Agile Toolkit Core

Release 0.1.0

Mar 17, 2023

Contents

1 Object Containers **3**
- 1.1 Containers . . . 4
- 1.2 Initializer Trait . . . 9
- 1.3 Factory Class . . . 10
- 1.4 AppScope Trait . . . 18
- 1.5 Dependency Injection Container . . . 19

2 Hooks **21**
- 2.1 Hook Trait . . . 22
- 2.2 Hook-based dynamic Methods . . . 25

3 Modelable Objects **29**
- 3.1 Modelable Trait . . . 30

4 Exceptions **31**

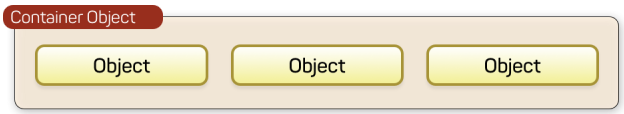
5 Others **33**
- 5.1 Debug Trait . . . 33
- 5.2 Writing ATK Docs . . . 34

Index **37**

Agile Core is a collection of PHP Traits for designing object-oriented frameworks. The main purpose for this project is to work as a foundation for [Agile UI](https://github.com/atk4/ui) and [Agile Data](https://github.com/atk4/data), but you are welcome to use some or all traits elsewhere.

CHAPTER 1

Object Containers



Within your application or framework you can quite often have requirement for using containers:

- Form containing fields
- Table containing columns
- Component containing sub-components
- Model object containing Field objects
- Application containing controllers

The natural solution to the problem is to create array like this:

```
public $fields = [];
```

After that you would need to create code for adding objects into container, removing, verify their existence etc.

CollectionTrait implements several handy methods which can be used to create necessary methods with minimum code footprint:

```
class Form
{
    use CollectionTrait;

    public $fields = [];

    public function addField(string $name, $seed = [])
    {
        $seed = Factory::mergeSeeds($seed, [FieldMock::class]);
```

(continues on next page)

(continued from previous page)

```
        $field = Factory::factory($seed, ['name' => $name]);

        return $this->_addIntoCollection($name, $field, 'fields');
    }

    // hasField, getField, removeField also can be added, see further docs.
}
```

Traits add multiple checks to prevert collisions between existing objects, call init() method, carry over $app and set $owner properties and calculate 'name' by combining it with the parent.

CollectionTrait only supports named object - you may not omit the name, however a more older implementation of *ContainerTrait* is used primarily for tracking Render Tree in ATK UI where name is optional and a unique name is guaranteed.

When $container is using *ContainerTrait*, this would be a typical code:

```
$child = $container->add(new ChildClass());

// $child is the object of ChildClass

$container->removeElement($child);
```

Although containers work with any objects, assigning additional traits to your ChildClass can extend the basic functionality.

- *InitializerTrait* will add automatic code execution when adding
- *AppScopeTrait* will pass value of $app property from container to child.
- *TrackableTrait* will let you assign unique names to object
- *Factory* will let you specify object class by Seed
- *DiContainerTrait* will let you with dependency injection

Just to clarify what Seed is:

```
$field = $form->add('TextArea');
```

In this scenario, even though a new object is added, we don't do it ourselves. We simply specify some information on how to create and what properties to inject into the object:

```
$field = $form->add([\Atk4\Ui\FormField\Password::class, 'icon' => ['lock', 'circular␣
↪inverted'], 'width' => 4);
```

The above code will determine the correct object to implement Password inside Form, instantiate it and then even add Icon object which is also defined through seed.

1.1 Containers

There are two relevant traits in the Container mechanics. Your "container" object should implement *ContainerTrait* and your child objects should implement *TrackableTrait* (if not, the $owner/$elements links will not be established)

If both parent and child implement *AppScopeTrait* then the property of `AppScopeTrait::app` will be copied from parent to the child also.

If your child implements *InitializerTrait* then the method *InitializerTrait::init* will also be invoked after linking is done.

You will be able to use *ContainerTrait::getElement()* to access elements inside container:

```
$object->add(new AnoterObject(), 'test');
$anotherObject = $object->getElement('test');
```

If you additionally use *TrackableTrait* together with *NameTrait* then your objects also receive unique "name". From example above:

- $object->name == "app_object_4"
- $anotherObject->name == "app_object_4_test"

1.1.1 Name Trait

trait NameTrait

Name trait only adds the 'name' property. Normally you don't have to use it because *TrackableTrait* automatically inherits this trait. Due to issues with PHP5 if both *ContainerTrait* and *TrackableTrait* are using *NameTrait* and then both applied on the object, the clash results in "strict warning". To avoid this, apply *NameTrait* on Containers only if you are NOT using *TrackableTrait*.

Properties

property NameTrait::$**name**

Name of the object.

Methods

None

1.1.2 CollectionTrait

trait CollectionTrait

This trait makes it possible for you to add child objects into your object, but unlike "ContainerTrait" you can use multiple collections stored as different array properties.

This class does not offer automatic naming, so if you try to add another element with same name, it will result in exception.

Example:

```
class Form
{
    use Core\CollectionTrait;

    protected $fields = [];

    public function addField(string $name, $seed = [])
    {
        $seed = Factory::mergeSeeds($seed, [FieldMock::class]);

        $field = Factory::factory($seed, ['name' => $name]);
```

(continues on next page)

(continued from previous page)

```
        return $this->_addIntoCollection($name, $field, 'fields');
    }

    public function hasField(string $name): bool
    {
        return $this->_hasInCollection($name, 'fields');
    }

    public function getField(string $name)
    {
        return $this->_getFromCollection($name, 'fields');
    }

    public function removeField(string $name)
    {
        $this->_removeFromCollection($name, 'fields');
    }
}
```

Methods

`CollectionTrait::_addIntoCollection` (*string $name*, *object $object*, *string $collection*)
Adds a new element into collection:

```
public function addField(string $name, $seed = [])
{
    $field = Factory::factory($seed);

    return $this->_addIntoCollection($name, $field, 'fields');
}
```

Factory usage is optional but would allow you to pass seed into addField()

`CollectionTrait::_removeFromCollection` (*string $name*, *string $collection*)
Remove element with a given name from collection.

`CollectionTrait::_hasInCollection` (*string $name*, *string $collection*)
Return object if it exits in collection and false otherwise

`CollectionTrait::_getFromCollection` (*string $name*, *string $collection*)
Same as _hasInCollection but throws exception if element is not found

`CollectionTrait::_shortenMl` (*$string $ownerName*, *string $itemShortName*)
Implements name shortening

Shortening is identical to :php:meth::*ContainerTrait::_shorten*.

Your object can this train together with ContainerTrait. As per June 2019 ATK maintainers agreed to gradually refactor ATK Data to use CollectionTrait for fields, relations, actions.

1.1.3 Container Trait

`trait ContainerTrait`
If you want your framework to keep track of relationships between objects by implementing containers, you can use *`ContainerTrait`*. Example:

```
class MyContainer extends OtherClass
{
    use Atk4\Core\ContainerTrait;

    public function add(object $obq, $args = []): object
    {
        $this->_addContainer($obj, is_string($args) ? ['name' => $args] : $args);

        return $obj;
    }
}

class MyItem
{
    use Atk4\Core\TrackableTrait;
    use Atk4\Core\NameTrait;
}

Now the instances of MyItem can be added to instances of MyContainer
and can keep track::

$parent = new MyContainer();
$parent->name = 'foo';
$parent->add(new MyItem(), 'child1');
$parent->add(new MyItem());

echo $parent->getElement('child1')->name;
// foo_child1

if ($parent->hasElement('child1')) {
    $parent->removeElement('child1');
}

foreach ($parent as $child) {
    $child->doSomething();
}
```

Child object names will be derived from the parent name.

Properties

property ContainerTrait::$**elements**

Contains a list of objects that have been "added" into the current container. The key is a "shot_name" of the child. The actual link to the element will be only present if child uses both *TrackableTrait* and *NameTrait* traits, otherwise the value of array key will be "true".

Methods

ContainerTrait::**add**(*$obj*, *$args = []*)

If you are using ContainerTrait only, then you can safely use this add() method. If you are also using factory, or initializer then redefine add() and call _addContainer, _addFactory,.

ContainerTrait::**_addContainer**(*object $element*, *array $args*) → void

Add element into container. Normally you should create a method add() inside your class that will execute this method. Because multiple traits will want to contribute to your add() method, you should see sample implementation in Object::add.

Your minimum code should be:

```
public function add(object $obj, $args = []): object
{
    $this->_addContainer($obj, is_string($args) ? ['name' => $args] : $args);

    return $obj;
}
```

$args be in few forms:

```
$args = ['child_name'];
$args = 'child_name';
$args = ['child_name', 'db' => $mydb];
$args = ['name' => 'child_name']; // obsolete, backward-compatible
```

Method will return the object. Will throw exception if child with same name already exist.

`ContainerTrait::`**`removeElement`**(*$shortName*)
Will remove element from $elements. You can pass either shortName or the object itself. This will be called if *`TrackableTrait::destroy`* is called.

`ContainerTrait::`**`_shorten`**(*$string $ownerName*, *string $itemShortName*)
Given the long owner name and short child name, this method will attempt to shorten the length of your children. The reason for shortening a name is to impose reasonable limits on overly long names. Name can be used as key in the GET argument or form field, so for a longer names they will be shortened.

This method will only be used if current object has `AppScope`, since the application is responsible for keeping shortenings.

`ContainerTrait::`**`getElement`**(*$shortName*)
Given a short-name of the element, will return the object. Throws exception if object with such shortName does not exist.

`ContainerTrait::`**`hasElement`**(*$shortName*)
Given a short-name of the element, will return true if object with such shortName exists, otherwise false.

`ContainerTrait::`**`_uniqueElementName`**()
Internal method to create unique name for an element.

1.1.4 Trackable Trait

`trait TrackableTrait`
Trackable trait implements a few fields for the object that will maintain it's relationship with the owner (parent).

When name is set for container, then all children will derive their names of the parent.

- Parent: foo
- Child: foo_child1

The name will be unique within this container.

Properties

`property` `TrackableTrait::$`**`owner`**
Will point to object which has add()ed this object. If multiple objects have added this object, then this will point to the most recent one.

property TrackableTrait::$**shortName**
When you add item into the owner, the "shortName" will contain short name of this item.

Methods

TrackableTrait::**getDesiredName**()
Normally object will try to be named after it's class, if the name is omitted. You can override this method to implement a different mechanics.

If you pass 'desired_name' => 'heh' to a constructor, then it will affect the preferred name returned by this method. Unlike 'name' => 'heh' it won't fail if another element with this name exists, but will add '_2' postfix.

TrackableTrait::**destroy**()
If object owner is set, then this will remove object from it's owner elements reducing number of links to the object. Normally PHP's garbage collector should remove object as soon as number of links is zero.

1.2 Initializer Trait

trait InitializerTrait

1.2.1 Introduction

With our traits objects now become linked with the "owner" and the "app". Initializer trait allows you to define a method that would be called after object is linked up into the environment.

Declare a object class in your framework:

```
class FormField
{
    use AppScopeTrait;
    use InitializerTrait;
    use NameTrait;
    use TrackableTrait;
}

class FormField_Input extends FormField
{
    public $value = null;

    protected function init(): void
    {
        parent::init();

        if ($_POST[$this->name) {
            $this->value = $_POST[$this->name];
        }
    }

    public function render()
    {
        return $this->getApp()->getTag('input/', ['name' => $this->name, 'value' =>
↪$value]);
    }
}
```

1.2.2 Methods

`InitializerTrait::init()`
: A blank init method that should be called. This will detect the problems when init() methods of some of your base classes has not been executed and prevents from some serious mistakes.

If you wish to use traits class and extend it, you can use this in your base class:

```
class FormField
{
    use AppScopeTrait;
    use InitializerTrait {
        init as _init
    }
    use TrackableTrait;
    use NameTrait;

    public $value = null;

    protected function init(): void
    {
        $this->_init(); // call init of InitializerTrait

        if ($_POST[$this->name) {
            $this->value = $_POST[$this->name];
        }
    }
}
```

1.3 Factory Class

class Factory

1.3.1 Introduction

This trait is used to initialize object of the appropriate class, handling things like:

- determining name of the class with ability to override
- passing argument to constructors
- setting default property values

Thanks to Factory trait, the following code:

```
$button = $app->add(['Button', 'A Label', 'icon' => 'book', 'action' =>␣
↪My\Action::class]);
```

can replace this:

```
$button = new \Atk4\Ui\Button('A Label');
$button->icon = new \Atk4\Ui\Icon('book');
$button->action = new My\Action();
$app->add($button);
```

Type Hinting

Agile Toolkit 2.1 introduces support for a new syntax. It is functionally identical to a short-hand code, but your IDE will properly set type for a *$button* to be *class Button* instead of *class View*:

```
$button = Button::addTo($view, ['A Label', 'icon' => 'book', 'action' =>␣
↪My\Action::class]);
```

The traditional *$view->add* will remain available, there are no plans to remove that syntax.

1.3.2 Class Name Resolution

An absolute/full class name must be always provided. Relative class name resolution was obsoleted/removed.

1.3.3 Seed

Using "class" as opposed to initialized object yields many performance gains, as initialization of the class may be delayed until it's required. For instance:

```
$model->hasMany('Invoices', Invoice::class);

// is faster than

$model->hasMany('Invoices', new Invoice());
```

That is due to the fact that creating instance of "Invoice" class is not required until you actually traverse into it using *$model->ref('Invoices')* and can offer up to 20% performance increase. But in some cases, you want to pass some information into the object.

Suppose you want to add a button with an icon:

```
$button = $view->add('Button');
$button->icon = new Icon('book');
```

It's possible that some call-back execution will come before button rendering, so it's better to replace icon with the class:

```
$button = $view->add('Button');
$button->icon = Icon::class;
```

In this case, however - it is no longer possible to pass the "book" parameter to the constructor of the Icon class.

This problem is solved in ATK with "Seeds".

A Seed is an array consisting of class name/object, named and numeric arguments:

```
$seed = [Button::class, 'My Label', 'icon' => 'book'];
```

Seed with and without class

There are two types of seeds - with class name and without. The one above contains the class and is used when user needs a flexibility to specify a class:

```
$app->add(['Button', 'My Label', 'icon' => 'book']);
```

The other seed type is class-less and can be used in situations where there are no ambiguity about which class is used:

```
$button->icon = ['book'];
```

Either of those seeds can be replaced with the Object:

```
$button = $app->add(new Button('My Label'));
$button->icon = new Icon('book');
```

If seed is a string then it would be treated as class name. For a class-less seed it would be treaded as a first argument to the construcor:

```
$button = $app->add('Button');
$button->icon = 'book';
```

Lifecycle of argument-bound seed

ATK only uses setters/getters when they make sense. Argument like "icon" is a very good example where getter is needed. Here is a typical lifecycle of an argument:

1. when object is created "icon" is set to null
2. seed may have a value for "icon" and can set it to string, array or object
3. user may explicitly set "icon" to string, array or object
4. some code may wish to interract with icon and will expect it to be object
5. recursiveRender() will expect icon to be also added inside $button's template

So here are some rules for ATK and add-ons:

- use class-less seeds where possible, but indicate so in the comments
- keep seed in its original form as long as possible
- use getter (getIcon()) which would convert seed into object (if needed)
- add icon object into render-tree inside recursiveRender() method

If you need some validation (e.g. icon and iconRight cannot be set at the same time by the button), do that inside recursiveRender() method or in a custom setter.

If you do resort to custom setters, make sure they return $this for better chaining.

Always try to keep things simple for others and also for yourself.

1.3.4 Factory

As mentioned juts above - at some point your "Seed" must be turned into Object. This is done by executing factory method.

`Factory::factory` (*$seed*, *$defaults = []*)

Creates and returns new object. If is_object($seed), then it will be returned and $defaults will only be sed if object implement DiContainerTrait.

In a conventional PHP, you can create and configure object before passing it onto another object. This action is called "dependency injecting". Consider this example:

```
$button = new Button('A Label');
$button->icon = new Icon('book');
$button->action = new Action(..);
```

Because Components can have many optional components, then setting them one-by-one is often inconvenient. Also may require to do it recursively, e.g. `Action` may have to be configured individually.

Agile Core implements a mechanism to make that possible through using Factory::factory() method and specifying a seed argument:

```
use Atk4\Ui\Button;

$button = Factory::factory([Button::Class, 'A Label', 'icon' => ['book'], 'action' =>␣
↪new Action(..)]);
```

Note that passing 'icon' => ['book'] will also use factory to initialize icon object.

Finally, if you are using IDE and type hinting, a preferred code would be:

```
use Atk4\Ui\Button;

$button = new Button('A Label');
Factory::factory($button, ['icon' => ['book'], 'action' => new Action(..)]);
```

This will properly set type to $button variable, while still setting properties for icon/action. More commonly, however, you would use this through the add() method:

```
use Atk4\Ui\Button;

$button = new Button('A Label');
$view->add([$button, 'icon' => ['book'], 'action' => new Action('..')]);
```

Seed Components

Class definition - passed as the `$seed[0]` and is the only mandatory component, e.g:

```
$button = Factory::factory([Button::class]);
```

Any other numeric arguments will be passed as constructor arguments:

```
$button = Factory::factory([Button::class, 'My Label', 'red', 'big']);

// results in

new Button('My Label', 'red', 'big');
```

Finally any named values inside seed array will be assigned to class properties by using `DiContainerTrait::setDefaults`.

Factory uses *array_shift* to separate class definition from other components.

Class-less seeds

You cannot create object from a class-less seed, simply because factory would not know which class to use. However it can be passed as a second argument to the factory:

```
$this->icon = Factory::factory([Icon::class, 'book'], $this->icon);
```

This will use class icon and first argument 'book' as default, but would use exitsing seed version if it was specified. Also it will preserve the object value of an icon.

Factory Defaults

Defaults array takes place of $seed if $seed is missing components. $defaults is using identical format to seed, but without the class. If defaults is not an array, then it's wrapped into [].

Array that lacks class is called defaults, e.g.:

```
$defaults = ['Label', 'My Label', 'big red', 'icon' => 'book'];
```

You can pass defaults as second argument to *Factory::factory()*:

```
$button = Factory::factory([Button::class], $defaults);
```

Executing code above will result in 'Button' class being used with 'My Label' as a caption and 'big red' class and 'book' icon.

You may also use `null` to skip an argument, for instance in the above example if you wish to change the label, but keep the class, use this:

```
$label = Factory::factory([null, 'Other Label'], $defaults);
```

Finally, if you pass key/value pair inside seed with a value of `null` then default value will still be used:

```
$label = Factory::factory(['icon' => null], $defaults);
```

This will result icon=book. If you wish to disable icon, you should use `false` value:

```
$label = Factory::factory(['icon' => false], $defaults);
```

With this it's handy to pass icon as an argument and don't worry if the null is used.

Precedence and Usage

When both seed and defaults are used, then values inside "seed" will have precedence:

- for named arguments any value specified in "seed" will fully override identical value from "defaults", unless if the seed's value is "null".
- for constructor arguments, the non-null values specified in "seed" will replace corresponding value from $defaults.

The next example will help you understand the precedence of different argument values. See my description below the example:

```
class RedButton extends Button
{
    protected $icon = 'book';

    protected function init(): void
    {
        parent::init();
```

(continues on next page)

(continued from previous page)

```
        $this->icon = 'right arrow';
    }
}

$button = Factory::factory([RedButton::class, 'icon' => 'cake'], ['icon' => 'thumbs up
↪']);
// Question: what would be $button->icon value here?
```

Factory will start by merging the parameters and will discover that icon is specified in the seed and is also mentioned in the second argument - $defaults. The seed takes precedence, so icon='cake'.

Factory will then create instance of RedButton with a default icon 'book'. It will then execute `DiContainerTrait::setDefaults` with the *['icon' => 'cake']* which will change value of $icon to *cake*.

The *cake* will be the final value of the example above. Even though *init()* method is set to change the value of icon, the *init()* method is only executed when object becomes part of RenderTree, but that's not happening here.

1.3.5 Seed Merging

`Factory::mergeSeeds` (*$seed*, *$seed2*, ...)

Two (or more) seeds can be merged resulting in a new seed with some combined properties:

1. **Class of a first seed will be selected. If specified as "null" will be picked** from next seed.
2. If string as passed as any of the argument it's considered to be a class
3. If object is passed as any of the argument, it will be used instead ignoring all classes and numeric arguments. All the key->value pairs will be merged and passed into setDefaults().

Some examples:

```
Factory::mergeSeeds(['Button', 'Button Label'], ['Message', 'Message label']);
// results in ['Button', 'Button Label']

Factory::mergeSeeds([null, 'Button Label'], ['Message', 'Message Label']);
// Results in ['Message', 'Button Label']);

Factory::mergeSeeds(['null, 'Label1', 'icon' => 'book'], ['icon' => 'coin', 'Button'],
↪ ['class' => ['red']]);
// Results in ['Button', 'Label1', 'icon' => 'book', 'class' => ['red']]
```

Seed merging can also be used to merge defaults:

```
Factory::mergeSeeds(['label 1'], ['icon' => 'book']);
// results in ['label 1', 'icon' => 'book']
```

When object is passed, it will take precedence and absorb all named arguments:

```
Factory::mergeSeeds(
    ['null, 'Label1', 'icon' => 'book'],
    ['icon' => 'coin', 'Button'],
    new Message('foobar'),
    ['class' => ['red']]
);
// result is
```

(continues on next page)

(continued from previous page)

```
// $obj = new Message('foobar');
// $obj->setDefaults(['icon' => 'book', 'class' => ['red']]);
```

If multiple objects are specified then early ones take precedence while still absorbing all named arguments.

Default and Seed objects

When object is passed as 2nd argument to Factory::factory() it takes precedence over all array-based seeds. If 1st argument of Factory::factory() is also object, then 1st argument object is used:

```
Factory::factory([Icon::class, 'book'], ['pencil']);
// book

Factory::factory([Icon::class, 'book'], new Icon('pencil'));
// pencil

Factory::factory(new Icon('book'), new Icon('pencil'));
// book
```

1.3.6 Usage in frameworks

There are several ways to use Seed Merging and Agile UI / Agile Data makes use of those patterns when possible.

Specify Icon for a Button

As you may know, Button class has icon property, which may be specified as a string, seed or object:

```
$button = $app->add(['Button', 'icon' => 'book']);
```

Well, to implement the button internally, render method uses this:

```
// in Form
$this->buttonSave = Factory::factory([Button::class], $this->buttonSave);
```

So the value you specify for the icon will be passed as:

- string: argument to constructor of *Button()*.
- array: arguments for constructors and inject properties
- object: will override return value

Specify Layout

The first thing beginners learn about Agile Toolkit is how to specify layout:

```
$app = new \Atk4\Ui\App('Hello World');
$app->initLayout('Centered');
```

The argument for initLayout is passed to factory:

```
$this->layout = Factory::factory($layout);
```

The value you specify will be treated like this:

- string: specify a class (prefixed by Layout)
- array: specify a class and allow to pass additional argument or constructor options
- object: will override layout

Form::addField and Table::addColumn

Agile UI is using form field classes from namespace Atk4UiFormField. A default class is 'Line' but there are several ways how it can be overridden:

- User can specify $ui['form'] / $ui['table'] property for model's field
- User can pass 2nd parameter to addField()
- Class can be inferred from field type

Each of the above can specify class name, so with 3 seed sources they need merging:

```
$seed = Factory::mergeSeeds($decorator, $field->ui, $inferred,␣
↪[\Atk4\Ui\FormField\Line::class, 'form' => $this]);
$decorator = Factory::factory($seed, null, 'FormField');
```

Passing an actual object anywhere will use it instead even if you specify seed.

Specify Form Field

addField, addButton, etc

Model::addField, Form::addButton, FormLayout::addHeader imply that the class of an added object is known so the argument you specify to those methods ends up being a factory's $default:

```
public function addButton($label)
{
    return $this->add(
        Factory::factory([Button::class, null, 'secondary'], $label);
        'Buttons'
    );
}
```

in this code factory will use a seed with a *null* for label, which means, that label will be actually taken from a second argument. This pattern enables 3 ways to use addButton():

```
$form->addButton('click me');
// Adds a regular button with specified label, as expected

$form->addButton(['click me', 'red', 'icon' => 'book']);
// Specify class of a button and also icon

$form->addButton(new MyButton('click me'));
// Use an object specified instead of a button
```

A same logic can be applied to addField:

```
$model->addField('is_vip', ['type' => 'boolean']);
// class = Field, type = boolean

$model->addField('is_vip', ['boolean'])
// new Field('boolean'), same result

$model->addField('is_vip', new MyBoolean());
// new MyBoolean()
```

and the implementation uses factory's default:

```
$field = Factory::factory($this->fieldSeed);
```

Normally the field class property is a string, which will be used, but it can also be array.

1.4 AppScope Trait

trait AppScopeTrait

1.4.1 Introduction

Typical software design will create the application scope. Most frameworks relies on "static" properties, methods and classes. This does puts some limitations on your implementation (you can't have multiple applications).

App Scope will pass the 'app' property into all the object that you're adding, so that you know for sure which application you work with.

1.4.2 Properties

property AppScopeTrait::$**app**
: Always points to current Application object

property AppScopeTrait::$**maxNameLength**
: When using mechanism for ContainerTrait, they inherit name of the parent to generate unique name for a child. In a framework it makes sense if you have a unique identifiers for all the objects because this enables you to use them as session keys, get arguments, etc.

  Unfortunately if those keys become too long it may be a problem, so ContainerTrait contains a mechanism for auto-shortening the name based around maxNameLength. The mechanism does only work if AppScopeTrait is used, $app property is set and has a maxNameLength defined. Minimum value is 40.

property AppScopeTrait::$**uniqueNameHashes**
: As more names are shortened, the substituted part is being placed into this hash and the value contains the new key. This helps to avoid creating many sequential prefixes for the same character sequence.

1.4.3 Methods

None

1.5 Dependency Injection Container

trait DiContainerTrait

Agile Core implements basic support for Dependency Injection Container.

1.5.1 What is Dependency Injection

As it turns out many PHP projects have built objects which hard-code dependencies on another object/class. For instance:

```
$book = new Book();
$book->name = 'foo';
$book->save(); // saves somewhere??
```

The above code uses some ORM notation and the book record is saved into the database. But how does Book object know about the database? Some frameworks thought it could be a good idea to use GLOBALS or STATIC. PHP Community is fighting against those patterns by using Dependency Injection which is a pretty hot topic in the community.

In Agile Toolkit this has never been a problem, because all of our objects are designed without hard dependencies, globals or statics in the first place.

"Dependency Injection" is just a fancy word for ability to specify other objects into class constructor / property:

```
$book = new Book($mydb);
$book['name'] = 'foo';
$book->save(); // saves to $mydb
```

1.5.2 What is Dependency Injection Container

By design your objects should depend on as little other objects as possible. This improves testability of objects, for instance. Typically constructor can be good for 1 or 2 arguments.

However in Agile UI there are components that are designed specifically to encapsulate many various objects. Crud for example is a fully-functioning editing solution, but suppose you want to use custom form object:

```
$crud = new Crud([
    'formEdit' => new MyForm(),
    'formAdd' => new MyForm(),
]);
```

In this scenario you can't pass all of the properties to the constructor, and it's easier to pass it through array of key/values. This pattern is called Dependency Injection Container. Theory states that developers who use IDEs extensively would prefer to pass "object" and not "array", however we typically offer a better option:

```
$crud = new Crud();
$crud->formEdit = new MyForm();
$crud->formAdd = new MyForm();
```

1.5.3 How to use DiContainerTrait

Calling this method will set object's properties. If any specified property is undefined then it will be skipped. Here is how you should use trait:

```
class MyObj
{
    use DiContainerTrait;

    public function __construct($defaults = [])
    {
        $this->setDefaults($defaults, true);
    }
}
```

You can also extend and define what should be done if non-property is passed. For example Button component allows you to pass value of $content and $class like this:

```
$button = new Button(['My Button Label', 'red']);
```

This is done by overriding setMissingProperty method:

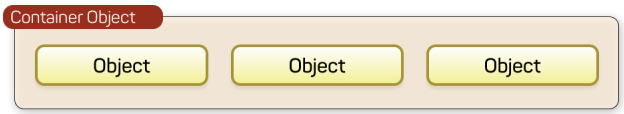
```
class MyObj
{
    use DiContainerTrait {
        setMissingProperty as private _setMissingProperty;
    }

    public function __construct($defaults = [])
    {
        $this->setDefaults($defaults, true);
    }

    protected function setMissingProperty($key, $value)
    {
        // do something with $key / $value

        // will either cause exception or will ignorance
        $this->_setMissingProperty($key, $value);
    }
}
```

CHAPTER 2

Hooks

When you look to make your framework / application extendable, the Hooks is a modern standard in PHP applications. This way a 3rd-party addon can execute code every time components are rendered or data is saved into database.

Our implementation of Hooks is based around storing callback references in array for your standard objects then executing them.



Yet HookTrait implements many much needed extensions to make hooks work great:

- define multiple hooking spot per object, e.g: 'beforeInit', 'beforeDelete' etc
- multiple call-back can be assigned to each spot
- callbacks are executed in order of numeric priority
- arguments can be passed to callbacks
- return values can be collected from callbacks
- callback may "Break Hook" preventing other callbacks from being executed

Once you assign *HookTrait* to AnyClass, you can start assigning and triggering callbacks:

```
$object = new AnyClass();

$object->onHook('test', function ($o) {
    echo 'hello';
});
$object->onHook('test', function ($o) {
    echo 'world';
});
```

(continues on next page)

(continued from previous page)

```
$object->hook('test'); // outputs: helloworld
```

2.1 Hook Trait

trait HookTrait

2.1.1 Introduction

HookTrait adds some methods into your class to registering call-backs that would be executed by triggering a hook. All hooks are local to the object, so if you want to have application-wide hook then use *app* property.

2.1.2 Hook Spots

Hook is described by a string identifier which we call hook-spot, which would normally be expressing desired action with prefixes "before" or "after if necessary.

Some good examples for hook spots are:

- beforeSave
- afterDelete
- validation

The framework or application would typically execute hooks like this:

```
$obj->hook('spot');
```

You can register multiple call-backs to be executed for the requested *spot*:

```
$obj->onHook('spot', function ($obj) {
    echo "Hook 'spot' is called!";
});
```

2.1.3 Adding callbacks

`HookTrait::`**`onHook`**(*$spot*, *$fx = null*, *array $args = []*, *int $priority = 5*)

Register a call-back method. Calling several times will register multiple callbacks which will be execute in the order that they were added.

2.1.4 Short way to describe callback method

There is a concise syntax for using $fx by specifying object only. In case $fx is omitted then $this object is used as $fx.

In this case a method with same name as $spot will be used as callback:

```
protected function init(): void
{
    parent::init();

    $this->onHookShort($spot, function (...$args) {
        $this->beforeUpdate(...$args);
    });
}

protected function beforeUpdate()
{
    // will be called from the hook
}
```

2.1.5 Callback execution order

$priority will make hooks execute faster. Default priority is 5, but if you add hook with priority 1 it will always be executed before any hooks with priority 2, 3, 5 etc.

Normally hooks are executed in the same order as they are added, however if you use negative priority, then hooks will be executed in reverse order:

```
$obj->onHook('spot', third, [], -1);

$obj->onHook('spot', second, [], -5);
$obj->onHook('spot', first, [], -5);

$obj->onHook('spot', fourth, [], 0);
$obj->onHook('spot', fifth, [], 0);

$obj->onHook('spot', ten, [], 1000);

$obj->onHook('spot', sixth, [], 2);
$obj->onHook('spot', seventh, [], 5);
$obj->onHook('spot', eight);
$obj->onHook('spot', nine, [], 5);
```

HookTrait::**hook**(*$spot*, *$args = null*)

execute all hooks in order. Hooks can also return some values and those values will be placed in array and returned by hook():

```
$mul = function ($obj, $a, $b) {
    return $a*$b;
};

$add = function ($obj, $a, $b) {
    return $a+$b;
};

$obj->onHook('test', $mul);
$obj->onHook('test', $add);

$res1 = $obj->hook('test', [2, 2]);
// res1 = [4, 4]
```

(continues on next page)

(continued from previous page)

```
$res2 = $obj->hook('test', [3, 3]);
// res2 = [9, 6]
```

2.1.6 Arguments

As you see in the code above, we were able to pass some arguments into those hooks. There are actually 3 sources that are considered for the arguments:

- first argument to callbacks is always the $object
- arguments passed as 3rd argument to onHook() are included
- arguments passed as 2nd argument to hook() are included

You can also use key declarations if you wish to override arguments:

```
// continue from above example

$pow = function ($obj, $a, $b, $power) {
    return pow($a, $power)+$pow($b, $power);
}

$obj->onHook('test', $pow, [2]);
$obj->onHook('test', $pow, [7]);

// execute all 3 hooks
$res3 = $obj->hook('test', [2, 2]);
// res3 = [4, 4, 8, 256]

$res4 = $obj->hook('test', [2, 3]);
// res3 = [6, 5, 13, 2315]
```

2.1.7 Breaking Hooks

`HookTrait::breakHook()`

When this method is called from a call-back then it will cause all other callbacks to be skipped.

If you pass $return argument then instead of returning all callback return values in array the $return will be returned by hook() method.

If you do not pass $return value (or specify null) then list of the values collected so far will be returned

Remember that adding breaking hook with a lower priority can prevent other call-backs from being executed:

```
$obj->onHook('test', function ($obj) {
    $obj->breakHook("break1");
});

$obj->onHook('test', function ($obj) {
    $obj->breakHook("break2");
}, [], -5);

$res3 = $obj->hook('test', [4, 4]);
// res3 = "break2"
```

breakHook method is implemented by throwing a special exception that is then caught inside hook() method.

2.1.8 Using references in hooks

In some cases you want hook to change certain value. For example when model value is set it may call normalization hook (methods will change $value):

```
public function set($field, $value)
{
    $this->hook('normalize', [&$value]);
    $this->data[$field] = $value;
}

$m->onHook('normalize', function (&$a) {
    $a = trim($a);
});
```

2.1.9 Checking if hook has callbacks

`HookTrait::`**`hookHasCallbacks`**`()`

This method will return true if at least one callback has been set for the hook.

2.2 Hook-based dynamic Methods

`DynamicMethodTrait` adds ability to add methods into objects dynamically. That's like a "trait" feature of a PHP, but implemented in run-time:

```
$object->addMethod('test', function ($o, $args) {
    echo 'hello, ' . $args[0];
});
$object->test('world'); // outputs: hello, world
```

There are also methods for removing and checking if methods exists, so:

```
method_exists($object, 'test');
// now should use
$object->hasMethod('test');

// and this way you can remove method
$object->removeMethod('test');
```

The implementation of dynamic methods relies on Hook trait, so to use it:

```
class AnyClass extends OtherClass
{
    use HookTrait;
    use DynamicMethodTrait;

    // .. your code ..
}
```

2.2.1 Dynamic Method Trait

`trait DynamicMethodTrait`

Introduction

Adds ability to add methods into objects dynamically. That's like a "trait" feature of a PHP, but implemented in run-time:

```
$object->addMethod('test', function ($o, $args) {
    echo 'hello, ' . $args[0];
});
$object->test('world');
```

Global Methods

If object has application scope *AppScopeTrait* and the application implements *HookTrait* then executing $object->test() will also look for globally-registered method inside the application:

```
$object->getApp()->addGlobalMethod('test', function ($app, $o, $args) {
    echo 'hello, ' . $args[0];
});

$object->test('world');
```

Of course calling test() on the other object afterwards will trigger same global method.

If you attempt to register same method multiple times you will receive an exception.

Dynamic Method Arguments

When calling dynamic method first argument which is passed to the method will be object itself. Dynamic method will also receive all arguments which are given when you call this dynamic method:

```
$m->addMethod('sum', function ($m, $a, $b) {
    return $a + $b;
});
echo $m->sum(3, 5); // 8
```

Properties

None

Methods

`DynamicMethodTrait::tryCall`(*$method*, *$arguments*)
Tries to call dynamic method, but doesn't throw exception if it is not possible.

`DynamicMethodTrait::addMethod`(*$name*, *$closure*)
Add new method for this object. See examples above.

`DynamicMethodTrait::hasMethod`(*$name*)
Returns true if object has specified method (either native or dynamic). Returns true also if specified methods is defined globally.

`DynamicMethodTrait::removeMethod`(*$name*)
Remove dynamically registered method.

DynamicMethodTrait::**addGlobalMethod**(*$name*, *$closure*)
Registers a globally-recognized method for all objects.

DynamicMethodTrait::**hasGlobalMethod**(*$name*)
Return true if such global method exists.

CHAPTER 3

Modelable Objects

[Agile Data](https://github.com/atk4/data) features a modern implementation for object modeling. You may extend [Model](http://agile-data.readthedocs.io/en/develop/model.html) class to define a business object, such as - Shopping-Bag:

```
class ShoppingBag extends \Atk4\Data\Model
{
    public $table = 'shopping_bag';

    protected function init(): void
    {
        parent::init();

        $this->hasOne('user_id', new User());
        $this->hasMany('Items', new Item())
            ->addField('total_price', ['aggregate' => 'sum', 'field' => 'price']);
    }
}
```

Such a model handles references to the user and items, is aware of storage details, but it is a non-visual object. Because Model does not know if you will need HTML or RestAPI to access it, it does not implement any visualization.

[Agile UI](https://github.com/atk4/ui) implements UI Components that can be binded together with a model and will render HTML in a way that User can understand and interact with.

To associate UI Component with Data Model a *setModel()* is used. But it's not only the UI Components that can be associated with the model. In fact "Authentication" controller can be associated with User model and RestAPI endpoints can be associated with models. This is why *setModel()* is implemented by a PHP Trait.

ModelableTrait allows you to associate your object with a Model:

```
$form->setModel('Order');

// or

$grid->setModel($order->ref('Items'), ['name', 'qty', 'price']);
```

3.1 Modelable Trait

trait ModelableTrait

3.1.1 Introduction

not yet implemented

3.1.2 Properties

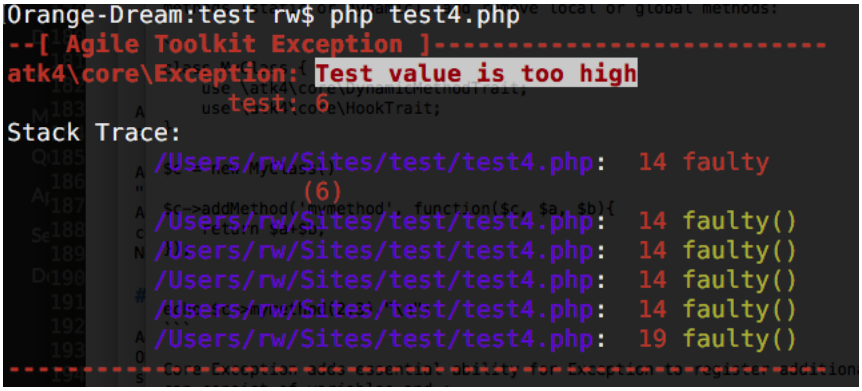
3.1.3 Methods

CHAPTER 4

Exceptions

Exceptions in most programming languages are cryptic and confusing, about 5% of the trace-back code is useful and it's obscured in most unique ways.

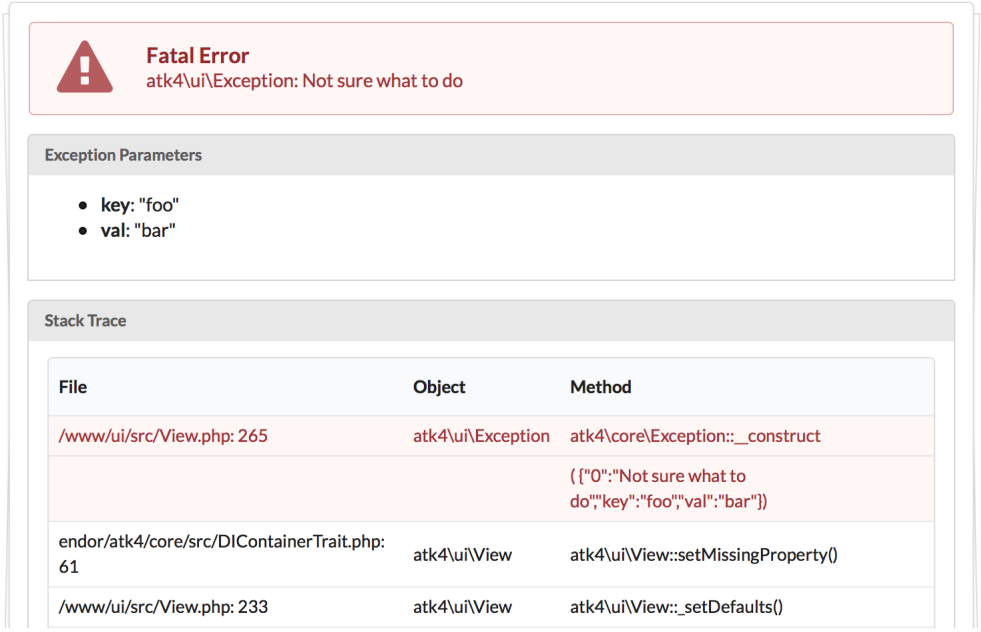
We've got an excellent solution by implementing exceptions our way. Simply look at how beautiful and clean they look:



The same can be said about web output:

Agile Core implements `Exception` class which offers many benefits compared to standard PHP exceptions:

- Pass additional information *(new Exception('Bad argument'))->addMoreInfo('arg', $arg')*
- Visualize in ASCII or HTML
- Better means of localization



Fatal Error
atk4\ui\Exception: Not sure what to do

Exception Parameters

- **key**: "foo"
- **val**: "bar"

Stack Trace

File	Object	Method
/www/ui/src/View.php: 265	atk4\ui\Exception	atk4\core\Exception::__construct
		({"0":"Not sure what to do","key":"foo","val":"bar"})
endor/atk4/core/src/DIContainerTrait.php: 61	atk4\ui\View	atk4\ui\View::setMissingProperty()
/www/ui/src/View.php: 233	atk4\ui\View	atk4\ui\View::_setDefaults()

CHAPTER 5

Others

5.1 Debug Trait

trait DebugTrait

5.1.1 Introduction

Agile Core implements ability for application to implement "debug", "info" and "messages". The general idea of them is that they can be generated in the depths of the code, but the application will receive and process this information based on the defined settings.

Sample scenario would be if some of the components tries to perform operation which fails and it is willing to pass information about this failure to the app. This is not as extreme as exception, but still, user needs to be able to find this information eventually.

Compatibility with PSR-3

Loggers as implemented by PSR-3 define message routing with various LogLevels, but it's intended for logging only. The Debug Trait covers a wider context as described below:

Debug

The design goal of Debug is to be able to display contextual debug information only when it's manually enabled. For instance, if you are having problem with user authentication, you should enable `$auth->debug()`. On other hand - if you wish to see persistence-related debug info, then `$db->debug()` will enable that.

Information logged through debug like this on any object that implements DebugTrait:

```
$this->debug('Things are bad');
$this->debug('User {user} created', ['user' => $user]);
```

The Application itself can use DebugTrait too and normally should do, making it possible to use `$this->getApp()->debug()`.

Various objects may implement DebugTrait and also invoke $this->debug(), but in most cases this will simply be ignored right away unless you manually enable debugging for the object:

```
$obj1->debug(); // enable debugging
$obj1->debug(false); // disable debugging
$obj1->debug(true); // also enables debugging

$obj1->debug('test1'); // will go to logger
$obj2->debug('test2'); // will not go to logger because debug is not enabled for this␣
↪object
```

Executing debug will look for `$this->getApp()` link and if the application implements `Psr\Log\LoggerInterface`, then `$this->getApp()->log()` will be called using LogLevel DEBUG.

Log

Log method will log message every time. DebugTrait implements the `log()` method which will either display information on the STDOUT (if `$this->getApp()` does not exist or does not implement PSR-3)

debugTraceChange

This method can help you find situations when a certain code is called multiple times and when it shouldn't. When called first time it will remember "trace" which is used to arrive at this point. Second time it will compare with the previous and will tell you where trace has diverged.

This method is pretty valuable when you try to find why certain areas of the code have executed multiple times.

5.1.2 Properties

5.1.3 Methods

5.2 Writing ATK Docs

New users of Agile Toolkit rely on documentation. To make it easier for the maintainers to update documentation - each component of ATK framework comes with a nice documentation builder.

5.2.1 Writing ATK Documentation

Open file "docs/index.rst" in your editor. Most editors will support "reSTructured Text" through add-on. The support is not perfect, but it works.

If you are updating a feature - find a corresponding ".rst" file. Your editor may be able to show you a preview. Modify or extend documentation as needed.

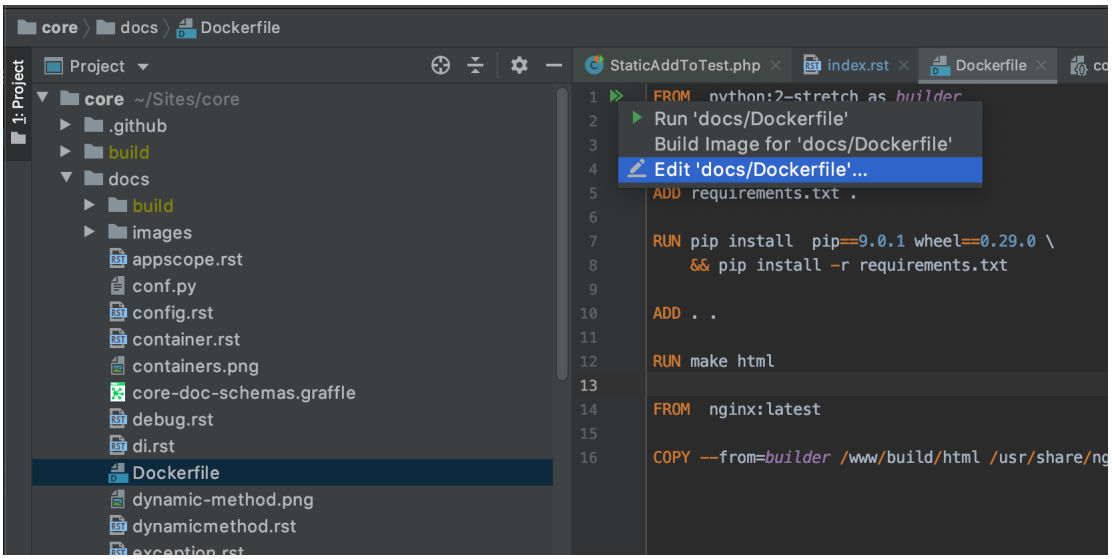
See also: http://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html

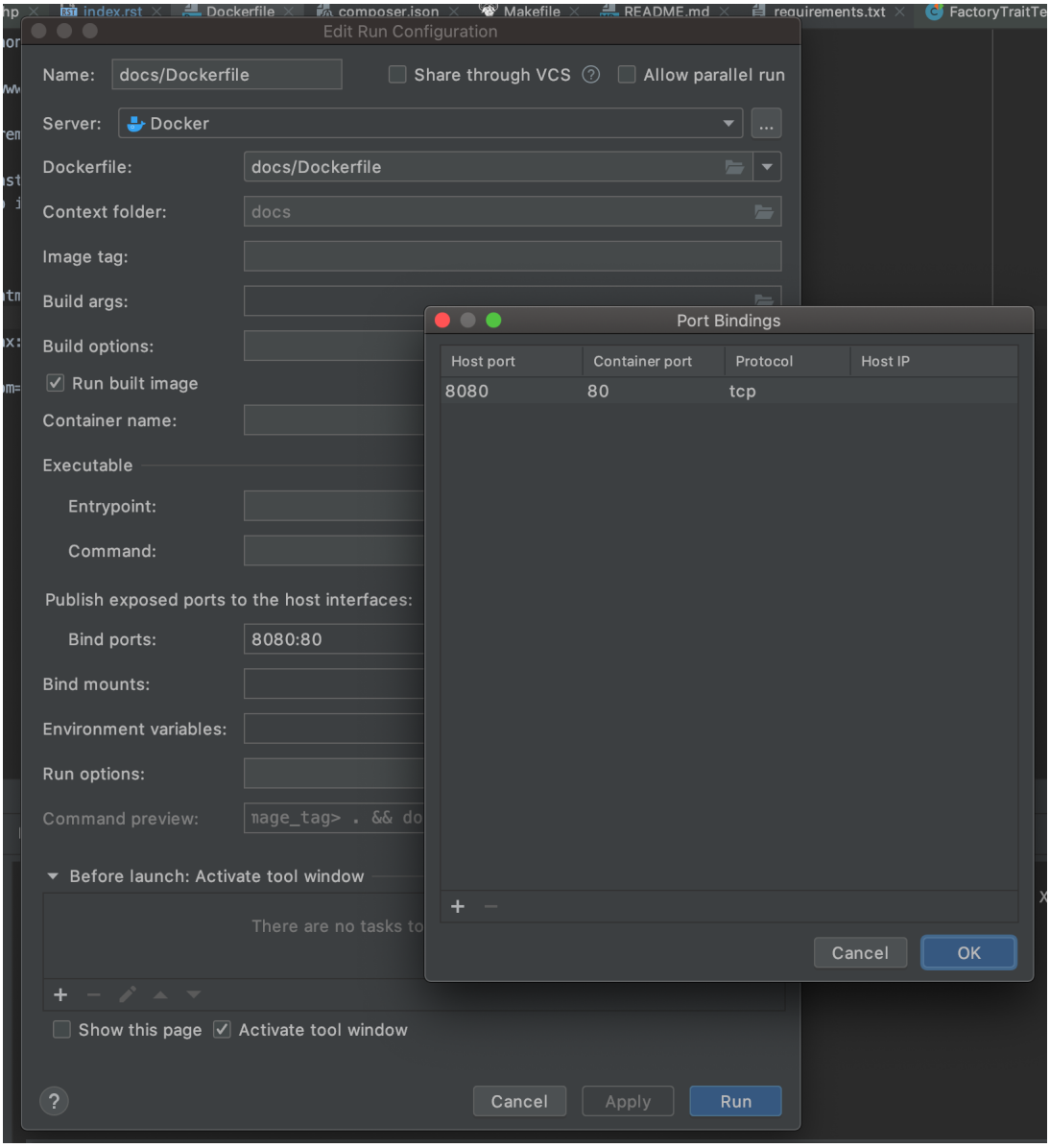
5.2.2 Building and Testing Documentation

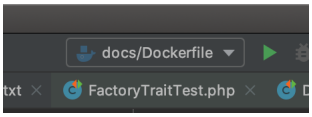
Make sure you have "Docker" installed, follow simple instructions in "docs/README.md".

Integrating PhpStorm

You can integrate PhpStorm build process like this:




Edit Run Configuration
Name: docs/Dockerfile
Share through VCS
Allow parallel run
Server: Docker
Dockerfile: docs/Dockerfile
Context folder: docs
Image tag:
Build args:
Build options:
Run built image
Container name:
Executable
Entrypoint:
Command:
Publish exposed ports to the host interfaces:
Bind ports: 8080:80
Bind mounts:
Environment variables:
Run options:
Command preview:
Before launch: Activate tool window
There are no tasks to
Show this page
Activate tool window
Cancel
Apply
Run
Port Bindings
Host port
Container port
Protocol
Host IP
8080
80
tcp
Cancel
OK


docs/Dockerfile
FactoryTraitTest.php

Index

Symbols

_addContainer() (*ContainerTrait method*), 7
_addIntoCollection() (*CollectionTrait method*), 6
_getFromCollection() (*CollectionTrait method*), 6
_hasInCollection() (*CollectionTrait method*), 6
_removeFromCollection() (*CollectionTrait method*), 6
_shorten() (*ContainerTrait method*), 8
_shortenMl() (*CollectionTrait method*), 6
_uniqueElementName() (*ContainerTrait method*), 8

A

add() (*ContainerTrait method*), 7
addGlobalMethod() (*DynamicMethodTrait method*), 26
addMethod() (*DynamicMethodTrait method*), 26
app (*AppScopeTrait property*), 18
AppScopeTrait (*trait*), 18

B

breakHook() (*HookTrait method*), 24

C

CollectionTrait (*trait*), 5
ContainerTrait (*trait*), 6

D

DebugTrait (*trait*), 33
destroy() (*TrackableTrait method*), 9
DiContainerTrait (*trait*), 19
DynamicMethodTrait (*trait*), 25

E

elements (*ContainerTrait property*), 7

F

Factory (*class*), 10
factory() (*Factory method*), 12

G

getDesiredName() (*TrackableTrait method*), 9
getElement() (*ContainerTrait method*), 8

H

hasElement() (*ContainerTrait method*), 8
hasGlobalMethod() (*DynamicMethodTrait method*), 27
hasMethod() (*DynamicMethodTrait method*), 26
hook() (*HookTrait method*), 23
hookHasCallbacks() (*HookTrait method*), 25
HookTrait (*trait*), 22

I

init() (*InitializerTrait method*), 10
InitializerTrait (*trait*), 9

M

maxNameLength (*AppScopeTrait property*), 18
mergeSeeds() (*Factory method*), 15
ModelableTrait (*trait*), 30

N

name (*NameTrait property*), 5
NameTrait (*trait*), 5

O

onHook() (*HookTrait method*), 22
owner (*TrackableTrait property*), 8

R

removeElement() (*ContainerTrait method*), 8
removeMethod() (*DynamicMethodTrait method*), 26

S

shortName (*TrackableTrait property*), 8

T

TrackableTrait (*trait*), 8
tryCall() (*DynamicMethodTrait method*), 26

U

uniqueNameHashes (*AppScopeTrait property*), 18