**A (really) tiny Javascript MVC microframework.**

**Downloads**

* [Download production version 0.8.0 (minified)](http://hay.github.io/stapes/stapes.min.js)
* [Download development version 0.8.0 (full source and annotated examples)](https://github.com/hay/stapes/archive/v0.8.0.zip)
* [Fork it and report bugs on Github](http://github.com/hay/stapes)

For node.js npm install stapes will do.

**Philosophy**

*Right. There's a zillion other Javascript MVC frameworks, why should i choose Stapes.js?*

Here's why i think you should use it: it's really tiny (just **2kb** minified and gzipped), easy to fit in an existing codebase, and doesn't force you to write your code in a specific style. Because of its small size it's **ideal for mobile development**.

*So, Stapes.js isn't a traditional MVC framework?*

No. Stapes doesn't have any pre-defined Models, Views or Controllers. It also doesn't have Collections, Routers, or stuff you might find in other libraries (remember, it's tiny :).

*Right.. so how do i write my code?*

Any way you want. Stapes has all the building blocks you need when writing a MVC app.

*Like what?*

Here's a list:

* A powerful yet simple event system using [on](http://hay.github.io/stapes/#m-on) and [emit](http://hay.github.io/stapes/#m-emit) methods. Usable on non-Stapes objects as well using [event mixins](http://hay.github.io/stapes/#m-mixinEvents).
* [Ecmascript Harmony-like classes](http://wiki.ecmascript.org/doku.php?id=harmony:classes) using [subclass](http://hay.github.io/stapes/#m-subclass).
* Event-triggering attributes using [get](http://hay.github.io/stapes/#m-get) and [set](http://hay.github.io/stapes/#m-set).
* Powerful [filter](http://hay.github.io/stapes/#m-filter) and [remove](http://hay.github.io/stapes/#m-remove) methods for attributes.
* Usable in traditional Javascript, as an AMD module (for [Require.js](http://requirejs.org/)) or on the server with [Node.js](http://www.nodejs.org/)
* [A simple mechanism](http://hay.github.io/stapes/#m-plugins) for writing plugins and extensions to extend the functionality of Stapes.
* Nice extensive documentation with lots of examples (you're reading that right now ;)

*But what about templating, persistence, routing, history management, etc?*

That's the flexibility of Stapes: you're free to use any library you want.

*Right, so this is your hobby project. Does this actually work in production?*

Stapes was written for a complex HTML5 media player in a high-traffic website, and it works nicely there, as well as in multiple other projects. Have you used Stapes in your project? [Let me know](mailto:hay@bykr.org), and i'll add you to this site as a showcase.

*Where do i report bugs, feature requests, patches?*

Please report all of those things on [Github](https://github.com/hay/stapes).

*Does this work in all browsers?*

Stapes should work on all current (mobile) browsers, as well as on older browsers like Internet Explorer 8. If you come accross any browser-specific bugs, please [report them](http://github.com/hay/stapes)on Github.

*Okay, seems promising. Where do i start?*

Read on...

**Introduction**

**If you're new to writing MVC apps you probably want to read this**[**beginner's tutorial**](http://www.haykranen.nl/2012/07/19/learn-mvc-create-a-javascript-todo-list-using-stapes-js-in-less-than-100-lines-code/)**, and write a todo app in less than 100 lines of code using Stapes**.

Write your Stapes modules like this

**var** Module **=** Stapes.subclass({

constructor : **function**(name) {

**this**.name **=** name;

}

sayName : **function**() {

console.log('My name is: ' **+** name);

}

});

Then, use it like this.

**var** module **=** **new** Module('Emmylou');

module.sayName(); *// 'My name is Emmylou'*

If you want to use private variables and functions, use Stapes like this

**var** Module **=** (**function**() {

**var** name;

**function** showPrivate() {

console.log("You shouldn't be seeing this", name);

}

**var** Module **=** Stapes.subclass({

constructor : **function**(aName) {

name **=** aName;

showPrivate();

}

});

**return** Module;

})();

**var** module **=** **new** Module('Emmylou');

However, note that this will only work with a single instance of the class because the scope stays the same. Creating an extra module will overwrite the private variables from the previous instance.

**Using event methods with any non-Stapes object or function**

It's possible to add the [event methods](http://hay.github.io/stapes/#m-events) to any regular Javascript object or function. See[Stapes.mixinEvents](http://hay.github.io/stapes/#m-mixinEvents) for more information.

**Using it together with RequireJS**

To avoid lots of global variables you can use a dependency loader like [Require.JS](http://www.requirejs.org/). This also makes the previous example easier to read.

A dependency loader also makes it a lot easier to modularize your code and (with the Require.js optimizer) easily minify and concat your app.

*// 'module.js'*

require(["path/to/Stapes"], **function**(Stapes) {

**var** name;

**function** showPrivate() {

console.log("You shouldn't be seeing this: ", name);

}

**var** Module **=** Stapes.subclass({

constructor : **function**(aName) {

name **=** aName;

showPrivate();

}

});

**return** Module;

});

*// somewhere else*

require(['module'], **function**(Module) {

**var** module **=** **new** Module('Elvis'); *// You shouldn't be seeing this: Elvis*

});

**Examples**

There are three examples available to get a taste on how to write a Stapes.js application.

* [A todo list](http://hay.github.io/stapes/examples/todos)
* [A todo list using Require.JS modules](http://hay.github.io/stapes/examples/todos-require)
* [A simple bouncing ball demo with HTML5 audio](http://hay.github.io/stapes/examples/bounce)

Code for these examples is available in the [development download](http://hay.github.io/stapes/#m-downloads).

Note that the two todo examples are also available from [TodoMVC](http://addyosmani.github.com/todomvc/labs/).

**API: Creation methods**

These methods aid in the creation and extension of classes, or *modules*. These terms are used interchangeably in this document.

**subclass**

**Module.subclass( [object] )**

**Stapes.subclass( [object, classOnly] )**

Create a new Stapes class that you can instantiate later on with new.

Note that until Stapes 0.6.0 the preferred way of creating new modules was by usingStapes.create. This has been replaced by subclass since 0.7.0. [Read this blogpost why](http://www.haykranen.nl/2013/01/13/stapes-0-7-0-and-the-end-of-create/).

You can add a constructor (the function that is run when you instantiate the class) property to the object. All other properties will become prototype methods. To add static methods to your class use [extend](http://hay.github.io/stapes/#m-extend). If you want to add more properties to the prototype of the class later on, use [proto](http://hay.github.io/stapes/#m-proto).

**var** Module **=** Stapes.subclass({

constructor : **function**(name) {

**this**.name **=** name;

},

getName : **function**() {

**return** **this**.name;

}

});

**var** module **=** **new** Module('emmylou');

module.getName(); *// 'emmylou'*

When calling subclass on a Stapes class you can extend it into a new class that will inherit all prototype properties of the parent class.

**var** Module **=** Stapes.subclass({

sayName : **function**() {

console.log('i say my name!');

}

});

**var** BetterModule **=** Module.subclass({

sayNameBetter : **function**() {

**this**.sayName(); *// Inherits 'sayName' from 'Module'*

console.log('i say my name better!');

}

});

**var** module **=** **new** BetterModule();

module.sayNameBetter(); *// 'i say my name! i say my name better!'*

Note that it's perfectly valid to call subclass without any arguments at all. In that case you'll simply get a new class with an empty constructor.

Also note that if you call subclass on a Module the child module **won't** automatically inherit the constructor of the parent. This is by design. To inherit the constructor of a parent class simply specify it in your subclass setup:

**var** Parent **=** Stapes.subclass({

constructor : **function**(name) {

**this**.name **=** name;

}

});

**var** Child **=** Parent.subclass({

*// inherit the constructor from parent*

constructor : Parent.prototype.constructor

});

Because subclass set ups the prototype chain correctly the instanceof parameter will work as expected.

module instanceof BetterModule; *// true*

module instanceof Module; *// true*

All modules automatically get a parent property that links back to the prototype of the parent. You can use this to override a method in a new class, but still call the method of the parent class, like the super method that is available in lots of languages. Unfortunately super is a reserved keyword in Javascript, so that's not usable in Stapes.

**var** BetterModule **=** Module.subclass({

*// Note that this method name is the same as the one in Module*

sayName : **function**() {

BetterModule.parent.sayName.apply(**this**, arguments);

console.log('i say my name better');

}

});

**var** module **=** **new** BetterModule();

module.sayName(); *// 'i say my name! i say my name better!'*

The optional classOnly flag can be set to true to make a class without any [event](http://hay.github.io/stapes/#m-events) or[data](http://hay.github.io/stapes/#m-data) methods. You will get all of the [creation](http://hay.github.io/stapes/#m-general) methods, so you can still subclass and extend your class. Use this flag to use Stapes as a simple class creation library.

**var** Class **=** Stapes.subclass({

constructor : **function**(name) {

**this**.name **=** name;

}

}, **true** **/\* <-- 'classOnly' flag \*/**);

'subclass' in Class; *// true*

'get' in Class; *// false*

**extend**

**module.extend( object[, object...])**

**Stapes.extend( object )**

Extend your class instance properties by giving an object. Keys with the same value will be overwritten. this will be set to the objects context.

Module.extend({

"names" : ['claire', 'alice']

"sayName" : **function**(i) {

console.log( "Hello " **+** **this**.names[i] **+** "!" );

}

});

**var** module **=** **new** Module();

module.sayName(**1**); *// 'alice'*

Note that using extend is exactly the same as directly assigning properties to the module:

Module.names **=** ['claire', 'alice'];

*// Is the same as*

Module.extend({

names : ['claire', 'alice'];

});

Both extend and [proto](http://hay.github.io/stapes/#m-proto) accept multiple objects as arguments:

**var** module **=** **new** (Stapes.subclass());

**var** instruments **=** { guitar : **true** };

**var** voices **=** { singing : **true** };

module.extend(instruments, voices);

console.log('guitar' in module, 'singing' in module); *// true, true*

extend is also useful for stuff like configuration properties.

**var** Module **=** **new** Stapes.subclass({

constructor : **function**(props) {

**this**.extend( props );

},

sayInfo : **function**() {

console.log(**this**.name **+** ' plays ' **+** **this**.instrument);

}

});

**var** module **=** **new** Module({

'name' : 'Johnny',

'instrument' : 'guitar'

});

module.sayInfo();

Stapes.extend() can be used for writing extra methods and properties that will be available on all Stapes modules, even after you have created them. This is very useful for writing plugins for functionality that isn't in Stapes.

Stapes.extend({

*// Sets an DOM element for views*

"setElement" : **function**(el) {

**this**.el **=** el;

**this**.$el **= $**(el); *// jQuery or Zepto reference*

}

});

**var** Module **=** Stapes.subclass();

**var** module **=** **new** Module();

module.setElement( document.getElementById("app") );

console.log( module.el ); *// "#app" DOM element*

console.log( module.$el ); *// jQuery or Zepto element*

All internal methods are available as the Stapes.\_ object, so if you want you can overwrite and hack virtually all Stapes behaviour. Note that these methods aren't documented, so you should take a look at the source.

**proto**

**Module.proto( object )**

Adds properties and methods to the prototype of the module.

Note that it's usally easier to directly add methods to the prototype using the [subclass](http://hay.github.io/stapes/#m-subclass)method. However,if you want to add methods to the prototype later on, proto might be useful.

**var** Module **=** Stapes.subclass();

Module.proto({

'sayName' : **function**() {

console.log('I say my name');

}

});

'sayName' in Module; *// false, not a static method*

'sayName' in Module.prototype; *// true*

*// Note that using proto is the same as:*

Module.prototype.sayName **=** **function**() {

console.log('I say my name');

};

Both proto and [extend](http://hay.github.io/stapes/#m-extend) accept multiple objects as arguments. This can be very handy for mixins.

**var** Module **=** Stapes.subclass();

**var** instruments **=** { guitar : **true** };

**var** voices **=** { singing : **true** };

Module.proto(instruments, voices);

**var** module **=** **new** Module();

console.log(module.guitar, module.singing); *// true, true*

**API: Events**

**on**

**module.on(eventName, handler [, context] )**

**module.on(object [, context] )**

**Stapes.on(object [, context] )**

**Stapes.on(eventName, handler [, context] )**

Add an event listener to a Stapes module or [object with mixed-in events](http://hay.github.io/stapes/#m-mixinEvents). When an event is [triggered](http://hay.github.io/stapes/#m-emit) the handler will be called with two arguments: data, which may contain any data, and an eventObject that contains some useful information about the event, such as the scope, target and event name.

As an optional third (when defining a single listener) or second (when defining multipe listeners using an object) parameter you can set what the value of this should be in the scope of the event handler. This is handy to prevent having to temporarily save the scope to a self or that variable or use the new bind property of EcmaScript 5.

**var** Module **=** Stapes.subclass();

**var** module **=** **new** Module();

module.on('ready', **function**() {

console.log('your module is ready!');

});

module.on({

"talk" : **function**() {

console.log('your module is talking!');

},

"walk" : **function**() {

console.log('your module is walking!');

},

"eat" : **function**( food ) {

console.log('your module is eating ' **+** (**this**.get('food') **||** 'nothing'));

}

});

**var** feedme **=** **new** Module();

*// Set the scope for the event handler to 'module'*

feedme.on({

"feed" : **function**( food ) {

**this**.set('food', food);

}

}, module);

module.emit('eat');

*// "your module is eating nothing"*

feedme.emit('feed', 'cookies');

module.emit('eat');

*// "your module is eating cookies"*

module.get('food');

*// "cookies"*

feedme.get('food');

*// null*

The on method on the global Stapes object can be used to listen to events from all defined modules.

Listening to the special all event on the Stapes global will trigger on **all** events thrown in all objects. Very useful for debugging, but be careful not to leave it in your production code :)

**var** Module **=** Stapes.subclass({

constructor : **function**(name) {

**this**.name **=** name;

},

beReady : **function**() {

**this**.emit('ready');

}

});

**var** module1 **=** **new** Module('module1');

**var** module2 **=** **new** Module('module2');

Stapes.on('ready', **function**(data, e) {

console.log('a ready event was triggered in: ' **+** e.scope.name);

});

module1.beReady(); *// 'a ready event was triggered in module 1';*

module2.beReady(); *// 'a ready event was triggered in module 2';*

The all event listener is also available on every module, so you can listen to all events from a specific module.

**var** Module **=** Stapes.subclass({

"go" : **function**() {

**this**.emit('foo');

**this**.emit('bar');

}

});

**var** module **=** **new** Module();

module.on("all", **function**(data, e) {

console.log(e.type);

});

module.go(); *// first 'foo', then 'bar'*

**off**

**module.off(eventType, handler)**

**module.off(eventType)**

**module.off()**

Removes event handlers from an object. Giving both an eventType and a handler will remove that specific handler from a module. Giving only an eventType will remove all handlers bound to that event type.

With no arguments at all, off() will remove *all* event handlers from a module.

**var** Module **=** Stapes.subclass();

**var** module **=** **new** Module();

**var** handler **=** **function**(){};

module.on({

"foo" : handler,

"bar" : **function**(){}

});

module.off("foo", handler); *// Removes only the specific handler for foo*

module.off("bar"); *// Removes all 'bar' handlers*

module.off(); *// Removes all handlers*

**emit**

**module.emit(eventName[, data])**

Trigger an event on the module. eventName can be a space seperated string if you want to trigger more events. data can be any Javascript variable you want, and will be passed to any event listeners

**var** Module **=** Stapes.subclass({

"sleep" : **function**() {

**this**.emit('sleeping', 'very deep');

}

});

**var** module **=** **new** Module();

module.on('sleeping', **function**(how) {

console.log("i'm sleeping " **+** how);

});

module.sleep(); *// "i'm sleeping very deep"*

**mixinEvents**

**Stapes.mixinEvents([object])**

It's possible to add Stapes' [event handling methods](http://hay.github.io/stapes/#m-events) to any Javascript object or function. This can be very handy if you want to create an object that only uses event handlers, or for an object or function that already exists and you don't want to convert to a Stapes module.

Here's how to add event methods to an object:

**var** module **=** {};

Stapes.mixinEvents(module);

module.on('sing', **function**() {

console.log("i'm singing!");

});

module.sing **=** **function**() {

**this**.emit('sing');

}

module.sing(); *// i'm singing!*

Stapes.mixinEvents returns the object, so you could write the first two lines from the previous example even shorter:

**var** module **=** Stapes.mixinEvents( {} );

No, wait! It can be even shorter! Without any arguments Stapes.mixinEvents returns a new object with mixed-in events.

**var** module **=** Stapes.mixinEvents();

You can also add event methods to a function:

**function** Module(what) {

**this**.what **=** what;

Stapes.mixinEvents(**this**);

}

Module.prototype.sing **=** **function**() {

**this**.emit('sing', **this**.what);

}

**var** m **=** **new** Module("Happy Birthday");

m.on('sing', **function**(what) {

console.log("i'm singing " **+** what **+** "!");

});

m.sing(); *// i'm singing Happy Birthday!*

Note that these events are also triggered on the main Stapes object, so you can useStapes.on to catch events from these mixed-in objects as well.

Stapes.on('sing', **function**(data, e) {

console.log("Singing from " **+** e.scope.name);

});

**var** module **=** Stapes.mixinEvents( {} );

module.name **=** "a cool module!";

module.sing **=** **function**() {

**this**.emit('sing');

}

module.sing(); *// 'Singing from a cool module!'*

**API: Data methods**

**each**

**module.each( function, [context] );**

Iterate over all attributes of a module.

When giving a context parameter this will be set as the this value in the iterator function. If context is not set it will be set to the module itself.

**var** Module **=** Stapes.subclass();

**var** singers **=** ['Johnny', 'Emmylou', 'Gram', 'June'];

**var** module **=** **new** Module();

module.msg **=** "I'll be your ";

module.push(singers);

module.each(**function**(singer) {

console.log(**this**.msg **+** singer);

});

*// Using the second context parameter of each()*

module.singers **=** **new** (Stapes.subclass());

module.singers.push( singers );

module.singers.each(**function**(singer) {

console.log(**this**.msg **+** singer);

}, module);

**filter**

**module.filter( function );**

Gets an array of attribute values using a custom function.

The callback function gets two arguments: the value of the attribute, and the original key.

module.set('singer', {

'name' : 'Elvis',

'instrument' : 'Guitar'

});

**var** singer **=** module.filter(**function**(item, key) {

**return** item.name **===** "Elvis" **&&** key **===** "singer";

}); *// [ { name : 'Elvis', instrument : 'Guitar' }]*

**get**

**module.get( key );**

**module.get( function );**

Gets an attribute by key. If the item is not available will return null

You can also use a function to get a specific key. This is comparable to [filter](http://hay.github.io/stapes/#m-filter), however, filter always returns an array of results, while get always returns a single result.

**var** Module **=** Stapes.subclass();

**var** module **=** **new** Module();

module.set({

'instrument': 'guitar',

'name': 'Johnny'

});

module.get('instrument'); *// 'guitar'*

module.get(**function**(value) {

**return** value **===** "guitar";

}); *// 'instrument'*

**getAll**

**module.getAll();**

Returns all the attributes of an module as an object. Handy for JSON serializing and persistence layers.

Note that this method returns a copy/clone instead of a reference.

**getAllAsArray**

**module.getAllAsArray();**

Returns all attributes as an array, so you can easily iterate. Note that the original key of the attribute is always available as a 'id' key in the the value, provided your value is an object.

Note that this method returns a copy/clone instead of a reference.

module.set({

"name" : "Johnny",

"instrument" : "guitar"

});

module.getAllAsArray().length; *// '2'*

**has**

**module.has( key );**

Checks if a key is available and returns true or false.

module.set('singer', 'Johnny');

module.has('singer'); *// true*

module.has('instrument'); *// false*

**map**

**module.map( function, [context] )**

Just like [each](http://hay.github.io/stapes/#m-each), map iterates over all attributes of a module. map returns a new array, where every attribute value of a module has been passed through function.

module.push([**1**, **2**, **3**]);

**var** arr **=** module.map(**function**(nr) {

**return** nr **\*** **2**;

});

console.log(arr); *// '[2, 4, 6]'*

By default map gets the module as the value of this, but this can be overwritten with the second argument.

**push**

**module.push( value, [silent] );**

**module.push( array, [silent] );**

Sets a value, automatically generates an unique uuid as a key.

You can also push an array of values.

Just as with set the optional silent flag will prevent any change events from being emitted.

For the rest of the behaviour of this method see [set](http://hay.github.io/stapes/#m-set).

m.push("foo");

m.getAll(); *// will look something like { "5323be61-afb8-4034-b408-51132756cd43" : "foo"}*

m.push([

"foo",

"bar",

"baz"

]);

push returns the module, so this method is chainable.

**remove**

**module.remove( key, [silent] );**

**module.remove( function );**

**module.remove();**

Deletes an attribute. Triggers remove and change events.

remove also triggers namespaced change and remove events (e.g. change:fooand remove:foo).

You can either use a key as an argument or a function

module.remove(**function**(item) {

**return** item.done **===** **true**;

});

It's possible to remove multiple keys in one go, simply space seperate them:

module.remove('singer instrument johnny');

If you do not want your remove to trigger an event set the optional silent flag to true.

module.remove('singer', **true** **/\* <-- silent flag \*/**);

Without any arguments, remove deletes all attributes in a module and triggers changeand remove events.

module.push([**1**,**2**,**3**]);

module.size(); *// 3*

module.remove();

module.size(); *// 0*

remove returns the module, so this method is chainable.

**set**

**module.set(key, value, [silent]);**

**module.set(object, [silent]);**

Sets an attribute. Use [push](http://hay.github.io/stapes/#m-push) if you want to 'push' a value with a random uuid, for collections.

To set multiple attributes in one go, use an object as an first argument.

Every attribute will trigger a change event. A key that doesn't exist will trigger acreate event, a key that does exist will trigger an update event.

All events will have the key of the attribute as their event value.

Special namespaced events will also be triggered. These events have a value ofeventType:key. So for example, a set on an attribute called 'name' will generate achange:name event. These events will have the attribute value instead of the key as a data argument in the event callback.

module.on({

"change" : **function**(key) {

console.log('Something happened with ' **+** key);

},

"change:name" : **function**(value) {

console.log('name was changed to ' **+** value);

},

"create" : **function**(key) {

console.log("New attribute " **+** key **+** " added!");

},

"update" : **function**(key) {

console.log("Attribute " **+** key **+** " was updated!");

}

});

module.set('name', 'Elvis'); *// will trigger 'change' and 'create' events*

module.set('name', 'Johnny'); *// will trigger 'change' and 'update' events*

module.set({

"name" : "Elvis",

"instrument" : "guitar"

});

If you do not want your set to trigger *any* events set the optional silent flag to true

module.set('singer', 'Johnny', **true** **/\* <-- silent flag \*/**);

This also works for objects:

module.set({

'singer' : 'Johnny',

'instrument' : 'guitar'

}, **true** **/\* silent \*/**);

To get the old or previous value of an attribute use the mutate event instead ofchange. There are both namespaced and general versions of this event, just likechange. Instead of the value of the attribute it will return an object with oldValue andnewValue properties. For convenience it also returns the key of the attribute.

module.set('name', 'Johnny');

module.on({

"mutate:name" : **function**(values) {

console.log(values.oldValue, values.newValue); *// "Johnny, Emmylou"*

},

"mutate" : **function**(values) {

*// Returns 'Emmylou'*

*// Note that this is identical to writing:*

*// module.on(*

*// 'change',*

*// function(key) {*

*// console.log( module.get(key) );*

*// }*

*// );*

console.log(values.newValue);

}

});

module.set('name', 'Emmylou');

Note that it's still perfectly fine to assign properties to the Stapes module directly, as long as you don't overwrite existing properties. All the data methods are useful if you want to do model-like stuff, but for ordinary properties that don't need change events setting attributes using get and set is fine.

module.on('change:name', **function**() {

*// Obviously, this event hander will never trigger*

});

module.name **=** "Elvis";

console.log(module.name); *// "Elvis"*

set returns the module, so this method is chainable

Stapes doesn't have a native way of doing validation, but you could overwrite the set(and possibly update) method to accomplish the same:

**var** Person **=** Stapes.subclass({

set : **function**(key, value) {

**if** (key **===** 'email') {

**if** (value.indexOf("@") **===** **-1**) {

**this**.emit('error', 'Invalid email adress');

} **else** {

Stapes.prototype.set.apply(**this**, arguments);

}

} **else** {

*// Normal set() behavior*

Stapes.prototype.set.apply(**this**, arguments);

}

}

});

**var** johnny **=** **new** Person();

johnny.on('error', **function**(msg) {

alert( msg ); *// 'Invalid email adress'*

});

johnny.set('email', 'invalid#email.com');

**size**

**module.size();**

Returns the number of attributes in a module.

module.push(['Johnny', 'Emmylou', 'June', 'Gram']);

console.log( module.size() ); *// '4'*

**update**

**module.update( key, fn, [silent] );**

**module.update( fn );**

Updates an attribute with a new value, based on the return value of a function.

Just as with [set](http://hay.github.io/stapes/#m-set) update will generate change and update events.

module.set('singer', 'Elvis');

module.update('singer', **function**(singer) {

**return** 'Johnny';

});

console.log( module.get('singer') ); *// 'Johnny';*

You can also pass a single function as an argument. In that case, update will run on all attributes in the module.

module.push([

{ "name" : "Johnny", "singer" : **false**},

{ "name" : "Emmylou", "singer" : **false**}

]);

module.update(**function**(item) {

item.singer **=** **true**;

**return** item;

});

The callback function in update gets two arguments: the value being modified, and the original key. The this value of the callback will be set to the module you're updating.

The silent flag will prevent any change events from firing.

update returns the module, so this method is chainable.

**Removed and deprecated stuff**

**Introduction**

These methods, modules and other things are either deprecated or have been removed. Don't use these features in new code, and remove any dependencies on them in your current code, in a future version they might be removed completely.

**create (removed)**

**Stapes.create()**

**module.create()**

Create a new instance of a Stapes object.

This method has been deprecated since 0.7.0 and was completely removed in 0.8.0.[Read this blogpost why](http://www.haykranen.nl/2013/01/13/stapes-0-7-0-and-the-end-of-create/).

**Stapes.util (removed)**

Until version 0.5.1 Stapes included a util module with Underscore-like utilities such aseach and map. This was removed in version 0.6.

If your code depends on the Stapes.util module you can use a [compatibility plugin](http://hay.github.io/stapes/plugins/#m-util).

**Miscellaneous**

**version**

**Stapes.version**

Returns the current version of Stapes. Note that this property is only available on the Stapes global variable, not on individual modules.

**Bugs and known limitations**

* Attributes are stored in an object internally. According to the Javascript spec objects don't guarantee that properties are returned in order. However, all browsers do return object properties in order except for [Chrome](http://stackoverflow.com/a/280861/152809) where numbered keys won't return in order.

In the future this will be fixed, but it requires a big rewrite. For now the best way to prevent this problem is to simply avoid using numbered attributes. So don't write something like module.set('1', 'one');

**History**