Deval Documentation

Release 1.0

Deval

Jan 22, 2023

Contents

CHAPTER 1

Welcome to Deval ‘s documentation!

Contents:

1.1 Overview

1.1.1 What is Deval?

Deval is a PHP template engine with support for partial evaluation at compilation to enable early error detection, optimize generated code and improve execution performance.

This user manual assumes you already know what a template engine is. Read more about template processors on Wikipedia if you want detailed information on this topic.

1.1.2 What is partial evaluation?

While most PHP template engines follow a “load variables and render” workflow Deval introduces an intermediate pre-compilation injection step to specify variables that you know won’t change on every rendering. By doing so, Deval will pre-evaluate your template and generate specialized code where all these invariants have been evaluated.

Template rendering workflow with Deval looks like this:

```
<?php

// Inject compile-time constants that don't change from one request to
// another. After this step Deval can pre-evaluate your template and store
// compiled result in its cache to save some processing power on next
// requests that will depend on the same set of constants
$renderer->inject($constants);

// Inject render-time variables into pre-processed template
$output = $renderer->render($variables);
```

(continues on next page)

(continued from previous page)

```
// Print rendered result
echo $output;
```

Deval will maintain a pre-evaluated version of each template file you use for every combination of constants you specify. For example if you chose to inject a `$language` constant into your template and use it to build localized strings Deval will cache a specialized version of your template for each language you support where strings have been pre-resolved.

Passing a value at compilation or rendering is a decision you'll take based on how many different values it can take and by how much pre-evaluating it improves performance. Deval offers a unified template syntax for all variables regardless of whether they've been specified at compilation or rendering, so you can change your mind without touching tour template code.

1.1.3 A simple template example

Let's keep out last example and pass current language as a constant to remove cost of strings localization from rendering. Out first simple template could look like this:

```
{{ $ locale(language, "users.list") }}
{{ for user in users }}
    - {{ $ user.login }}
{{ empty }}
    {{ $ locale(language, "no.users") }}
{{ end }}
```

We'll see about template syntax later and for now you only need to know about these two constructs:

- `{{ $ expression }}` prints the result of `expression` after evaluating it ;
- `{{ for i in c }}...{{ empty }}...{{ end }}` is a foreach-like loop.

After injecting a `locale` function and a `language` string as constants, Deval will compile and cache a PHP snippet similar to this one:

```
Registered user:
<?php $_counter = 0; foreach ($users as $user) { ?>
    - <?php echo $user->login; ?>
<?php ++$_counter; } if ($_counter === 0) { ?>
    No users registered.
<?php } ?>
```

As you can see all statements depending on variables `locale` and `language` have been evaluated in generated code, as their value was known at compile time. Other variables have been left untouched and Deval expects you to specify their value when rendering the template (and will raise an error if you don't).

1.1.4 Safety & performance

Deval keeps track of variables referenced in your template and will detect uninitialized ones before rendering anything, as opposed to the usual runtime error detection you get with native PHP or some template engines. While this is not a perfect static error detection solution (for example Deval can't detect runtime type errors) it's a good compromise between safety and ease of writing.

The compile-time evaluation mentioned in previous section also applies to every constant expression declared in your templates, meaning you won't get any performance penalty for factorizing code. In the following template code example:

```
{{ for i in range(0, 4) }}
    Rank {{ $ i + 1 }} / 5: {{ $ players[i].name }}
{{ end }}
```

Deval will unroll the "for" loop as it depends only on known values and compile a PHP snippet equivalent to this one:

```
Rank 1 / 5: <?php echo $players[0]->name; ?>
Rank 2 / 5: <?php echo $players[1]->name; ?>
Rank 3 / 5: <?php echo $players[2]->name; ?>
Rank 4 / 5: <?php echo $players[3]->name; ?>
Rank 5 / 5: <?php echo $players[4]->name; ?>
```

You shouldn't worry about this when writing template code as Deval will take care of pre-evaluating as much code as possible with the information it as been given.

1.2 Getting started

1.2.1 Install the library

You can choose to download lastest Deval release or build it from source.

Option 1: download latest release

Download latest Deval release, unzip it somewhere inside your project directory and skip to next section to see how to import Deval into your project.

Option 2: build from source

You'll need following requirements to build Deval from source:

- PHP 5.6.0 or above
- Node.js 6.11.0 or above

Install Node.js & npm, clone Deval repository, browse to cloned directory and run `npm install` command to install dependencies and build:

```
$ git checkout https://github.com/r3c/deval.git
$ cd deval
$ npm install
```

Once build, copy content of the `src` folder somwhere in your project and move to next section.

1.2.2 Import into your project

Require file `deval.php` file to start using the library ; here is a minimal code example that shows most common features. Save the snippet below into some `sample.php` file, we'll dig into details about each line in next sections:

```
<?php

require 'lib/deval/deval.php';

```

(continues on next page)

(continued from previous page)

```
// Create a new renderer for template file 'template/users.deval' and use
// directory 'cache' to store pre-evaluated intermediate results (your web
// user must have read/write permission on this directory)
$renderer = new Deval\CacheRenderer('template/users.deval', 'cache/');

// Inject some standard functions into template e.g. 'find', 'sort', etc.
$renderer->inject(Deval\Builtin::deval());

// Inject compile-time constants that will be used for pre-evaluation. Only
// serializable values can be used here, see API documentation about the
// inject method for more details
$renderer->inject(array('language' => 'en-us'));

// Inject render-time variables and print rendered output
echo $renderer->render(array('users' => array('Jane', 'John')));
```

In this example we created an instance of *Deval\CacheRenderer* which is a production renderer able to cache pre-evaluated code for optimal performance. You may also want to put performance aside, for example when developing your project, and use a *Deval\FileRenderer* instead that won't cache anything and reprocess your templates on each call, or even a *Deval\StringRenderer* that will let you specify your template code as a string instead of reading it from a file:

```
// Create a simple file-based renderer with no caching optimization
$renderer = new Deval\FileRenderer('template/users.deval');

// Create a string-based renderer, still with no caching
$renderer = new Deval\StringRenderer($my_template_code);
```

By default you don't have access to PHP functions from within a template (we'll explain why exactly in *Functional constraint* section), which is why we added a first call to method *Deval\Renderer::inject* in previous example. It's here to inject a few common functions in our template and make them available before compilation to enable many early optimizations. A full list of functions injected from *Deval\Builtin::deval* can be found in the *Deval flavor functions* section.

1.2.3 Write a template

Now you're ready for writing a template. Create a new text file in your favorite editor and type in some contents:

```
Note: this page should be displayed using {{ $ language }} locale.

Users list: {{ $ join(", ", users) }}
```

Save this file as `template/users.deval` (relative to your previously created `sample.php` file) to match the name we used in previous example. Don't forget to create a `cache/` directory to store pre-evaluated results and browse to your `sample.php` file. Result should look like this:

```
Note: this page should be displayed using en-us locale.

Users list: Jane, John
```

As you can guess the `join` function we used in our template is one of the builtin ones we mentionned earlier, and is similar to PHP's standard implode function.

One last note before jumping into more details: if you're curious you can have a look at the content of your `cache/` folder, it should now contain a `.php` file generated from your template. Edit this file and see the note about page

locale includes a literal `en-us` part which has been pre-evaluated since it was injected as a constant. The `users` variable however still exists and is expected to be provided at rendering. If you change your sample code and switch language value to `"fr-fr"` (or anything different from `"en-us"`) then display the page again, you'll see a second generated file appearing in `cache/` folder to store this second pre-evaluated variant of your template.

Now you have all the basics, continue to next section to read about language syntax and how to write real-life templates.

1.3 Template language

1.3.1 Template syntax

Template language in Deval consists in mixed text literal and special statements enclosed within a pair of double braces:

- Everything you type outside from these braces will be rendered without modification except the backslash `\` character used for escaping.
- Anything you type within a pair of `{{` and `}}` will be processed by Deval and produce an output depending on the inner contents.

Most statements start a block that must be ended with a `{{ end }}` marker. The contents you put between such block and its corresponding end marker is called "body" and can contain both text literal and other nested statements.

Backslashes can be used to prevent special characters from being interpreted. Write `\{{` in a template to print a literal pair of opening braces without having them interpreted as the beginning of a Deval statement, or `\\` to print a literal backslash.

C-style comments (between a `/*` and `*/` pair) are also allowed anywhere in Deval statements.

1.3.2 Statements

Print an expression

To print result of an expression `expr`, use the "$" statement:

```
{{ $ variable }}
{{ $ "literal" }}
{{ $ 3 * 5 }}
{{ $ f(x) + y }}
```

Printed value will be inserted without modification into current document, meaning you may need to enclose it with an escaping function (e.g. HTML escaping if you're printing an HTML document) to avoid leaking unescaped characters. The *Expression wrapping* statement may be useful in this context.

As expression `expr` is transformed to a string before being inserted into current document, make sure it's a primitive type (e.g. boolean, integer) or an object that implements `__toString()` method.

Variable assignment

Use "let" statement to declare and assign variables:

```
{{ let name = expr * expr }}{{ $ name }}{{ end }}

{{ let a = f(x), b = g(a) }}
    {{ $ a /* will print the result of f(x) */ }}
    {{ $ b /* will print the result of g(f(x)) */ }}
{{ end }}
```

Variables created this way are accessible within the statement body and override any pre-existing variable with the same name until the end of the "let" block. You can declare as many variables as you want with a single "let" statement, and each one of them can depend on the ones created before it.

Conditional branches

To conditionally hide a block use the "if" statement:

```
{{ if expr1 }}
    expr1 is true
{{ end }}

{{ if expr2 }}
    expr2 is true
{{ elseif expr3 }}
    expr2 is false but expr3 is true
{{ else }}
    both expr2 and expr3 are false
{{ end }}
```

Expressions used as conditions in "if" or "elseif" statements are converted to boolean just like in PHP, see boolean casting in PHP manual for details.

Collection enumeration

Iterate through a collection's keys and values using the "for" statement:

```
{{ for value in ["a", "b", "c"] }}
    {{ $ value }}
{{ end }}

{{ for key, value in pairs }}
    key = {{ $ key }}, value = {{ $ value }}
{{ end }}
```

Objects can be enumerated as well, as long as they implement the traversable interface. An optional "empty" clause can be added and will be displayed if enumerated collection was empty:

```
{{ for value in collection }}
    {{ $ value }}
{{ empty }}
    Collection is empty.
{{ end }}
```

Inclusion & extension

Deval offers two styles of template composition mechanism: inclusion and extension.

Inclusion through "include" statement can be used to import contents of another template into current one ; Deval will act as if contents from imported template was copy-pasted to replace the "include" statement itself:

```
{{ include path/to/other/template.deval }}
```

Extension is a bit more complex and allow reusing the layout of a template while replacing parts of its content. Start by defining an outer template and insert a few "label" statements where you'll want to replace contents:

```
<html>
    <head>
        <title>{{ label title }}</title>
    </head>
    <body>
        {{ label body }}
    </body>
</html>
```

Then write another template that will extend first one and define contents for each "label" block:

```
{{ extend outer.deval }}
    {{ block title }}
        This is my page title!
    {{ block body }}
        And here is some text contents.
{{ end }}
```

Contents specified after each block will replace matching "label" block from extended template.

Path specified in both "include" or "extend" blocks are relative to current template. Use backslash \ character to escape any special character or whitespace in your path.

Expression wrapping

You'll most probably want to escape unsafe values (e.g. user input) before printing their contents from your templates. While this can easily be done by injecting an escaping function and using it to wrap all the expressions you want to print with "$" statements, the "wrap" block provides a nice solution to factorize some code:

```
{{ wrap escape }}
    Every {{ $ expression }} printed within this wrap {{ $ block }} will
    be passed {{ $ through }} an escaping function.
{{ end }}
```

Is equivalent to:

```
Every {{ $ escape(expression) }} printed within this wrap {{ $ escape(block) }}
will be passed {{ $ escape(through) }} an escaping function.
```

The "wrap" statement takes any function as its unique parameter and calls it on every value printed by inner "$" statements. Multiple "wrap" statements can be nested, resulting in functions being called from the innermost to outermost "wrap" statement. A common use case for HTML generation is using a "wrap" statement at the very beginning of the template with an injected function such as htmlentities as variable `html` you can be sure nothing is left unescaped before printing.

When doing so, you may occasionally want to cancel wrapping for printing a safe HTML snippet from some variable without wrapping it. This can be achieved with the "unwrap" statement that cancels the innermost parent "wrap" one:

```
{{ wrap html }}
        <p>This {{ $ variable }} will be HTML-escaped.</p>
        {{ unwrap }}
                <p>This {{ $ raw }} one won't so make sure it doesn't contain␣
↪unvalidated user input!</p>
        {{ end }}
        <p>We're back in {{ $ safe }} context here with auto-escaping enabled.</p>
{{ end }}
```

1.3.3 Expressions

Literal constants

Deval supports following literals in expressions:

- Boolean values e.g. `true` or `false`
- Floating point numbers e.g. `5.32` or `.17`
- Integer numbers e.g. `0` or `42`
- Character strings e.g. `""` or `"hello"` (double quotes only)
- Values arrays e.g. `["a", "b", "c"]` or `[[1, 2], [3, 4]]`
- Key-values arrays e.g. `["i": 3, "j": 7]`
- Undefined value aka `null`

Note only double quotes are accepted for strings, single quotes have no defined meaning in Deval.

Symbol references

Variables can be referenced by their name without the usual "$" character found in PHP scripts. Remember "$" in Deval is used to write a print construct instead!

Any variable you inject for compile-time or runtime evaluation is referenced using the same unified syntax.

Function calls

Invoke functions the same way you do in PHP, passing arguments (if any) between a pair of parenthesis:

```
{{ $ join(":", [1, 2, 3]) }}
```

Note there is no syntactic difference between variables and functions in Deval as there is in PHP, both don't require a "$" character. Functions behave as variables and can be assigned to symbols, passed as arguments or used as array items just like any regular variable.

Member access

Access array items or object properties by appending index or property name between square brackets to your expression:

```
{{ $ array[i] }}
{{ $ dictionary["key"] }}
{{ $ matrix[x * 5][y + 3] }}
```

When accessing property with name as a constant string you can replace square brackets and quotes by a single dot:

```
{{ $ dictionary.key }}
```

Accessing a non-existent member in Deval won't trigger in an error but evaluates to `null` instead.

Due to the fact Deval handles names in a different way than PHP it can't make the difference between properties and functions having the same name within some class, nor it can make the difference between static and instance properties. This means if your object `o` has both a property `$o->member` and a method `$o->member()` then writing `o.member` in a template will lead to undefined result.

Mathematical & logical

Following operators can be used in Deval, by order of precedence:

- `-value`, `!value`: negate or apply boolean "not" on given value
- `a * b`, `a / b`, `a % b`: multiply, divide or get modulo of operands
- `a + b`, `a - b`: add or substract operands
- `a == b`, `a != b`, `a < b`, `a <= b`, `a > b`, `a >= b`: compare operands
- `a && b`: return `a` if `a` was equivalent to `true` or `b` otherwise
- `a || b`: return `a` if `a` was equivalent to `false` or `b` otherwise

Comparison operators are always strict, meaning == and `!=` are equivalent to triple-equal operator in PHP and will consider two values as different when they're of different types. If you want to test equality between a number and its string representation you first need to cast one of the operands (e.g. with one of the *Deval flavor functions*).

Boolean "and" and "or" operators are different from the ones found in PHP as they don't return a `true` or `false` value but one of their operand instead, after checking for their equivalence to true or false using PHP boolean casting rules. This property allows more than just boolean arithmetics:

```
You have {{ $ i }} new message{{ $ i > 1 && "s" }}
```

Boolean test in the above code will result in `false` if `i` is lower or equal to `1` or `"s"` otherwise, printing an "s" after "message" only when needed. Another nice example is this one:

```
{{ $ test && x || y }}
```

Due to operator precedence the above statement will print the value of `x` if both `test` and `x` are true or `y` otherwise. This makes it a close equivalent to a ternary operator having same result except when `test` is true but `x` is not.

Moment control

In some rare situations you may want to control whether an expression must be evaluated at compile-time or at runtime. Moment operators `(+)` and `(-)` offer a solution to this problem by either forcing an expression to be evaluated at compile-time (or raising an error if it can't) or delaying its evaluation to runtime:

```
{{ if (+)state == 1 }}
    do something
{{ end }}
{{ $ ((-)lookup)("something") }}
```

In this example we force `state` to be evaluated at compile-time, meaning its value cannot be left unknown and Deval will be able to either eliminate the test when generating code (because it's known to be successful) or the entire branch otherwise.

In the following statement we prevent the `lookup` function from being invoked even if its value is known at compile-time, maybe because it depends on some global context being setup first. Note the use of parenthesis: removing them would cause the operator to apply on the result of the function call instead (because operator precedence is lower than function invoke), which is not what we wanted here.

Lambda definition

Deval supports definition of lambda functions:

```
{{ let return_three = () => 3 }}
    {{ $ return_three() /* will print 3 */ }}
{{ end }}
{{ let absolute = (i) => i < 0 && -i || i }}
    {{ $ absolute(-5) /* will print 5 */ }}
    {{ $ absolute(7) /* will print 7 */ }}
{{ end }}
{{ let sum = (a, b) => a + b }}
    {{ $ sum(3, 8) /* will print 11 */ }}
{{ end }}
```

Lambda functions can also capture variables from surrounding context (closure):

```
{{ let
    upper_limit = 3,
    pair = find (items, (i) => i < upper_limit) }}
    index = {{ $ pair[0] }}, value = {{ $ pair[1] }}
{{ end }}
```

Given a `find` function like the one available in *Deval flavor functions* and an `items` array of integers, this code would search for the first item lower than 3 and print its index and value.

1.3.4 Options

Deval configuration can be modified through an optional *`Deval\Setup`* parameter passed when creating an instance of *`Deval\CacheRenderer`* or equivalent:

```
<?php

require 'lib/deval/deval.php';

$setup = Deval\Setup();
$setup->style = 'deindent'; // Change some option

$renderer = new Deval\CacheRenderer('template/users.deval', 'cache/', $setup);
```

Following sections list available options.

Whitespace control

The way Deval handles whitespaces in the literal text parts of your templates can be modified through the *Deval\Setup::$style* property. You can use some of Deval predefined styles or pass a PHP function to define your own. Use ',' as a separator between names if you want to apply more than one Deval predefined style:

```
<?php

[ ... ]

$setup->style = 'deindent'; // Use predefined style 'deindent'
// or
$setup->style = 'deindent,collapse'; // Use 'deindent' then 'collapse' ; good choice␣
↪when generating HTML
// or
$setup->style = function ($s) { return trim ($s); }; // Use custom function
```

Predefined styles are:

- deindent: remove a line break character (\n) followed by whitespaces at the beginning and end of each literal text block. This style allows you to indent your template code without adding unwanted blank characters into generated code. As a side-effect you may experience blocks being collapsed when you want to preserve some whitespaces ; this can be fixed by either adding additional line breaks or inserting spaces through Deval statements e.g. `{{ $ " " }}` that won't be removed.
- collapse: replace any sequence of one or more whitespaces by a single one. This setting is useful to generate a more compact output when targeting a language that ignores repeated whitespaces such as HTML.
- preserve: do not remove nor replace any whitespace from template literals.

Default value of style option is 'deindent'.

PHP compatibility

Use the *Deval\Setup::$version* property to change target PHP compatibility version. PHP versions 7 and above unlocked language constructs that were not possible on previous versions (e.g. `$f()()`), saving Deval from resorting to less effective workarounds.

Default value of this option is current executing PHP version (obtained through `PHP_VERSION` constant), meaning you should not have to touch this option except in rare situations if you use Deval to generate and execute code that don't run using the same PHP version.

1.4 Builtin functions

1.4.1 Functional constraint

Deval's ability to evaluate code at compilation or delay it to runtime relies on template code not having any side-effect. Side-effects would make evaluation not predictable by depending on inputs that Deval have no control on, and therefore could break your templates by producing different results depending on when or in which order functions are evaluated. This is why the template language and all *Deval flavor functions* are pure.

Deval has no way to make sure functions you inject are pure, so this responsibility is left to the developer. Remember it's technically possible to inject non-pure functions into a Deval template but you'll most probably break something if you do so, unless you really know what you're doing.

1.4.2 List of builtin functions

Deval offers two predefined "flavors" of builtin functions you can inject:

- Preferred "deval" flavor: a minimal set of pure functions you can safely use in your templates without any risk of unexpected behavior ;
- Alternative "php" flavor: all standard PHP functions made directly available in your templates. Make sure you only use the pure ones, otherwise you may experience unreliable results as explained above.

To inject builtin functions into your template, use the same *Deval\Renderer::inject* method we saw in previous sections:

```
// Inject "deval" flavor builtin functions into your template
$renderer->inject(Deval\Builtin::deval());

// or

// Inject "php" flavor builtin functions into your template
$renderer->inject(Deval\Builtin::php());
```

Deval flavor functions

Following functions are available in *Deval\Builtin::deval* flavor:

array (*value1*[, *value2*[, ...]])
Convert parameters into arrays and merge them together. This function can be used to cast any value to an array (equivalent to PHP's `(array)` cast) and/or concatenate several values as a single array.

Parameters **valueN** (*mixed*) – any value

Returns concatenated parameters $valueN after converting them to arrays

bool (*value*)
Convert input value into boolean, just like a PHP boolean cast would do.

Parameters **value** (*mixed*) – any value

Returns input value converted to boolean

cat (*value1*[, *value2*[, ...]])
Concatenate arrays or strings (all arguments will be treated as arrays or strings depending on the type of first argument).

Parameters **valueN** (*mixed*) – string or array

Returns concatenated values

compare (*value1*, *value2*)
Compare two values and return -1 if first one is lower than second one, 1 if first one is greater than second one, or 0 otherwise. When used on numeric values comparison uses numerical order. When used on strings comparison uses alphabetical order. Two values of different types are always different but the order between them is undefined.

Parameters **valueN** (*mixed*) – any value

Returns comparison order

default (*value*, *fallback*)
Shorthand function to test whether a value is defined and not null.

Parameters **value** (*mixed*) – input value

Returns `value` if defined and not null, `fallback` otherwise

filter(*items*[, *predicate*])

Filter items from an array based on a predicate. If predicate is not specified then `(item) => bool(item)` is used, meaning function will return an array with all items which are equivalent to true using PHP boolean casting rules.

Parameters

- **items** (*any_array*) – input items
- **predicate** (*function*) – predicate callback

Returns array of all items for which `predicate(item)` is true

find(*items*[, *predicate*])

Find first item from an array matching given predicate. If predicate is not specified then `(item) => true` is used, meaning function will return first item from the array.

Parameters

- **items** (*any_array*) – input items
- **predicate** (*function*) – predicate callback

Returns first item from array for which `predicate(item)` is true

flip(*items*)

Return an array where keys and values have been swapped (similar to PHP function array_flip).

Parameters **items** (*any_array*) – input items

Returns array with swapped keys and values

float(*value*)

Convert input value into floading point number, just like a PHP float cast would do.

Parameters **value** (*mixed*) – any value

Returns input value converted to floating point number

group(*items*[, *get_key*[, *get_value*[, *merge*]]])

Group array items together, optionally transforming keys and values and handling key collisions using callback functions. This function will process every key and value from input array and apply specified `get_key` and `get_value` callbacks on them, passing them `value` and `key` as arguments. Resulting key and value are inserted into output array, using `merge` callback to resolve conflict when two values share the same key and passing it both previous and current value as arguments.

This very versatile function can be used in multiple situations depending on the callback you specify. For example when used with default callbacks it will act as a "unique" function and remove duplicates, by using values as keys and solving conflicts by keeping first encountered value.

Parameters

- **items** (*any_array*) – input items
- **get_key** (*function*) – key transform callback, returns `value` if not specified
- **get_value** (*function*) – value transform callback, returns `value` if not specified
- **merge** (*function*) – merge conflict handling callback, returns previous value if not specified

Returns grouped array

int (*value*)
Convert input value into integer number, just like a PHP int cast would do.

Parameters **value** (*mixed*) – any value

Returns input value converted to integer number

join (*items*[, *separator*])
Join array items together in a string using an optional separator (similar to PHP function implode).

Parameters

- **items** (*any_array*) – input items
- **separator** (*string*) – separator, empty string is used if undefined

Returns joined array items as a single string

keys (*items*)
Extract keys from array and make another array out of them (similar to PHP function array_keys).

Parameters **items** (*any_array*) – input items

Returns input item keys

length (*value*)
Return length of an array (number of items) or a string (number of characters).

Parameters **value** (*mixed*) – input array or string

Returns length of input value

map (*items*, *apply*)
Returns an array after applying given callback to all its values, leaving keys unchanged (similar to PHP function array_map).

Parameters **items** (*any_array*) – input items

Returns array of (key, apply(value)) pairs

max (*value1*[, *value2*[, ...]])
Returns highest value in given array when given a single argument, or highest argument when given more than one (similar to PHP function max).

Parameters **valueN** (*mixed*) – array (if one argument) or scalar value (if more)

Returns greatest value or argument

min (*value1*[, *value2*[, ...]])
Returns lowest value in given array when given a single argument, or lowest argument when given more than one (similar to PHP function min).

Parameters **valueN** (*mixed*) – array (if one argument) or scalar value (if more)

Returns lowest value or argument

php (*symbol*)
Access PHP global variable, constant or function by name. Prepend "#" to name to access a constant or "$" to access a variable. Class members can be accessed by prepending their namespace followed by "::" to the symbol name. This function allows you to escape from a safe pure context, so all precautions listed in *Functional constraint* section apply to it.

Parameters **symbol** (*string*) – name of the symbol to access

Returns symbol value

```
{{ $ php("implode")(",", [1, 2]) /* access PHP function */ }}
{{ $ php("#PHP_VERSION") /* access PHP constant */ }}
{{ $ php("$_SERVER")["PHP_SELF"] /* access PHP variable */ }}
{{ $ php("My\\SomeClass::$field") /* access class variable */ }}
{{ $ php("OtherClass::#VALUE") /* access class constant */ }}
```

range (*start*, *stop*[, *step*])
Build a sequence of numbers between given boundaries (inclusive), using a step increment between each value (similar to PHP function **range_**).

Parameters

- **start** (*integer*) – first value of the sequence
- **stop** (*integer*) – last value of the sequence
- **step** (*integer*) – increment between numbers, 1 will be used in not specified

Returns sequence array

reduce (*items*, *callback*[, *initial*])
Reduce array items to a scalar value using a callback function (similar to PHP function array_reduce).

Parameters

- **items** (*any_array*) – input items
- **callback** (*function*) – callback function producing result from aggregated value and current item value
- **initial** (*mixed*) – value used as initial aggregate, `null` if not specified

Returns final aggregated value

replace (*value*, *replacements*)
Replace all occurrences of `replacements` keys by corresponding values (similar to PHP function str_replace but takes a single key-value array for replacements instead of two separate arrays).

Parameters

- **value** (*string*) – original string
- **replacements** (*any_array*) – replacements key-value pairs

Returns string with all keys from `replacements` replaced

reverse (*value*)
Reverse elements in an array (similar to PHP function array_reverse) or characters in a string (similar to PHP function strrev).

Parameters **value** (*mixed*) – input array or string

Returns reversed array or string

slice (*value*, *offset*[, *count*])
Extract delimited slice from given array or string starting at given offset.

Parameters

- **value** (*mixed*) – input array or string
- **offset** (*integer*) – beginning offset of extracted slice
- **count** (*integer*) – length of extracted slice, or extract to the end if not specified

Returns extracted array or string slice

sort (*items*[, *compare*])
Sort input array using optional comparison callback.

Parameters

- **items** (*any_array*) – input items
- **callback** (*function*) – items comparison function, see usort for specification

Returns sorted array

split (*string*, *separator*[, *limit*])
Split string into array using a separator string (similar to PHP function explode).

Parameters

- **string** (*string*) – input string
- **separator** (*string*) – separator string
- **limit** (*integer*) – maximum number of items in output array

Returns array of split strings

str (*value*)
Convert input value into string, just like a PHP string cast would do.

Parameters **value** (*mixed*) – any value

Returns input value converted to string

values (*items*)
Extract values from array and make another array out of them (similar to PHP function array_values).

Parameters **items** (*any_array*) – input items

Returns input item values

void ()
Empty function which always returns `null`, for use as a default placeholder in Deval statements.

Returns null

zip (*keys*, *values*)
Create a key-value array from given list of keys and values (similar to PHP function array_combine). Input arrays `keys` and `values` must have the same length for this function to work properly.

Parameters

- **keys** (*any_array*) – items to be used as array keys
- **values** (*any_array*) – items to be used as array values

Returns key-value array

PHP flavor functions

If you chose to use *Deval\Builtin::php* flavor, all standard PHP functions are available in your templates. Proceed with caution! Using any non-pure function e.g. rand could make your template unreliable as you don't control when exactly it's going to be called nor how many times.

```
{{ if strlen(input) == 0 }}
    Please enter a non-empty value!
{{ end }}
```

1.5 API reference

1.5.1 Renderer

Common interface

class Deval**Renderer**
Main Deval class used to compile and render template after injecting values to it.

inject(*$constants*)
Inject values into template. **Only serializable values can be injected** as Deval may need to store them in intermediate document. If you need to inject functions, give them a name (do not use anonymous functions) and inject this name into Deval.

Parameters

- **$constants** (*array*) – key-value array to inject into template, each value will be injected with name taken from its associated key

render(*$variables = array()*)
Evaluate and render template after optionally specifying runtime values. Any value, serializable or not, can be passed to this method. If you need to inject method, pass an array with class name (for static methods) or instance (for instance methods) as first item and method name as second item.

Parameters

- **$variables** (*array*) – key-value array to inject into template, similar to the one from *Deval\Renderer::inject*

Returns evaluated and rendered template as a string

Implementations

class Deval**CacheRenderer**
Implementation for production usage, using an intermediate cache for pre-evaluated template code.

_construct(*$path*, *$directory*, *$setup = null*)
Create a new *Deval\CacheRenderer* instance.

Parameters

- **$path** (*string*) – path to input template file
- **$directory** (*string*) – directory where cached files will be stored
- **$setup** (Deval\Setup) – renderer configuration

class Deval**FileRenderer**
Implementation for development usage, perform rendering from template file on each request with no caching.

_construct(*$path*, *$setup = null*)
Create a new *Deval\FileRenderer* instance.

Parameters

- **$path** (*string*) – path to input template file
- **$setup** (Deval\Setup) – renderer configuration

class Deval**StringRenderer**
Implementation for development usage, perform rendering from template string on each request with no caching.

_construct (*$source*, *$setup = null*)
Create a new *Deval\StringRenderer* instance.

Parameters

- **$source** (*string*) – template source code
- **$setup** (Deval\Setup) – renderer configuration

1.5.2 Builtin

class Deval**Builtin**
Utility class providing different "flavors" of builtin functions.

deval()
Returns Deval favor builtin functions, see *Deval flavor functions* for details.

Returns functions array

php()
Returns PHP favor builtin functions, see *PHP flavor functions* for details.

Returns functions array

1.5.3 Setup

class Deval**Setup**
Configuration class for *Deval\Renderer*.

property style
Whitespace handling option, see *Whitespace control* for details.

property version
PHP compatibility option, see *PHP compatibility* for details.

1.6 Credits

1.6.1 Thanks

- David Majda & Futago-za Ryuu for PEG.js
- Nordth & James Nylen for phpegjs

CHAPTER 2

Indices and tables

- genindex
- search

PHP Namespace Index

d

Index

Symbols

A

B

C

D

F

G

I

J

K

L

M

P

R

S

V

Z