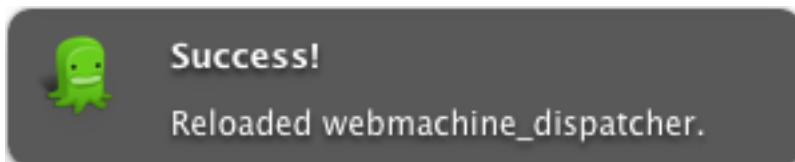


---

## Stay in Sync

### What is Sync?

Sync is a developer utility. It recompiles and reloads your Erlang code on-the-fly. With Sync, you can code without friction.



What does “code without friction” mean? It means that with Sync running, you no longer need to worry about running `make`, or `c:l(Module)` again. Write code, save the file, and watch as Erlang automatically detects your changes, recompiles the code, and reloads the module.

### How can I use Sync?

#### Install via rebar dependency

```
1 {deps, [  
2     {sync, ".*",  
3         {git, "https://github.com/rustyio/sync.git", {branch, "  
4         master"}}}  
5 ]}.
```

#### Manual install

```
1 cd $ERL_LIBS  
2 git clone git@github.com:rustyio/sync.git  
3 (cd sync; make)
```

The recommended approach is to put sync in your `$ERL_LIBS` directory.

Then, go in the Erlang console of an application you are developing, run `sync:go()` . You can also start sync using `application:start(sync)` . but this will require you to have sync’s dependencies started as well: `syntax_tools` and `compiler`.

It’s generally just recommended to do `sync:go()` .

Starting up:

---

```
1 (rustyio@127.0.0.1)6> sync:go().
2
3 Starting Sync (Automatic Code Compiler / Reloader)
4 Scanning source files...
5 ok
6 08:34:18.609 [info] Application sync started on node 'rustyio@127.0.0.1'
```

Successfully recompiling a module:

```
1 08:34:43.255 [info] /Code/Webmachine/src/webmachine_dispatcher.erl:0:
   Recompiled.
2 08:34:43.265 [info] webmachine_dispatcher: Reloaded! (Beam changed.)
```

Warnings:

```
1 08:35:06.660 [info] /Code/Webmachine/src/webmachine_dispatcher.erl:33:
   Warning: function dispatch/3 is unused
```

Errors:

```
1 08:35:16.881 [info] /Code/Webmachine/src/webmachine_dispatcher.erl:196:
   Error: function reconstitute/1 undefined
2 /Code/Webmachine/src/webmachine_dispatcher.erl:250: Error: syntax error
   before: reconstitute
```

## Stopping and Pausing

You can stop the `sync` application entirely (wiping its internal state) with `sync:stop()`. You can then restart the application with a new state using `sync:go()`

If, however, you would rather pause `sync` so that it will not update anything during some period, you can pause the scanner with `sync:pause()`. You might do this while upgrading you wish not to have immediately loaded until everything is complete. Calling `sync:go()` once again will unpause the scanner.

Bear in mind that running `pause()` will not stop files that are currently being compiled.

## Specifying folders to sync

To your erlang `config` add

```
1 [
2     {sync, [
3         {src_dirs, {strategy(), [src_dir_descr()]}}
```

---

```
4     ]}
5 ].
```

```
1 -type strategy() :: add | replace.
```

If `strategy()` is `replace`, sync will use ONLY specified dirs to sync. If `strategy()` is `add`, sync will add specific dirs to list of dirs to sync.

```
1 -type src_dir_descr() :: { Dir :: file:filename(), [Options ::
    compile_option()]}.
```

You probably want to specify `outdir` compile option.

For example

```
1 [
2     {sync, [
3         {src_dirs, {replace, [{"./priv/plugins", [{outdir, "./priv/
4             plugins_bin"}]}}}
5     ]}
6 ].
```

## Console Logging

By default, sync will print success / warning / failure notifications to the erlang console. You can control this behaviour with the `log` configuration options.

### Valid Values For `log`

- `all`: Print all console notifications
- `none`: Print no console notifications
- `[success | warnings | errors]`: A list of any combination of the atoms `success`, `warnings`, or `errors`. Example: `[warnings, errors]` will only show warnings and errors, but suppress success messages.
- `true`: An alias to `all`, kept for backwards compatibility
- `false`: An alias to `none`, kept for backwards compatibility
- `skip_success`: An alias to `[errors, warnings]`, kept for backwards compatibility.

The `log` value can be specified in any of the following ways:

#### 1. Loaded from a `.config file`

```
1 {log, all},
2 {log, [success, warnings]},
```

---

## 2. As an environment variable called from the erlang command line:

```
1 erl -sync log all
2 erl -sync log none
```

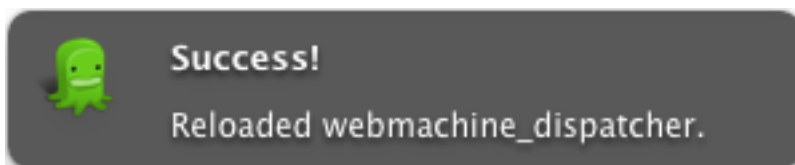
## 3. From within the Erlang shell:

```
1 sync:log(all);
2 sync:log(none);
3 sync:log([errors, warnings]);
```

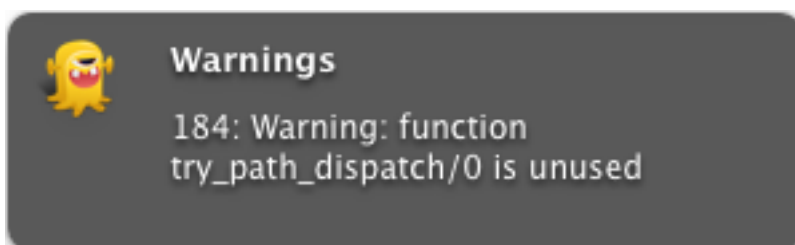
## Desktop Notifications

Sync can pop success / warning / failure notifications onto your desktop to keep you informed of compilation results. All major operating systems are supported: Mac via Growl, Linux via Libnotify, Windows via Notifu and Emacs via lwarn / message command. Below are Growl screenshots.

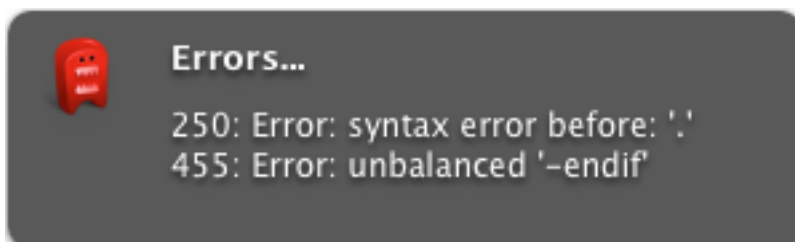
Success:



Warnings:



Errors:



---

## Disabling Desktop Notifications

Desktop notifications follow the same conventions as the console logging above, and can be selectively enabled or disabled with the `growl` configuration variable:

### Valid Values For `growl`

- `all`: Print all console notifications
- `none`: Print no console notifications
- `[success | warnings | errors]`: A list of any combination of the atoms `success`, `warnings`, or `errors`. Example: `[warnings, errors]` will only show warnings and errors, but suppress success messages.
- `true`: An alias to `all`, kept for backwards compatibility
- `false`: An alias to `none`, kept for backwards compatibility
- `skip_success`: An alias to `[errors, warnings]`, kept for backwards compatibility.

The `growl` value can be specified in any of the following ways:

#### 1. Loaded from a `.config` file

```
1 {growl, all},  
2 {growl, [success, warnings]},
```

#### 2. As an environment variable called from the `erlang` command line:

```
1 erl -sync growl all  
2 erl -sync growl none
```

#### 3. From within the Erlang shell:

```
1 sync:growl(all);  
2 sync:growl(none);  
3 sync:growl([errors, warnings]);
```

## Troubleshooting Growl Notifications

Sync attempts to auto-detect the notification package to use via the `os:type()` command.

If this isn't working for you, or you would like to override the default, use the `executable` configuration parameter:

```
1 {executable, TYPE}
```

---

Where `TYPE` is: + `'auto'` Autodetermine (default behaviour) + `'growlnotify'` for Mac / Growl. + `'notification_center'` for Mac OS X built-in Notification Center. + `'notify-send'` for Linux / libnotify. + `'notifu'` for Windows / Notifu. + `'emacscclient'` for Emacs notifications.

Like all configuration parameters, this can also be specified when launching Erlang with:

```
1 erl -sync executable TYPE
```

## Remote Server “Patching”

If you are developing an application that runs on an Erlang cluster, you may need to recompile a module on one node, and then broadcast the changed module to other nodes. Sync helps you do that with a feature called “patching.”

To use the patching feature:

1. Connect to any machine in your cluster via distributed erlang. A simple `net_adm:ping(Node)` should suffice.
2. Run `sync:patch()`. This will start the Sync application if it’s not already started, and enable “patch mode”.
3. Start editing code.

Sync will detect changes to code, recompile your modules, and then sent the updated modules to every Erlang node connected to your cluster. If the module already exists on the node, then it will be overwritten on disk with the new .beam file and reloaded. If the module doesn’t exist on the new node, then it is simply updated in memory.

## How does Sync work?

Upon startup, Sync gathers information about loaded modules, ebin directories, source files, compilation options, etc.

Sync then periodically checks the last modified date of source files. If a file has changed since the last scan, then Sync automatically recompiles the module using the previous set of compilation options. If compilation was successful, it loads the updated module. Otherwise, it prints compilation errors to the console.

Sync also periodically checks the last modified date of any beam files, and automatically reloads the file if it has changed.

---

The scanning process adds 1% to 2% CPU load on a running Erlang VM. Much care has been taken to keep this low. Shouldn't have to say this, but this is for development mode only, don't run it in production.

## Sync Post-hooks

You can register a post-hook to run after Sync reloads modules. This can allow you to run tests on modules if you like, or anything else for that matter.

You can register a post-hook with:

```
1 sync:onsync(fun(Mods) ->
2     io:format("Reloaded Modules: ~p~n",[Mods])
3     end).
```

This will simply print the list of modules that were successfully recompiled.

For example, if you wanted to automatically run a unit test on each reloaded module that has a `test()` function exported, you could do the following:

```
1 RunTests = fun(Mods) ->
2     [Mod:test() || Mod <- Mods, erlang:function_exported(Mod, test, 0)]
3 end,
4 sync:onsync(RunTests).
```

A post-hook can also be specified as a `{Module,Function}` tuple, which assumes `Module:Function/1`

*Note:* Currently, only one post-hook can be registered at a time.

## Unregistering a post-hook

To unregister a post-hook, call

```
1 sync:clear_onsync().
```

## Registering Automatic Tests

There is a handy shortcut to enable automatic tests using the Sync post-hooks, you can simply call:

```
1 sync:enable_autotest().
```

---

## Whitelisting/Excluding modules from the scanning process

Sometimes you may want to focus only on a few modules, or prevent some modules from being scanned by sync. To achieve this modify `whitelisted_modules` or `excluded_modules` configuration parameter in the node's config file.

Beyond specifying modules one by one, identified by atoms, you can also specify them in bulk, identified by regular expressions, but with a slower sync.

## Moving Application Location

Previously, if an entire application (like a reltool-generated release) was moved from one location to another, sync would fail to recompile files that were changed until all the beams were remade. While this is typically as simple as typing `rebar compile`, it was still a hassle.

The solution to this was to enable the ability for sync to “heal” the paths when it turned out they had been moved.

The way this works is by checking if the `source` path inside the beam is a file that exists, and by checking if that path is a descendant of the root of your application. If sync has been set to fix the paths, and module's source is pointing at a path that isn't a descendant of the current working directory, it will attempt to find the correct file.

You can change how this will be handled with a `non_descendants` setting in the config:

- `fix`: Fix any file that isn't a descendant
- `allow`: Use the original path in the module, regardless of its location, recompiling only if the original location changes.
- `ignore`: If a file is not a descendant, sync will completely ignore it.

## Sample Configuration File

Please note that sync loads with the following defaults:

```
1  [  
2      {sync, [  
3          {growl, all},  
4          {log, all},  
5          {non_descendants, fix},  
6          {executable, auto},  
7          {whitelisted_modules, []},  
8          {excluded_modules, []}
```



---

```
9     ]}  
10 ].
```

You can view a full sample configuration file (`sync.sample.config`) that you're free to include in your application. Remember to use the `-config` switch for the `erl` program:

```
1 erl -config sync.config
```