

## NAME

Module::Pluggable - automatically give your module the ability to have plugins

## SYNOPSIS

Simple use Module::Pluggable -

```
package MyClass;
use Module::Pluggable;
```

and then later ...

```
use MyClass;
my $mc = MyClass->new();
# returns the names of all plugins installed under MyClass::Plugin::*
my @plugins = $mc->plugins();
```

## EXAMPLE

Why would you want to do this? Say you have something that wants to pass an object to a number of different plugins in turn. For example you may want to extract meta-data from every email you get sent and do something with it. Plugins make sense here because then you can keep adding new meta data parsers and all the logic and docs for each one will be self contained and new handlers are easy to add without changing the core code. For that, you might do something like ...

```
package Email::Examiner;

use strict;
use Email::Simple;
use Module::Pluggable require => 1;

sub handle_email {
    my $self = shift;
    my $email = shift;

    foreach my $plugin ($self->plugins) {
        $plugin->examine($email);
    }

    return 1;
}
```

.. and all the plugins will get a chance in turn to look at it.

This can be trivially extended so that plugins could save the email somewhere and then no other plugin should try and do that. Simply have it so that the `examine` method returns 1 if it has saved the email somewhere. You might also want to be paranoid and check to see if the plugin has an `examine` method.

```
foreach my $plugin ($self->plugins) {
    next unless $plugin->can('examine');
    last if $plugin->examine($email);
}
```

And so on. The sky's the limit.

## DESCRIPTION

Provides a simple but, hopefully, extensible way of having 'plugins' for your module. Obviously this isn't going to be the be all and end all of solutions but it works for me.

Essentially all it does is export a method into your namespace that looks through a search path for .pm files and turn those into class names.

Optionally it instantiates those classes for you.

## ADVANCED USAGE

Alternatively, if you don't want to use 'plugins' as the method ...

```
package MyClass;
use Module::Pluggable sub_name => 'foo';
```

and then later ...

```
my @plugins = $mc->foo();
```

Or if you want to look in another namespace

```
package MyClass;
use Module::Pluggable search_path => ['Acme::MyClass::Plugin',
'MyClass::Extend'];
```

or directory

```
use Module::Pluggable search_dirs => ['mylibs/Foo'];
```

Or if you want to instantiate each plugin rather than just return the name

```
package MyClass;
use Module::Pluggable instantiate => 'new';
```

and then

```
# whatever is passed to 'plugins' will be passed
# to 'new' for each plugin
my @plugins = $mc->plugins(@options);
```

alternatively you can just require the module without instantiating it

```
package MyClass;
use Module::Pluggable require => 1;
```

since requiring automatically searches inner packages, which may not be desirable, you can turn this off

```
package MyClass;
use Module::Pluggable require => 1, inner => 0;
```

You can limit the plugins loaded using the except option, either as a string, array ref or regex

```
package MyClass;
use Module::Pluggable except => 'MyClass::Plugin::Foo';
```

or

```
package MyClass;
use Module::Pluggable except => ['MyClass::Plugin::Foo',
'MyClass::Plugin::Bar'];
```

or

```
package MyClass;
use Module::Pluggable except => qr/^MyClass::Plugin::(Foo|Bar)$/;
```

and similarly for only which will only load plugins which match.

Remember you can use the module more than once

```
package MyClass;
use Module::Pluggable search_path => 'MyClass::Filters' sub_name =>
'filters';
use Module::Pluggable search_path => 'MyClass::Plugins' sub_name =>
'plugins';
```

and then later ...

```
my @filters = $self->filters;
my @plugins = $self->plugins;
```

## INNER PACKAGES

If you have, for example, a file **lib/Something/Plugin/Foo.pm** that contains package definitions for both `Something::Plugin::Foo` and `Something::Plugin::Bar` then as long as you either have either the **require** or **instantiate** option set then we'll also find `Something::Plugin::Bar`. Nifty!

## OPTIONS

You can pass a hash of options when importing this module.

The options can be ...

### sub\_name

The name of the subroutine to create in your namespace.

By default this is 'plugins'

### search\_path

An array ref of namespaces to look in.

### search\_dirs

An array ref of directories to look in before @INC.

### instantiate

Call this method on the class. In general this will probably be 'new' but it can be whatever you want. Whatever arguments are passed to 'plugins' will be passed to the method.

The default is 'undef' i.e just return the class name.

### require

Just require the class, don't instantiate (overrides 'instantiate');

**inner**

If set to 0 will **not** search inner packages. If set to 1 will override `require`.

**only**

Takes a string, array ref or regex describing the names of the only plugins to return. Whilst this may seem perverse ... well, it is. But it also makes sense. Trust me.

**except**

Similar to `only` it takes a description of plugins to exclude from returning. This is slightly less perverse.

**package**

This is for use by extension modules which build on `Module::Pluggable`: passing a `package` option allows you to place the plugin method in a different package other than your own.

**file\_regex**

By default `Module::Pluggable` only looks for `.pm` files.

By supplying a new `file_regex` then you can change this behaviour e.g

```
file_regex => qr/\.(plugin$)/
```

**include\_editor\_junk**

By default `Module::Pluggable` ignores files that look like they were left behind by editors. Currently this means files ending in `~` (`~`), the extensions `.swp` or `.swo`, or files beginning with `.#`.

Setting `include_editor_junk` changes `Module::Pluggable` so it does not ignore any files it finds.

**METHODS****search\_path**

The method `search_path` is exported into you namespace as well. You can call that at any time to change or replace the `search_path`.

```
$self->search_path( add => "New::Path" ); # add  
$self->search_path( new => "New::Path" ); # replace
```

**FUTURE PLANS**

This does everything I need and I can't really think of any other features I want to add. Famous last words of course

Recently tried fixed to find inner packages and to make it 'just work' with PAR but there are still some issues.

However suggestions (and patches) are welcome.

**AUTHOR**

Simon Wistow <simon@thegestalt.org>

**COPYING**

Copyright, 2006 Simon Wistow

Distributed under the same terms as Perl itself.

**BUGS**

None known.

**SEE ALSO**

*File::Spec, File::Find, File::Basename, Class::Factory::Util, Module::Pluggable::Ordered*