docs.zendframework.com

Using the EventManager - tutorials

1. Docs »
2. Component Tutorials »
3. Using the EventManager

This tutorial explores the features of zend-eventmanager in-depth.

Terminology

- An **Event** is a named action.
- A **Listener** is any PHP callback that reacts to an *event*.
- An **EventManager** *aggregates* listeners for one or more named events, and *triggers* events.

Typically, an *event* will be modeled as an object, containing metadata surrounding when and how it was triggered, including the event name, what object triggered the event (the "target"), and what parameters were provided. Events are *named*, which allows a single *listener* to branch logic based on the event.

Getting started

The minimal things necessary to start using events are:

- An `EventManager` instance
- One or more listeners on one or more events
- A call to `trigger()` an event

A basic example looks something like this:

```
use Zend\EventManager\EventManager;
```

```
$events = new EventManager();
$events->attach('do', function ($e) {
    $event = $e->getName();
    $params = $e->getParams();
    printf(
        'Handled event "%s", with parameters %s',
        $event,
        json_encode($params)
    );
});

$params = ['foo' => 'bar', 'baz' => 'bat'];
$events->trigger('do', null, $params);
```

The above will result in the following:

```
Handled event "do", with parameters {"foo":"bar","baz":"bat"}
```

> **Closures are not required**
>
> Throughout this tutorial, we use closures as listeners. However, any valid PHP callback can be attached as a listeners: PHP function names, static class methods, object instance methods, functors, or closures. We use closures within this post for illustration only.

Event instances

`trigger()` is useful as it will create a `Zend\EventManager\Event` instance for you. You may want to create such an instance manually; for instance, you may want to re-use the same event instance to trigger multiple events, or you may want to use a custom instance.

`Zend\EventManager\Event`, which is the shipped event type and the one used by the `EventManager` by default has a constructor that accepts the same three arguments passed to `trigger()`:

```
use Zend\EventManager\Event;

$event = new Event('do', null, $params);
```

When you have an instance available, you will use a different `EventManager` method to trigger the event:

`triggerEvent()` . As an example:

```
$events->triggerEvent($event);
```

Event targets

If you were paying attention to the first example, you will have noted the `null` second argument both when calling `trigger()` as well as creating an `Event` instance. Why is it there?

Typically, you will compose an `EventManager` within a class, to allow triggering actions within methods. The middle argument to `trigger()` is the "target", and in the case described, would be the current object instance. This gives event listeners access to the calling object, which can often be useful.

```
 use Zend\EventManager\EventManager;
use Zend\EventManager\EventManagerAwareInterface;
use Zend\EventManager\EventManagerInterface;

class Example implements EventManagerAwareInterface
{
    protected $events;

    public function setEventManager(EventManagerInterface $events)
    {
        $events->setIdentifiers([
            __CLASS__,
            get_class($this),
        ]);
        $this->events = $events;
    }

    public function getEventManager()
    {
        if (! $this->events) {
            $this->setEventManager(new EventManager());
        }
        return $this->events;
    }

    public function doIt($foo, $baz)
    {
        $params = compact('foo', 'baz');
```

```
        $this->getEventManager()->trigger(__FUNCTION__, $this, $params);
    }

}

$example = new Example();

$example->getEventManager()->attach('doIt', function($e) {
    $event  = $e->getName();
    $target = get_class($e->getTarget());
    $params = $e->getParams();
    printf(
        'Handled event "%s" on target "%s", with parameters %s',
        $event,
        $target,
        json_encode($params)
    );
});

$example->doIt('bar', 'bat');
```

The above is basically the same as the first example. The main difference is that we're now using that middle argument in order to pass the target, the instance of `Example`, on to the listeners. Our listener is now retrieving that (`$e->getTarget()`), and doing something with it.

If you're reading this critically, you should have a new question: What is the call to `setIdentifiers()` for?

One aspect that the `EventManager` implementation provides is an ability to compose a `SharedEventManagerInterface` implementation.

`Zend\EventManager\SharedEventManagerInterface` describes an object that aggregates listeners for events attached to objects with specific *identifiers*. It does not trigger events itself. Instead, an `EventManager` instance that composes a `SharedEventManager` will query the `SharedEventManager` for listeners on identifiers it's interested in, and trigger those listeners as well.

How does this work, exactly?

Consider the following:

```
 use Zend\EventManager\SharedEventManager;

$sharedEvents = new SharedEventManager();
$sharedEvents->attach('Example', 'do', function ($e) {
    $event  = $e->getName();
    $target = get_class($e->getTarget());
    $params = $e->getParams();
    printf(
        'Handled event "%s" on target "%s", with parameters %s',
        $event,
        $target,
        json_encode($params)
    );
});
```

This looks almost identical to the previous example; the key difference is that there is an additional argument at the *start* of the list, `'Example'`. This code is saying, "Listen to the 'do' event of the 'Example' target, and, when notified, execute this callback."

This is where the `setIdentifiers()` argument of `EventManager` comes into play. The method allows passing a string, or an array of strings, defining the name or names of the context or targets the given instance will be interested in. If an array is given, then any listener on any of the targets given will be notified.

So, getting back to our example, let's assume that the above shared listener is registered, and also that the `Example` class is defined as above. We can then execute the following:

```
 $example = new Example();
$example->getEventManager()->setSharedManager($sharedEvents);
$example->do('bar', 'bat');
```

and expect the following to be `echo`'d:

```
 Handled event "do" on target "Example", with parameters
{"foo":"bar","baz":"bat"}
```

Now, let's say we extended `Example` as follows:

```
 class SubExample extends Example
{
```

```
}
```

One interesting aspect of our `setEventManager()` method is that we defined it to listen both on `__CLASS__` and `get_class($this)`. This means that calling `do()` on our `SubExample` class would also trigger the shared listener! It also means that, if desired, we could attach to specifically `SubExample`, and listeners attached to only the `Example` target would not be triggered.

Finally, the names used as contexts or targets need not be class names; they can be some name that only has meaning in your application if desired. As an example, you could have a set of classes that respond to "log" or "cache" — and listeners on these would be notified by any of them.

> **Use class names as identifiers**
>
> We recommend using class names, interface names, and/or abstract class names for identifiers. This makes determining what events are available easier, as well as finding which listeners might be attaching to those events. Interfaces make a particularly good use case, as they allow attaching to a group of related classes a single operation.

At any point, if you do not want to notify shared listeners, pass a `null` value to `setSharedManager()`:

```
$events->setSharedManager(null);
```

and they will be ignored. If at any point, you want to enable them again, pass the `SharedEventManager` instance:

```
$events->setSharedManager($sharedEvents);
```

Wildcards

So far, with both a normal `EventManager` instance and with the `SharedEventManager` instance, we've seen the usage of string event and string target names to which we want to attach. What if you want to attach a listener to multiple events or targets?

The answer is to supply an array of events or targets, or a wildcard, `*`.

Consider the following examples:

```
$events->attach(
    ['foo', 'bar', 'baz'],
    $listener
);


$events->attach(
    '*',
    $listener
);


$sharedEvents->attach(
    ['Foo', 'Bar', 'Baz'],
    'doSomething',
    $listener
);


$sharedEvents->attach(
    '*',
    'doSomething',
    $listener
);


$sharedEvents->attach(
    ['Foo', 'Bar', 'Baz'],
    ['foo', 'bar', 'baz'],
    $listener
);


$sharedEvents->attach(
    ['Foo', 'Bar', 'Baz'],
    '*',
    $listener
);


$sharedEvents->attach(
    '*',
```

```
    ['foo', 'bar', 'baz'],
    $listener
);


$sharedEvents->attach(
    '*',
    '*',
    $listener
);
```

The ability to specify multiple targets and/or events when attaching can slim down your code immensely.

> **Wildcards can cause problems**
>
> Wildcards, while they simplify listener attachment, can cause some problems. First, the listener must either be able to accept any incoming event, or it must have logic to branch based on the type of event, the target, or the event parameters. This can quickly become difficult to manage.
>
> Additionally, there are performance considerations. Each time an event is triggered, it loops through all attached listeners; if your listener cannot actually handle the event, but was attached as a wildcard listener, you're introducing needless cycles both in aggregating the listeners to trigger, and by handling the event itself.
>
> We recommend being specific about what you attach a listener to, in order to prevent these problems.

Listener aggregates

Another approach to listening to multiple events is via a concept of listener aggregates, represented by `Zend\EventManager\ListenerAggregateInterface`. Via this approach, a single class can listen to multiple events, attaching one or more instance methods as listeners.

This interface defines two methods, `attach(EventManagerInterface $events)` and `detach(EventManagerInterface $events)`. You pass an `EventManager` instance to one and/or the other, and then it's up to the implementing class to determine what to do.

The trait `Zend\EventManager\ListenerAggregateTrait` defines a `$listeners` property and common logic for detaching an aggregate's listeners. We'll use that to demonstrate creating an aggregate logging listener:

```
 use Zend\EventManager\EventInterface;
use Zend\EventManager\EventManagerInterface;
use Zend\EventManager\ListenerAggregateInterface;
use Zend\EventManager\ListenerAggregateTrait;
use Zend\Log\Logger;

class LogEvents implements ListenerAggregateInterface
{
    use ListenerAggregateTrait;

    private $log;

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

    public function attach(EventManagerInterface $events)
    {
        $this->listeners[] = $events->attach('do', [$this, 'log']);
        $this->listeners[] = $events->attach('doSomethingElse', [$this,
'log']);
    }

    public function log(EventInterface $e)
    {
        $event  = $e->getName();
        $params = $e->getParams();
        $this->log->info(sprintf('%s: %s', $event,
json_encode($params)));
    }
}
```

Attach the aggregate by passing it an event manager instance:

```
 $logListener = new LogEvents($logger);
$logListener->attach($events);
```

Any events the aggregate attaches to will then be notified when triggered.

Why bother? For a couple of reasons:

- Aggregates allow you to have stateful listeners. The above example demonstrates this via the composition of the logger; another example would be tracking configuration options.

- Aggregates make detaching listeners easier, as you can detach all listeners a class defines at once.

Introspecting results

Sometimes you'll want to know what your listeners returned. One thing to remember is that you may have multiple listeners on the same event; the interface for results must be consistent regardless of the number of listeners.

The `EventManager` implementation by default returns a `Zend\EventManager\ResponseCollection` instance. This class extends PHP's `SplStack`, allowing you to loop through responses in reverse order (since the last one executed is likely the one you're most interested in). It also implements the following methods:

- `first()` will retrieve the first result received

- `last()` will retrieve the last result received

- `contains($value)` allows you to test all values to see if a given one was received, and returns a boolean `true` if found, and `false` if not.

- `stopped()` will return a boolean value indicating whether or not a short-circuit occured; more on this in the next section.

Typically, you should not worry about the return values from events, as the object triggering the event shouldn't really have much insight into what listeners are attached. However, sometimes you may want to short-circuit execution if interesting results are obtained. (zend-mvc uses this feature to check for listeners returning responses, which are then returned immediately.)

Short-circuiting listener execution

You may want to short-circuit execution if a particular result is obtained, or if a listener determines that something is wrong, or that it can return something quicker than the target.

As examples, one rationale for adding an `EventManager` is as a caching mechanism. You can trigger one event early in the method, returning if a cache is found, and trigger another event late in the method, seeding the cache.

The `EventManager` component offers two ways to handle this, depending on whether you have an event instance already, or want the event manager to create one for you.

- `triggerEventUntil(callable $callback, EventInterface $event)`
- `triggerUntil(callable $callback, $eventName, $target = null, $argv = [])`

In each case, `$callback` will be any PHP callable, and will be passed the return value from the most recently executed listener. The `$callback` must then return a boolean value indicating whether or not to halt execution; boolean `true` indicates execution should halt.

Your consuming code can then check to see if execution was short-circuited by using the `stopped()` method of the returned `ResponseCollection` .

Here's an example:

```
 public function someExpensiveCall($criteria1, $criteria2)
{
    $params  = compact('criteria1', 'criteria2');
    $results = $this->getEventManager()->triggerUntil(
        function ($r) {
            return ($r instanceof SomeResultClass);
        },
        __FUNCTION__,
        $this,
        $params
    );

    if ($results->stopped()) {
        return $results->last();
    }


}
```

With this paradigm, we know that the likely reason of execution halting is due to the last result meeting the test callback criteria; as such, we return that last result.

The other way to halt execution is within a listener, acting on the `Event` object it receives. In this case, the listener calls `stopPropagation(true)`, and the `EventManager` will then return without notifying any additional listeners.

```
 $events->attach('do', function ($e) {
    $e->stopPropagation();
    return new SomeResultClass();
});
```

This, of course, raises some ambiguity when using the trigger paradigm, as you can no longer be certain that the last result meets the criteria it's searching on. As such, we recommend that you standardize on one approach or the other.

Keeping it in order

On occasion, you may be concerned about the order in which listeners execute. As an example, you may want to do any logging early, to ensure that if short-circuiting occurs, you've logged; if implementing a cache, you may want to return early if a cache hit is found, and execute late when saving to a cache.

Each of `EventManager::attach()` and `SharedEventManager::attach()` accept one additional argument, a *priority*. By default, if this is omitted, listeners get a priority of 1, and are executed in the order in which they are attached. However, if you provide a priority value, you can influence order of execution.

- *Higher* priority values execute *earlier*.
- *Lower* (negative) priority values execute *later*.

To borrow an example from earlier:

```
 $priority = 100;
$events->attach('Example', 'do', function($e) {
    $event  = $e->getName();
    $target = get_class($e->getTarget());
    $params = $e->getParams();
    printf(
```

```
            'Handled event "%s" on target "%s", with parameters %s',
            $event,
            $target,
            json_encode($params)
        );
}, $priority);
```

This would execute with high priority, meaning it would execute early. If we changed `$priority` to `-100`, it would execute with low priority, executing late.

While you can't necessarily know all the listeners attached, chances are you can make adequate guesses when necessary in order to set appropriate priority values.

We advise avoiding setting a priority value unless absolutely necessary.

Custom event objects

As noted earlier, an `Event` instance is created when you call either `trigger()` or `triggerUntil()`, using the arguments passed to each; additionally, you can manually create an instance. Why would you do so, however?

One thing that looks like a code smell is when you have code like this:

```
$routeMatch = $e->getParam('route-match', false);
if (! $routeMatch) {

}
```

The problems with this are several:

- Relying on string keys for event parameters is going to very quickly run into problems — typos when setting or retrieving the argument can lead to hard to debug situations.
- Second, we now have a documentation issue; how do we document expected arguments? how do we document what we're shoving into the event?
- Third, as a side effect, we can't use IDE or editor hinting support — string keys give these tools nothing to work with.

Similarly, consider how you might represent a computational result of a method when triggering an event. As an example:

```
$params['__RESULT__'] = $computedResult;
$events->trigger(__FUNCTION__ . '.post', $this, $params);
```

```
$result = $e->getParam('__RESULT__');
if (! $result) {

}
```

Sure, that key may be unique, but it suffers from a lot of the same issues.

The solution is to create *custom event types*. As an example, zend-mvc defines a custom `MvcEvent`; this event composes the application instance, the router, the route match, the request and response instances, the view model, and also a result. We end up with code like this in our listeners:

```
 $response = $e->getResponse();
$result   = $e->getResult();
if (is_string($result)) {
    $content = $view->render('layout.phtml', ['content' => $result]);
    $response->setContent($content);
}
```

As noted earlier, if using a custom event, you will need to use the `triggerEvent()` and/or `triggerEventUntil()` methods instead of the normal `trigger()` and `triggerUntil()`.

Putting it together: Implementing a caching system

In previous sections, I indicated that short-circuiting is a way to potentially implement a caching solution. Let's create a full example.

First, let's define a method that could use caching. You'll note that in most of the examples, we use `__FUNCTION__` as the event name; this is a good practice, as it makes code completion simpler, maps event names directly to the method triggering the event, and typically keeps the event names unique. However, in the case of a caching example, this might lead to identical events being triggered, as we will be triggering multiple events from the same method. In such cases, we recommend adding a semantic suffix:

`__FUNCTION__ . 'pre'`, `__FUNCTION__ . 'post'`, `__FUNCTION__ . 'error'`, etc. We will use this convention in the upcoming example.

Additionally, you'll notice that the `$params` passed to the event are usually the parameters passed to the method. This is because those are often not stored in the object, and also to ensure the listeners have the exact same context as the calling method. In the upcoming example, however, we will be triggering an event using the *results of execution*, and will need a way of representing that. We have two possibilities:

- Use a "magic" key, such as `__RESULT__`, and add that to our parameter list.
- Create a custom event that allows injecting the result.

The latter is a more correct approach, as it introduces type safety, and prevents typographical errors. Let's create that event now:

```
use Zend\EventManager\Event;

class ExpensiveCallEvent extends Event
{
    private $criteria1;
    private $criteria2;
    private $result;

    public function __construct($target, $criteria1, $criteria2)
    {

        $this->setName('someExpensiveCall');
        $this->setTarget($target);
        $this->criteria1 = $criteria1;
        $this->criteria2 = $criteria2;
    }

    public function getCriteria1()
    {
        return $this->criteria1;
    }

    public function getCriteria2()
    {
        return $this->criteria2;
```

```
    }

    public function setResult(SomeResultClass $result)
    {
        $this->result = $result;
    }

    public function getResult()
    {
        return $this->result;
    }
}
```

We can now create an instance of this within our class method, and use it to trigger listeners:

```
 public function someExpensiveCall($criteria1, $criteria2)
{
    $event = new ExpensiveCallEvent($this, $criteria1, $criteria2);
    $event->setName(__FUNCTION__ . '.pre');
    $results = $this->getEventManager()->triggerEventUntil(
        function ($r) {
            return ($r instanceof SomeResultClass);
        },
        $event
    );

    if ($results->stopped()) {
        return $results->last();
    }




    $event->setName(__FUNCTION__ . '.post');
    $event->setResult($calculatedResult);
    $this->events()->triggerEvent($event);
    return $calculatedResult;
}
```

Before triggering either event, we set the event name in the instance to ensure the correct listeners are notified. The first trigger checks to see if we get a result class returned, and, if so, we return it. The second trigger is a fire-and-forget; we don't care what is returned, and only want to notify listeners of the result.

To provide some caching listeners, we'll need to attach to each of the `someExpensiveCall.pre` and `someExpensiveCall.post` events. In the former case, if a cache hit is detected, we return it. In the latter, we store the value in the cache.

The following listeners attach to the `.pre` and `.post` events triggered by the above method. We'll assume `$cache` is defined, and is a zend-cache storage adapter. The first listener will return a result when a cache hit occurs, and the second will store a result in the cache if one is provided.

```
$events->attach('someExpensiveCall.pre', function (ExpensiveCallEvent
$e) use ($cache) {
    $key = md5(json_encode([
        'criteria1' => $e->getCriteria1(),
        'criteria2' => $e->getCriteria2(),
    ]));

    $result = $cache->getItem($key, $success);

    if (! $success) {
        return;
    }

    $result = new SomeResultClass($result);
    $e->setResult($result);
    return $result;
});

$events->attach('someExpensiveCall.post', function (ExpensiveCallEvent
$e) use ($cache) {
    $result = $e->getResult();
    if (! $result instanceof SomeResultClass) {
        return;
    }

    $key = md5(json_encode([
        'criteria1' => $e->getCriteria1(),
        'criteria2' => $e->getCriteria2(),
    ]));

    $cache->setItem($key, $result);
});
```

> **ListenerAggregates allow stateful listeners**
>
> The above could have been done within a `ListenerAggregate`, which would have allowed keeping the `$cache` instance as a stateful property, instead of importing it into closures.

Another approach would be to move the body of the method to a listener as well, which would allow using the priority system in order to implement caching.

If we did that, we'd modify the `ExpensiveCallEvent` to omit the `.pre` suffix on the default event name, and then implement the class that triggers the event as follows:

```
 public function setEventManager(EventManagerInterface $events)
{
    $this->events = $events;
    $events->setIdentifiers([__CLASS__, get_class($this)]);
    $events->attach('someExpensiveCall', [$this,
'doSomeExpensiveCall']);
}

public function someExpensiveCall($criteria1, $criteria2)
{
    $event = new ExpensiveCallEvent($this, $criteria1, $criteria2);
    $this->getEventManager()->triggerEventUntil(
        function ($r) {
            return $r instanceof SomeResultClass;
        },
        $event
    );
    return $event->getResult();
}

public function doSomeExpensiveCall(ExpensiveCallEvent $e)
{

    $e->setResult($calculatedResult);
}
```

Note that the `doSomeExpensiveCall` method does not return the result directly; this allows what was originally our `.post` listener to trigger. You'll also notice that we return the result from the `Event` instance; this is why the first listener passes the result into the event, as we can then use it from the calling

method!

We will need to change how we attach the listeners; they will now attach directly to the `someExpensiveCall` event, without any suffixes; they will also now use priority in order to intercept before and after the default listener registered by the class. The first listener will listen at priority `100` to ensure it executes before the default listener, and the second will listen at priority `-100` to ensure it triggers after we already have a result:

```
 $events->attach('someExpensiveCall', function (ExpensiveCallEvent $e)
use ($cache) {

}, 100);

$events->attach('someExpensiveCall', function (ExpensiveCallEvent $e)
use ($cache) {

}, -100);
```

The workflow ends up being approximately the same, but eliminates the conditional logic from the original version, and reduces the number of events to one.

The alternative, of course, is to have the object compose a cache instance and use it directly. However, the event-based approach allows:

- Re-using the listeners with multiple events.
- Attaching multiple listeners to the event; as an example, to implement argument validation, or to add logging.

The point is that if you design your object with events in mind, you can add flexibility and extension points without requiring decoration or class extension.

Conclusion

zend-eventmanager is a powerful component. It drives the workflow of zend-mvc, and is used in many Zend Framework components to provide hook points for developers to manipulate the workflow. It can be a powerful tool in your development toolbox.
