Ouzo Documentation

Release

Ouzo developers

Jul 21, 2017

Framework

CHAPTER 1

Tutorials

Start with 5 minutes tutorial, read about project structure and then dive deeper into more advanced Ouzo topics.

Project structure explained

Let's walk through the code and see how it works.

Routes

File `myproject/config/routes.php` contains configuration of routing. You can run `./console ouzo:routes` to see all routes exposed by your app.

`Route::get('/', 'users#index');` instructs Ouzo that requests to '/' are handled by method **index** in **UsersController**.

Controller

```
class UsersController extends Controller
{
    public function init()
    {
        $this->layout->setLayout('sample_layout');
    }

    public function index()
    {
        $this->view->users = User::all();
```

```
        $this->view->render();
    }
    ...
```

Function **init** sets layout used by this controller. The default layout adds "Ouzo Framework!" banner and includes bootstrap files.

In the **index** function, we fetch and assign all users to the **users** view variable. You can access this variable in a view as a field (`$this->users`).

In the next line we render a view. By default view name is derived from controller and method names. In this case it will be `Users/index` which means file `View/Users/index.phtml` will be used. You can render other views by passing a parameter to the render method.

```
class UsersController extends Controller
{
    ...
    public function edit()
    {
        $this->view->user = User::findById($this->params['id']);
        $this->view->render();
    }

    public function update()
    {
        $user = User::findById($this->params['id']);
        if ($user->updateAttributesIfValid($this->params['user'])) {
            $this->redirect(userPath($user->id), "User updated");
        } else {
            $this->view->user = $user;
            $this->view->render('Users/edit');
        }
    }
    ...
```

Method **edit** is called when edition page is requested. It assigns `user` variable and renders view.

Method **update** is called when updated user form is submitted. It loads a user by id and then tries to update it. If update succeeds we return redirect to the user page with message *"User updated"*. If update fails we use `$user` variable containing new values to render edition page. It's important that we use the same `$user` variable on which $user->updateAttributesIfValid was called. It will contain values submitted by browser and validation errors that prevented successful update.

Model

```
class User extends Model
{
    public function __construct($attributes = [])
    {
        parent::__construct([
            'attributes' => $attributes,
            'fields' => ['login', 'password']
        ]);
    }
```

```
    public function validate()
    {
        parent::validate();
        $this->validateNotBlank($this->login, 'Login cannot be blank', 'login');
    }
}
```

User class is mapped to the **users** table, primary key defaults to **id** and sequence to **users_id_seq**. Parameter **fields** defines columns that will be exposed as model attributes. You can pass additional options to override the default mapping.

```
parent::__construct([
    'table' => 'other_name'
    'primaryKey' => 'other_id',
    'sequence' => 'other_sequence'
    'attributes' => $attributes,
    'fields' => ['login', 'password']
]);
```

Function **validate** is called by function **isValid** and **updateAttributesIfValid**. **validateNotBlank** takes a value to validate, error message and a field that is highlighted in red when validation fails.

View

`Application/View/Users/edit.phtml` contains users edition page.

```
<?php echo renderPartial('Users/_form', [
    'user' => $this->user,
    'url' => userPath($this->user->id),
    'method' => 'PUT',
    'title' => 'Edit user'
]);
```

Function **renderPartial** displays a fragment of php code using variables passed in the second argument. By convention partials names start with underscore. We extracted `Users/_form` partial so that we can use the same form for user creation and update.

`Users/_form` looks as follows:

```
<?php echo showErrors($this->user->getErrors()); ?>

<div class="panel panel-default">
    <div class="panel-heading"><?php echo $this->title; ?></div>
    <div class="panel-body">
        <?php $form = formFor($this->user); ?>
        <?php echo $form->start($this->url, $this->method, ['class' => 'form-
↪horizontal']); ?>

        <div class="form-group">
            <?php echo $form->label('login', ['class' => 'control-label col-lg-2']); ?
↪>

            <div class="col-lg-10">
                <?php echo $form->textField('login') ?>
            </div>
```

```
        </div>

        <div class="form-group">
            <?php echo $form->label('password', ['class' => 'control-label col-lg-2
↪']); ?>

            <div class="col-lg-10">
                <?php echo $form->passwordField('password'); ?>
            </div>
        </div>

        <div class="form-group">
            <div class="col-lg-offset-2 col-lg-10">
                <button type="submit" class="btn btn-primary">Save</button>
                <?php echo linkButton(['name' => 'cancel', 'value' => 'Cancel', 'url'␣
↪=> usersPath(), 'class' => "btn btn-default"]); ?>
            </div>
        </div>

        <?php echo $form->end(); ?>
    </div>
</div>
```

Function **showErrors** displays validation errors set on our model. In the line #6 we create a form for the user model. Method `$form->start` displays form html element for the given url.

Lines:

```
$form->label('login', ['class' => 'control-label col-lg-2']);
//<label for="user_login" class="control-label col-lg-2">Login</label>
$form->textField('login');
//<input type="text" id="user_login" name="user[login]" value="thulium">
```

display label and text input for user's login.

Label text is taken from translations (`locales/en.php`) by a key that is a concatenation of the model and field names. In this case it's *'user.login'*.

Routes

Concept of routings

When your application receives a request e.g.:

`GET` **`/users/12`**

it needs to be matched to a controller and action. This case can be served by the following route rule:

```
Route::get('/users/:id', 'users#show');
```

This request will be dispatched to the `users` controller's and `show` action with `[id => 12]` in `params`.

Basic types of routes

GET route

```
Route::get('/users/add' 'users#add');
```

HTTP request method must be `GET`, then router finds `users` controller and `add` action.

POST route

```
Route::post('/users/create' 'users#create');
```

HTTP request method must be `POST`, then router finds `users` controller and `create` action. POST parameters are also available in `$this->params`.

DELETE route

```
Route::delete('/users/destroy' 'users#destroy');
```

HTTP request method must be `DELETE`, then router finds `users` controller and `destroy` action.

PUT route

```
Route::put('/users/update' 'users#update');
```

HTTP request method must be `PUT`, then router finds `users` controller and `edit` action.

Any route

```
Route::any('/users/show_items' 'users#show_items');
```

HTTP request must be one of `GET`, `POST`, `PUT`, `PATCH` or `DELETE`.

Route parameters

In Ouzo you can use parametrized URLs.

```
Route::get('/users/show/id/:id/name/:name' 'users#show');
```

This route provides mapping between HTTP verbs to controller and action. Parameters will be available in `$this->params` as map - `[id => value, name => value]`. E.g.:

`GET` **`/users/show/id/12/name/John`**

will dispatch to `users` controller, `show` action and map of parameters `[id => 12, name => John]`.

Resource route

This type of route simplifies mapping of RESTful controllers.

```
Route::resource('phones');
```

This route creates a default REST routing:

URL Helper	HTTP Verb	Path	Controller#Action
phonesPath	GET	/phones	phones#index
freshPhonePath	GET	/phones/fresh	phones#fresh
editPhonePath	GET	/phones/:id/edit	phones#edit
phonePath	GET	/phones/:id	phones#show
phonesPath	POST	/phones	phones#create
phonePath	PUT	/phones/:id	phones#update
phonePath	PATCH	/phones/:id	phones#update
phonePath	DELETE	/phones/:id	phones#destroy

Options

as

You can rename generated routes using `as` option:

```
Route::get('/agents', 'agents#index', ['as' => 'my_name']);
```

Grouping routes

Ouzo provides functionality to grouping routes. You can handle that case using:

```
Route::group("api", function() {
    GroupedRoute::post('/users/:id/archive', 'users#archive');
    GroupedRoute::resource('users');
    GroupedRoute::get('/users/:id/orders', 'users#orders');
});
```

Above example is equivalent for the:

```
Route::post('/api/users/:id/archive', 'api/users#archive');
Route::resource('api/users');
Route::get('/api/users/:id/orders', 'api/users#orders');
```

Console tool

Listing defined routes

Ouzo provides a command tool to display all defined routes. You can execute `./console ouzo:routes` in terminal to produce output with registered routes. This is a sample output:

```
+-----------------+-----------+---------------------------------------+----------------
↪---+
| URL Helper      | HTTP Verb | Path                                  | Controller
↪#Action |
+-----------------+-----------+---------------------------------------+----------------
↪---+
| indexIndexPath  | GET       | /                                     | index#index  ␣
↪    |
|                 | ALL       | /users                                | users        ␣
↪    |
|                 |           |   except:                             |              ␣
↪    |
|                 |           |     new                               |              ␣
↪    |
|                 |           |     select_outbound_for_user          |              ␣
↪    |
| indexAgentsPath | GET       | /agents/index                         | agents#index ␣
↪    |
| indexAgentsPath | POST      | /agents/index                         | agents#index ␣
↪    |
|                 | ALL       | /photos                               | photos       ␣
↪    |
| indexAgentsPath | ANY       | /agents/index                         | agents#index ␣
↪    |
| phonesPath      | GET       | /phones                               | phones#index ␣
↪    |
| freshPhonePath  | GET       | /phones/fresh                         | phones#fresh ␣
↪    |
| editPhonePath   | GET       | /phones/:id/edit                      | phones#edit  ␣
↪    |
| phonePath       | GET       | /phones/:id                           | phones#show  ␣
↪    |
| phonesPath      | POST      | /phones                               | phones#create ␣
↪    |
| phonePath       | PUT       | /phones/:id                           | phones#update ␣
↪    |
| phonePath       | PATCH     | /phones/:id                           | phones#update ␣
↪    |
| phonePath       | DELETE    | /phones/:id                           | phones#destroy␣
↪    |
| myNamePath      | GET       | /agents                               | agents#index ␣
↪    |
| showAgentsPath  | GET       | /agents/show/id/:id/call_id/:call_id  | agents#show  ␣
↪    |
+-----------------+-----------+---------------------------------------+----------------
↪---+
```

This tool can display routes per controller. Used with `-c` parameter - `./console ouzo:routes -c=phones`, produces output:

```
+-----------------+-----------+---------------------------------------+----------------
↪---+
| URL Helper      | HTTP Verb | Path                                  | Controller
↪#Action |
+-----------------+-----------+---------------------------------------+----------------
↪---+
| phonesPath      | GET       | /phones                               | phones#index  ␣
↪    |
| freshPhonePath  | GET       | /phones/fresh                         | phones#fresh  ␣
↪    |
| editPhonePath   | GET       | /phones/:id/edit                      | phones#edit   ␣
↪    |
| phonePath       | GET       | /phones/:id                           | phones#show   ␣
↪    |
| phonesPath      | POST      | /phones                               | phones#create ␣
↪    |
| phonePath       | PUT       | /phones/:id                           | phones#update ␣
↪    |
| phonePath       | PATCH     | /phones/:id                           | phones#update ␣
↪    |
| phonePath       | DELETE    | /phones/:id                           | phones#destroy␣
↪    |
+-----------------+-----------+---------------------------------------+----------------
↪---+
```

Generating the UriHelper functions

Route tool can generate `UriHelper` functions too. Used with `-g`, parameter creates or overwrites file `Application/Helper/GeneratedUriHelper.php` which should be included in `UriHelper.php` in the same location. To generate this file use `./console ouzo:routes -g`. E.g.:

Route:

```
Route::get('/agents', 'agents#index', ['as' => 'my_name']);
```

Displayed:

```
| myNamePath  | GET       | /agents                           | agents#index      |
```

Can be used in application:

```
$agentsUrl = myNamePath();
```

ORM

Model definition

This code will map `Category` class to a `categories` table with *id* as a primary key and one column *name*.

```
class Category extends Model
{
    public function __construct($attributes = [])
    {
```

```
        parent::__construct([
            'attributes' => $attributes,
            'fields' => ['name']
        ]);
    }
}
```

`Model` constructor accepts the following parameters:

- `table` - defaults to pluralized class name. E.g. customer_orders for `CustomerOrder`
- `primaryKey` - defaults to `id`
- `sequence` - defaults to `table_primaryKey_seq`
- `hasMany` specification of a has-many relation e.g. `['name' => ['class' => 'Class', 'foreignKey' => 'foreignKey']]`
- `hasOne` specification of a has-one relation e.g. `['name' => ['class' => 'Class', 'foreignKey' => 'foreignKey']]`
- `belongsTo` specification of a belongs-to relation e.g. `['name' => ['class' => 'Class', 'foreignKey' => 'foreignKey']]`
- `fields` - mapped column names
- `attributes` - array of `column => value`
- `beforeSave` - function to invoke before *insert* or *update*
- `afterSave` - function to invoke after *insert* or *update*

Columns specified by **'fields'** parameter are exposed with magic getter and setter.

Working with model objects

Creating new instances

You can create an instance using Model's constructor or `` `Model::newInstance `` method. They both take an array of attributes as an optional parameter.

```
$user = new User();
$user = new User(['name' => 'bob']);

$user = User::newInstance(['name' => 'bob']);
```

Instances created using constructor and `Model::newInstance` method are not inserted into db. Validation is also not performed.

If you want to create, validate and save an instance, you can use `Model::create` method.

```
$user = User::create(['name' => 'bob']);
```

If validation fails, `ValidationException` is thrown.

Saving and updating

You can save a new instance using `insert` method. It returns the value of the primary key of the newly inserted row. You can update an existing object using `update` method. If you are not sure if an object was already saved you can call `insertOrUpdate` method.

```
$product = new Product();
$product->name = 'Phone';

$id = $product->insert();

$product->name = 'Super Phone';
$product->update();

$product->name = 'Phone';
$product->insertOrUpdate();
```

Before and after save callbacks

You can call defined methods before/after save or update.

```
class Product
{
    private $_fields = ['description', 'name', 'id_category', 'id_manufacturer', 'sale
↪'];

    public function __construct($attributes)
    {
        parent::__construct([
                'attributes' => $attributes,
                'fields' => $this->_fields,
                'beforeSave' => 'addExclamationMarkToDescription'
            ]);
    }

    function addExclamationMarkToDescription()
    {
        if ($this->description) {
            $this->description .= '!';
        }
    }
}
```

All saves or updates will be adding an exclamation mark to description. This callback accepts following types of callback:

- string e.g. `'methodName'`
- array e.g. `['methodName1', 'methodName2']`
- lambda e.g. `function() { ... }`

Update of multiple records

You can update specific columns in records matching given criteria.

```
$affectedRows = User::where(['name' => 'bob'])
                 ->update(['name' => 'eric']);
```

Issued sql query:

```
UPDATE users set name = ? WHERE name = ? Params: ['eric', 'bob']
```

Default field values

You can define default values for fields in two ways - using **string** or **anonymous function**.

```
[
    'description' => 'no desc',
    'name' => function() {
        return 'no name';
    }
]
```

Now if you create a new model object these fields will be set to their default values.

```
class ModelWithDefaults extends Model {
    public function __construct($attributes = []) {
        parent::__construct([
            'attributes' => $attributes,
            'fields' => [
                'description' => 'no desc',
                'name' => function() {
                    return 'no name';
                }
            ]
        ]);
    }
}

$modelWithDefaults = new ModelWithDefaults();
echo $modelWithDefaults->description; // no desc
echo $modelWithDefaults->name; // no name
```

Validation

You can validate the state of objects with `Model::validate` method. Just override it in you model and implement all necessary checks.

```
public function validate()
{
    parent::validate();
    $this->validateNotBlank($this->name, 'Name cannot be blank.', 'name');
    $this->validateTrue($this->accepted, 'Accepted should be true');
}
```

Second parameter specifies the message that will be used in the case of error. Third parameter specifies the field name so that the corresponding input can be highlighted in the html form.

You can check if a model object is valid by calling `Model::isValid` method. If validation fails it returns false and sets errors attribute. You can then see what was wrong calling `getErrors` (for error messages) or

`getErrorFields` (for invalid fields).

If your object has relations to other objects and you want to validate them altogether you can call `validateAssociated` method passing other objects.

```
public function validate()
{
    parent::validate();
    $this->validateAssociated($this->child);
}
```

Validation is provided by `Validatable` class. You can easily add validation to other classes by extending `Validatable`.

Fetching objects

findById

Loads object for the given primary key. If object does not exist, exception is thrown

findByIdOrNull

Loads object for the given primary key. If object does not exist, null is returned.

findBySql

Executes a native sql and returns an array of model objects created by passing every result row to the model constructor.

- `$nativeSql` - database specific sql
- `$params` - bind parameters

```
User::findBySql('select * from users');
User::findBySql('select * from users where login like ?', "%cat%");
```

Normally, there's no reason to use `findBySql` as Ouzo provides powerful query builder described in another section.

Relations

Relations are used to express associations between Models. You can access relation objects using Model properties (just like other attributes). Relation object are lazy-loaded when they are accessed for the first time and cached for subsequent use.

For instance, if you have a `User` model that belongs to a `Group`:

```
$group = Group::create(['name' => 'Admin']);
$user = User::create(['login' => 'bob', 'group_id' => $group->id]);
```

You can access user's group as follows: `echo $user->group->name;`

Ouzo supports 3 types of associations:

- **Belongs to** - expresses 1-1 relationship. It's specified by `belongsTo` parameter. Use `belongsTo` in a class that contains the foreign key.
- **Has one** - expresses 1-1 relationship. It's specified by `hasOne` parameter. Use `hasOne` in a class that contains the key referenced by the foreign key.
- **Has many** - expresses One-to-many relationship. It's specified by `hasMany` parameter.

Relations are defined by following parameters:

- **class** - name of the associated class.
- **foreignKey** - foreign key.
- **referencedColumn** - column referenced by the foreign key. By default it's the primary key of the referenced class.

Note that **foreignKey** and **referencedColumn** mean different things depending on the relation type.

Let's see an example.

We have products that are assigned to exactly one category, and categories that can have multiple products.

```
class Category extends Model
{
    public function __construct($attributes = [])
    {
        parent::__construct([
            'hasMany' => [
                'products' => ['class' => 'Product', 'foreignKey' => 'category_id']
            ],
            'attributes' => $attributes,
            'fields' => ['name']]);
    }
}
```

`foreignKey` in `Category` specifies column in `Product` that references the `categories` table. Parameter `referencedColumn` was omitted so the Category's primary key will be used by default.

```
class Product extends Model
{
    public function __construct($attributes = [])
    {
        parent::__construct([
            'attributes' => $attributes,
            'belongsTo' => [
                'category' => ['class' => 'Category', 'foreignKey' => 'category_id'],
            ],
            'fields' => ['description', 'name', 'category_id']]);
    }
}
```

`foreignKey` in `Product` specifies column in `Product` that references the `categories` table. Parameter `referencedColumn` was omitted so again the Category's primary key will be used.

Inline Relation

If you want to join your class with another class without specifying the relation in the constructor, you can pass a relation object to the `join` method

```
User::join(Relation::inline([
  'class' => 'Animal',
  'foreignKey' => 'name',
  'localKey' => 'strange_column_in_users'
]))->fetchAll();
```

Cyclic relations

Normally, it suffices to specify **class** and **foreignKey** parameters of a relation. However, if your models have cycles in relations (e.g. User can have a relation to itself) you have to specify **referencedColumn** as well (Ouzo is not able to get primary key name of the associated model if there are cycles).

Conditions in relations

If you want to customize your relation you can use **conditions** mechanism. For example, to add a condition use string or array:

```
'hasOne' => [
    'product_named_billy' => [
        'class' => 'Test\Product',
        'foreignKey' => 'id_category',
        'conditions' => "products.name = 'billy'"
    ]
]
```

you can use a closure too:

```
'products_ending_with_b_or_y' => [
    'class' => 'Test\Product',
    'foreignKey' => 'id_category',
    'conditions' => function () {
        return WhereClause::create("products.name LIKE ? OR products.name LIKE ?", ['
↪%b', '%y']);
    }
]
```

Sorted hasMany relation

You specify order of elements in hasMany relation:

```
'hasMany' => [
    'products_ordered_by_name' => [
        'class' => 'Test\Product',
        'foreignKey' => 'id_category',
        'order' => "products.name ASC"
    ]
]
```

You can also order relation by multiple columns:

```
'product_ordered_by_name' => [
    'class' => 'Test\Product',
    'foreignKey' => 'id_category',
    'order' => ["products.name ASC", "products.description DESC"]
]
```

Query builder

It's a fluent interface that allows you to programmatically build queries.

Fully-fledged example:

```
$orders = Order::alias('o')
        ->join('product->category', ['p', 'ct'])
        ->innerJoin('customer', 'c')
        ->where([
            'o.tax'  => [7, 22],
            'p.name' => 'Reno',
            'ct.name' => 'cars'])
        ->with('customer->preferences')
        ->offset(10)
        ->limit(12)
        ->order(['ct.name asc', 'p.name desc'])
        ->fetchAll();
```

Where

Single parameter

Simplest way to filter records is to use where clause on Model class e.g.

```
User::where(['login' => 'ouzo'])->fetch();
```

In the above example we are searching for a user, who has login set to ouzo. You can check the log files (or use Stats class in debug mode) to verify that the database query is correct:

```
SELECT users.* FROM users WHERE login = ? Params: ["ouzo"]
```

Alternative syntax:

```
User::where('login = ?', 'ouzo')->fetch();
```

Multiple parameters

You can specify more than one parameter e.g.

```
User::where(['login' => 'ouzo', 'password' => 'abc'])->fetch();
```

Which leads to:

```
SELECT users.* FROM users WHERE (login = ? AND password = ?) Params: ["ouzo", "abc"]
```

Alternative syntax:

```
User::where('login = ? AND password = ?', ['ouzo', 'abc'])->fetch();
```

Restrictions

You can use restriction mechanism to build where conditions. Usage:

```
Product::where(['name' => Restrictions::like('te%')])->fetch()
```

Supported restrictions:

- **between**

`['count' => Restrictions::between(1, 3)]` produces `SELECT * FROM table WHERE (count >= ? AND count <= ?) Params: [1, 3]`

Between restriction handles four modes: INCLUSIVE (default), EXCLUSIVE, RIGHT_EXCLUSIVE, LEFT_EXCLUSIVE. Just pass it as a third parameter e.g. `Restrictions::between(1, 3, Between::EXCLUSIVE)`.

- **equalTo**

`['name' => Restrictions::equalTo('some name')]` produces `SELECT * FROM table WHERE name = ? Params: ["some name"]`

- **notEqualTo**

`['name' => Restrictions::notEqualTo('some name')]` produces `SELECT * FROM table WHERE name <> ? Params: ["some name"]`

- **greaterOrEqualTo**

`['count' => Restrictions::greaterOrEqualTo(3)]` produces `SELECT * FROM table WHERE count >= ? Params: [3]`

- **greaterThan**

`['count' => Restrictions::greaterThan(3)]` produces `SELECT * FROM table WHERE count > ? Params: [3]`

- **lessOrEqualTo**

`['count' => Restrictions::lessOrEqualTo(3)]` produces `SELECT * FROM table WHERE count <= ? Params: [3]`

- **lessThan**

`['count' => Restrictions::lessThan(3)]` produces `SELECT * FROM table WHERE count < ? Params: [3]`

- **like**

`['name' => Restrictions::like("some%")]` produces `SELECT * FROM table WHERE name LIKE ? Params: ["some%"]`

- **isNull**

`['name' => Restrictions::isNull()]` produces `SELECT * FROM table WHERE name IS NULL`

- **isNotNull**

`['name' => Restrictions::isNotNull()]` produces `SELECT * FROM table WHERE name IS NOT NULL`

- **regexp**

```
['name' => Restrictions::regexp('so.*')]
```

for Postgres driver produces

```
SELECT * FROM table WHERE name ~ ? Params:  ["so.*"]
```

for MySql driver produces

```
SELECT * FROM table WHERE name REGEXP ? Params:  ["so.*"]
```

for Sqlite driver produces

```
SELECT * FROM table WHERE name REGEXP ? Params:  ["so.*"]
```

Note: For Sqlite driver `sqlite3-pcre` package must be installed to support regular expressions.

Parameters chaining

Where clauses can be chained e.g.

```
User::where(['login' => 'ouzo'])
    ->where(['password' => 'abc'])
    ->fetch();
```

SQL query will be exactly the same as in the previous example.

OR operator

Where clauses are chained with AND operator. In order to have OR operator you need to use `Any::of` function e.g.

```
User::where(Any::of(['login' => 'ouzo', 'password' => 'abc']))
    ->fetch();
```

Query:

```
SELECT users.* FROM users WHERE login = ? OR password = ? Params: ["ouzo", "abc"]
```

You can use parameters chaining as described in previous section and combine `Any:of` with standard `where`.

If you wish to use multiple values for the same key, you can use Restrictions:

```
User::where(Any::of(['login' => [Restrictions::equalTo('ouzo'), Restrictions::equalTo(
↪'rules')]]))
    ->fetch();
```

Multiple values

If you want to search for any of values equal to given parameter:

```
User::where(['login' => ['ouzo', 'admin']])->fetch();
```

It results in:

```
SELECT users.* FROM users WHERE login IN (?, ?) Params: ["ouzo", "admin"]
```

It is not possible to use alternative syntax for this type of query.

Note: Please, remember that if you want to retrieve more than one record you need to use fetchAll instead of fetch:

```
User::where(['login' => ['ouzo', 'admin']])->fetchAll();
```

Retrieve all records

All records of given type can be fetched by using empty where clause:

```
User::where()->fetchAll();
```

Or shortened equivalent:

```
User:all();
```

Join

Types:

- `Model::join` or `Model::leftJoin` - left join,
- `Model::innerJoin` - inner join,
- `Model::rightJoin` - right join.

Relation definition

As a first step relations have to be defined inside a Model class. Let's say there is User, which has one Product. User definition needs `hasOne` relation:

```
class User extends Model
{
    public function __construct($attributes = [])
    {
        parent::__construct([
            'attributes' => $attributes,
            'hasOne' => ['product' => [
                                    'class' => 'Product',
                                    'foreignKey' => 'user_id']],
            'fields' => ['login', 'password']]);
    }
}
```

The relation name is `product`, it uses `Product` class and is mapped by user_id column in the database.

Single join

Now `join` can be used to retrieve User together with Product:

```
User::join('product')->fetch();
```

Query:

```
SELECT users.*, products.* FROM users
LEFT JOIN products ON products.user_id = users.id
```

Product can be referred from User object:

```
$user = User::join('product')->fetch();
echo $user->product->name;
```

Join can be combined with other parts of query builder (where, limit, offset, order etc.) e.g.

```
User::join('product')->where(['products.name' => 'app'])->fetch();
```

Query:

```
SELECT users.*, products.* FROM users
LEFT JOIN products ON products.user_id = users.id
WHERE products.name = ? Params: ["app"]
```

Multiple joins / join chaining

You can chain join clauses:

```
User::join('product')
   ->join('group')->fetchAll();
```

Nested joins

You can join models through other models with nested joins.

Let's assume that you have Order that has Product and Product has Category:

```
$order = Order::join('product->category')->fetch();
```

```
SELECT orders.*, products.*, categories.*
FROM orders
LEFT JOIN products ON products.id = orders.product_id
LEFT JOIN categories ON categories.id = products.category_id
```

Returned order will contain fetched product and that product will contain category. The following code will echo category's name without querying db:

```
echo $order->product->category->name;
```

Aliasing

Normally if you want to reference a table in the query builder you have to use the table name. When you join multiple Models it may be cumbersome. That is when aliases come in handy.

```
$product = Product::alias('p')
        ->join('category', 'c')
        ->where(['p.name' => 'a', 'c.name' => 'phones'])
        ->fetch();
```

```
SELECT p.*, c.*
FROM products AS p
LEFT JOIN categories AS c ON c.id = p.category_id
WHERE p.name = 'a' and c.name = 'phones'
```

If you want to alias tables in nested join you can pass array of aliases as a second parameter of `join` method.

```
$orders = Order::alias('o')
        ->join('product->category', ['p', 'c'])
        ->where([
            'o.tax'  => 7
            'p.name' => 'Reno',
            'c.name' => 'cars'])
        ->fetchAll();
```

With

`ModelQueryBuilder::with` method instructs ouzo to fetch results with their relations.

The following code will return products with their categories.

```
$products = Product::where()->with('category')->fetchAll();
```

Ouzo will query db for products, then load all corresponding categories with one query.

```
SELECT products.* FROM products
SELECT categories.* FROM categories WHERE id IN (?, ?, ..,) Params: [product1.
↪category_id, product2.category_id, ..., productN.category_id]
```

You can chain `with` methods. You can also use `with` to fetch nested relations.

```
$orders = Order::where()
   ->with('product->category')
   ->fetchAll();
```

Ouzo will first load all matching orders, then their products, and then products' categories:

```
SELECT orders.* FROM orders
SELECT products.* FROM products WHERE id IN (?, ?, ...)
SELECT categories.* FROM categories WHERE id IN (?, ?, ...)
```

For `hasOne` and `belongsTo` relations you can use `join` instead. However, joins with `hasMany` relations will not fetch associated objects so `with` is the only way of fetching them eagerly.

Count

Count all records

Counting all records of given type:

```
User::count()
```

As a result integer with size is returned. Query:

```
SELECT count(*) FROM users
```

Count with where

Count method accepts same arguments as where e.g.

```
User::count(['login' => 'ouzo']);
```

Query:

```
SELECT count(*) FROM users WHERE login = ? Params: ["ouzo"]
```

Limit and offset

Limit

In order to limit number of records to retrieve use `limit` method with integer argument:

```
User::where()->limit(10)->fetch();
```

It returns first 10 records:

```
SELECT users.* FROM users LIMIT ? Params: [10]
```

Offset

Usually used with `limit` method, it sets offset (integer) from which records will be retrieved:

```
User::where()->offset(5)->fetch();
```

Query:

```
SELECT users.* FROM users OFFSET ? Params: [5]
```

Combined with `limit`:

```
User::where()->limit(10)->offset(5)->fetch();
```

Query:

```
SELECT users.* FROM users LIMIT ? OFFSET ? Params: [10, 5]
```

Order

Order by one column

To sort the result:

```
User::where()->order('login')->fetch();
```

Query:

```
SELECT users.* FROM users ORDER BY login
```

Order by multiple columns

If array is given as an argument the method sorts by multiple columns:

```
User::where()->order(['login', 'id'])->fetch();
```

Query:

```
SELECT users.* FROM users ORDER BY login, id
```

Sort direction

Ascending or descending:

```
User::where()->order(['login asc', 'id desc'])->fetch();
```

Query:

```
SELECT users.* FROM users ORDER BY login asc, id desc
```

Transactions

You can control transactions manually:

```
Db::getInstance()->beginTransaction();
try {
    Db::getInstance()->commitTransaction();
    //do something
} catch (Exception $e) {
    Db::getInstance()->rollbackTransaction();
}
```

You can run a callable object in a transaction:

```
$result = Db::getInstance()->runInTransaction(function() {
   //do something
   return $result;
});
```

You can also proxy an object so that all methods become transactional:

```
$user = new User(['name' => 'bob']);
$transactionalUser = Db::transactional($user);

$transactionalUser->save(); //runs in a transaction
```

Locking

If you want to lock a particular rows in a table with SELECT ... FOR UPDATE use lockForUpdate:

```
User::where(['name' => 'bob'])->lockForUpdate()->fetch();
```

Tests

Controller test case

Ouzo provides `ControllerTestCase` which allows you to verify that:

- there's a route for a given url
- controllers methods work as expected
- views are rendered without errors

```
<?php
class UsersControllerTest extends ControllerTestCase
{
    /**
     * @test
     */
    public function shouldRenderIndex()
    {
        //when
        $this->get('/users');

        //then
        $this->assertRenders('Users/index');
    }

    /**
     * @test
     */
    public function shouldRedirectToIndexOnSuccessInCreate()
    {
        //when
        $this->post('/users', [
```

```
            'user' => [
                'login' => 'login'
            ]]
        );

        //then
        $this->assertRedirectsTo(usersPath());
    }
}
```

Methods

- `get($url)` - mock GET request for given url
- `post($url, $data)` - mock POST request with data for given url
- `put($url, $data)` - mock PUT request with data for given url
- `patch($url)` - mock PATCH request for given url
- `delete($url)` - mock DELETE request for given url
- `getAssigned($name)` - get value of $name variable assigned to the rendered view.
- `getRenderedJsonAsArray()` - get returned JSON as array
- `getResponseHeaders()` - get all response header

Assertions

- `assertRedirectsTo($path)`
- `assertRenders($viewName)` - asserts that the given view was rendered
- `assertAssignsModel($variable, $modelObject)` - asserts that a model object was assigned to a view
- `assertDownloadsFile($file)`
- `assertAssignsValue($variable, $value)`
- `assertRenderedContent()` - returns StringAssert for rendered content.
- `assertRenderedJsonAttributeEquals($attribute, $equals)`
- `assertResponseHeader($expected)`

Database test case

Ouzo provides `DbTransactionalTestCase` class that takes care of transactions in tests. This class starts a new transaction before each test case and rolls it back afterwards.

```
<?php
class UserTest extends DbTransactionalTestCase
{
    /**
     * @test
```

```
     */
    public function shouldPersistUser()
    {
        //given
        $user = new User(['name' => 'bob']);

        //when
        $user->insert();

        //then
        $storedUser = User::where(['name' => 'bob'])->fetch();
        $this->assertEquals('bob', $storedUser->name);
    }
}
```

Model assertions

`Assert::thatModel` allows you to check if two model objects are equal.

Sample usage

```
<?php
class UserTest extends DbTransactionalTestCase
{
    /**
     * @test
     */
    public function shouldPersistUser()
    {
        //given
        $user = new User(['name' => 'bob']);

        //when
        $user->insert();

        //then
        $storedUser = User::where(['name' => 'bob'])->fetch();
        Assert::thatModel($storedUser)->isEqualTo($user);
    }
}
```

Assertions

- `isEqualTo($expected)` - compares all attributes. If one model has loaded a relation and other has not, they are considered not equal. Attributes not listed in model's fields are also compared
- `hasSameAttributesAs($expected)` - compares only attributes listed in Models fields

String assertions

`Assert::thatString` allows you to check strings as a fluent assertions.

Sample usage

```
Assert::thatString("Frodo")
     ->startsWith("Fro")->endsWith("do")
     ->contains("rod")->doesNotContain("fro")
     ->hasSize(5);

Assert::thatString("Frodo")->matches('/Fro\w+/');
Assert::thatString("Frodo")->isEqualToIgnoringCase("frodo");
Assert::thatString("Frodo")->isEqualTo("Frodo");
Assert::thatString("Frodo")->isEqualNotTo("asd");
```

Assertions

- `contains($substring)` - check that string contains substring
- `doesNotContain($substring)` - check that string does not contains substring
- `startsWith($prefix)` - check that string is start with prefix
- `endsWith($postfix)` - check that string is end with postfix
- `isEqualTo($string)` - check that string is equal to expected
- `isEqualToIgnoringCase($string)` - check that string is equal to expected (case insensitive)
- `isNotEqualTo($string)` - check that string not equal to expected
- `matches($regex)` - check that string is fit to regexp
- `hasSize($length)` - check string length
- `isNull()` - check a string is null
- `isNotNull()` - check a string is not null
- `isEmpty()` - check a string is empty
- `isNotEmpty()` - check a string is not empty

Array assertions

`Assert::thatArray` is a fluent array assertion to simplify your tests.

Sample usage

```
<?php
$animals = ['cat', 'dog', 'pig'];
Assert::thatArray($animals)->hasSize(3)->contains('cat');
Assert::thatArray($animals)->containsOnly('pig', 'dog', 'cat');
Assert::thatArray($animals)->containsExactly('cat', 'dog', 'pig');
```

Note: Array assertions can also be used to examine array of objects. Methods to do this is `onProperty` and `onMethod`.

Using `onProperty`:

```
<?php
$object1 = new stdClass();
$object1->prop = 1;

$object2 = new stdClass();
$object2->prop = 2;

$array = [$object1, $object2];
Assert::thatArray($array)->onProperty('prop')->contains(1, 2);
```

Using `onMethod`:

```
Assert::thatArray($users)->onMethod('getAge')->contains(35, 24);
```

Assertions

- `contains($element ..)` - vararg elements to examine that array contains them
- `containsOnly($element ..)` - vararg elements to examine that array contains **only** them
- `containsExactly($element ..)` - vararg elements to examine that array contain **exactly** elements in pass order
- `hasSize($expectedSize)` - check size of the array
- `isNotNull()` - checks if array is not null
- `isEmpty()` - checks if array is empty
- `isNotEmpty()` - checks if array is not empty
- `containsKeyAndValue($elements)`
- `containsSequence($element ..)` - checks if vararg sequence exists in the array
- `excludes($element ..)`
- `hasEqualKeysRecursively(array $array)`
- `isEqualTo($array)`
- `keys()` - next assertions will be against array keys, not values

Exception assertions

CatchException enables you to write a unit test that checks that an exception is thrown.

Sample usage

```
//given
$foo = new Foo();

//when
CatchException::when($foo)->method();

//then
CatchException::assertThat()->isInstanceOf("FooException");
```

Assertions

- `isInstanceOf($exception)`
- `isEqualTo($exception)`
- `notCaught()`
- `hasMessage($message)`

Session assertions

`Assert::thatSession` class comes with a handy method to test your session content.

Sample usage

```
// when
Session::set('key1', 'value1');
       ->set('key2', 'value2');

// then
Assert::thatSession()
      ->hasSize('2')
      ->contains('key2' => 'value2');
```

Note: This assert has the same method as `Assert::thatArray`.

Testing time-dependent code

We do recommend you to use Clock instead of DateTime. Clock provides time travel and time freezing capabilities, making it simple to test time-dependent code.

```
//given
Clock::freeze('2011-01-02 12:34');

//when
$result = Clock::nowAsString('Y-m-d');
```

```
//then
$this->assertEquals('2011-01-02', $result);
```

See also:

Clock

Mocking

Ouzo provides a Mockito like mocking library that allows you to write tests in BDD (given when then) or AAA (arrange act assert) fashion.

You can stub method calls:

```
$mock = Mock::create();
Mock::when($mock)->method(1)->thenReturn('result');

$result = $mock->method(1);

$this->assertEquals("result", $result);
```

And then verify interactions:

```
//given
$mock = Mock::create();

//when
$mock->method("arg");

//then
Mock::verify($mock)->method("arg");
```

Unlike other PHP mocking libraries you can verify interactions ex post facto which is more natural and fits BDD or AAA style.

Note: Mock::mock() is an alias for Mock::create(). You can use those methods interchangeably.

If you use type hinting and the mock has to be of a type of a Class, you can pass the required type to `Mock::create` method.

```
$mock = Mock::create('Foo');

$this->assertTrue($mock instanceof Foo);
```

You can stub a method to throw an exception;

```
Mock::when($mock)->method()->thenThrow(new Exception());
```

Verification that a method was not called:

```
Mock::verify($mock)->neverReceived()->method("arg");
```

Making sure that there were no interactions:

```
Mock::verifyZeroInteractions($mock);
```

You can stub multiple calls in one call to thenReturn:

```
$mock = Mock::create();
Mock::when($mock)->method(1)->thenReturn('result1', 'result2');
Mock::when($mock)->method()->thenThrow(new Exception('1'), new Exception('2'));
```

Both thenReturn and thenThrow accept multiples arguments that will be returned/thrown in subsequent calls to a stubbed method.

```
$mock = Mock::create();

Mock::when($mock)->method()->thenReturn('result1', 'result2');

$this->assertEquals("result1", $mock->method());
$this->assertEquals("result2", $mock->method());
```

You can stub a method to return value calculated by a callback function:

```
Mock::when($mock)->method(Mock::any())->thenAnswer(function (MethodCall $methodCall) {
  return $methodCall->name . ' ' . Arrays::first($methodCall->arguments);
});
```

Argument matchers

- Mock::any() - matches any value for an argument at the given position

```
Mock::verify($mock)->method(1, Mock::any(), "foo");
```

- Mock::anyArgList() - matches any possible arguments. It means that all calls to a given method will be matched.

```
Mock::verify($mock)->method(Mock::anyArgList());
```

- Mock::argThat() - returns an instance of FluentArgumentMatcher that can chain methods from *Functions*.

```
Mock::verify($mock)->method(Mock::argThat()->extractField('name')->equals('Bob'));
```

```
Mock::verify($mock)->method('first arg', Mock::argThat()->isInstanceOf('Foo'));
```

In rare cases, you may need to write your own argument matcher:

```
class MyArgumentMatcher implements Ouzo\Tests\Mock\ArgumentMatcher {
    public function matches($argument) {
        return $argument->name == 'Bob' || $argument->surname == 'Smith';
    }
}

Mock::verify($mock)->method(new MyArgumentMatcher());
```

Stream stubbing

In some cases you may need to to stub stream wrappers e.g. `php://input`. Ouzo provides a special class for this `StreamStub`.

- First, you have to register your wrapper:

```
StreamStub::register('you_wrapper_name');
```

- Then write something to the wrapper:

```
StreamStub::$body = 'some content';
```

- When you're finished using stream you must unregistered it:

```
StreamStub::unregister();
```

Comprehensive example:

```
StreamStub::register('some_name');
StreamStub::$body = '{"name":"jonh","id":123,"ip":"127.0.0.1"}';

$object = YourClass::readJsonFromWrapper('some_name://input');

$this->assertEquals('john', $object->name);
StreamStub::unregister();
```

Functional programming

Ouzo provides many utility classes that facilitate functional programming in php.

- *Arrays* contains facade for php arrays functions. You will never wonder if array_filter has array or closure as the first parameter.
- *Functions* contains static utility methods returning closures that can be used with Arrays and FluentArray.
- *FluentArray* provides an interface for manipulating arrays in a chained fashion.
- *FluentIterator* provides an interface for manipulating iterators in a chained fashion.
- *FluentFunctions* provides an interface for composing functions in a chained fashion.

Example 1

Let's assume that you have a User class that has a method isCool. You have an array of users and want to check if any of them is cool.

Pure php:

```
function isAnyCool($users) {
    foreach($users as $user) {
        if ($user->isCool()) {
            return true;
        }
    }
    return false;
}
```

Ouzo:

```
function isAnyCool($users) {
    return Arrays::any($users, function($user) {
        return $user->isCool();
    });
}
```

or using Functions::extract():

```
$anyCool = Arrays::any($users, Functions::extract()->isCool());
```

Similarly, you may want to check if all of them are cool:

```
$allCool = Arrays::all($users, Functions::extract()->isCool());
```

See also:

Arrays::groupBy()

Arrays::toMap()

Example 2

Let's assume that you have a User class that has a list of addresses. Each address has a type (like: home, invoice etc.) and User has `getAddress($type)` method.

Now, let's write a code that given a list of users, returns a lists of unique non-empty cities from users' home addresses.

Pure php:

```
$cities = array_unique(array_filter(array_map(function($user) {
   $address = $user>getAddress('home');
   return $address? $address->city : null;
}, $users)));
```

Ouzo:

```
$cities = FluentArray::from($users)
         ->map(Functions::extract()->getAddress('home')->city)
         ->filter(Functions::notEmpty())
         ->unique()
         ->toArray();
```

See also:

FluentArray

Functions::extract()

Example 3

If the array/iterator is very long and you are interested only in a small subset or processing is time consuming, you may want to use FluentIterator so that all operations are performed lazily (and only if necessary).

```
$activityReports = FluentIterator::from($users)
        ->filter(activeInLastMonth())
        ->map(createActivityReport())
        ->limit(10)
        ->toArray();
```

See also:

FluentIterator

Composing functions

Class `FluentFunctions` allows you to easily compose functions from `Functions`.

```
$usersWithSurnameStartingWithB =
      Arrays::filter($users, FluentFunctions::extractField('surname')->startsWith('B
↪'));
```

is equivalent of:

```
$usersWithSurnameStartingWithB = Arrays::filter($users, function($user) {
    $extractField = Functions::extractField('name');
    $startsWith = Functions::startsWith('B');
    return $startsWith($extractField($product));
});
```

Another example:

```
$bobs = Arrays::filter($users, FluentFunctions::extractField('name')->equals('Bob'));
```

See also:

FluentFunctions

Autoloading classes

Ouzo is compliant with PSR-4 specification. By default newly created project derived from ouzo-app will have PSR-4 structure as well. However, you can use any class loading method: PSR-4, PSR-0, classmap or whatever.

Note: There are three types of classes which Ouzo is expecting to be in specific locations:

- Controllers - under `\Application\Controller`
- Models - under `\Application\Model`
- Widgets - under `\Application\Widget`

Changing the defaults

If you wish to change the defaults, it can be done easily with configuration settings:

```
$config['namespace']['controller'] = '\\My\\New\\Controller';
$config['namespace']['model'] = '\\My\\New\\Model';
$config['namespace']['widget'] = '\\My\\New\\Widget';
```

Config

Ouzo has primary configurations locations in `config` directory. Inside this folder exists configuration for *prod* and *test* environments.

Custom config in Bootstrap

Ouzo has options to add custom config class, which can override default defined config values.

Note: **TODO:** Extract interface for custom configs.

```
$bootstrap = new Bootstrap();
$bootstrap->addConfig(new MyNewConfig());
$bootstrap->runApplication();
```

Config in session

Ouzo can handle config per user (using session mechanism). To override or set config value you must add value to `$_SESSION['config']` e.g.:

```
Session::set('config', 'db', 'host', '127.0.0.1');
```

Override rules

Thus config may be loaded form multiple locations there are rules for overriding:

Default config is override by the custom config.

Example:

```
$default['db']['host'] = 'localhost';
$default['db']['port'] = '5432';

$custom['db']['port'] = '1122';
```

Result (after override):

```
Array
(
    [db] => Array
        (
            [host] => localhost
            [port] => 1122
        )

)
```

Override default config by the custom can be also override by the session values.

Example:

```
$default['db']['host'] = 'localhost';
$default['db']['port'] = '5432';

$custom['db']['port'] = '1122';

Session::set('config', 'db', 'host', '127.0.0.1');
```

Result (after override):

```
Array
(
    [db] => Array
        (
            [host] => 127.0.0.1
            [port] => 1122
        )

)
```

Methods

getValue

Returns nested config value. If value does not exist it will return empty array.

Parameters: `string $keys..`

Example:

```
$host = Config::getValue('db', 'host'); //search $config['db']['host'] = 'localhost';
```

Result: `localhost`

---

getPrefixSystem

Returns defined prefix system.

Example:

```
//$config['global']['prefix_system'] = 'my_super_system';
$prefix = Config::getPrefixSystem();
```

Result: `my_super_system`

---

all

Returns all defined config parameters.

---

registerConfig

Note: **TODO:** Extract interface for custom configs.

---

overrideProperty

Override config property during runtime, may be useful in tests.

Parameters: `string $keys..`

Example:

```
//$config['key']['sub_key'] = 'value';
Config::overrideProperty('key', 'sub_key')->with('new value');
$value = Config::getValue('key', 'sub_key');
```

Result: `new value`

---

clearProperty

Clear override property to the default value.

Parameters: `string $keys..`

Example:

```
//$config['key']['sub_key'] = 'value';
Config::overrideProperty('key', 'sub_key')->with('new value');
Config::clearProperty('key', 'sub_key');
```

Result: `value`

---

revertProperty

Revert config last override value.

Parameters: `string $keys..`

Example:

```
//$config['key']['sub_key'] = 'value';
Config::overrideProperty('key1', 'sub_key')->with('first');
Config::overrideProperty('key1', 'sub_key')->with('second');
Config::revertProperty('key1', 'sub_key');
```

Result: `first`

FormHelper

View helper methods for generating form markup.

---

escapeText

Convert special characters to HTML entities

Parameters: `$text`

---

escapeNewLine

Changes new lines to `<br>` and converts special characters to HTML entities.

Parameters: `$text`

---

linkTo

Creates a link tag.

Parameters: `$name`, `$href`, `$attributes = []`

Example:

```
linkTo("Name", "url", ['class' => 'btn'])
```

Result:

```
<a href="url" class="btn">Name</a>
```

---

linkButton

Creates a button tag.

Parameters: `$params`

---

labelTag

Creates a label tag.

Parameters: `$name`, `$content`, `$attributes = []`

Example:

```
labelTag("name", "A Label", ['class' => 'pretty'])
```

Result:

```
<label for="name" class="pretty">A Label</label>
```

---

hiddenTag

Creates a hidden input tag.

Parameters: `$name`, `$value`, `$attributes = []`

Example:

```
hiddenTag("name", "value", ['id' => 'my-id'])
```

Result:

```
<input type="hidden" id="my-id" name="name" value="value">
```

---

textFieldTag

Creates a text input tag.

Parameters: `$name`, `$value`, `$attributes = []`

Example:

```
textFieldTag("name", "value", ['id' => 'my-id'])
```

Result:

```
<input type="text" id="my-id" name="name" value="value">
```

---

textAreaTag

Creates a textarea tag.

Parameters: `$name`, `$content`, `$attributes = []`

Example:

```
textAreaTag("name", "Content", ['id' => 'my-id'])
```

Result:

```
<textarea id="my-id" name="name">Content</textarea>
```

---

checkboxTag

Creates a checkbox input tag.

Parameters: `$name`, `$value`, `$checked`, `$attributes = []`

Example:

```
checkboxTag("name", "true", true,  ['class' => 'my-class'])
```

Result:

```
<input name="name" type="hidden" value="0">
<input type="checkbox" value="true" id="name" name="name" class="my-class" checked="">
```

---

selectTag

Creates a select tag.

Parameters: `$name`, `$items = []`, `$value`, `$attributes = []`, `$promptOption = null`

Example:

```
selectTag('status', ['bob' => 'Bob', 'fred' => 'Fred'], ['bob'], ['class' => "my-
↪select"])
```

Result:

```
<select id="status" name="status" class="my-select">
    <option value="bob" selected="">Bob</option>
    <option value="fred">Fred</option>
</select>
```

---

passwordFieldTag

Creates a password input tag.

Parameters: `$name`, `$value`, `$attributes = []`

Example:

```
passwordFieldTag("name", "value",  ['class' => 'my-class'])
```

Result:

```
<input type="password" value="value" id="name" name="name" class="my-class" />
```

---

radioButtonTag

Creates radio tag.

Parameters: `$name`, `$value`, `$attributes = []`

Example:

```
radioButtonTag('age', 33);
```

Result:

```
<input type="radio" id="age" name="age" value="33"/>
```

---

formTag

Creates a form tag.

Parameters: `$url`, `$method = 'POST'`, `$attributes = []`

Example:

```
formTag('url', 'post', ['class' => "my-select"])
```

Result:

```
<form class="my-select" action="url" method="POST">
```

---

endFormTag

Creates end form tag.

Example:

```
endFormTag()
```

Result:

```
</form>
```

---

formFor

Creates *ModelFormBuilder* for specific model object.

Parameters: `$model`

I18n

Locales

Locale files are placed under `locales` directory. File names are used as locale names, e.g.

- `en.php` - contains English words
- `pl.php` - contains Polish words

Locale files are simple arrays. Each translations has got its corresponding label in array.

```
return [
    'ouzo' => 'Ouzo',
    'framework' => 'Framework'
];
```

Translating based on label

Translations are found by label. So for the previous example, locales can be used such as: `echo I18n:t('ouzo');`

It will print `Ouzo`. When label is not found in array, label itself is returned.

Using `t` function in views

`ViewHelper` defines `t` function. It is a convenient alias for `I18n::t`:

```
<?= t('ouzo') ?>
```

Hierarchical labels

In order to create more complex structures multi-dimensional arrays are supported, e.g.:

```
return [
    'hello' => [
        'world' => 'Hi, world!'
    ]
];
```

Each level of array is combined by dot when using in `t` method:

```
echo I18n::t('hello.world');
```

Parametrization

Ouzo supports translation parameters. There can be as many parameters given as we need. Parameters are referenced in translations by `%{name}`. E.g.

```
return [
    'introduction' => 'My name is %{name}.'
];
```

Usage:

```
<?= t('introduction', ['name' => 'Jon Snow']) ?>
```

Pluralization

Whenever there is a need to distinguish between singular and plural forms, `pluralizeBasedOn` comes in handy.

First, we need to specify all forms (number of forms is determined by locale, e.g. 2 for English):

```
return [
    'dog' => '%{count} dog|%{count} dogs'
];
```

Usage:

```
<?= t('dog', ['count' => $count], pluralizeBasedOn($count)) ?>
```

It will print:

- `1 dog` for count = 1
- `2 dogs` for count = 2
- `3 dogs` for count = 3
- etc.

`pluralizeBasedOn` is a method in `I18n` class as well as function available in views (as an alias defined in `ViewHelper`).

Configuring language

`I18n` determines current language by configuration parameter named `language`. By default `en` is used.

Getting labels

All labels can be retrieved by:

```
$labels = I18n::labels();
```

If we want particular level of translations, we can specify it as a parameter:

```
$labels = I18n::labels('hello');
```

PhpStorm IDE support

Ouzo PhpStorm plugin is a must-have if you work with multi-language project. Check it out at: https://plugins.jetbrains.com/plugin/7565?pr= (it can be installed directly from PhpStorm's settings). It contains a number of handy functions and refactorings which makes it very easy to create and manage translations in your apps.

Session

Session is facade for session handling. Session data is stored in files. Path can be set in configuration if you want to change your system's default ($config['session']['path']).

---

startSession

To initialize session use:

```
Session::startSession();
```

Note: You don't need to call it if you use Ouzo Controllers - it is done automatically.

---

get

To get a variable from session use:

```
$value = Session::get('key');
```

---

all

To get all session variables use:

```
$array = Session::all();
```

---

set

To set a variable in session use:

```
Session::set('key', 'value');
```

Result is:

```
array(1) {
  'key' =>
  string(5) "value"
}
```

Note: Set methods can be chained:

```
Session::set('key', 'value')->set('another', 'value');
```

---

push

To add an element to an array stored in session use:

```
Session::push('key', 'value1');
Session::push('key', 'value2');
```

Result is:

```
array(1) {
  'key' =>
  array(2) {
    [0] =>
    string(6) "value1"
    [1] =>
    string(6) "value2"
  }
}
```

---

remove

To remove a variable from session use:

```
$value = Session:remove('key');
```

---

has

To check if a variable exists in session use:

```
$value = Session:has('key');
```

---

flush

To remove all variables from session just flush it:

```
Session:flush();
```

Nested keys

All session handling methods (except of all and flush) support nested keys e.g.

```
Session::get('key1', 'key2', 'value');
Session::set('key1', 'key2', 'value');
Session::push('key1', 'key2', 'value');
Session::remove('key1', 'key2');
Session::has('key1', 'key2');
```

You can specify as many keys as you want. Last argument in get, set and push is the value.

See also:

Session assertions

ModelFormBuilder

ModelFormBuilder simplifies implementation of forms for model objects.

```
<? $form = formFor($this->user); ?>
  <?= $form->start(userPath($this->user->id), 'post'); ?>

  <?= $form->label('login'); ?>
  <?= $form->textField('login'); ?>

  <?= $form->passwordField('password'); ?>

  <?= $form->checkboxField('cool'); ?>
  <?= $form->hiddenField('hidden_field'); ?>

  <?= $form->textArea('description'); ?>
```

```
  <?= $form->selectField('role', ['admin' => 'Admin', 'user' => 'User']); ?>

<?= $form->end(); ?>
```

In the first line

```
<? $form = formFor($this->user); ?>
```

we create a form for *user* object. All methods in ModelFormBuilder take field name as the first parameter and optionally array of options (class, id etc.)

Input values are taken from model object. Input names are inferred from model class name and field name. For instance, User's field *login* will have `name="user[login]"` and `id="user_login"`.

label

Creates a label tag.

Parameters: `$field`, `$options = []`

Example:

```
$form->label("name", ['class' => 'pretty'])
//=> <label for="name" class="pretty">A Label</label>
// assuming that there's a translation for modelName.name e.g. user.name => A Label
```

hiddenField

Creates a hidden input tag.

Parameters: `$field`, `$options = []`

Example:

```
$form->hiddenField("name", ['id' => 'my-id'])
//=> <input type="hidden" id="my-id" name="user[name]" value="">
```

textField

Creates a text input tag.

Parameters: `$name`, `$value`, `$attributes = []`

Example:

```
$form->textField('login')
//=> <input type="text" id="user_login" name="user[login]" value="thulium">
```

textArea

Creates a textarea tag.

Parameters: `$field`, `$options = []`

Example:

```
$form->textArea("name")
//=> <textarea id="user_name" name="user[name]"></textarea>
```

checkboxField

Creates a checkbox input tag.

Parameters: `$field, $options = []`

Example:

```
$form->checkboxField("cool",  ['class' => 'my-class'])
//=>
//<input type="checkbox" value="1" id="user_cool" name="user[cool]" class="my-class">
//<input name="user[cool]" type="hidden" value="0">
```

selectField

Creates a select tag.

Parameters: `$field, $items = [], $options = [], $promptOption = null`

Example:

```
$form->selectField('person', ['bob' => 'Bob', 'fred' => 'Fred'], ['class' => "my-
↪select"], 'select person')
//=>
//<select id="user_person" name="user[person]" class="my-select">
//  <option value="" selected="">select person</option>
//  <option value="bob">Bob</option><option value="fred">Fred</option>
//</select>
```

passwordField

Creates a password input tag.

Parameters: `$field, $options = []`

Example:

```
$form->passwordField("name",  ['class' => 'my-class'])
//=>
//<input type="password" id="user_password" name="user[password]" value="value">
```

Arrays

Helper functions that can operate on arrays.

---

contains

Returns true if array contains given element. Comparison is based on *Objects::equal()*

Parameters: `array $array`, `mixed $element`

Example:

```
$result = Arrays::contains([1, 2, 3], 2);
```

Result: `true`

containsAll

Returns true if array contains all given elements. Comparison is based on *Objects::equal()*

Parameters: `array $array`, `array $elements`

Example:

```
$result = Arrays::containsAll([1, 2, 3], [1, 2]);
```

Result: `true`

---

keyExists

Checks is key exists in an array.

Parameters: `array $elements`, `$key`

Example:

```
$array = ['id' => 1, 'name' => 'john'];
$return = Arrays::keyExists($array, 'name');
```

Result: `true`

---

concat

Merges array of arrays into one array. Unlike flatten, concat does not merge arrays that are nested more that once.

Parameters: `array $arrays`

Example:

```
$result = Arrays::concat([[1, 2], [3, 4]]);
```

Result:

```
Array
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 4
)
```

---

getValue

Returns the element for the given key or a default value otherwise.

Parameters: `array $elements, $key, $default = null`

Example:

```
$array = ['id' => 1, 'name' => 'john'];
$value = Arrays::getValue($array, 'name');
```

Result: `john`

Example:

```
$array = ['id' => 1, 'name' => 'john'];
$value = Arrays::getValue($array, 'surname', '--not found--');
```

Result: `--not found--`

---

sort

Returns a new array sorted using given comparator. The comparator function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. To obtain comparator one may use `Comparator` class (for instance `Comparator::natural()` which yields ordering using comparison operators).

Parameters: `array $array, $comparator`

Example:

```
class Foo
{
    private $value;

    public function __construct($value)
    {
      $this->value = $value;
    }

    public function getValue()
    {
      return $this->value;
    }
}
```

```
$values = [new Foo(1), new Foo(3), new Foo(2)];
$sorted = Arrays::sort($values, Comparator::compareBy('getValue()'));
```

Result:

```
Array
(
    [0] => class Foo (1) {
               private $value => int(1)
           }
    [1] => class Foo (1) {
               private $value => int(2)
           }
    [2] => class Foo (1) {
               private $value => int(3)
           }
)
```

---

first

This method returns the first value in the given array .

Parameters: `array $elements`

Example:

```
$array = ['one', 'two' 'three'];
$first = Arrays::first($array);
```

Result: `one`

---

firstOrNull

This method returns the first value or `null` if array is empty.

Parameters: `array $elements`

Example:

```
$array = [];
$return = Arrays::firstOrNull($array);
```

Result: `null`

---

last

This method returns the last value in the given array.

Parameters: `array $elements`

Example:

```
$array = ['a', 'b', 'c'];
$last = Arrays::last($array);
```

Result: `c`

---

any

Returns true if at least one element in the array satisfies the predicate.

Parameters: `array $elements, $predicate`

Example:

```
$array = ['a', true, 'c'];
$any = Arrays::any($array, function ($element) {
    return is_bool($element);
});
```

Result: `true`

---

all

Returns true if every element in array satisfies the predicate.

Parameters: `array $elements, $predicate`

Example:

```
$array = [1, 2];
$all = Arrays::all($array, function ($element) {
    return $element < 3;
});
```

Result: `true`

---

find

Finds first element in array that is matched by function. Returns null if element was not found.

Parameters: `array $elements, callable $function`

---

count

Returns the number of elements for which the predicate returns true.

Parameters: `array $elements, $predicate`

Example:

```
$array = [1, 2, 3];
$count = Arrays::count($array, function ($element) {
   return $element < 3;
});
```

Result: `2`

---

filter

This method filters array using function. Result contains all elements for which function returns `true`.

Parameters: `$elements`, `$function`

Example:

```
$array = [1, 2, 3, 4];
$result = Arrays::filter($array, function ($value) {
    return $value > 2;
});
```

Result:

```
Array
(
    [2] => 3
    [3] => 4
)
```

---

filterByKeys

Filters array by keys using the predicate.

Example:

```
$array = ['a1' => 1, 'a2' => 2, 'c' => 3];
$filtered = Arrays::filterByKeys($array, function ($elem) {
    return $elem[0] == 'a';
});
```

Result:

```
Array
(
    [a1] => 1
    [a2] => 2
)
```

---

filterByAllowedKeys

Returns an array containing only the given keys.

Example:

```
$array = ['a' => 1, 'b' => 2, 'c' => 3];
$filtered = Arrays::filterByAllowedKeys($array, ['a', 'b']);
```

Result:

```
Array
(
    [a] => 1
    [b] => 2
)
```

---

filterNotBlank

Returns a new array without blank elements.

Parameters: `array $elements`

---

map

This method maps array values using the function. It invokes the function for each value in the array and creates a new array containing the values returned by the function.

Parameters: `array $elements`, `$function`

Example:

```
$array = ['k1', 'k2', 'k3'];
$result = Arrays::map($array, function ($value) {
    return 'new_' . $value;
});
```

Result:

```
Array
(
    [0] => new_k1
    [1] => new_k2
    [2] => new_k3
)
```

---

mapKeys

This method maps array keys using the function. It invokes the function for each key in the array and creates a new array containing the keys returned by the function.

Parameters: `array $elements, $function`

Example:

```
$array = [
     'k1' => 'v1',
     'k2' => 'v2',
     'k3' => 'v3'
];
$arrayWithNewKeys = Arrays::mapKeys($array, function ($key) {
     return 'new_' . $key;
});
```

Result:

```
Array
(
     [new_k1] => v1
     [new_k2] => v2
     [new_k3] => v3
)
```

---

mapEntries

This method maps array values using the function which takes key and value as parameters. Invokes the function for each value in the array. Creates a new array containing the values returned by the function.

Parameters: `array $elements, $function`

Example:

```
$array = ['a' => '1', 'b' => '2', 'c' => '3'];
$result = Arrays::mapEntries($array, function ($key, $value) {
    return $key . '_' . $value;
});
```

Result:

```
Array
(
    [a] => a_1
    [b] => b_2
    [c] => c_3
)
```

---

toMap

This method creates associative array using key and value functions on array elements.

Parameters: `array $elements`, `$keyFunction`, `$valueFunction = null`

Example:

```
$array = range(1, 2);
$map = Arrays::toMap($array, function ($elem) {
    return $elem * 10;
}, function ($elem) {
    return $elem + 1;
});
```

Result:

```
Array
(
    [10] => 2
    [20] => 3
)
```

Note: If `$valueFunction` is not given Functions::identity() is used.

```
$users = [new User('bob'), new User('john')];
$usersByName = Arrays::toMap($users, function ($user) {
    return $user->name;
});
```

`$usersByName` will contain associative array with users indexed by their names.

Note: You can Functions::extractField provided by ouzo:

```
$usersByName = Arrays::toMap($users, Functions::extractField('name'));
```

---

combine

Returns a new array with `$keys` as array keys and `$values` as array values.

Parameters: `array $keys`, `array $values`

Example:

```
$keys = ['id', 'name', 'surname'];
$values = [1, 'john', 'smith'];
$combined = Arrays::combine($keys, $values);
```

Result:

```
Array
(
    [id] => 1
```

```
    [name] => john
    [surname] => smith
)
```

orderBy

This method sorts elements in array using order field.

Parameters: `array $elements, $orderField`

Example:

```
$obj1 = new stdClass();
$obj1->name = 'a';
$obj1->description = '1';

$obj2 = new stdClass();
$obj2->name = 'c';
$obj2->description = '2';

$obj3 = new stdClass();
$obj3->name = 'b';
$obj3->description = '3';

$array = [$obj1, $obj2, $obj3];
$sorted = Arrays::orderBy($array, 'name');
```

Result:

```
Array
(
    [0] => stdClass Object
        (
            [name] => a
            [description] => 1
        )

    [1] => stdClass Object
        (
            [name] => b
            [description] => 3
        )

    [2] => stdClass Object
        (
            [name] => c
            [description] => 2
        )

)
```

uniqueBy

Removes duplicate values from an array. It uses the given expression to extract value that is compared.

Parameters: `array $elements, $selector`

Example:

```
$a = new stdClass();
$a->name = 'bob';

$b = new stdClass();
$b->name = 'bob';

$array = [$a, $b];
$result = Arrays::uniqueBy($array, 'name');
```

Result:

```
Array
(
    [0] => $b
)
```

---

groupBy

Groups elements in array using given function. If `$orderField` is set, grouped elements will be also sorted.

Parameters: `array $elements, $keyFunction, $orderField = null`

Example:

```
$obj1 = new stdClass();
$obj1->name = 'a';
$obj1->description = '1';

$obj2 = new stdClass();
$obj2->name = 'b';
$obj2->description = '2';

$obj3 = new stdClass();
$obj3->name = 'b';
$obj3->description = '3';

$array = [$obj1, $obj2, $obj3];
$grouped = Arrays::groupBy($array, Functions::extractField('name'));
```

Result:

```
Array
(
    [a] => Array
        (
            [0] => stdClass Object
                (
                    [name] => a
```

```
                    [description] => 1
                )

        )

    [b] => Array
        (
            [0] => stdClass Object
                (
                    [name] => b
                    [description] => 2
                )

            [1] => stdClass Object
                (
                    [name] => b
                    [description] => 3
                )

        )

)
```

reduce

Method to reduce an array elements to a string value.

Parameters: `array $elements`, `callable $function`

setNestedValue

Sets nested value.

Parameters: `array $array`, `array $keys`, `$value`

Example:

```
$array = [];
Arrays::setNestedValue($array, ['1', '2', '3'], 'value');
```

Result:

```
Array
(
     [1] => Array
         (
             [2] => Array
                 (
                     [3] => value
                 )
         )
)
```

getNestedValue

Returns nested value when found, otherwise returns null.

Parameters: `array $array, array $keys`

Example:

```
$array = ['1' => ['2' => ['3' => 'value']]];
$value = Arrays::getNestedValue($array, ['1', '2', '3']);
```

Result: `value`

---

hasNestedKey

Checks if array has a nested key.

Parameters: `array $array, array $keys, $flags = null`

Example:

```
$array = ['1' => ['2' => ['3' => 'value']]];
$value = Arrays::hasNestedKey($array, ['1', '2', '3']);
```

Result: `true`

Example with null values:

```
$array = ['1' => ['2' => ['3' => null]]];
$value = Arrays::hasNestedKey($array, ['1', '2', '3'], Arrays::TREAT_NULL_AS_VALUE);
```

Result: `true`

Note: It's possible to check array with null values using flag `Arrays::TREAT_NULL_AS_VALUE`.

---

removeNestedKey

Returns array with removed keys even are nested.

Parameters: `array $array, array $keys`

Example:

```
$array = ['1' => ['2' => ['3' => 'value']]];
Arrays::removeNestedKey($array, ['1', '2']);
```

Result:

```
Array
(
    [1] => Array
        (
        )
)
```

Note: It's possible to remove keys when they don't have any children using flag `Arrays::REMOVE_EMPTY_PARENTS`.

Example:

```
$array = ['1' => ['2' => ['3' => 'value']]];
Arrays::removeNestedKey($array, ['1', '2'], Arrays::REMOVE_EMPTY_PARENTS);
```

Result:

```
Array
(
)
```

removeNestedValue

Deprecated since version 1.0.

Use *Arrays::removeNestedKey()* instead.

findKeyByValue

This method returns a key for the given value.

Parameters: `array $elements`, `$value`

Example:

```
$array = [
    'k1' => 4,
    'k2' => 'd',
    'k3' => 0,
    9 => 'p'
];
$key = Arrays::findKeyByValue($array, 0);
```

Result: `k3`

flatten

Returns a new array that is a one-dimensional flattening of the given array.

Parameters: `array $elements`

Example:

```
$array = [
    'names' => [
        'john',
        'peter',
        'bill'
    ],
    'products' => [
        'cheese',
        ['milk', 'brie']
    ]
];
$flatten = Arrays::flatten($array);
```

Result:

```
Array
(
    [0] => john
    [1] => peter
    [2] => bill
    [3] => cheese
    [4] => milk
    [5] => brie
)
```

---

flattenKeysRecursively

Returns a flattened array of keys with corresponding values.

Parameters: `array $array`

Example:

```
$array = [
     'customer' => [
         'name' => 'Name',
         'phone' => '123456789'
     ],
     'other' => [
         'ids_map' => [
             '1qaz' => 'qaz',
             '2wsx' => 'wsx'
         ],
         'first' => [
             'second' => [
                 'third' => 'some value'
             ]
         ]
     ]
];
$flatten = Arrays::flattenKeysRecursively($array)
```

Result:

```
Array
(
    [customer.name] => Name
    [customer.phone] => 123456789
    [other.ids_map.1qaz] => qaz
    [other.ids_map.2wsx] => wsx
    [other.first.second.third] => some value
)
```

---

intersect

Computes the intersection of arrays.

Parameters: `array $array1, array $array2`

---

recursiveDiff

Returns a recursive diff of two arrays

Parameters: `array $array1, array $array2`

Example:

```
$array1 = ['a' => ['b' => 'c', 'd' => 'e'], 'f'];
$array2 = ['a' => ['b' => 'c']];
$result = Arrays::recursiveDiff($array1, $array2);
```

Result:

```
Array
(
    [a] => Array
        (
            [d] => e
        )
    [0] => f
)
```

---

toArray

Makes an array from element. Returns the given argument if it's already an array.

Parameters: `$element`

Example:

```
$result = Arrays::toArray('test');
```

Result:

```
Array
(
    [0] => test
)
```

---

isAssociative

Checks if the given array is associative. An array is considered associative when it has at least one string key. **Parameters:** `array $array`

Example:

```
$result = Arrays::isAssociative([1 => 'b', 'a' => 2, 'abc'])
```

Result: `true`

---

shuffle

Returns shuffled array with retained key association.

Parameters: `array $array`

Example:

```
$result = Arrays::shuffle([1 => 'a', 2 => 'b', 3 => 'c']);
```

Result:

```
Array
(
    [3] => c
    [1] => a
    [2] => b
)
```

---

randElement

Returns a random element from the given array.

Parameters: `array $elements`

Example:

```
$array = ['john', 'city', 'small'[;
$rand = Arrays::randElement($array);
```

Result: *rand element from array*

Strings

underscoreToCamelCase

Changes underscored string to the camel case.

Parameters: `$string`

Example:

```
$string = 'lannisters_always_pay_their_debts';
$camelcase = Strings::underscoreToCamelCase($string);
```

Result: `LannistersAlwaysPayTheirDebts`

---

camelCaseToUnderscore

Changes camel case string to underscored.

Parameters: `$string`

Example:

```
$string = 'LannistersAlwaysPayTheirDebts';
$underscored = Strings::camelCaseToUnderscore($string);
```

Result: `lannisters_always_pay_their_debts`

---

removePrefix

Returns a new string without the given prefix.

Parameters: `$string`, `$prefix`

Example:

```
$string = 'prefixRest';
$withoutPrefix = Strings::removePrefix($string, 'prefix');
```

Result: `Rest`

---

removePrefixes

Removes prefixes defined in array from string.

Parameters: `$string`, `array $prefixes`

Example:

```
$string = 'prefixRest';
$withoutPrefix = Strings::removePrefixes($string, ['pre', 'fix']);
```

Result: `Rest`

---

removeSuffix

Returns a new string without the given suffix.

Parameters: `$string`, `$suffix`

Example:

```
$string = 'JonSnow';
$withoutSuffix = Strings::removeSuffix($string, 'Snow');
```

Result: `Jon`

---

startsWith

Checks if string starts with `$prefix`.

Parameters: `$string`, `$prefix`

Example:

```
$string = 'prefixRest';
$result = Strings::startsWith($string, 'prefix');
```

Result: `true`

---

endsWith

Checks if string ends with `$suffix`.

Parameters: `$string`, `$suffix`

Example:

```
$string = 'StringSuffix';
$result = Strings::endsWith($string, 'Suffix');
```

Result: `String`

---

equalsIgnoreCase

Determines whether two strings contain the same data, ignoring the case of the letters in the strings.

Parameters: `$string1`, `$string2`

Example:

```
$equal = Strings::equalsIgnoreCase('ABC123', 'abc123')
```

Result: `true`

---

remove

Removes all occurrences of a substring from string.

Parameters: `$string`, `$stringToRemove`

Example:

```
$string = 'winter is coming???!!!';
$result = Strings::remove($string, '???');
```

Result: `winter is coming!!!`

---

appendSuffix

Adds suffix to the string.

Parameters: `$string`, `$suffix = ''`

Example:

```
$string = 'Daenerys';
$stringWithSuffix = Strings::appendSuffix($string, ' Targaryen');
```

Result: `Daenerys Targaryen`

appendIfMissing

Adds suffix to the string, if string does not end with the suffix already.

Parameters: `$string`, `$suffix = ''`

Example:

```
$string = 'Daenerys  Targaryen';
$unmodified = Strings::appendIfMissing($string, ' Targaryen');
```

Result: `Daenerys Targaryen`

appendPrefix

Adds prefix to the string.

Parameters: `$string, $prefix = ''`

Example:

```
$string = 'Targaryen';
$stringWithPrefix = Strings::appendPrefix($string, 'Daenerys ');
```

Result: `Daenerys Targaryen`

prependIfMissing

Adds prefix to the string, if string does not start with the prefix already.

Parameters: `$string, $prefix = ''`

Example:

```
$string = 'Queen Daenerys Targaryen';
$unmodified = Strings::prependIfMissing($string, 'Queen ');
```

Result: `Queen Daenerys Targaryen`

---

tableize

Converts a word into the format for an Ouzo table name. Converts `'ModelName'` to `'model_names'`.

Parameters: `$class`

Example:

```
$class = "BigFoot";
$table = Strings::tableize($class);
```

Result: `BigFeet`

---

escapeNewLines

Changes new lines to `<br>` and converts special characters to HTML entities.

Parameters: `$string`

Example:

```
$string = "My name is <strong>Reek</strong> \nit rhymes with leek";
$escaped = Strings::escapeNewLines($string);
```

Result: `My name is &lt;strong&gt;Reek&lt;/strong&gt; <br />\nit rhymes with leek`

---

htmlEntityDecode

Alias for `html_entity_decode()` with UTF-8 and defined flag `ENT_COMPAT`.

Parameters: `$text`

---

htmlEntities

Alias for `htmlentities()` with UTF-8 and flags `ENT_COMPAT` and `ENT_SUBSTITUTE` (`ENT_IGNORE` for php <= 5.3).

Parameters: `$text`

---

equal

Checks if string representations of two objects are equal.

Parameters: `$object1`, `$object2`

Example:

```
$result = Strings::equal('0123', 123);
```

Result: `false`

---

isBlank

Checks if string is blank.

Parameters: `$string`

Example:

```
Strings::isBlank('word'); // false
Strings::isBlank('0');    // false

Strings::isBlank("\n");    // true
Strings::isBlank('   ');   // true
Strings::isBlank(PHP_EOL); // true
```

---

isNotBlank

Checks if string is not blank. This method has a reverse effect of *Strings::isBlank()*.

Parameters: `$string`

Example:

```
$result = Strings::isNotBlank('0');
```

Result: `true`

---

abbreviate

Abbreviates a string using ellipsis.

Parameters: `$string`, `$maxWidth`

Example:

```
$result = Strings::abbreviate('ouzo is great', 5);
```

Result: `ouzo ...`

---

trimToNull

Removes control characters from both ends of this string returning null if the string is empty (“”) after the trim or if it is null.

Parameters: `$string`

Example:

```
$result = Strings::trimToNull('  ');
```

Result: `null`

---

sprintfAssoc

Replaces all occurrences of placeholder in string with values from associative array.

Parameters: `$string`, `$params`

Example:

```
$sprintfString = "This is %{what}! %{what}? This is %{place}!";
$assocArray = [
  'what' => 'madness',
  'place' => 'Sparta'
];
```

Result: `This is madness! madness? This is Sparta!`

---

sprintAssocDefault

Replaces all occurrences of placeholder in string with values from associative array. When no value for placeholder is found in array, a default empty value is used if not otherwise specified.

Parameters: `$string`, `array $params`, `$default = ''`

Example:

```
$sprintfString = "This is %{what}! %{what}? This is %{place}!";
$assocArray = [
  'what' => 'madness',
  'place' => 'Sparta'
];
```

Result: `This is madness! madness? This is Sparta!`

---

contains

Checks if string contains substring.

Parameters: `$string`, `$substring`

---

substringBefore

Gets the substring before the first occurrence of a separator. The separator is not returned.

If the separator is not found, the string input is returned.

Parameters: `$string`, `$separator`

Example:

```
$string = 'winter is coming???!!!';
$result = Strings::substringBefore($string, '?');
```

Result: `winter is coming`

---

substringAfter

Gets the substring after the first occurrence of a separator. The separator is not returned.

If the separator is not found, the string input is returned.

Parameters: `$string`, `$separator`

Example:

```
$string = 'abc+efg+hij';
$result = Strings::substringAfter($string, '+');
```

Result: `efg+hij`

Objects

Helper functions that can operate on any php Object.

toString

Returns a string representation of the given object. If the given object implements `__toString` method it will be used.

Parameters: `$var`

Example:

```
Objects::toString('string'); //=> "string"
Objects::toString(null); //=> null
Objects::toString(1); //=> 1
Objects::toString(true); //=> true

Objects::toString(['a', 1]); //=> ["a", 1]

Objects::toString(['key' => 'value1', 'key2' => 'value2']);
//=> [<key> => "value1", <key2> => "value2"]

$object = new stdClass();
$object->field1 = 'field1';
$object->field2 = 'field2';

Objects::toString($object);
//=> stdClass {<field1> => "field1", <field2> => "field2"}
```

getValue

Returns value of a field or default if the field does not exist or is null.

Parameters: `$object`, `$field`, `$default = null`

Example:

```
$object = new stdClass();
$object->field1 = 'value';

$result = Objects::getValue($object, 'field1');
```

Returns: `'value'`

```
$object = new stdClass();

$result = Objects::getValue($object, 'field1', 'not found');
```

Returns: `'not found'`

setValueRecursively

Sets value of a nested field.

Parameters: `$object`, `$names`, `$value`

Example:

```
$object = new stdClass();
$object->field1 = new stdClass();
Objects::setValueRecursively($object, 'field1->field2', 'value')

echo $object->field1->field2
```

will echo `'value'`.

getValueRecursively

Returns value of a nested field or default if the field does not exist.

The `$names` parameter can also contain method calls e.g.: `'field->method()->field'`

Parameters: `$object`, `$names`, `$default = null`

Example:

```
$object = new stdClass();
$object->field1 = new stdClass();
$object->field1->field2 = 'value';

$result = Objects::getValueRecursively($object, 'field1->field2');
```

Result: `'value'`

Example2:

```
$object = new stdClass();
$object->field1 = new stdClass();

$result = Objects::getValueRecursively($object, 'field1->field2->field3', 'not found
↪');
```

Result: `'not found'`

equal

Returns true if $a is equal to $b. Comparison is based on the following rules:

- same type + same type = strict check
- object + object = loose check
- array + array = compares arrays recursively with these rules
- string + integer = loose check (`'1' == 1`)
- boolean + string (`'true'` or `'false'`) = loose check
- `false` in other cases (`'' != null`, `'' != 0`, `'' != false`)

Parameters: `mixed $a`, `mixed $b`

Example:

```
$result = Objects::equal(['1'], ['1']));
```

Result: `true`

Functions

Static utility methods returning closures.

---

extractId

Returns a function object that calls `getId` method on its argument.

Example: `$ids = Arrays::map($models, Functions::extractId());`

---

extractField

Returns a function object that returns a value of the given field of its argument.

Parameters: `field`

Example:

```
$users = [User::new(['name' => 'bob']), User::new(['name' => 'john'])];
$names = Arrays::map($users, Functions::extractField('name'));
```

---

extractFieldRecursively

Returns a function object that returns a value of the given nested field of its argument.

Parameters: `$fields`

Example:

```
$object = new stdClass();
$object->field1 = new stdClass();
$object->field1->field2 = 'value';

$fun = Functions::extractFieldRecursively('field1->field2');
$result = $fun($object);
```

Result: `value`

Example: `$groupNames = Arrays::map($users, Functions::extractFieldRecursively('group->name')`

Note: It can also call functions: `$groupFullNames = Arrays::map($users, Functions::extractFieldRecursively('group->getFullName()'));`

---

extractExpression

Returns a function object that returns a result of the expression evaluated for its argument. It's a more efficient equivalent of *Functions::extractField* and *Functions::extractFieldRecursively* (it examines the given expression and returns the most suitable function).

If `$expression` is a function object, it is returned unchanged.

Parameters: `$expression`

---

identity

Returns a function object that always returns the argument.

Example:

```
$fun = Functions::identity()
$result = $fun('bob');
```

Result: `bob`

---

constant

Creates a function that returns value for any input.

Example:

```
$fun = Functions::constant('john')
$result = $fun('bob');
```

Result: `john`

---

throwException

Creates a function that throws $exception for any input.

Example:

```
$fun = Functions::throwException(new Exception('error'))
$result = $fun('bob');
```

Throws: `Exception('error')`

---

trim

Returns a function object that trims its arguments.

---

not

Returns a function object that negates result of supplied predicate.

Parameters: `$predicate`

Example: `$isNotArrayFunction = Functions::not(Functions::isArray());`

---

isArray

Returns a function object (predicate) that returns true if its argument is an array.

---

isInstanceOf

Returns a function object (predicate) that returns true if its argument is an instance of the given type.

Parameters: `$type`

---

prepend

Returns a function object that prepends the given prefix to its arguments.

Parameters: `$prefix`

---

append

Returns a function object that appends the given suffix to its arguments.

Parameters: `$suffix`

---

notEmpty

Returns a function object (predicate) that returns true if its argument is not empty.

---

notBlank

Returns a function object (predicate) that returns true if its argument is not blank.

---

notNull

Returns a function object (predicate) that returns true if its argument is not null.

---

removePrefix

Returns a function object that removes the given prefix from its arguments.

Parameters: `$prefix`

---

startsWith

Returns a function object (predicate) that returns true if its argument starts with the given prefix.

Parameters: `$prefix`

---

formatDateTime

Returns a function object that format date time its arguments.

Parameters: `$format = Date::DEFAULT_TIME_FORMAT`

---

compose

Returns the composition of two functions. Composition is defined as the function h such that h(a) == A(B(a)) for each a.

Parameters: `$functionA`, `$functionB`

---

toString

Returns a function object that calls `Objects::toString` on its argument.

---

extract

Fluent builder for a callable that extracts a value from its argument.

The callable object returned by this method records all actions performed on it. Then when it is invoked, it replays those actions on the invocation argument.

Parameters: `$type` - optional type hint for PhpStorm dynamicReturnType plugin.

Example:

Let's assume that you have a User class that has a list of addresses. Each address has a type (like: home, invoice etc.) and User has getAddress($type) method.

Now, let's write a code that given a list of users, returns a lists of cities from users' home addresses.

```
$cities = Arrays::map($users, function($user) {
  return $user>getAddress('home')->city;
});
```

It gets more complicated when some users don't have home address:

```
$cities = Arrays::map($users, function($user) {
  $address = $user>getAddress('home');
  return $address? $address->city : null;
});
```

We can write it in one line using Functions::extract:

```
$cities = Arrays::map($users, Functions::extract()->getAddress('home')->city);
```

Additionally, if you use PhpStorm dynamicReturnType plugin you can pass type as the first argument of Functions::extract.

```
Arrays::map($users, Functions::extract('User')->getAddress('home')->city);
```

```
$cities = Arrays::map($users, Functions::extract('User')->...
//ctrl+space will show you all methods/properties of the User class
```

Extractor can also extract array values:

```
Arrays::map($users, Functions::extract('User')->addresses['home']);
```

---

surroundWith

Returns a function object that surround with given character its arguments.

Parameters: `$character`

---

equals

Comparison is based on *Objects::equal()*

Parameters: `$object`

---

notEquals

Comparison is based on *Objects::equal()*

Parameters: `$object`

---

random

Returns a function that generates random numbers. Optional parameters `$min` and `$max` specify range (inclusive).

Parameters: `$min`, `$max`

Example:

```
Functions::random();
Functions::random(10, 20);
```

FluentArray

`FluentArray` provides an interface for manipulating arrays in a chained fashion. It's inspired by FluentIterable from guava library.

Example:

```
$result = FluentArray::from($users)
            ->map(Functions::extractField('name'))
            ->filter(Functions::notEmpty())
            ->unique()
            ->toArray();
```

Example above returns an array of non empty unique names of users.

---

from

Returns a FluentArray that wraps the given array.

Parameters: `array $array`

---

map

Returns a FluentArray that applies function to each element of this FluentArray.

Parameters: `$function`

---

mapKeys

Returns a FluentArray that applies $function to each key of this FluentArray.

Parameters: `$function`

---

filter

Returns a FluentArray that contains only elements that satisfy a predicate.

Parameters: `$function`

---

filterNotBlank

Return a FluentArray that applies function Arrays::filterNotBlank on each of element.

---

filterByKeys

Returns a FluentArray that contains only elements with keys that satisfy a predicate.

Parameters: `$function`

---

filterByAllowedKeys

Returns a FluentArray containing only the given keys.

Parameters: `$allowedKeys`

---

unique

Returns a FluentArray that contains unique elements.

---

uniqueBy

Removes duplicate values from an array. It uses the given expression to extract value that is compared.

Parameters: `$selector`

Example:

```
$a = new stdClass();
$a->name = 'bob';

$b = new stdClass();
$b->name = 'bob';

$array = [$a, $b];
$result = FluentArray::from($array)->uniqueBy('name')->toArray();
```

Result:

```
Array
(
    [0] => $b
)
```

---

keys

Returns a FluentArray that contains array of keys of the original FluentArray.

---

values

Returns a FluentArray that contains array of values of the original FluentArray.

---

flatten

Returns a FluentArray that contains flattened array of the original FluentArray.

---

intersect

Returns a FluentArray that contains only elements of the original FluentArray that occur in the given $array.

Parameters: `array $array`

---

reverse

Returns a FluentArray that contains elements of the original FluentArray in reversed order.

---

toMap

This method creates associative array using key and value functions on array elements. If `$valueFunction` is not given the result will contain original elements as values.

Parameters: `$keyFunction, $valueFunction = null`

Example:

```
$array = range(1, 2);
$map = FluentArray::from($array)->toMap(function ($elem) {
           return $elem * 10;
       }, function ($elem) {
            return $elem + 1;
       });
```

Result:

```
Array
(
    [10] => 2
    [20] => 3
)
```

---

toArray

Returns elements of this FluentArray as php array.

---

firstOr

Returns the first element of this FluentArray or `$default` if FluentArray is empty.

Parameters: `$default`

---

toJson

Encodes FluentArray elements to json.

---

limit

Returns a FluentArray with the first `$number` elements of this FluentArray.

Parameters: `$number`

Example:

```
$array = [1, 2, 3];
$result = FluentArray::from($array)->limit(2)->toArray();
```

Result:

```
Array
(
    [0] => 1,
    [1] => 2,
)
```

---

skip

Returns a FluentArray that skips its first `$number` elements.

Parameters: `$number`

Example:

```
$array = [1, 2, 3];
$result = FluentArray::from($array)->skip(2)->toArray();
```

Result:

```
Array
(
    [0] => 3
)
```

sort

Returns a FluentArray with its elements sorted using the given comparator

Parameters: `$comparator`

Example:

```
$array = [3, 1, 2];
$result = FluentArray::from($array)->sort(Comparator::natural())->toArray();
```

Result:

```
Array
(
    [0] => 1,
    [1] => 2,
    [2] => 3
)
```

flip

Returns a FluentArray with its elements flipped (keys replaced with values).

Example:

```
$array = ['a', 'b', 'c'];
$result = FluentArray::from($array)->flip()->toArray();
```

Result:

```
Array
(
    ['a'] => 0,
    ['b'] => 1,
    ['c'] => 2
)
```

FluentIterator

Interface for manipulating iterators in a chained fashion. It's inspired by FluentIterable from guava library.

```
$result = FluentIterator::fromArray([1, 2, 3])
          ->cycle()
          ->limit(10)
          ->reindex()
          ->toArray(); // [1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

```
$result = FluentIterator::fromFunction(Functions::random(0, 10))
          ->limit(10)
          ->toArray(); // returns array of 10 random numbers between 0 and 10␣
↪(inclusive)
```

All methods are applied lazily during iteration or call to `toArray`.

FluentIterator works great with php 5.5 generators:

```
function fibonacci() {
    $i = 0;
    $k = 1;
    while(true) {
        yield $k;
        $k = $i + $k;
        $i = $k - $i;
    }
}
```

Get 10th Fibonacci number:

```
$number = FluentIterator::from(fibonacci())->skip(9)->first();
```

Display first ten fibonacci numbers that are greater than 100:

```
$iterator = FluentIterator::from(fibonacci())
    ->filter(function($number) {
        return $number > 100;
    })
    ->limit(10);

foreach($iterator as $number) {
```

```
    echo $number, ", ";
}
```

---

from

Returns a fluent iterator that wraps `$iterator`

Parameters: `Iterator $iterator`

---

fromArray

Returns a fluent iterator for `$array`.

Parameters: `array $array`

---

fromFunction

Returns a fluent iterator that uses $function to generate elements. `$function` takes one argument which is the current position of the iterator.

Parameters: `callable $function`

---

cycle

Returns a fluent iterator that cycles indefinitely over the elements of this fluent iterator.

---

batch

Returns a fluent iterator returning elements of this fluent iterator grouped in chunks of `$chunkSize`

Parameters: `$chunkSize`

---

filter

Returns a fluent iterator returning elements of this fluent iterator that satisfy a predicate.

Parameters: `callable $predicate`

---

map

Returns a fluent iterator that applies function to each element of this fluent iterator.

Parameters: `callable $function`

---

limit

Returns a fluent iterator returning the first `$number` elements of of this fluent iterator.

Parameters: `$number`

---

skip

Returns a fluent iterator returning all but first `$number` elements of this fluent iterator.

Parameters: `$number`

---

reindex

Returns an iterator that indexes elements numerically starting from 0

---

firstOr

Returns the first element or defaultValue if the iterator is empty.

Parameters: `$default`

---

first

Returns the first element in iterator or throws an Exception if iterator is empty

---

toArray

Copies elements of this fluent iterator into an array.

FluentFunctions

Fluent interface for function composition.

Methods in `FluentFunctions` return instance of `FluentFunction` that contains all functions from *Functions*.

Calls to FluentFunction can be chained. The resultant function calls chained function in the order they were specified.

For example:

```
$functionC = FluentFunctions::functionA()->functionB();
```

results in a functionC such that for each argument x functionC(x) == functionB(functionA(x)).

Example

Let's create a function that extracts field 'name' from the given argument, then removes prefix 'super', adds ' extra' at the beginning, appends '! ' and surrounds result with `"***"`.

```
$function = FluentFunctions::extractField('name')
      ->removePrefix('super')
      ->prepend(' extra')
      ->append('! ')
      ->surroundWith("***");

$product = new Product(['name' => 'super phone']);

$result = Functions::call($function, $product); //=> '*** extra phone! ***'
```

Comparators

Comparators are used to determine the order of objects in `Arrays::sort`. It is a flexible mechanism to compare objects. Ouzo provides various comparators out of the box and the ability to write your custom comparators.

`Comparator` class is a facade which contains all comparators: * natural * reverse * compareBy * compound

Natural order

As simple as it gets:

```
$result = Arrays::sort([1, 3, 2], Comparator::natural());
```

It sorts given array in a natural order, so the result would be `1, 2, 3`.

Reverse

It is a comparator according to which order of elements is reversed. It expects another comparator as a parameter. E.g.

```
$result = Arrays::sort([1, 3, 2], Comparator::reverse(Comparator::natural()));
```

Result is obviously a reversed array of natural order, which is `3, 2, 1`. Any comparator may be passed as a parameter. Combining comparators? Just imagine the possibilities!

Compare by

Compares objects by using values computed using given expressions. Expressions should comply with format accepted by `Functions::extractExpression`.

Imagine you have `Product` and you want to sort it by its `name` property. Not a problem:

```
$product1 = new Product(['name' => 'b']);
$product2 = new Product(['name' => 'c']);
$product3 = new Product(['name' => 'a']);

$result = Arrays::sort([$product1, $product2, $product3], Comparator::compareBy('name
↪'));
```

In case you haven't heard of Ouzo's assertions, here is the simplest way to test if the above is true:

```
$result = Assert::thatArray($result)->onProperty('name')->containsExactly('a', 'b', 'c
↪');
```

You can specify as many expressions as you need, e.g.

```
Comparator::compareBy('name', 'description', 'price');
```

Compound

Combines comparators into one, ordered by first comparator. If two values are equal according to the first comparator (tie), then tie breakers resolve conflicts. Second provided comparator is the first tie breaker, third is the second tie breaker and so on.

Example:

```
$product1 = new Product(['name' => 'a', 'description' => '2']);
$product2 = new Product(['name' => 'b', 'description' => '2']);
$product3 = new Product(['name' => 'a', 'description' => '1']);

$result = Arrays::sort([$product1, $product2, $product3],
    Comparator::compound(
        Comparator::reverse(Comparator::compareBy('name')),
        Comparator::compareBy('description')
    )
);
```

Now, let's analyze it:

1. products are sorted by `name` property (a, a, b)
2. reversed (b, a, a)
3. there is a conflict (a = a)
4. so a tie breaker goes to work
5. ties are sorted by `description` property (b, a1, a2)

Voila!

Custom comparators

If you want to write your own comparator the only thing you need to do is to create a class with `__invoke` method implemented.

Comparator returns an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Take a look at `Ouzo\Utilities\Comparator` classes for more details.

Iterators

forArray

Returns an iterator containing the elements of `$array`.

Parameters: `array $array`

---

generate

Returns an iterator that uses `$function` to generate elements. `$function` takes one argument which is the current position of the iterator.

Parameters: `callable $function`

---

cycle

Returns an iterator that cycles indefinitely over the elements of `$iterator`.

Parameters: `Iterator $iterator`

---

batch

Returns the elements of `$iterator` grouped in chunks of `$chunkSize`

Parameters: `Iterator $iterator`, `$chunkSize`

---

filter

Returns the elements of `$iterator` that satisfy a predicate.

Parameters: `Iterator $iterator`, `callable $predicate`

---

map

Returns an iterator that applies function to each element of `$iterator`.

Parameters: `Iterator $iterator`, `callable $function`

---

firstOr

Returns the first element or defaultValue if the iterator is empty.

Parameters: `Iterator $iterator`, `$default`

first

Returns the first element in iterator or throws an Exception if iterator is empty

---

limit

Creates an iterator returning the first `$number` elements of the given iterator.

Parameters: `Iterator $iterator`, `$number`

---

skip

Creates an iterator returning all but first `$number` elements of the given iterator.

Parameters: `Iterator $iterator`, `$number`

---

reindex

Returns an iterator that indexes elements numerically starting from 0

Parameters: `Iterator $iterator`

---

toArray

Copies an iterator's elements into an array.

Parameters: `Iterator $iterator`

Cache

Simple request scope cache.

---

get

Returns object from cache. If there's no object for the given key and $functions is passed, $function result will be stored in cache under the given key.

Parameters: `$key`, `$function = null`

Example:

```
$countries = Cache::get("countries", function() {
    //expensive computation that returns a list of countries
    return Country::all();
})
```

---

put

Stores the given object in the cache.

Parameters: `$key`, `$object`

---

contains

Returns true if cache contains an object for the given key.

Parameters: `$key`

---

memoize

Caches the result of the given closure using filename:line as a key.

Parameters: `$function`

Example:

```
$countries = Cache::memoize(function() {
    //expensive computation that returns a list of countries
    return Country::all();
})
```

---

size

Returns number of stored objects.

---

clear

Clear all items stored in cache.

Suppliers

Static utility methods returning suppliers.

---

memoize

Returns a supplier which caches the callback result and returns that value on subsequent calls to `get()`.

```
class Command
{
    public function getNumber()
    {
        return rand();
    }
}

$command = new Command();

$supplier = Suppliers::memoize(function () use ($command) {
    return $command->getNumber(); //returns 1102808477
});

echo $supplier->get(); //1102808477
echo $supplier->get(); //1102808477
```

---

memoizeWithExpiration

Returns a supplier which caches the callback result and removes the cached value after specified time. Subsequent calls to `get()` return the cached value if expiration time has not passed. Time is passed in seconds.

```
class Command
{
    public function getNumber()
    {
        return rand();
    }
}
```

```php
$command = new Command();

$supplier = Suppliers::memoizeWithExpiration(function () use ($command) {
    return $command->getNumber(); //returns 1102808477
}, 10);

echo $supplier->get(); //1102808477
echo $supplier->get(); //1102808477
//after 10 seconds
echo $supplier->get(); //1302561906
```

Path

It is a utility designed to simplify path related operations, such as joining or normalizing paths.

---

join

Allows you to join all given path parts together using system specific directory separator. It ignores empty arguments and excessive separators.

Example:

```php
echo Path::join('/disk', 'my/dir', 'file.txt');
```

Result: `/disk/my/dir/file.txt`

---

joinWithTemp

Similar to `Path::join`, but additionaly it adds system specific temporary directory path at the beginning.

Example:

```php
echo Path::joinWithTemp('/disk', 'my/dir', 'file.txt');
```

Result: `/tmp/disk/my/dir/file.txt`

---

normalize

It normalizes given path by removing unncessary references to parent directories (i.e. "..") and removing double slashes.

Example:

```php
echo Path::normalize('/disk/..//photo.jpg');
```

Result: `/photo.jpg`

Clock

Clock is a better DateTime.

Clock has plus<interval>($count) and minus<interval>($count) methods that return a modified copy of a Clock object.

Example:

```
$string = Clock::now()
    ->plusYears(1)
    ->plusMonths(2)
    ->minusDays(3)
    ->format();
```

Clock provides time travel and time freezing capabilities, making it simple to test time-dependent code.

Clock::freeze sets time to a specific point so that each subsequent call to Clock::now() will return fixed time.

Example:

```
//given
Clock::freeze('2011-01-02 12:34');

//when
$result = Clock::nowAsString('Y-m-d');

//then
$this->assertEquals('2011-01-02', $result);
```

You can obtain a Clock set to a specific point in time.

```
$result = Clock::at('2011-01-02 12:34');
```

You can convert Clock to DateTime:

```
$result = Clock::now()->toDateTime();
```

You can convert Clock to a string using the specified format:

```
$result = Clock::now()->format('Y-m-d H:i:s');
```

You can check if time represented by one Clock instance if before or after another Clock instance time.

```
$result1 = Clock::now()->isBefore(Clock::at('2011-01-02 12:34'));
$result2 = Clock::now()->isAfter(Clock::at('2011-01-02 12:34'));
```

Joiner

Joins array text elements together with a separator. Returns a string.

on

Static method to create a `Joiner` object and define a separator.

Parameters: `$separator`

---

join

Returns a string containing array elements joined together with a separator.

Parameters: `array $array`

Example:

```
$result = Joiner::on(', ')->join([1 => 'A', 2 => 'B', 3 => 'C']);
```

Result: `'A, B, C'`

---

skipNulls

Returns a `Joiner` that skips null elements.

Example:

```
$result = Joiner::on(', ')->skipNulls()->join(['A', null, 'C']);
```

Result: `'A, C'`

---

map

Returns a `Joiner` that transforms array elements before joining.

Example:

```
$result = Joiner::on(', ')->map(function ($key, $value) {
        return "$key => $value";
    })->join([1 => 'A', 2 => 'B', 3 => 'C']);
```

Result:: `'1 => A, 2 => B, 3 => C'`

---

mapValues

Returns a `Joiner` that transforms array values before joining.

Example:*

```
$result = Joiner::on(', ')->mapValues(function ($value) {
        return "val = $value";
    })->join([1 => 'A', 2 => 'B', 3 => 'C']);
```

Result: `'val = A, val = B, val = C'`

TimeAgo

This class is intended to format date in a "time ago" manner. `TimeAgo` class returns key calculated for date e.g. current date returns `timeAgo.justNow`. Additionally it returns array with parameters for the key e.g. `timeAgo.yesterdayAt` has parameter named `label` which contains value.

Example:

```
$currentDate = '2012-02-20 12:00';
Clock::freeze($currentDate);
$timeAgo = TimeAgo::create('2012-02-20 11:00');

$timeAgo->getKey(); //timeAgo.todayAt
$timeAgo->getParams(); //['label' => '11:00']
```

Note: Returned key can be used with Ouzo's *I18n* methods to do the translation .

---

timeAgo.justNow

Returned when difference between current date and given date is *less or equal than 60 seconds*.

Params: `none`

---

timeAgo.minAgo

Returned when difference between current date and given date is *greater than 60 seconds* and *less or equal than 60 minutes*.

Params: `['label' => $minutesAgo]`

---

timeAgo.todayAt

Returned when *day is the same* and difference between current date and given date is *greater than 60 minutes* and *less or equal than 24 hours*.

Params: `['label' => $date->format('H:i')]`

---

timeAgo.yesterdayAt

Returned when *day is yesterday*.

Params: `['label' => $date->format('H:i')]`

---

timeAgo.thisYear

Returned when *year is the same*.

Params: `['day' => $date->format('j'), 'month' => 'timeAgo.month.' . $date->format('n')]`

Note: Parameter `month` has value in a format: `timeAgo.month.<number of month>`. Number of month is between 1..12 (January..December).

Date is returned

If date is returned e.g. `2014-01-10` it means that *date is before the current year*.

Params: `none`

Model generator

Model generator is a console tool for creating Model classes for existing database tables. Generator reads information about database table and transforms it into Ouzo's Model class.

Note: Currently there is a support for MySQL and PostgreSQL.

Basic example

Change current path to project directory (e.g. myproject):

```
cd myproject
```

Generate Model class body for table **users** containing three columns: *id*, *login*, *password*:

```
./console ouzo:model_generator users
```

The command should output a model class **User**:

```
--------------------------------
Database name: thulium_1
Class name: PhoneParam
Class namespace: Application\Model
--------------------------------
<?php
namespace Application\Model;

use Ouzo\Model;

/**
 * @property string login
 * @property string password
*/
class User extends Model
{
    private $_fields = ['login', 'password'];

    public function __construct($attributes = [])
    {
        parent::__construct([
            'table' => 'users',
            'primaryKey' => 'id',
            'attributes' => $attributes,
            'fields' => $this->_fields
        ]);
    }
}
Saving class to file: '/path/to/myproject/Application/Model/User.php'
```

As you can see `$_fields` lists all `users` table columns (except for id which is specified by `primaryKey` parameter).

Note: You could save the generated class to a file by specifying `-f=/path/to/file.php` option. If not specified namespace and class name is used.

Arguments

```
table                       Table name.
```

Options

```
--class (-c)            Class name. If not specified class name is generated based on␣
↪table name.
--file (-f)             Class file path. If not specified namespace and class name is␣
↪used.
--namespace (-s)        Class namespace (e.g 'Model\MyModel'). Hint: Remember to escape␣
↪backslash (\\)! (default: "Model").
--remove-prefix (-p)    Remove prefix from table name when generating class name␣
↪(default: "t").
```

```
--output-only (-o)     Do not save file on disk, only display output.
--short-arrays (-a)    Use shorthand array syntax (PHP 5.4).
```

Note: If no option is specified application will print the help message.

CHAPTER 2

PhpStorm plugins

- Ouzo framework plugin
- DynamicReturnTypePlugin - for Mock and CatchException. You have to copy dynamicReturnTypeMeta.json to your project root.

Index

G