$val` argument to fetch the hive key content as well. This could save an additional `get` call.

```
if ($f3->exists('foo',$value)) {
    echo $value; // bar
}
```

devoid

Return TRUE if the hive key is empty and not cached

```
bool devoid ( string $key )
```

The `devoid` function also checks the Cache backend storage, if the key was not found in the hive.

Notice: devoid uses PHP's `empty()` function to determine if the hive key is empty and not cached.

Usage:

```
$f3->set('foo','');
$f3->set('bar',array());
$f3->set('baz',array(),10);

$f3->devoid('foo'); // true
$f3->devoid('bar'); // true
$f3->devoid('baz'); // false
```

clear

Unset hive key, key no longer exists

```
void clear ( string $key )
```

To remove a hive key and its value completely from memory:

```
$f3->clear('foobar');
$f3->clear('myArray.param1'); // removes key `param1` from array `myArray`
```

If the given hive key was cached before, it will be cleared from cache too.

Some more special usages:

```
$f3->clear('SESSION'); // destroys the user SESSION
$f3->clear('COOKIE.foobar'); // removes a cookie
$f3->clear('CACHE'); // clears all cache contents
```

Notice: Clearing all cache contents at once is not supported for the XCache cache backend

mset

Multi-variable assignment using associative array

```
void mset ( array $vars [, string $prefix = '' [, integer $ttl = 0 ]] )
```

Usage:

```
$f3->mset(
    array(
        'var1'=>'value1',
        'var2'=>'value2',
        'var3'=>'value3',
    )
);

echo $f3->get('var1'); // value1
echo $f3->get('var2'); // value2
echo $f3->get('var3'); // value3
```

You can append all key names using the 2nd argument `$prefix` .

```
$f3->mset(
    array(
        'var1'=>'value1',
        'var2'=>'value2',
        'var3'=>'value3',
    ),
    'pre_'
);

echo $f3->get('pre_var1'); // value1
echo $f3->get('pre_var2'); // value2
echo $f3->get('pre_var3'); // value3
```

To cache all vars, set a positive numeric integer value to `$ttl` in seconds.

hive

Return the whole hive contents as an array

```
array hive ()
```

Usage:

```
printf ("A Busy Hive: <pre>%s</pre>", var_export( $f3->hive(), true ) );
```

copy

Copy contents of a hive variable to another

```
mixed copy ( string $src, string $dst )
```

Return a writable reference to a new `$dst` hive key. If `$dst` already exists in the hive, it simply gets overwritten.

Usage:

```
$f3->set('foo','123');
$f3->set('bar','barbar');
$bar = $f3->copy('foo','bar'); // bar = '123'
$bar = 456;
$f3->set('foo','789');
echo $f3->get('bar'); // '456'
```

concat

Concatenate string to hive string variable

```
string concat ( string $key, string $val )
```

Return result of the concatenation. **Note**: If `$key` does not exist in the hive, it is automatically created in the hive.

Usage:

```
$f3->set('count', 99);
$f3->set('item', 'beer');
$text = $f3->concat('count', ' bottles of '.$f3->get('item'));
$text .= ' on the wall';
$f3->concat('wall', $f3->get('count')); // new 'wall' hive key is created
echo $f3->get('wall'); // 99 bottles of beer on the wall
```

flip

Swap keys and values of hive array variable

```
array flip ( string $key )
```

Usage:

```
$f3->set('data', array(
    'foo1' => 'bar1',
    'foo2 '=> 'bar2',
    'foo3' => 'bar3',
));
$f3->flip('data');

print_r($f3->get('data'));
/* output:
Array
(
    [bar1] => foo1
    [bar2] => foo2
    [bar3] => foo3
)
*/
```

push

Add element to the end of hive array variable

```
mixed push ( string $key, mixed $val )
```

Usage:

```
$f3->set('fruits',array(
    'apple',
    'banana',
    'peach',
));
$f3->push('fruits','cherry');

print_r($f3->get('fruits'));
/* output:
Array
(
    [0] => apple
    [1] => banana
    [2] => peach
    [3] => cherry
)
*/
```

pop

Remove last element of hive array variable

```
mixed pop ( string $key )
```

Usage:

```
$f3->set('fruits',array(
    'apple',
    'banana',
    'peach'
));
$f3->pop('fruits'); // returns "peach"

print_r($f3->get('fruits'));
/* output:
Array
(
    [1] => apple
    [2] => banana
)
*/
```

unshift

Add element to the beginning of hive array variable

```
mixed unshift ( string $key, string $val )
```

Usage:

```
$f3->set('fruits',array(
    'apple',
    'banana',
    'peach'
));
$f3->unshift('fruits','cherry');

print_r($f3->get('fruits'));
/* output:
Array
(
    [0] => cherry
    [1] => apple
    [2] => banana
    [3] => peach
)
*/
```

shift

Remove first element of a hive array variable

```
array|NULL shift ( string $key )
```

Return the left-shifted hive array variable, or `NULL` if the hive array variable is empty or is not an array.

> **Notice**: `shift` use the PHP function `array_shift()`. It means that all numerical array keys of the hive array variable will be modified to start counting from zero while literal keys won't be touched

Example:

```
$f3->set('fruits', array(
    'crunchy'=>'apples',
    '11'=>'bananas',
    '6'=>'kiwis',
    'juicy'=>'peaches'
));
$f3->shift('fruits'); // returns "apples"
print_r($f3->get('fruits'));
/* output:
Array
(
    [0] => bananas
    [1] => kiwis
    [juicy] => peaches
)
*/
```

merge

Merge array with hive array variable

```
array merge ( string $key, array $src [, bool $keep = FALSE])
```

Return the resulting array of the merge. (Does not touch the value of the hive key)

Example:

```
$f3->set('foo', array('blue','green'));
$bar = $f3->merge('foo', array('red', 'yellow'));

/* $bar now is:
array (size=4)
  [0] => string 'blue' (length=4)
  [1] => string 'green' (length=5)
  [2] => string 'red' (length=3)
  [3] => string 'yellow' (length=6)
*/
```

When the parameter `$keep` is *TRUE*, the origin `$key` in the HIVE is also updated.

extend

Extend hive array variable with default values from $src

```
array extend ( string $key, array $src [, bool $keep = FALSE])
```

This method provides a simple way to add some default values to an array:

```
$f3->set('settings',[
    'foo'=> 1,
    'bar'=> 'baz',
    'colors'=>[
        'blue'=>1,
        'green'=>2
    ]
]);
$f3->set('defaults',[
    'foo'=>0,
    'zzz'=>2,
    'colors'=>[
        'red'=>3,
        'blue'=>4
    ],
]);
$settings = $f3->extend('settings','defaults');
print_r($settings);

/*
Array (
    [foo] => 1
    [zzz] => 2
    [colors] => Array (
            [red] => 3
            [blue] => 1
            [green] => 2
        )
    [bar] => baz
)
*/
```

When the parameter `$keep` is *TRUE*, the origin `$key` in the HIVE is also updated.

Encoding & Conversion

fixslashes

Convert backslashes to slashes

```
string fixslashes ( string $str )
```

Usage:

```
$filepath = __FILE__; // \www\mysite\myfile.txt
$filepath = $f3->fixslashes($filepath); // /www/mysite/myfile.txt
```

split

Split comma-, semi-colon, or pipe-separated string

```
array split ( string $str [, $noempty = TRUE ] )
```

Usage:

```
$data = 'value1,value2;value3|value4';
print_r($f3->split($data));
/* output:
Array
(
    [0] => value1
    [1] => value2
    [2] => value3
    [3] => value4
)
*/
```

NB: by default, empty values are filtered out. Set `$noempty` to FALSE to prevent this behaviour.

stringify

Convert PHP expression/value to compressed exportable string

```
string stringify ( mixed $arg [, array $stack = NULL ] )
```

This function allows you to convert any PHP expression, value, array or any object to a compressed and exportable string.

The `$detail` parameter controls whether to walk recursively into nested objects or not.

Example with a simple 2D array:

```
$elements = array('water','earth','wind','fire');
$fireworks = array($elements, shuffle($elements), array_flip($elements));
echo $f3->stringify($fireworks);
// Outputs e.g.:
"[['water','earth','wind','fire'],true,['earth'=>0,'water'=>1,'wind'=>2,'fire'=>3]]"


$car = new stdClass();
$car->color = 'green';
$car->location = array('35.360496','138.727798');
echo $f3->stringify($car);
// Outputs e.g.:
"stdClass::__set_state(['color'=>'green','location'=>['35.360496','138.727798']])"
```

CSV

Flatten array values and return as CSV string

```
string csv ( array $args )
```

Usage:

```
$elements = array('water','earth','wind','fire');
echo $f3->csv($elements); // displays 'water','earth','wind','fire' // including sing
le quotes
```

camelcase

Convert snake_case string to camelCase

```
string camelcase ( string $str )
```

Usage:

```
$str_s_c = 'user_name';
$f3->camelcase($str_s_c); // returns 'userName'
```

snakecase

Convert camelCase string to snake_case

```
string snakecase ( string $str )
```

Usage:

```
$str_CC = 'userName';
$f3->snakecase($str_CC); // returns 'user_name'
```

sign

Return -1 if specified number is negative, 0 if zero, or 1 if the number is positive

```
int sign ( mixed $num )
```

hash

Generate 64bit/base36 hash

```
string hash ( string $str )
```

Generates a hash for a given string (length between 11 and 13)

Example:

```
$f3->hash('foobar'); // returns '3vrllw03cko4s' (length=13)
```

base64

Return Base64-encoded equivalent

```
string base64 ( string $data, string $mime )
```

Example:

```
echo $f3->base64('<h1>foobar</h1>','text/html');
// data:text/html;base64,PGgxPmZvb2JhcjwvaDE+
```

encode

Convert special characters to HTML entities

```
string encode ( string $str )
```

Encodes symbols like `&` `<` `>` and other chars, based on your applications ENCODING setting. (default: UTF-8)

Example:

```
echo $f3->encode("we <b>want</b> 'sugar & candy'");
// we &lt;b&gt;want&lt;/b&gt; 'sugar &amp; candy'
```

decode

Convert special HTML entities back to characters

```
string decode ( string $str )
```

Example:

```
echo $f3->decode("we &lt;b&gt;want&lt;/b&gt; 'sugar &amp; candy'");
// we <b>want</b> 'sugar & candy'
```

clean

Remove HTML tags (except those enumerated) and non-printable characters to mitigate XSS/code injection attacks

```
string clean ( mixed $var [, string $tags = NULL ] )
```

`$var` can be either a `string` or an `array`. In the latter case, it will be recursively traversed to clean each and every element.

`$tags` defines a list (as per the split syntax) (base#split) of *allowed* html tags that will **not** be removed.

Advice: It is recommended to use this function to sanitize submitted form input.

Examples:

```
$foo = "99 bottles of <b>beer</b> on the wall. <script>alert('scripty!')</script>";
echo $f3->clean($foo); // "99 bottles of beer on the wall. alert('scripty!')"


$foo = "<h1><b>nice</b> <span>news</span> article <em>headline</em></h1>";
$h1 = $f3->clean($foo,'h1,span'); // <h1>nice <span>news</span> article headline</h1>
```

scrub

Similar to clean(), except that variable is passed by reference

```
string scrub ( mixed &$var [, string $tags = NULL ] )
```

Example:

```
$foo = "99 bottles of <b>beer</b> on the wall. <script>alert('scripty!')</script>";
$bar = $f3->scrub($foo);
echo $foo; // 99 bottles of beer on the wall. alert('scripty!')
```

serialize

Return string representation of PHP value

```
string serialize ( mixed $arg )
```

Depending on the SERIALIZER (quick-reference#serializer) system variable, this method converts anything into a portable string expression. Possible values are **igbinary** and **php**. F3 checks on startup if igbinary is available and prioritize it, as the igbinary extension works much faster and uses less disc space for serializing. Check igbinary on github (https://github.com/igbinary/igbinary).

unserialize

Return PHP value derived from string

```
string unserialize ( mixed $arg )
```

See serialize (base#serialize) for further description.

Localisation

Note: F3 follows ICU formatting rules (http://userguide.icu-project.org/formatparse/) **without** requiring the PHP's `intl` module.

format

Return locale-aware formatted string

```
string format( string $format [, mixed $arg0 [, mixed $arg1...]] )
```

The `$format` string contains one or more placeholders identified by a position index enclosed in curly braces, the starting index being 0. The placeholders are replaced by the values of the provided arguments.

```
echo $f3->format('Name: {0} - Age: {1}','John',23); //outputs the string 'Name: John
- Age: 23'
```

The formatting can get more precise if the expected type is provided within placeholders.

Current supported types are:

- date
- date,short
- date,long
- date,custom,{exp}
- time
- time,short
- time,custom,{exp}
- number,integer
- number,currency
- number,currency,int
- number,currency,{char}
- number,percent
- number,decimal,{int}
- plural,{exp}

```
echo $f3->format('Current date: {0,date} - Current time: {0,time}',time());
//outputs the string 'Current date: 04/12/2013 - Current time: 11:49:57'
echo $f3->format('Created on: {0,date,custom,%A, week: %V }', time());
//outputs the string 'Created on: Monday, week 45'
echo $f3->format('{0} is displayed as a decimal number while {0,number,integer} is ro
unded',12.54);
//outputs the string '12.54 is displayed as a decimal number while 13 is rounded'
echo $f3->format('Price: {0,number,currency}',29.90);
//outputs the string 'Price: $29.90'
echo $f3->format('Percentage: {0,number,percent}',0.175);
//outputs the string 'Percentage: 18%' //Note that the percentage is rendered as an i
nteger
echo $f3->format('Decimal Number: {0,number,decimal,2}', 0.171231235);
//outputs the string 'Decimal Number: 0,17'
```

The **plural** type syntax is a little bit less straightforward since it allows you to associate a different output depending on the input quantity.

The plural type syntax must start with `0, plural,` followed by a list of plural keywords associated with the desired output. The accepted keywords are '*zero*', '*one*', '*two*' and '*other*'.

Furthermore, you can insert the matching numeral in your output strings thanks to the # sign that will be automatically replaced by the matching number, as illustrated in the example below:

```
$cart_dialogs= '{0, plural,'.
    'zero   {Your cart is empty.},'.
    'one    {One (#) item in your cart.},'.
    'two    {A pair of items in your cart.},'.
    'other  {There are # items in your cart.}
}';

echo $f3->format($cart_dialogs,0); // displays 'Your cart is empty.'
echo $f3->format($cart_dialogs,1); // displays 'One (1) item in your cart.'
echo $f3->format($cart_dialogs,2); // displays 'A pair of items in your cart.'
echo $f3->format($cart_dialogs,3); // displays 'There are 3 items in your cart.'
```

Each plural keyword is optional and you can for example omit the plural keyword 'two' if the 'other' one fits that case. Of course, if you omit'em all, only the numerals will be displayed. As a general rule, keep at least the 'other' plural keyword as a fallback.

Automatic formatting of hive variables

> **Nice to Remember:** F3, for your convenience, and to tremendously ease the use of formatting in your templates, you can fetch and format variables from the hive in a single command:

Example:

```
$f3->set('a_books_story',
    '{0, plural,'.
        'zero   {There is n#thing on the table.},'.
        'one    {A book is on the table.},'.
        'two    {Two (#) books are on this table.},'.
        'other  {There are # books on the table.}'.
    '}'
);
echo $f3->get('a_books_story',0); // displays 'There is n0thing on the table.'
echo $f3->get('a_books_story',1); // displays 'A book is on the table.'
echo $f3->get('a_books_story',2); // displays 'Two (2) books are on this table.'
echo $f3->get('a_books_story',7); // displays 'There are 7 books on the table.'
```

language

Assign/auto-detect language

```
string language ( string $code )
```

This function is used while booting the framework to auto-detect the possible user language, by looking at the HTTP request headers, specifically the Accept-Language header sent by the browser.

Use the LANGUAGE (quick-reference#language) system variable to get and set languages, since it also handles dependencies like setting the locales using php setlocale(LC_ALL,...) (http://php.net/manual/en/function.setlocale.php) and changing dictionary files. The FALLBACK (quick-

reference#fallback) system variable defines a default language, that will be used, if none of the detected languages are available as a dictionary file.

Example:

```
$f3->get('LANGUAGE'); // 'de-DE,de,en-US,en'
$f3->set('LANGUAGE', 'es-BR,es');
$f3->get('LANGUAGE'); // 'es-BR,es,en' the fallback language is added at the end of t
he list
```

> The LANGUAGE variable accepts the same kind of string as the HTTP Accept-Language header, which is a comma separated list of 2-letter language codes optionally followed by **a hypen** and a 2-letter country code.
>
> **NB:** Even though POSIX locales use an underscore as a separator (es_BR.UTF-8), you should define the LANGUAGE variable with a hyphen (es-BR). Dictionary files follow the same rule (es-BR.php / es-BR.ini).

System locales

In order to ensure that the format (base#format) method, and other locale-aware php functions, work like expected, the system locale is loaded automatically by this function. E.g:

```
$f3->set('ENCODING','UTF-8');
$f3->set('LANGUAGE','it-IT');// the locale it_IT.UTF-8 will be automatically loaded u
sing setlocale
```

Some more examples:

```
$f3->LANGUAGE='en' => F3 will try to load 2 locales:
    en.UTF-8
    en
$f3->LANGUAGE='en-US' => F3 will try to load 4 locales:
    en_US.UTF-8
    en_US
    en.UTF-8
    en
$f3->LANGUAGE='en-US,en-GB' => F3 will try to load 6 locales:
    en_US.UTF-8
    en_US
    en_GB.UTF-8
    en_GB
    en.UTF-8
    en
```

NB: The first locale found on the server is loaded. Make sure your server does have all the locales installed, which you want to support in your application. On most linux machines you can check that using `locale -a` and install new locales with `dpkg-reconfigure locales` . Some apache webserver configurations maybe need a restart afterwards to work with the new locales.

lexicon

Transfer lexicon entries to hive

```
array lexicon ( string $path )
```

This function is used while booting the framework to auto-load the dictionary files, located within the defined path in `LOCALES` var.

```
$f3->set('LOCALES','dict/');
```

A dictionary file can be a php file returning a key-value paired associative array, or an .ini-style formatted config file. Read the guide about language files here (views-and-templates#multilingual-support).

Routing

alias

Assemble url from alias name

```
string alias ( string $name [, array|string $params = array() [, array|string $query
= array() ]] )
```

Example:

```
$f3->route('GET @complex:/resize/@format/*/sep/*','App->nowhere');
$f3->alias('complex','format=20x20,2=foo/bar,3=baz.gif');
$f3->alias('complex',array('format'=>'20x20',2=>'foo/bar',3=>'baz.gif'));
// Both examples return '/resize/20x20/foo/bar/sep/baz.gif'
```

build

Replace tokenized URL with current route's token values

```
string build ( string $url [, array $params = array() ] )
```

Example:

```
// for instance the route is '/subscribe/@channel' with PARAMS.channel = 'fatfree'

echo $f3->build('@channel'); // displays 'fatfree'
echo $f3->build('/get-it/now/@channel'); // displays '/get-it/now/fatfree'
echo $f3->build('/subscribe/@channel');  // displays '/subscribe/fatfree'
```

If you'd like to specifically define the tokens in the given `$url`, you can use the `$params` argument for that:

```
$f3->build('/resize/@format/*/sep/*',array(
  'format'=>'200x200',// 1=>'200x200' also works
  2=>'foo/bar',
  3=>'baz.gif'
));
// returns: /resize/200x200/foo/bar/sep/baz.gif
```

mock

Mock an HTTP request

```
mixed mock ( string $pattern [, array $args = NULL [, array $headers = NULL [, string
$body = NULL ]]] )
```

This emulates a HTTP request based upon the verb and resource defined by `$pattern` .

- `$args` gets exported as the matching verb's global variable (`$_GET` , `$_POST` or `$_REQUEST`)
- `$headers` gets exported as global HTTP headers (`$_SERVER[HTTP_…]`)
- The HTTP request body `$body` gets exported as the HIVE's `BODY` variable for verbs not equal to `GET` or `HEAD` . If `$body` is undefined, `$args` gets URL-encoded and exported as `BODY`
- Appending `[ajax]` to `$pattern` mocks AJAX calls
- Appending `[sync]` to `$pattern` mocks ordinary (synchronous) calls
- Appending `[cli]` to `$pattern` mocks command-line (CLI) calls
- Named routes and tokens (routing-engine#NamedRoutes) are valid resources

Basic usage example:

```
$f3->mock('GET /page/view [ajax]');
```

Basic usage example with a named route and token:

```
$f3->route('GET @grub:/food/@id', /* … */);
$f3->mock('GET @grub(@id=bread)');
```

Unit test (unit-testing) as advanced usage example:

```
$f3->route('GET|POST|PUT @grub:/food/@id/@quantity', /* … */);
$f3->mock('POST /food/sushki/134?a=1',array('b'=>2));
$test->expect(
    $_GET==array('a'=>1) && $_POST==array('b'=>2) && $_REQUEST==array('a'=>1,'b'=>2)
&& $f3->get('BODY')=='b=2',
    'Request body and superglobals $_GET, $_POST, $_REQUEST correctly set on mocked P
OST'
);
```

parse

Parse a string containing key-value pairs and return them as an array

```
array parse ( string $str )
```

Basic usage example:

```
$array = $f3->parse('framework=f3 , speed=fast , features=full');
echo $array['framework'];  // 'f3'
echo $array['speed'];  // 'fast'
```

Storing parsed string containing key-value pairs into The Hive as key-value pairs using mset method

Example:

```
$array = $f3->parse('framework=f3 , speed=fast , features=full');
$f3->mset($array);
echo $f3->get('framework');  // 'f3'
echo $f3->get('speed');      // 'fast'
```

route

Bind a route pattern to a given handler

```
null route ( string|array $pattern, callback $handler [, int $ttl = 0 [, int $kbps =
0 ]] )
```

Basic usage example:

```
$f3->route('POST /login','AuthController->login');
```

Route Pattern

The `$pattern` var describes a route pattern, that consists of the request type(s) and a request URI, both separated by a space char.

Verbs

Possible request type (Verb) definitions, that F3 will process, are: **GET**, **POST**, **PUT**, **DELETE**, **HEAD**, **PATCH**, **CONNECT**.

You can combine multiple verbs, to use the same route handler for all of them. Simply separate them by a pipe char, like `GET|POST`.

Tokens

The request URI may contain one or more **token(s)**, that a meant for defining dynamic routes. Tokens are indicated by a `@` char prior their name and can optionally be wrapped by single curly brackets `{}`. See this examples:

```
$f3->route('GET|HEAD /@page','PageController->display'); // ex: /about
$f3->route('POST /@category/@thread','ForumThread->saveAnswer'); // /games/battlefiel
d3
$f3->route('GET /image/@width-@height/@file','ImageCompressor->render'); // /image/30
0-200/mario.jpg
$f3->route('GET /image/{@width}x{@height}/@file','ImageCompressor->render'); // /imag
e/300x200/mario.jpg
```

After processing the incoming request URI (initiated by run (base#run)), you'll find the value of each of those tokens in the `PARAMS` system variable as named key, like `$f3->get('PARAMS.file')` // 'mario.jpg'

> **Notice:** Routes and their according verbs are grouped by their URL pattern. Static routes **precede** routes with dynamic tokens or wildcards.

Wildcards

You can also define wildcards (`/*`) in your routing URI. Furthermore, you can use them in combination with `@` tokens.

```
$f3->route('GET /path/*/@page', function($f3,$params) {
    // called URI: "/path/cat/page1"
    // $params is the same as $f3->get('PARAMS');
    $params[0]; // contains the full route path. "/path/cat/page"
    $params[1]; // and further numeric keys in PARAMS hold the catched wildcard paths
and tokens. in this case "cat"
    $params['page']; // is your last segment, in this case "page1"
});
```

> **Notice:** The `PARAMS` var contains all tokens as named key, and additionally all tokens and wildcards as numeric key, depending on their order of appearance.

The route above also works with sub categories. Just call `/path/cat/subcat/page1`

```
$params[1]; // now contains "/cat/subcat"
```

It even works with some more sub-levels. You just need to explode this value with a `/` -delimiter to handle your sub-categories. Something like `/path/*/@pagetitle/@pagenum` is also quite easy.

It becomes complicated when you try to use more than one wildcard, because only the first `/*` -wildcard can hold unlimited path-segments. Any further wildcards can only contain exactly one part between the slashes (`/`). So try to keep it simple.

Groups

It's possible to assign multiple routes to the same route handler, using an array of routes in `$pattern` . It would look like this:

```
$f3->route(
  array(
    'GET /archive',
    'GET /archive/@year',
    'GET /archive/@year/@month',
    'GET /archive/@year/@month/@day'
  ),
  function($f3,$params){
    $params+=array('year'=>2013,'month'=>1,'day'=>1); //default values
    //etc..
  }
);
```

Named Routes

Since `F3 v3.2.0` you may also assign a name to your routes. Therefore follow this pattern:

```
$f3->route('GET @beer_list: /beer', 'Beer->list');
```

Names are inserted after the route VERB and preceded by an @ symbol. All named routes can be accessed by the ALIASES (quick-reference#aliases) system variable for further processing in templates or for rerouting. Check out the User Guide about creating named routes (routing-engine#named-routes) for additional information.

Furthermore you can use an existing route name to shorten additional route definitions:

```
$f3->route('GET @beer_details: /beer/@id', 'Beer->get');
$f3->route('POST @beer_details', 'Beer->saveComment');
$f3->route('PUT @beer_details', 'Beer->savePhoto');
```

Route Handler

Can be a callable class method like 'Foo->bar' or 'Foo::bar', a function name, or an anonymous function.

F3 automatically passes the framework instance and the route tokens to route handler controller classes. E.g:

```
$f3->set('hello','world');
$f3->route('GET /foo/@file','Bar->baz');

class Bar {
    function baz($f3,$args) {//<-- $f3 is the framework instance, $args are the route
tokens
        echo $f3->get('hello');
        echo $args['file'];
    }
}
```

Caching

The 3rd argument `$ttl` defines the caching time in seconds. Setting this argument to a positive value will call the expire (base#expire) function to set cache metadata in the HTTP response header. It also caches the route response, but only GET and HEAD requests are cacheable.

If `CACHE` is turned off, `$ttl` will only control the browser cache using expire (base#expire) header metadata. If `CACHE` is turned on, and there is a positive `$ttl` value set for the current request URI handler, F3 additionally will cache the output for this page, and refresh it when `$ttl` expires. Read more about it here (https://groups.google.com/d/msg/f3-framework/lwaqZjtwCvU/PC-gK8Ki9PMJ) and here (https://groups.google.com/d/msg/f3-framework/lwaqZjtwCvU/LDUlPhQfc84J).

Bandwidth Throttling

Set the 4th argument `$kbps` to your desired speed limit, to enable throttling. Read more (optimization#bandwidth-throttling) about it in the user guide.

reroute

Reroute to specified URI

```
null reroute ( [ string|array $url = NULL [, bool $permanent = FALSE [, bool $die = T
RUE ]]] )
```

Examples of usage:

```
// an old page is moved permanently
$f3->route('GET|HEAD /obsoletepage', function($f3) {
    $f3->reroute('/newpage', true);
});

// whereas a Post/Redirect/Get pattern would just redirect temporarily
$f3->route('GET|HEAD /login', 'AuthController->viewLoginForm');
$f3->route('POST /login', function($f3) {
    if( AuthController->checkLogin == true )
        $f3->reroute('/members', false);
    else
        $f3->reroute('/login', false);
});

// if no $url parameter is set, it'll reroute to the current route with GET verb
$f3->route('POST /search', function($f3) {
    // back to search form, if an empty term was submitted
    if ($f3->devoid('POST.search_term'))
        $f3->reroute();
});

// we can also reroute to external URLs
$f3->route('GET /partners', function($f3) {
    $f3->reroute('http://externaldomain.com');
});

// it's also possible to reroute to named routes
$f3->route('GET @beer_list: /beer', 'Beer->list');
$f3->route('GET /old-beer-page', function($f3) {
    $f3->reroute('@beer_list');
});

// even with dynamic parameter in your named route
$f3->route('GET @beer_producers: /beer/@country/@village', 'Beer->byproducer');
$f3->route('GET /old-beer-page', function($f3) {
    $f3->reroute('@beer_producers(@country=Germany,@village=Rhine)');
});
// but it'll also work with any unnamed tokenized URL
$f3->reroute('/beer/@country/@village',TRUE)
```

When `$url` is an array, it's uses for alias rerouting. You can specify the alias name in the 1st array element, its parameters in the 2nd and additional query elements as 3rd array element:

```
$f3->reroute(['filter','a=foo,b=bar','time=3']);
// or
$f3->reroute(['filter',['a'=>'foo','b'=>'bar'],['time'=>3]]);

// equivalent to:
$f3->reroute('@filter(a=foo,b=bar)?time=3');
```

The `$die` parameter can be used to disable immediate redirecting, instead the script continues after the appropriate headers were set. This can be useful for unit testing.

redirect

Redirect a route to another URL

```
null redirect ( string|array $pattern, string $url [, bool $permanent = TRUE ] )
```

This is a little shortcut method between *route* and *reroute* methods to defined routes that are just meant to redirect the client. The `$pattern` argument accepts the same values as the route (base#route) method does. For `$url` you can use anything that would also be accepted in the appropriate reroute (base#reroute) method argument.

Example of usage:

```
// redirect old pages
$f3->redirect('GET /oldpage', '/newpage');
// jump to absolut URLs
$f3->redirect('GET /external-link', 'http://subdomain.domain.com');
// temporarily redirect to another named route
$f3->redirect('GET /login', '@member_area', false);
// redirect one named route to another
$f3->redirect('GET @member_welcome', '@member_area');
```

This can also be configured in config files (framework-variables#ConfigurationFiles) within a `[redirects]` section.

map

Provide ReST interface by mapping HTTP verb to class method

```
null map ( string $url, string $class [, int $ttl = 0 [, int $kbps = 0 ]] )
```

Its syntax works slightly similar to the **route** function, but you need not to define a HTTP request method in the 1st argument, because they are mapped as functions in the Class you prodive in the `$class` argument. Read more about the REST support in the User Guide (routing-engine#rest-representational-state-transfer).

Example of usage:

```
$f3->map('/news/@item','News');

class News {
    function get() {}
    function post() {}
    function put() {}
    function delete() {}
}
```

run

Match routes against incoming URI and call their route handler

```
null run ()
```

Example of usage:

```
$f3 = require __DIR__.'/lib/base.php';
$f3->route('GET /',function(){
    echo "Hello World";
});
$f3->run();
```

After processing the incoming request URI, the routing pattern that matches that URI is saved in the `PATTERN` var, the current HTTP request URI in the `URI` var and the request method in the `VERB` var. The `PARAMS` var will contains all tokens as named keys, and additionally all tokens and wildcards as numeric keys, depending on their order of appearance.

Generate a 404 error when a tokenized class doesn't exist.

> **Notice:** If a static and dynamic route pattern both match the current URI, then the *static* route pattern has priority.

call

Execute callback/hooks (supports 'class->method' format)

```
mixed|false call ( callback $func [, mixed $args = NULL [, string $hooks = '' ]] )
```

This method provides that facility to invoke callbacks and their arguments. F3 recognizes these as valid callbacks:

- Anonymous/lambda functions (aka closures)
- `array('class','method')`
- `class::method` static method
- `class->method` equivalent to `array(new class,method)`

`$args` if specified provides a means of executing the callback with parameters.

`$hooks` is used by the `route()` method to specify pre- and post-execution functions, i.e. `beforeroute()` and `afterroute()`. (refer to the section Event Handlers (routing-engine#event-handlers) for more explanations about `beforeroute()` and `afterroute()`)

chain

Execute specified callbacks in succession; Apply same arguments to all callbacks

```
array chain ( array|string $funcs [, mixed $args = NULL ] )
```

This method invokes several callbacks in succession:

```
echo $f3->chain('a; b; c', 0);

function a($n) {
    return 'a: '.($n+1); // a: 1
}

function b($n) {
    return 'b: '.($n+1); // b: 1
}

function c($n) {
    return 'c: '.($n+1); // c: 1
}
```

relay

Execute specified callbacks in succession; Relay result of previous callback as argument to the next callback

```
array relay ( array|string $funcs [, mixed $args = NULL ] )
```

This method invokes callback in succession like chain (base#chain) but applies the result of the first function as argument of the succeeding function, i.e.:

```
echo $f3->relay('a; b; c', 0);

function a($n) {
    return 'a: '.($n+1); // a: 1
}

function b($n) {
    return 'b: '.($n+1); // b: 2
}

function c($n) {
    return 'c: '.($n+1); // c: 3
}
```

File System

copy

Native PHP Function

```
bool copy ( string $source , string $dest [, resource $context ] )
```

To copy a file, use the native `copy()` (http://php.net/manual/en/function.copy.php) PHP function. Shown here for convenience.

delete (unlink)

Native PHP Function

```
bool unlink ( string $filename [, resource $context ] )
```

To delete a file, use the native `unlink()` (http://php.net/manual/en/function.unlink.php) PHP function. Shown here for convenience.

move (rename using diff path)

Native PHP Function

```
bool rename ( string $oldname , string $newname [, resource $context ] )
```

To move a file, use the native `rename()` (http://php.net/manual/en/function.rename.php) PHP function changing the path in the second argument. Shown here for convenience.

mutex

Create mutex, invoke callback then drop ownership when done

```
mixed mutex ( string $id, callback $func [, mixed $args = NULL ] )
```

A Mutual Exclusion (mutex) (https://en.wikipedia.org/wiki/Mutual_exclusion) is a cross-platform mechanism for synchronizing access to resources, in such a way that a process that has acquired the mutex gains exclusive access to the resource. Other processes trying to acquire the same mutex will be in a suspended state until the mutex is released.

```
$f3->mutex('test',function() {
    // Critical section
    session_start();
    $contents=file_get_contents('mutex');
    sleep(5);
    file_put_contents('mutex',$contents.date('r').' '.session_id()."\n");
});
```

Arguments can be passed to the callback:

```
$f3->mutex('my-mutex-id',function($path,$str) {
    @file_put_contents($path,$str);
},array('/foo/bar/facts.txt','F3 is cool'));
```

read

Read file (with option to apply Unix LF as standard line ending)

```
string|FALSE read ( string $file [, bool $lf = FALSE ] )
```

Uses the `file_get_contents()` (http://www.php.net/file_get_contents) PHP function to read the entire `$file` and return it as a string. Returns `FALSE` on failure.

IF `$lf` is `TRUE`, an EOL conversion to UNIX LF format is performed on all the lines ending.

rel

Return path relative to the base directory

```
string rel ( string $url )
```

Example:

```
$f3->set('BASE','http://fatfreeframework.com/');
echo $f3->rel( 'http://fatfreeframework.com/gui/img/supported_dbs.jpg' ); // 'gui/im
g/supported_dbs.jpg'
```

rename

Native PHP Function

```
bool rename ( string $oldname , string $newname [, resource $context ] )
```

To rename a file, use the native `rename()` (http://php.net/manual/en/function.rename.php) PHP function. Shown here for convenience.

write

Exclusive file write

```
int|FALSE write ( string $file, mixed $data [, bool $append = FALSE ] )
```

Uses the `file_put_contents()` (http://www.php.net/file_put_contents) PHP function with the `LOCK_EX` PHP flag to acquire an exclusive lock on the file while proceeding to the writing.

If `$append` is `TRUE` and the `$file` already exits, the `$data` is appended to the file content instead of overwriting it.

Misc

instance

Return class instance

```
Base $f3 = \Base::instance();
```

This is used to grab the framework instance at any point of your code.

abort

Disconnect HTTP client

```
void abort ( )
```

This method could be handy to abort the connection to the client and send the current response to it, but actually continue script execution after this point for doing time consuming tasks like sending SMTP mails or generating large PDF files without any lag for the client.

blacklisted

Lookup visitor's IP against common DNS blacklist services

```
bool blacklisted ( string $ip )
```

This function get called while bootstrapping the application and will lookup the visitors IP against common DNS blacklist services defined by the DNSBL system variable (quick-reference#dnsbl). This is very useful to protect your application against Spam bots or DOS attacks.

Return `TRUE` if the IPv4 `$ip` address is present in DNSBL (quick-reference#dnsbl)

compile

Convert JS-style token to PHP expression

```
string compile ( string $str )
```

This method is mainly used by the Preview class (preview), the lightweight template engine, to convert tokens to variables.

Example:

```
$f3->compile('@RAINBOW.cyan'); // returns: $RAINBOW['cyan']
```

config

Configure framework according to .ini-style file settings

```
null config ( string $file [, bool $allow = FALSE ])
```

This will parse a configuration file, provided by `$file` and setup the framework with variables and routes. If the 2nd argument `$allow` is provided, template strings are interpreted, thus making dynamic configs possible:

```
[globals]
foo = bar
baz = {{@foo}}
```

See the user guide section about configuration files (framework-variables#configuration-files) to get a full description about how to setup your ini file.

constants

Convert class constants to array

```
array constants ( object | string $class [, string $prefix = '' ] )
```

Example, providing we have a `Foo\Bar` class defined as below:

```
namespace Foo;
class Bar {
  const STATUS_Inactive=0;
  const STATUS_Active=1;
  const E_NotFound='Not found';
}
```

you can access all the constants prefixed by `STATUS_` :

```
$constants=$f3->constants('Foo\Bar','STATUS_');
var_dump($constants);
// outputs:
array(2) {
  ["Inactive"]=>int(0)
  ["Active"]=>int(1)
}
```

NB: passing an object also works: `$f3->constants($bar,'STATUS_')` .

dump

Dump (_echo_) expression with syntax highlighting

```
null dump ( mixed $expr )
```

NOTICE: The syntax highlighting depends on the DEBUG level (quick-reference#debug).

error

Execute error handler

```
null error ( int $code [, string $text = '' [, array $trace = NULL [, int $level = 0
]]] )
```

Calling this function logs an error and executes the ONERROR (quick-reference#onerror) handler if defined. Otherwise it will display a default error page in HTML for synchronous requests, or gives a JSON string for AJAX requests.

expire

Send cache metadata to HTTP client

```
void expire ( [ int $secs = 0 ] )
```

There is little need to call this method directly because it is automatically invoked at runtime by the framework, depending on whether the page should be cached or otherwise. The framework sends the necessary HTTP cache control headers to the browser so you don't need to send it yourself.

```
$f3->expire(0); // sends 'Cache-Control: no-cache, no-store, must-revalidate'
```

highlight

Apply syntax highlighting

```
string highlight ( string $text )
```

Applies syntax highlighting to a given string and returns the highlighted string.

Example:

```
$highlighted_code = $f3->highlight( '$fatfree->rocks(\'FAST\' AND $light)' );
```

Returns:

```
<code><span class="variable">$fatfree</span><span class="object_operator">-&gt;</span
><span class="string">rocks</span><span>(</span><span class="constant_encapsed_strin
g">'FAST'</span><span class="whitespace"> </span><span class="logical_and">AND</span>
<span class="whitespace"> </span><span class="variable">$light</span><span>)</span></
code>
```

> Keep in mind you need the `code.css` stylesheet to correctly see the syntax highlighting in your browser pages. You can include it in your pages with <link href="code.css" rel="stylesheet" /> (code.css is bundled into the framework 'lib/' folder)

status

Send HTTP/1.1 status header; Return text equivalent of status code

```
string status ( int $code )
```

Use this method for sending various HTTP status messages (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html) to the client, e.g.

```
$f3->status(404); // Sends a '404 Not Found' client error
$f3->status(407); // Sends a '407 Proxy Authentication Required' client error
$f3->status(503); // Sends a '503 Service Unavailable' server error
```

unload

Execute framework/application shutdown sequence

```
null unload ( string $cwd )
```

First, changes PHP's current directory to directory `$cwd`, writes session data and ends the session by calling the `session_commit()` PHP function.

Then it shutdowns the application and calls the shutdown handler defined in UNLOAD (quick-reference#unload).

As a final fallback, an HTTP error `500` is raised if one of the following `E_ERROR, E_PARSE, E_CORE_ERROR, E_COMPILE_ERROR` is detected and not handled by the shutdown handler.

recursive

Invoke callback recursively for all data types

```
mixed recursive( mixed $arg , callable $func [, array $stack = NULL ] )
```

The recursive method takes your mixed `$arg` and applies the `$func` callback to any deep nested array value or object property that can be found and returns a copy of `$arg`. Here a little example:

```
// ensure proper UTF-8 sequences before encoding
echo json_encode($f3->recursive($out,function($val){
    return mb_convert_encoding($val,'UTF-8','UTF-8');
}));
```

> **Notice:** To invoke the callback `$func` on objects `PHP >= 5.4` is required. Otherwise objects are returned without further processing. Please keep in mind that the returned objects are clones and the callback is only applied to accessible non-static properties.

until

Loop until callback returns TRUE (for long polling)

```
mixed until ( callable $func [, array $args = null [, int $timeout = 60 ] ] )
```

The callback (base#call) `$func` gets called once a second with argument `$args` until one of the following conditions terminates the loop:

- HTTP Client disconnects
- Time out after `$timeout` seconds
- Time out one second before reaching PHP's configured `max_execution_time`

- Session reopening fails
- Callback `$func` returns a value boolean casted equal to `true`

F3 closes the session after and reopens the session before calling the callback `$func` to prevent session locking, which would otherwise block concurrent requests with the same session.

Notice: `$timeout` is limited by the PHP configuration `max_execution_time`.