Services

Cache

Overview

A HubzeroCacheManager object is available for managing cache data storage and retrieval.

The Cache service comes with a wrapper class to easily work with multiple cache storage driver instances from a single object.

```
$manager = new HubzeroCacheManager($app);
```

Throughout this documentation the Cache facade will be used as it provides a convenient, terse access to the underlying implementations of the cache manager.

Drivers

A number of cache drivers are available.

Custom Drivers

The extend method on the HubzeroCacheManager can be used to extend the cache facility. It is used to bind a custom driver resolver to the manager. The following example demonstrates how to register a new cache driver named "example":

```
Cache::extend('example', function($config)
{
    return new ExampleStore;
});
```

The first argument passed to the extend method is the name of the driver, which will correspond to the driver option in the config/cache.php configuration file. The second argument is a Closure that should return an HubzeroCacheStorageStorageInterface instance. The closure is passed an array of configuration values.

Retrieving Items

The get method on the Cache facade is used to retrieve items from the cache. If the item does

not exist in the cache, null will be returned.

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

A default value can be passed as a second argument to the get method. This value will be returned if the cache store fails to find an item associated with the specified key or the data had expired. The default may be also be a closure:

```
$value = Cache::get('key', 'default');

$value = Cache::get('key', function() { return 'default'; });
```

The all method can be used to retrieve **all** items in the cache store.

```
$data = Cache::all();
```

Checking For Item Existence

The has method may be used to determine if an item exists in the cache:

```
if (Cache::has('key'))
{
    //
}
```

Storing Items

The put method on the Cache object is used to store items in the cache. When placing an item in the cache, the number of minutes for which the value should be cached will also need to specified:

```
Cache::put('key', 'value', $minutes);
```

Alternatively, the add method will only add the item to the cache if it does not already exist in the cache store:

```
Cache::add('key', 'value', $minutes);
```

The forever method may be used to store an item in the cache permanently. These values must be manually removed from the cache using the forget method:

```
Cache::forever('key', 'value');
```

Removing Items

You may remove items from the cache using the forget method on the Cache object:

```
Cache::forget('key');
```

Everything may be removed from the cache store by calling the clean method.

```
Cache::clean();
```

To limit the clean method to specific group of cached data, such as just cached data for the Tags component, a cache group name may be passed. In the example below, this will only remove cached data for the "tags" cache group.

```
Cache::clean('tags');
```

Finally, there is a gc (for "Garbage Collection") method for removing expired data.

```
Cache::gc();
```

Filesystem

Overview

The main entry point for the file system API is the HubzeroFilesystemManager. When working with file systems, this is the class that methods will be invoked on. The Manager makes use of the adapter pattern which helps eliminate the inconsistencies of the different file systems by providing a common interface. So, wether using a Local, Ftp, or Dropbox adapter, the method calls and returned data types will be the same.

Adapters

Local

This is the default file system adapter.

```
$adapter = new HubzeroFilesystemAdapterLocal();

$filesystem = new HubzeroFilesystemFilesystem($adapter);
```

FTP

```
$adapter = new HubzeroFilesystemAdapterFtp(array(
    'host' => 'ftp.example.com',
    'port' => 21,
    'username' => 'username',
    'password' => 'password',
    'root' => '/path/to/root',
));

$filesystem = new HubzeroFilesystemFilesystem($adapter);
```

None

A base adapter used primarily for testing.

```
$adapter = new HubzeroFilesystemAdapterNone();

$filesystem = new HubzeroFilesystemFilesystem($adapter);
```

Retrieving Files

The read method may be used to retrieve the raw string contents of a given file:

```
Filesystem::read('file.jpg');
```

The exists method may be used to determine if a given file exists:

```
if ( ! Filesystem::exists('file.jpg'))
{
    throw new Exception('File not found.');
}
```

Storing Files

The write method may be used to store a file on disk. This method accepts two arguments: the file path and the contents to write.

```
Filesystem::write('file.jpg', $contents);
```

Copy / Move

The copy method may be used to copy an existing file to a new location on the disk:

```
Filesystem::copy('old/file1.jpg', 'new/file1.jpg');
```

The move method may be used to move an existing file to a new location:

```
Filesystem::move('old/file1.jpg', 'new/file1.jpg');
```

Prepend / Append

The prepend method allows for easily prepending contents to the beginning of an existing file.

```
Filesystem::prepend('file.log', 'Prepended Text');
```

Similarly, the append method allows for easily appending contents to the end of a file.

```
Filesystem::append('file.log', 'Appended Text');
```

Removing Files

The delete method accepts a single filename to remove from the system:

```
Filesystem::delete('file.jpg');
```

Directories

soon

Macros

If a feature is not included in the Filesystem class, it can be extended through macros. A macro can extend basic, existing functionality to perform more complex tasks. One example of this would be a macro for creating a directory tree.

Macros must implement HubzeroFilesystemMacroInterface and generally consist of at least two methods: getMethod and handle.

```
class Example extends HubzeroFilesystemMacroBase
{
 public function getMethod()
 {
  return 'examplify';
 }

 public function handle($path)
```

```
 {
  $data = $this->filesystem->read($path);

  return $data . 'EXAMPLE';
 }
}
```

The getMethod method returns the name of the call to the filesystem that will invoke the macro.

The handle method does all the real work. When a macro is invoked, the filesystem object calls handle on the macro and passes it all arguments it received from invoking call.

```
// Add the macro to the filesystem object
$filesystem->addMacro(new Example);

// Append 'EXAMPLE' to a file's contents
$content = $filesystem->examplify('path/to/file');
```

Language

Session

Overview

In its simplest form, a PHP session allows data to be stored temporarily on the server and accessed throughout a user's time on the site. When that user leaves the site or is inactive for a certain amount of time, the data is destroyed.

The Session class has already taken care of many aspects of session storage. It provides a very simple interface to store and retrieve data from the user's session.

Storing Data

Also, when multiple extensions run on a site, there is a possibility of running into naming conflicts in session variables. For this reason, Session allows for the specification of a namespace that a var should be placed under.

```
Session::set('cart', $cart, 'uniqueName');
```

Retrieving Data

Much like in the Request library, a default value can be specified in the get() method as the second argument.

```
// Return an empty array if no data found
$cart = Session::get('cart', array());
```

Events

Overview

Events provide a simple observer implementation, allowing one to subscribe and listen for events in the application.

The Event

When an event is triggered, it's identified by a unique name (e.g. system.onRoute), which any number of listeners might be listening to. An Event instance is also created and passed to all of the listeners.

Naming Conventions

The unique event name can be any string, but optionally follows a few simple naming conventions:

- use only lowercase letters, numbers, dots (.) and underscores (_);
- prefix names with a namespace followed by a dot (e.g. kernel.);
- end names with a verb that indicates what action is being taken (e.g. request).

Event Object

When the dispatcher notifies listeners, it passes an actual Event object to those listeners. The base Event class is very simple: it contains a method for stopping event propagation, methods for adding and removing arguments (i.e., data to be passed to the listeners), and a method for adding to the response.

Often times, data about a specific event needs to be passed along with the Event object so that the listeners have needed information.

```
// Creating an Event called "onSomething".
$event = new HubzeroEventsEvent('system.onDoSomething');

// Adding an argument named "foo" with value "bar".
$event->addArgument('foo', 'bar');
```

Arguments can be added (if not already existing), forcefully set, retrieved, or removed.

addArgument

Add an event argument, only if it is not existing.

setArgument
: Set the value of an event argument. If the argument already exists, it will be overridden.

removeArgument
: Remove an event argument.

getArgument
: Get an event argument value.

hasArgument
: Tell if the given event argument exists.

```
class EventListener
{
    public function onDoSomething($event)
    {
        // Check that the event has the necessary 'foo' argument
        if ( ! $event->hasArgument('foo'))
        {
            return;
        }

        // Get the 'foo' argument
        $foo = $event->getArgument('foo');
    }
}
```

The Dispatcher

The dispatcher is the central object of the event dispatcher system. In general, a single dispatcher is created, which maintains a registry of listeners. When an event is triggered via the dispatcher, it notifies all listeners registered with that event:

```
$dispatcher = new HubzeroEventsDispatcher();
```

Registering Listeners

Registering an event listener can be done simply by passing the listener object to the addListener method.

```
$dispatcher->addListener(new SystemListener);
```

By default, the listener will be registered to all events matching its method names. If the listener contains methods that should not be registered, a defined list of events may be passed.

```
$dispatcher->addListener(
    new SystemListener,
    array(
        'onBeforeRoute' => HubzeroEventsPriority::NORMAL,
        'onAfterRoute' => HubzeroEventsPriority::NORMAL,
    )
);
```

In the above example, the SystemListener object is registered as an even listener along with a specified list of events that it listens to. When a defined list of events is passed, a listener's priority for a given Event is also specified.

When a Listener is added without specifying the event names, it is registered with a NORMAL priority to all events. If some listeners have the same priority for a given event, they will be called in the order they were added to the Dispatcher.

It is also possible to register a closure or anonymous function as an event listener:

```
$dispatcher->addListener(
    function($event) {
        $foo = $event->getArgument('foo');
        // ... do cool things here ...
    },
    array(
        'onBeforeRoute' => HubzeroEventsPriority::NORMAL
    )
);
```

Triggering Events

Once listeners and events have been registered, the events can be triggered. The listeners will be called in a queue according to their priority for that Event.

```
// Triggering the onBeforeRoute Event.
$dispatcher->trigger('onBeforeRoute');
```

The trigger method not only accepts an event name but can also accept a custom Event object.

```
// Creating an event called "onBeforeRoute" with a "foo" argument.
$event = new HubzeroEventsEvent('onAfterSomething');
$event->setArgument('foo', 'bar');

$dispatcher->trigger($event);
```

Arguments or data to be passed to the listener can be specified in an array as a second parameter to the trigger.

```
// Triggering the onBeforeRoute Event.
$dispatcher->trigger('onBeforeRoute', array($foo, $bar));
```

In the example above, the two variables $foo and $bar are added as arguments to the event object and passed to the listener. This is functionally equivalent to the following:

```
// Creating an event called "onBeforeRoute" with a "foo" argument.
$event = new HubzeroEventsEvent('onAfterSomething');
$event->setArgument('foo', $foo);
$event->setArgument('bar', $bar);

$dispatcher->trigger($event);
```

Stopping Events

In some cases, it may make sense for a listener to prevent any other listeners from being called. That is, the listener needs to be able to tell the dispatcher to stop all propagation of the event to future listeners. This can be accomplished from inside a listener by calling the stop() method on the event:

```
class SystemListener
{
    public function onBeforeRoute(Event $event)
    {
        // Stopping the Event propagation.
        $event->stop();
    }
```

```
}
```

When stopping the Event propagation, the next listeners in the queue won't be called.

It is possible to detect if an event was stopped by using the isStopped() method which returns a boolean value:

```
class SystemListener
{
    public function onBeforeRoute(Event $event)
    {
        // Stopping the Event propagation.
        $event->stop();

        if ($event->isStopped())
        {
            //...
        }
    }
}
```