Finite Documentation

Release 1.1

Yohan Giarelli

April 18, 2016

Contents

CHAPTER 1

Use with Symfony

1.1 Installation

```
$ composer require yohang/finite
```

1.1.1 Register the bundle

Register the bundle in your AppKernel:

```
<?php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new\Finite\Bundle\FiniteBundle\FiniteFiniteBundle(),
        // ...
    );
}
```

1.2 Defining your stateful class

As we want to track the state of an object (or the multiples states, but this example will focus on object with single state-graph), create your class if it doesn't already exists. This class has simply to implements `Finite\StatefulInterface`.

This part is covered in *Define your object*.

1.3 Configuration

```
finite_finite:

    document_workflow:
        class: MyDocument  # You class FQCN
        states:
```

```
            draft:    { type: initial, properties: { visible: false } }
            proposed: { type: normal,  properties: { visible: false } }
            accepted: { type: final,   properties: { visible: true  } }
            refused:  { type: final,   properties: { visible: false } }

        transitions:
            propose:  { from: draft,    to: proposed }
            accept:   { from: proposed, to: accepted }
            refuse:   { from: proposed, to: refused  }
```

At this point, your graph is ready and you can start using your workflow on your object.

1.4 Controller / Service usage

Finite define several services into the Symfony DIC. The easier to use is `finite.context`.

1.4.1 Example

```
<?php

$context = $this->get('finite.context');
$context->getState($document); // return "draft", or... the current state if different
$context->getProperties($document); // array:1 [ 'visible' => false ]
$context->getTransitions($document); // array:2 [ 0 => "propose", 1 => "refuse" ]
$context->hasProperty($document, 'visible'); // true
$context->getFactory(); // Return an instance of FiniteFactory, used to instantiate the state machine
$context->getStateMachine($document); // Returns a initialized StateMachine instance for $document


// Throw a 404 if document isn't visible
if (!$this->get('finite.context')->getProperties($document)['visible']) {
    throw $this->createNotFoundException(
        sprintf('The document "%s" is not in a visible state.', $document->getName())
    );
}
```

1.5 Twig usage

Although the Twig Extension is not Symfony-specific at all, when using the Symfony Bundle, Finite functions are automatically accessible in your templates.

```
{{ dump(finite_state(document)) }} {# "draft" #}
{{ dump(finite_transitions(document)) }} {# array:2 [ 0 => "propose", 1 => "refuse" ] #}
{{ dump(finite_properties(document)) }} {# array:1 [ 'visible' => false ] #}
{{ dump(finite_has(document, 'visible')) }} {# true #}
{{ dump(finite_can(document, 'accept')) }} {# true #}


{# Display reachable transitions #}
{% for transition in finite_transitions(document) %}
    <a href="{{ path('document_apply_transition', {transition: transition}) }}">
        {{ transition }}
```

```
    </a>
{% endfor %}


{# Display an action if available #}
{% if finite_can(document, 'accept') %}
    <button type="submit" name="accept">
        Accept this document
    </button>
{% endif %}
```

1.5.1 Example

1.6 Using callbacks

The state machine is built around a a very flexible and powerful events / callbacks system. Events dispatched with the EventDispatcher and works as the Symfony kernel events.

1.6.1 Events

finite.set_initial_state: This event is fired when initializing a state machine with an object which does not have a defined state. It allows you to manage the default initial state of your object.

finite.initialize: Fired when the StateMachine is initialized for an object (event if the current object state is known)

finite.test_transition: Fired when testing if a transition can be applied, when you call `StateMachine#can` or `StateMachine#apply`. This event is an instance of `Finite\Event\TransitionEvent` and can be rejected, which leads to a non-appliable transition. This is one of the most useful event, as it allows you to introduce business code for allowing / rejecting transitions

finite.test_transition.[transition_name]: Same as `finite.test_transition` but with the concerned transition in the event name.

finite.test_transition.[graph].[transition_name]: Same as `finite.test_transition` but with the concerned graph and transition in the event name.

finite.pre_transition: Fired before applying a transition. You can use it to prepare your object for a transition.

finite.pre_transition.[transition_name]: Same as `finite.pre_transition` but with the concerned transition in the event name.

finite.pre_transition.[graph].[transition_name]: Same as `finite.pre_transition` but with the concerned graph and transition in the event name.

finite.post_transition: Fired after applying a transition. You can use it to execute the business code you have to execute when a transition is applied.

finite.post_transition.[post_transition]: Same as `finite.post_transition` but with the concerned transition in the event name.

finite.post_transition.[graph].[transition_name]: Same as `finite.post_transition` but with the concerned graph and transition in the event name.

1.6.2 Callbacks

Callbacks are a simplified mechanism allowing you to plug your domain services on the finite events. You can see it as a way to listen to events without defining a listener class that just redirects the events to your services.

Using YAML configuration

```
finite_finite:

    document_workflow:
        class: MyDocument
        states:
            # ...
        transitions:
            # ...

        callbacks:
            before:
                # Will call the `sendPublicationMail` method of `@app.mailer.document` service
                # When the `accept` transition is applied
                send_publication_mail:
                    disabled: false # default value
                    on: accept
                    do: [ @app.mailer.document, 'sendPublicationMail' ]

                # Will call the `sendNotAnymoreProposedEmail` method of `@app.mailer.document` servic
                # When any transition from the `proposed` state is applied.
                # This condition can be negated by prefixing a `-` before the state name
                # And the same exists for the destination transitions (with `to: `)
                send_publication_mail:
                    disabled: false # default value
                    from: ['proposed']
                    do: [ @app.mailer.document, 'sendNotAnymoreProposedEmail' ]
```

1.7 Configuration reference

```
finite_finite:

    # Prototype
    name: # internal name of your graph, not used
        class:                ~           # Required, FQCN of your class
        graph:                default     # Name of your graph, keep default if using a single graph
        property_path:        finiteState # The property of your class used to store the state


        states:
            # Prototype
            name:            # Required, Name of your state
                type: normal # State type, in "initial", "normal", "final"
                properties:  # Properties array.
                    # Prototype
                    name:                ~


        transitions:
```

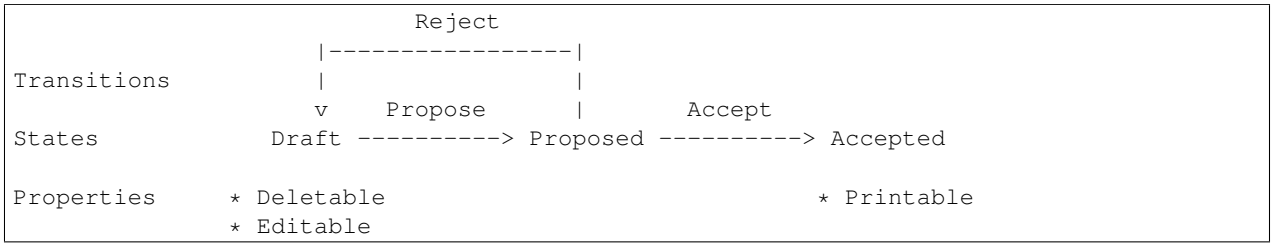
```
            # Prototype
            name:            # Required, Name of your transition
                from: []     # Required, states the transition can come from
                to:   ~      # Required, state where the transition go
                properties: # Properties array.
                    # Prototype
                    name:                ~

        callbacks:

            before: # Pre-transition callbacks
                # Prototype
                name:
                    do:       ~ # Required. The callback.
                    on:       ~ # On which transition to trigger the callback. Default null
                    from:     ~ # From which states are we triggering the callback. Default null
                    to:       ~ # To which states are we triggering the callback. Default null
                    disabled: false

            after: # Post-transition callbacks
                # Prototype
                name:
                    on:                  ~
                    do:                  ~
                    from:                ~
                    to:                  ~
                    disabled:            false
```

CHAPTER 2

Basic graph

2.1 Goal

In this example, we'll see a basic Document workflow, following this graph :



2.2 Implement the document class

```php
<?php

class Document implements Finite\StatefulInterface
{
    private $state;

    public function getFiniteState()
    {
        return $this->state;
    }

    public function setFiniteState($state)
    {
        $this->state = $state;
    }
}
```

2.3 Configure your graph

```php
<?php

$loader = new Finite\Loader\ArrayLoader([
```

```
    'class'   => 'Document',
    'states'  => [
        'draft' => [
            'type'       => Finite\State\StateInterface::TYPE_INITIAL,
            'properties' => ['deletable' => true, 'editable' => true],
        ],
        'proposed' => [
            'type'       => Finite\State\StateInterface::TYPE_NORMAL,
            'properties' => [],
        ],
        'accepted' => [
            'type'       => Finite\State\StateInterface::TYPE_FINAL,
            'properties' => ['printable' => true],
        ]
    ],
    'transitions' => [
        'propose' => ['from' => ['draft'], 'to' => 'proposed'],
        'accept'  => ['from' => ['proposed'], 'to' => 'accepted'],
        'reject'  => ['from' => ['proposed'], 'to' => 'draft'],
    ],
]);

$document = new Document;
$stateMachine = new Finite\StateMachine\StateMachine($document);
$loader->load($stateMachine);
$stateMachine->initialize();
```

At this point, your Workflow / State graph is fully accessible to the state machine, and you can start to work with your workflow.

2.4 Working with workflow

2.4.1 Current state

```
<?php
// Get the name of the current state
$stateMachine->getCurrentState()->getName();
// string(5) "draft"

// List the currently accessible properties, and their values
$stateMachine->getCurrentState()->getProperties();
// array(2) {
//     'deletable' => bool(true)
//     'editable' => bool(true)
// }

// Checks if "deletable" property is defined
$stateMachine->getCurrentState()->has('deletable');
// bool(true)

// Checks if "printable" property is defined
$stateMachine->getCurrentState()->has('printable');
// bool(false)
```

2.4.2 Available transitions

```php
<?php

// Retrieve available transitions
var_dump($stateMachine->getCurrentState()->getTransitions());
// array(1) {
//      [0] => string(7) "propose"
// }

// Check if we can apply the "propose" transition
var_dump($stateMachine->getCurrentState()->can('propose'));
// bool(true)

// Check if we can apply the "accept" transition
var_dump($stateMachine->getCurrentState()->can('accept'));
// bool(false)
```

2.4.3 Apply transition

```php
<?php

// Trying to apply a not accessible transition
try {
    $stateMachine->apply('accept');
} catch (\Finite\Exception\StateException $e) {
    echo $e->getMessage();
}
// The "accept" transition can not be applied to the "draft" state.

// Applying a transition
$stateMachine->apply('propose');
$stateMachine->getCurrentState()->getName();
// string(7) "proposed"
$document->getFiniteState();
// string(7) "proposed"
```

CHAPTER 3

Events / Callbacks

3.1 Overview

Finite use the Symfony EventDispatcher component to notify each actions done by the State Machine.

You can use the event system directly with callbacks in your configuration, or by attaching listeners to the event dispatcher.

3.2 Implement your document class and define your workflow

See Basic graph.

3.3 Use callbacks

Callbacks can be defined directly in your State Machine configuration. The can be called before or after the transition apply, and their definition use the following pattern :

```
<?php

$definition = [
    'from' => [],  // a string or an array of string that represent the initial states that trigger t
    'to'   => [],  // a string or an array of string that represent the target states that trigger th
    'on'   => [],  // a string or an array of string that represent the transition names that trigger
    'do'   => function($object, Finite\Event\TransitionEvent $e) {
        // The callback
    }
];
```

from and *to* parameters can be any state names. Prefix by - to process an exclusion. By default, callbacks matchs all the events.

3.3.1 Example :

```
<?php

[
    'from' => ['all', '-proposed'],
```

```
    'do'    => function($object, Finite\Event\TransitionEvent $e) {
        // callback code
    }
];
```

Will match any transition that don't begin on the *proposed* state.

3.3.2 Full example :

```
<?php

$loader       = new Finite\Loader\ArrayLoader([
    'class'       => 'Document',
    'states'      => [
        'draft'    => [
            'type'       => Finite\State\StateInterface::TYPE_INITIAL,
            'properties' => ['deletable' => true, 'editable' => true],
        ],
        'proposed' => [
            'type'       => Finite\State\StateInterface::TYPE_NORMAL,
            'properties' => [],
        ],
        'accepted' => [
            'type'       => Finite\State\StateInterface::TYPE_FINAL,
            'properties' => ['printable' => true],
        ]
    ],
    'transitions' => [
        'propose' => ['from' => ['draft'], 'to' => 'proposed', 'properties' => ['foo' => 'bar']],
        'accept'  => ['from' => ['proposed'], 'to' => 'accepted'],
        'reject'  => ['from' => ['proposed'], 'to' => 'draft'],
    ],
    'callbacks' => [
        'before' => [
            [
                'from' => '-proposed',
                'do'   => function(\Finite\Event\TransitionEvent $e) {
                    echo 'Applying transition '.$e->getTransition()->getName(), "\n";
                    if ($e->has('foo')) {
                        echo "Parameter \"foo\" is defined\n";
                    }
                }
            ],
            [
                'from' => 'proposed',
                'do'   => function() {
                    echo 'Applying transition from proposed state', "\n";
                }
            ]
        ],
        'after' => [
            [
                'to' => ['accepted'], 'do' => [$document, 'display']
            ]
        ]
    ]
]);
```

```
$stateMachine->apply('propose');
// => "Applying transition propose"
// => "Parameter "foo" is defined"

$stateMachine->apply('reject');
// => "Applying transition from proposed state"

$stateMachine->apply('propose');
// => "Applying transition propose"
// => "Parameter "foo" is defined"

$stateMachine->apply('accept');
// => "Applying transition from proposed state"
// => "Hello, I'm a document and I'm currently at the accepted state."
```

3.4 Use event dispatcher

If you prefer, you can use directly the event dispatcher.

Here is the available events :

```
finite.initialize      => Dispatched at State Machine initialization
finite.test_transition => Dispatched when testing if a transition can be applied
finite.pre_transition  => Dispatched before a transition
finite.post_transition => Dispatched after a transition

finite.test_transition.{transitionName} => Dispatched when testing if a specific transition can be ap
finite.pre_transition.{transitionName}  => Dispatched before a specific transition
finite.post_transition.{transitionName} => Dispatched after a specific transition

finite.test_transition.{graph}.{transitionName} => Dispatched when testing if a specific transition
finite.pre_transition.{graph}.{transitionName}  => Dispatched before a specific transition in a speci
finite.post_transition.{graph}.{transitionName} => Dispatched after a specific transition in a specif
```

3.4.1 Example :

```
<?php

$stateMachine->getDispatcher()->addListener('finite.pre_transition', function(\Finite\Event\Transitio
    echo 'This is a pre transition', "\n";
});
$stateMachine->apply('propose');
// => "This is a pre transition"
```

3.4.2 Example testing transitions:

```
<?php

$stateMachine->getDispatcher()->addListener('finite.test_transition', function(\Finite\Event\Transiti
    $e->reject();
});
```

```
try {
    $stateMachine->apply('propose');
}
catch (Finite\StateMachine\Exception\StateException $e) {
    echo 'The transition did not apply', "\n";
}

// => "The transition did not apply"
```

CHAPTER 4

Transitions properties

As the second argument, *StateMachine#apply* and *StateMachine#test* will accept an array of properties to be passed to the dispatched event, and accessible by the listeners.

Default properties can be defined with your state graph.

```
$stateManager->apply('some_event', array('something' => $value));
```

In your listeners you just have to call `$event->getProperties()` to access the passed data.

```
<?php

namespace My\AwesomeBundle\EventListener;

use Finite\Event\TransitionEvent;

class TransitionListener
{
    /**
     * @param TransitionEvent $event
     */
    public function someEvent(TransitionEvent $event)
    {
        $entity = $event->getStateMachine()->getObject();
        $params = $event->getProperties();

        $entity->setSomething($params['something']);
    }
}
```

4.1 Default properties

```
'transitions' => array(
    'finish'    => array(
        'from' => array('middle'),
        'to'   => 'end',
        'properties' => array('foo' => 'bar'),
        'configure_properties' => function (OptionsResolver $resolver) {
            $resolver->setRequired('baz');
        }
    )
)
```

CHAPTER 5

A PHP Finite State Machine

Finite is a state machine library that gives you ability to manage the state of a PHP object through a graph of states and transitions.

CHAPTER 6

Overview

6.1 Define your workflow / state graph

```php
<?php

$document     = new MyDocument;
$stateMachine = new Finite\StateMachine\StateMachine;
$loader       = new Finite\Loader\ArrayLoader([
    'class'  => 'MyDocument',
    'states' => [
        'draft'    => ['type' => 'initial', 'properties' => []],
        'proposed' => ['type' => 'normal',  'properties' => []],
        'accepted' => ['type' => 'final',   'properties' => []],
        'refused'  => ['type' => 'final',   'properties' => []],
    ],
    'transitions' => [
        'propose' => ['from' => ['draft'],    'to' => 'proposed'],
        'accept'  => ['from' => ['proposed'], 'to' => 'accepted'],
        'refuse'  => ['from' => ['proposed'], 'to' => 'refused'],
    ]
]);

$loader->load($stateMachine);
$stateMachine->setObject($document);
$stateMachine->initialize();
```

6.2 Define your object

```php
<?php

class MyDocument implements Finite\StatefulInterface
{
    private $state;
    public function getFiniteState()
    {
        return $this->state;
    }
    public function setFiniteState($state)
    {
        $this->state = $state;
```

```
    }
}
```

6.3 Work with states & transitions

```php
<?php

echo $stateMachine->getCurrentState();
// => "draft"

var_dump($stateMachine->can('accept'));
// => bool(false)

var_dump($stateMachine->can('propose'));
// => bool(true)

$stateMachine->apply('propose');
echo $stateMachine->getCurrentState();
// => "proposed"
```

CHAPTER 7

Contribute

Contributions are welcome !

Finite follows PSR-2 code, and accept pull-requests on the GitHub repository.

If you're a beginner, you will find some guidelines about code contributions at Symfony.