Scripting the Vim editor, Part 5: Event-driven scripting and automation

Automate your workflow with Vim's autocommands

Damian Conway, Dr. March 03, 2010

Why repeat yourself? You can configure Vim's comprehensive event model to execute time-saving scripts whenever particular editing events—such as loading a file or switching between editor modes—occur. This article, the fifth in a series, describes how events work in Vim, explores a selection of useful event types, and then gets you started with attaching specific scripts to particular events. The end result is a more automated workflow configured precisely to your needs.

View more content in this series

Vim's event model

Vim's editing functions behave as if they are event-driven. For performance reasons, the actual implementation is more complex than that, with much of the event handling optimized away or handled several layers below the event loop itself, but you can still think of the editor as a simple while loop responding to a series of editing events.

Whenever you start a Vim session, open a file, edit a buffer, change your editing mode, switch windows, or interact with the surrounding filesystem, you are effectively queuing an event that Vim immediately receives and handles.

For example, if you start Vim, edit a file named demo.txt, swap into Insert mode, type in some text, save the file, and then exit, your Vim session receives a series of events like what is shown in Listing 1.

Listing 1. Event sequence in a simple Vim editing session

```
> vim
    BufWinEnter     (create a default window)
    BufEnter        (create a default buffer)
    VimEnter        (start the Vim session):edit example.txt
    BufNew          (create a new buffer to contain demo.txt)
    BufAdd          (add that new buffer to the session's buffer list)
    BufLeave        (exit the default buffer)
    BufWinLeave     (exit the default window)
```

```
    BufUnload       (remove the default buffer from the buffer list)
    BufDelete       (deallocate the default buffer)
    BufReadCmd      (read the contexts of demo.txt into the new buffer)
    BufEnter        (activate the new buffer)
    BufWinEnter     (activate the new buffer's window)i
    InsertEnter     (swap into Insert mode)
Hello
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)
    CursorMovedI    (insert a character)<ESC>
    InsertLeave     (swap back to Normal mode):wq
    BufWriteCmd     (save the buffer contents back to disk)
    BufWinLeave     (exit the buffer's window)
    BufUnload       (remove the buffer from the buffer list)
    VimLeavePre     (get ready to quit Vim)
    VimLeave        (quit Vim)
```

More interestingly, Vim provides "hooks" that allow you to intercept any of these editing events. So you can cause a particular Vimscript command or function to be executed every time a specific event occurs: every time Vim starts, every time a file is loaded, every time you leave Insert mode … or even every time you move the cursor. This makes it possible to add automatic behaviors almost anywhere throughout the editor.

Vim provides notifications for 78 distinct editing events, which fall into eight broad categories: session start-up and clean-up events, file-reading events, file-writing events, buffer-change events, option-setting events, window-related events, user-interaction events, and asynchronous notifications.

To see the complete list of these events, type `:help autocmd-events` on the Vim command line. For detailed descriptions of each event, see `:help autocmd-events-abc`.

This article explains how events work in Vim and then introduces a series of scripts for automating editing events and behaviours.

Event handling with autocommands

The mechanism Vim provides for intercepting events is known as the *autocommand*. Each autocommand specifies the type of event to be intercepted, the name of the edited file in which such events are to be intercepted, and the command-line mode action to be taken when the event is detected. The keyword for all this is `autocmd` (which is often abbreviated to just `au`). The usual syntax is:

```
autocmd  EventName  filename_pattern   :command
```

The event name is one of the 78 valid Vim event names (as listed under `:help autocmd-events`). The filename pattern syntax is similar -- but not identical -- to a normal shell pattern (see `:help autocmd-patterns` for details). The command is any valid Vim command, including calls to Vimscript functions. The colon at the start of the command is optional, but it’s a good idea to include it; doing so makes the command easier to locate in the (usually complex) argument list of an `autocmd`.

For example, you could surrender all remaining dignity and specify an event handler for the `FocusGained` event by adding the following to your `.vimrc` file:

```
autocmd  FocusGained  *.txt   :echo 'Welcome back, ' . $USER . '! You look great!'
```

`FocusGained` events are queued whenever a Vim window becomes the window system's input focus, so now whenever you swap back to your Vim session, if you're editing any file whose name matches the filename pattern `*.txt`, then Vim will automatically execute the specified `echo` command.

You can set up as many handlers for the same event as you wish, and all of them will be executed in the sequence in which they were originally specified. For example, a far more useful automation for `FocusGained` events might be to have Vim briefly emphasize the cursor line whenever you swap back to your editing session, as shown in Listing 2.

Listing 2. A useful automation for FocusGained events

```
autocmd  FocusGained  *.txt   :set cursorline
autocmd  FocusGained  *.txt   :redraw
autocmd  FocusGained  *.txt   :sleep 1
autocmd  FocusGained  *.txt   :set nocursorline
```

These four autocommands cause Vim to automatically highlight the line containing the cursor (`set cursorline`), reveal that highlighting (`redraw`), wait one second (`sleep 1`), and then switch the highlighting back off (`set nocursorline`).

You can use any series of commands in this way; you can even break up a single control structure across multiple autocommands. For example, you could set up a global variable (`g:autosave_on_focus_change`) to control an "autosave"mechanism that automatically writes any modified `.txt` file whenever the user swaps away from Vim to some other window (causing a `FocusLost` event to be queued):

Listing 3. Autocommand to autosave when leaving an editor window

```
autocmd  FocusLost  *.txt   :    if &modified && g:autosave_on_focus_change
autocmd  FocusLost  *.txt   :    write
autocmd  FocusLost  *.txt   :    echo "Autosaved file while you were absent"
autocmd  FocusLost  *.txt   :    endif
```

Multi-line autocommands like this require that you repeat the essential event-selector specification (i.e., `FocusLost *.txt`) multiple times. Hence they are generally unpleasant to maintain, and more error-prone. It's much cleaner and safer to factor out any control structure, or other command sequences, into a separate function and then have a single autocommand call that function. For example:

Listing 4. A cleaner way to handle multi-line autocommands

```
function! Highlight_cursor ()
    set cursorline
    redraw
    sleep 1
    set nocursorline
endfunction
function! Autosave ()
   if &modified && g:autosave_on_focus_change
       write
       echo "Autosaved file while you were absent"
   endif
endfunction

autocmd  FocusGained  *.txt   :call Highlight_cursor()
autocmd  FocusLost    *.txt   :call Autosave()
```

Universal and single-file autocommands

So far, all the examples shown have restricted event handling to files that matched the pattern `*.txt`. Obviously, that implies that you can use any file-globbing pattern to specify the files to which a particular autocommand applies. For example, you could make the previous cursor-highlighting `FocusGained` autocommand apply to any file simply by using the universal file-matching pattern `*` as the filename filter:

```
" Cursor-highlight any file when context-switching ...
autocmd  FocusGained  *          :call Highlight_cursor()
```

Alternatively, you can restrict commands to a single file:

```
" Only cursor-highlight for my .vimrc ...
autocmd  FocusGained  ~/.vimrc   :call Highlight_cursor()
```

Note that this also implies that you can specify different behaviors for the same event, depending on which file is being edited. For example, when the user turns their attention elsewhere, you might choose to have text files autosaved, or have Perl or Python scripts check-pointed, while a documentation file might be instructed to reformat the current paragraph, as shown in Listing 5.

Listing 5. What to do when the user’s attention is elsewhere

```
autocmd  FocusLost  *.txt   :call Autosave()
autocmd  FocusLost  *.p[ly] :call Checkpoint_sourcecode()
autocmd  FocusLost  *.doc  :call Reformat_current_para()
```

Autocommand groups

Autocommands have an associated namespace mechanism that allows them to be gathered into autocommand groups, whence they can be manipulated collectively.

To specify an autocommand group, you can use the `augroup` command. The general syntax for the command is:

```
augroup GROUPNAME
    " autocommand specifications here ...
augroup END
```

The group's name can be any series of non-whitespace characters, except "`end`"or "`END`", which are reserved for specifying the end of a group.

Within an autocommand group, you can place any number of autocommands. Typically, you would group commands by the event they all respond to, as shown in Listing 6.

Listing 6. Defining a group for autocommands responding to FocusLost events

```
augroup Defocus
    autocmd  FocusLost  *.txt   :call Autosave()
    autocmd  FocusLost  *.p[ly] :call Checkpoint_sourcecode()
    autocmd  FocusLost  *.doc   :call Reformat_current_para()
augroup END
```

Or you might group a series of autocommands relating to a single filetype, such as:

Listing 7. Defining a group of autocommands for handling text files

```
augroup TextEvents
    autocmd  FocusGained  *.txt   :call Highlight_cursor()
    autocmd  FocusLost    *.txt   :call Autosave()
augroup END
```

Deactivating autocommands

You can remove specific event handlers using the `autocmd!` command (that is, with an exclamation mark). The general syntax for this command is:

```
autocmd!  [group]  [EventName [filename_pattern]]
```

To remove a single event handler, specify all three arguments. For example, to remove the handler for `FocusLost` events on `.txt` files from the `Unfocussed` group, use:

```
autocmd!  Unfocussed  FocusLost  *.txt
```

Instead of a specific event name, you can use an asterisk to indicate that every kind of event should be removed for the particular group and filename pattern. If you wanted to remove all events for `.txt` files within the `Unfocussed` group, you would use:

```
autocmd!  Unfocussed      *      *.txt
```

If you leave off the filename pattern, then every handler for the specified event type is removed. You could remove all the `FocusLost` handlers in the `Unfocussed` group like so:

```
autocmd!  Unfocussed  FocusLost
```

If you also leave out the event name, then every event handler in the specified group is removed. So, to turn off all event handling specified in the `Unfocussed` group:

```
autocmd!  Unfocussed
```

Finally, if you omit the group name, the autocommand removal applies to the currently active group. The typical use of this option is to "clear the decks" within a group before setting up a series of autocommands. For example, the `Unfocussed` group is better specified like so:

Listing 8. Making sure a group is empty before adding new autocommands

```
augroup Unfocussed
    autocmd!

    autocmd  FocusLost  *.txt   :call Autosave()
    autocmd  FocusLost  *.p[ly] :call Checkpoint_sourcecode()
    autocmd  FocusLost  *.doc   :call Reformat_current_para()
augroup END
```

Adding an `autocmd!` to the start of every group is important because autocommands do not statically declare event handlers; they dynamically create them. If you execute the same `autocmd` twice, you get two event handlers, both of which will be separately invoked by the same combination of event and filename from that point onward. By starting each autocommand group with an `autocmd!`, you wipe out any existing handlers within the group so that subsequent `autocmd` statements replace any existing handlers, rather than augmenting them. This, in turn, means that your script can be executed as many times as necessary (or your `.vimrc` can be `source`'d repeatedly) without multiplying event-handling entities unnecessarily.

Some practical examples

The appropriate use of autocommands can make your editing life vastly easier. Let's look at a few ways you can use autocommands to streamline your editing process and remove existing frustrations.

Managing simultaneous edits

One of the most useful features of Vim is that it automatically detects when you attempt to edit a file that is currently being edited by some other instance of Vim. That often happens in multi-window environments, where you're already editing a file in another terminal; or in multi-user setups, where someone else is already working on a shared file. When Vim detects a second attempt to edit a particular file, you get the following request:

```
Swap file ".filename.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (Q)uit, (A)bort: _
```

Depending on the environment in which you're working, your fingers probably automatically hit one of those options every time, without much conscious thought on your part. For example, if you rarely work on shared files, you probably just hit `q` to terminate the session, and then go hunting for the terminal window where you're already editing the file. On the other hand, if you typically edit shared resources, perhaps your fingers are trained to immediately hit **<ENTER>**, in order to select the default option and open the file read-only.

With autocommands, however, you can completely eliminate the need to see, recognize, and respond to that message, simply by automating the response to the `SwapExists` event that triggers

it. For example, if you never want to edit files that are already being edited elsewhere, you could add the following to your `.vimrc`:

Listing 9. Automatically quitting on simultaneous edits

```
augroup NoSimultaneousEdits
    autocmd!
    autocmd  SwapExists  *  :let v:swapchoice = 'q'
augroup END
```

This sets up an autocommand group, and removes any previous handlers (via the `autocmd!` command). It then installs a handler for the `SwapExists` event on any file (using the universal file pattern: `*`). That handler simply assigns the response `'q'` to the special `v:swapchoice` variable. Vim consults this variable prior to displaying the "`swapfile exists`"message. If the variable has been set, it uses the value as the automatic response and doesn't bother showing the message. So now you'll never see the `swapfile` message; your Vim session will just automatically quit if you try to edit a file that's being edited elsewhere.

Alternately, if you'd prefer always to open already edited files in read-only mode, you can simply change the `NoSimultaneousEdits` group to:

Listing 10. Automating read-only access to existing files

```
augroup NoSimultaneousEdits
    autocmd!
    autocmd  SwapExists  *  :let v:swapchoice = 'o'
augroup END
```

More interestingly, you could arrange to select between these two (or any other) alternatives, based on the location of the file being considered. For example, you might prefer to auto-quit files in your own subdirectories, but open shared files under `/dev/shared/` as read-only. You could do that with the following:

Listing 11. Automating a context-sensitive response

```
augroup NoSimultaneousEdits
    autocmd!
    autocmd  SwapExists  ~/*            :let v:swapchoice = 'q'
    autocmd  SwapExists  /dev/shared/*  :let v:swapchoice = 'o'
augroup END
```

That is: if the full filename begins with the home directory, followed by anything at all (`~/*`), then preselect the "quit" behaviour; but if the full filename starts with the shared directory (`/dev/shared/*`), then preselect the "read-only" behaviour instead.

Autoformatting code consistently

Vim has good support for automatic edit-time code layout (see `:help indent.txt` and `:help filter`). For example, you can turn on the `'autoindent'` and `'smartindent'` options and have Vim re-indent your code blocks automatically as you type. Or you can hook your own language-specific code reformatter to the standard `=` command by setting the `'equalprg'` option.

Unfortunately, Vim doesn't have an option or a command to deal with one of the commonest code-formatting situations: being forced to read someone else's abysmally *malformatted* code. Specifically, there's no built-in option to tell Vim to automatically sanitize the formatting of any code file you open.

That's okay because it's trivially easy to set up an autocommand to do that instead.

For example, you could add the following autocommand group to your `.vimrc`, so that C, Python, Perl, and XML files are automatically run through the appropriate code formatter whenever you open a file of the corresponding type, as shown in Listing 12.

Listing 12. Beautiful code, on autocommand

```
augroup CodeFormatters
    autocmd!

    autocmd  BufReadPost,FileReadPost   *.py    :silent %!PythonTidy.py
    autocmd  BufReadPost,FileReadPost   *.p[lm] :silent %!perltidy -q
    autocmd  BufReadPost,FileReadPost   *.xml   :silent %!xmlpp –t –c –n
    autocmd  BufReadPost,FileReadPost   *.[ch]  :silent %!indent
augroup END
```

All of the autocommands in the group are identical in structure, differing only in the filename extensions to which they apply and the corresponding pretty-printer they invoke.

Note that the autocommands do not name a single event to be handled. Instead, each one specifies a list of events. Any `autocmd` can be specified with a comma-separated list of event types, in which case the handler will be invoked for any of the events listed.

In this case, the events listed for each handler are `BufReadPost` (which is queued whenever an existing file is loaded into a new buffer) and `FileReadPost` (which is queued immediately after any `:read` command is executed). These two events are often specified together because between them they cover the most common ways of loading the contents of an existing file into a buffer.

After the event list, each autocommand specifies the file suffix(es) to which it applies: Python's `.py`, Perl's `.pl` and `.pm`, XML's `.xml`, or the `.c` and `.h` files of C. Note that, as with events, these filename patterns could also have been specified as a comma-separated list, rather than a single pattern. For example, the Perl handler could have been written:

```
autocmd  BufReadPost,FileReadPost   *.pl,*.pm   :silent %!perltidy -q
```

or the C handler could be extended to handle common C++ variants (`.C`, `.cc`, `.cxx`, etc.) as well, like so:

```
autocmd  BufReadPost,FileReadPost   *.[chCH],*.cc,*.hh,*.[ch]xx  :silent %!indent
```

As usual, the final component of each autocommand is the command to be executed. In each case, it is a global filter command (**`%!filter_program`**), which takes the entire contents of the file (`%`) and pipes it out (`!`) to the specified external program (one of: `PythonTidy.py`, `perltidy`, `xmlpp`,

or `indent`). The output of each program is then pasted back into the buffer, replacing the original contents.

Normally, when filter commands like these are used, Vim automatically displays a notification after the command completes, like so:

```
42 lines filtered
Press ENTER or type command to continue_
```

To prevent this annoyance, each of the autocommands prefixes its action with a `:silent`, which neutralizes any ordinary information messages, but still allows error messages to be displayed.

Opportunistic code autoformatting

Vim has excellent support for automatically formatting C code as you type it, but it offers less support for other languages. That's not entirely Vim's fault; some languages -- yes, Perl, I'm looking at you -- can be extremely hard to format correctly on the fly.

If Vim doesn't give you adequate support for autoformatting source code in your preferred language, you can easily have your editor invoke an external utility to do that for you.

The simplest approach is to make use of the `InsertLeave` event. This event is queued whenever you exit from `Insert` mode (most commonly, immediately after you hit `<ESC>`). You can easily set up a handler that reformats your code every time you finish adding to it, like so:

Listing 13. Invoking PerlTidy after every edit

```
function! TidyAndResetCursor ()
    let cursor_pos = getpos('.')
    %!perltidy -q
    call setpos('.', cursor_pos)
endfunction

augroup PerlTidy
    autocmd!
    autocmd InsertLeave *.p[lm]  :call TidyAndResetCursor()
augroup END
```

The `TidyAndResetCursor()` function first makes a record of the current cursor position, by storing the cursor information returned by the built-in `getpos()` in the variable `cursor_pos`. It then runs the external `perltidy` utility over the entire file (`%!perltidy -q`), and finally restores the cursor to its original position, by passing the saved cursor information to the built-in `setpos()` function.

Inside the `PerlTidy` group, you then just set up a single autocommand that calls `TidyAndResetCursor()` every time the user leaves `Insert` mode within any Perl file.

This same code pattern could be adapted to perform any appropriate action each time you insert text. For example, if you were working on a very unreliable system and wished to maximize your ability to recover files (see `:help usr_11.txt`) if something went wrong, you could arrange for Vim to update its swap-file every time you left `Insert` mode, like so:

```
augroup UpdateSwap
    autocmd!
    autocmd  InsertLeave  *  :preserve
augroup END
```

Timestamping files

Another useful set of events are `BufWritePre`, `FileWritePre`, and `FileAppendPre`. These "`Pre`" events are queued just before your Vim session writes a buffer back to disk (as a result of a command such as `:write`, `:update`, or `:saveas`). A `BufWritePre` event occurs just before the entire buffer is written, a `FileWritePre` occurs just before part of a buffer is written (that is, when you specify a range of lines to be written: `:1,10write`). A `FileAppendPre` occurs just before a `:write` command is used to append rather than replace; for example:

```
:write >> logfile.log).
```

For all three types of events, Vim sets the special line-number aliases `'[` and `']` to the range of lines being written. These aliases can then be used in the range specifier of any subsequent command, to ensure that autocommand actions are applied only to the relevant lines.

Typically, you would set up a single handler that covered all three types of pre-writing event. For example, you could have Vim automatically update an internal timestamp whenever a file was written (or appended) to disk, as shown in Listing 14.

Listing 14. Automatically updating an internal timestamp whenever a file is saved

```
function! UpdateTimestamp ()
    '[,']s/^This file last updated: \zs.*/\= strftime("%c") /
endfunction

augroup TimeStamping
    autocmd!

    autocmd BufWritePre,FileWritePre,FileAppendPre  *  :call UpdateTimestamp()
augroup END
```

The `UpdateTimestamp()` function performs a substitution (`s/.../.../`) on every line being written, by specifically limiting the range of the substitution to between `'[` and `']` like so: `'[,']s/.../.../`. The substitution looks for lines starting with "`This file last updated:`", followed by anything (`.*`). The `\zs` before the `.*` causes the substitution to pretend that the match only started after the colon, so only the actual timestamp is replaced.

To update the timestamp, the substitution uses the special `\= escape` sequence in the replacement text. This escape sequence tells Vim to treat the replacement text as a Vimscript expression, evaluating it to get the actual replacement string. In this case, that expression is a call to the built-in `strftime()` function, which returns a standard timestamp string of the form: "`Fri Oct 23 14:51:01 2009`". This string is then written back into the timestamp line by the substitution command.

All that remains is to set up an event handler (`autocmd`) for all three event types (`BufWritePre`,`FileWritePre`,`FileAppendPre`) in any file (`*`) and have it invoke the appropriate timestamping function (`:call UpdateTimestamp()`). Now, any time a file is written, any timestamp in the lines being saved will be updated to the current time.

Note that Vim provides two other sets of events that you can use to modify the behavior of write operations. To automate some action that should happen after a write, you can use `BufWritePost`, `FileWritePost`, and `FileAppendPost`. To completely replace the standard write behavior with your own script, you can use `BufWriteCmd`, `FileWriteCmd`, and `FileAppendCmd` (but consult `:help Cmd-event` first for some important caveats).

Table-driven timestamps

Of course, you could also create much more elaborate mechanisms to handle files with different timestamping conventions. For example, you might prefer to specify the various timestamp signatures and their replacements in a Vim dictionary (see the previous article in this series) and then loop through each pair to determine how the timestamp should be updated. This approach is shown in Listing 15.

Listing 15. Table-driven automatic timestamps

```
let s:timestamps = {
\  'This file last updated: \zs.*'             :  'strftime("%c")',
\  'Last modification: \zs.*'                  :  'strftime("%Y%m%d.%H%M%S")',
\  'Copyright (c) .\{-}, \d\d\d\d-\zs\d\d\d\d' :  'strftime("%Y")',
\}

function! UpdateTimestamp ()
    for [signature, replacement] in items(s:timestamps)
        silent! execute "'[,']s/" . signature . '/\= ' . replacement . '/'
    endfor
endfunction
```

Here, the for loop iterates through each timestamp's signature/replacement pair in the `s:timestamps` dictionary, like so:

```
for [signature, replacement] in items(s:timestamps)
```

It then generates a string containing the corresponding substitution command. The following substitution command is identical in structure to the one in the previous example, but is here constructed by interpolating the signature/replacement pair into a string:

```
"'[,']s/" . signature . '/\= ' . replacement . '/'
```

Finally, it executes the generated command silently:

```
silent! execute "'[,']s/" . signature . '/\= ' . replacement . '/'
```

The use of `silent!` is important because it ensures that any substitutions that don't match will not result in the annoying `Pattern not found` error message.

Note that the last entry in `s:timestamps` is a particularly useful example: it automatically updates the year-range of any embedded copyright notices, whenever a file containing them is written.

Filename-driven timestamps

Instead of listing all possible timestamp formats in a single table, you might prefer to parameterize the `UpdateTimestamp()`function and then create a series of distinct `autocmds` for different filetypes, as shown in Listing 16.

Listing 16. Context-sensitive timestaming for different filetypes

```
function! UpdateTimestamp (signature, replacement)silent! execute "'[,']s/" . a:signature . '/\= ' .
 a:replacement . '/'
endfunction

augroup Timestamping
    autocmd!

    " C header files use one timestamp format ... autocmd BufWritePre,FileWritePre,FileAppendPre  *.h
        \ :call UpdateTimestamp('This file last updated: \zs.*', 'strftime("%c")')" C code files use
 another ...autocmd BufWritePre,FileWritePre,FileAppendPre  *.c
        \ :call UpdateTimestamp('Last update: \zs.*', 'strftime("%Y%m%d.%H%M%S")')
augroup END
```

In this version, the signature and replacement components are passed explicitly to `UpdateTimestamp()`, which then generates a string containing the single corresponding substitution command and executes it. Within the `Timestamping` group, you then set up individual autocommands for each required file type, passing the appropriate timestamp signature and replacement text for each.

Conjuring directories

Autocommands can be useful even before you begin editing. For example, when you start editing a new file, you will occasionally see a message like this one:

```
"dir/subdir/filename" [New DIRECTORY]
```

This means that the file you specified (in this case `filename`) does not exist and that the directory it's supposed to be in (in this case `dir/subdir`) doesn't exist either.

Vim will happily allow you to ignore this warning (many users don't even recognize that it is a warning) and continue to edit the file. But when you try to save it you'll be confronted with the following unhelpful error message:

```
"dir/subdir/filename" E212: Can't open file for writing.
```

Now, in order to save your work, you have to explicitly create the missing directory before writing the file into it. You can do that from within Vim like so:

```
:write
"dir/subdir/filename" E212: Can't open file for writing.
:call mkdir(expand("%:h"),"p")
:write
```

Here, the call to the built-in `expand()` function is applied to `"%:h"`, where the `%` means the current filepath (in this case `dir/subdir/filename`), and the `:h` takes just the "head" of that path, removing the filename to leave the path of the intended directory (`dir/subdir`). The call to Vim's built-in `mkdir()` then takes this directory path and creates all the interim directories along it (as requested by the second argument, `"p"`).

Realistically, though, most Vim users would be more likely to simply escape to the shell to build the necessary directories. For example:

```
:write
"dir/subdir/filename" E212: Can't open file for writing.
:! mkdir -p dir/subdir/
:write
```

Either way, it's a hassle. If you're eventually going to have to create the missing directory anyway, why not have Vim notice up-front that it doesn't exist, and simply create it for you before you even start? That way, you'll never encounter the obscure [`New DIRECTORY`] hint; nor will your workflow be later interrupted by an equally mysterious `E212` error.

To have Vim take care of prebuilding non-existent directories, you could hook a handler into the `BufNewFile` event, which is queued whenever you start to edit a file that does not yet exist. Listing 17 shows the code you would add to your `.vimrc` file to make this work.

Listing 17. Unconditionally autocreating non-existent directories

```
augroup AutoMkdir
    autocmd!
    autocmd  BufNewFile  *  :call EnsureDirExists()
augroup END
function! EnsureDirExists ()
    let required_dir = expand("%:h")
    if !isdirectory(required_dir)
        call mkdir(required_dir, 'p')
    endif
endfunction
```

The `AutoMkdir` group sets up a single autocommand for `BufNewFile` events on any kind of file, calling the `EnsureDirExists()` function whenever a new file is edited. `EnsureDirExists()` first determines the directory being requested by expanding the "head"of the current filepath: `expand("%:h")`. It then uses the built-in `isdirectory()` function to check whether the requested directory exists. If not, it attempts to create the directory using Vim's built-in `mkdir()`.

Note that, if the `mkdir()` call can't create the requested directory for any reason, it will produce a slightly more precise and informative error message:

```
E739: Cannot create directory: dir/subdir
```

Conjuring directories more carefully

The only problem with this solution is that, occasionally, autocreating non-existent subdirectories is exactly the wrong thing to do. For example, suppose you requested the following:

```
> vim /share/sites/corporate/root/.htaccess
```

You had intended to create a new access control file in the already existing subdirectory `/share/corporate/website/root/`. Except, of course, because you got the path wrong, what you actually did was create a new access control file in the formerly non-existent subdirectory `/share/website/corporate/root/`. And because that happened automatically, with no warnings of any kind, you might not even realize the mistake. At least, not until the misapplied access control precipitates some online disaster.

To guard against errors like this, you might prefer that Vim be a little less helpful in autocreating missing directories. Listing 18 shows a more elaborate version of `EnsureDirExists()`, which still detects missing directories but now asks the user what to do about them. Note that the autocommand set-up is exactly the same as in Listing 17; only the `EnsureDirExists()` function has changed.

Listing 18. Conditionally autocreating non-existent directories

```
augroup AutoMkdir
    autocmd!
    autocmd  BufNewFile  *  :call EnsureDirExists()
augroup END
function! EnsureDirExists ()
    let required_dir = expand("%:h")
    if !isdirectory(required_dir)
        call AskQuit("Directory '" . required_dir . "' doesn't exist.", "&Create it?")

        try
            call mkdir( required_dir, 'p' )
        catch
            call AskQuit("Can't create '" . required_dir . "'", "&Continue anyway?")
        endtry
    endif
endfunction

function! AskQuit (msg, proposed_action)
    if confirm(a:msg, "&Quit?\n" . a:proposed_action) == 1
        exit
    endif
endfunction
```

In this version of the function, `EnsureDirExists()` locates the required directory and detects whether it exists, exactly as before. However, if the directory is missing, `EnsureDirExists()` now calls a helper function: `AskQuit()`. This function uses the built-in `confirm()` function to inquire whether you want to exit the session or autocreate the directory. `"Quit?"` is presented as the first option, which also makes it the default if you just hit `<ENTER>`.

If you do select the `"Quit?"`option, the helper function immediately terminates the Vim session. Otherwise, the helper function simply returns. In that case, `EnsureDirExists()` continues to execute, and attempts to call `mkdir()`.

Note, however, that the call to `mkdir()` is now inside a `try...endtry` construct. This is -- as you might expect -- an exception handler, which will now catch the `E739` error that is thrown if `mkdir()` is unable to create the requested directory.

When that error is thrown, the `catch` block will intercept it and will call `AskQuit()` again, informing you that the directory could not be created, and asking whether you still want to continue. For more details on Vim's extensive exception handling mechanisms see: `:help exception-handling`.

The overall effect of this second version of `EnsureDirExists()` is to highlight the non-existent directory but require you to explicitly request that it be created (by typing a single `'c'` when prompted to). If the directory cannot be created, you are again warned and given the option of continuing with the session anyway (again, by typing a single `'c'` when asked). This also makes it trivially easy to escape from a mistaken edit (simply by hitting `<ENTER>` to select the default `"Quit?"` option at either prompt).

Of course, you might prefer that continuing was the default, in which case, you would just change the first line of `AskQuit()` to:

```
if confirm(a:msg, a:proposed_action . "\n&Quit?") == 2
```

In this case the proposed action would be the first alternative, and hence the default behaviour. Note that `"Quit?"` is now the second alternative, so the response now has to be compared against the value 2.

Looking ahead

Autocommands can save you a great deal of effort and error by automating repetitive actions that you would otherwise have to perform yourself. A productive way to get started is to take a mental step back as you edit and watch for repetitive patterns of usage that might be suitably automated using Vim's event-handling mechanisms. Scripting those patterns into autocommands may require some extra work up front, but the automated actions will repay your investment every day. By automating everyday actions you'll save time and effort, avoid errors, smooth your workflow, eliminate trivial stressors, and thereby improve your productivity.

Though your autocommands will probably start out as simple single-line automations, you may soon find yourself redesigning and elaborating them, as you think of better ways to have Vim do more of your grunt work. In this fashion, your event handlers can grow progressively smarter, safer, and more perfectly adapted to the way you want to work.

As Vim scripts like these become more complex, however, you also will need better tools to manage them. Adding 10 or 20 lines to your `.vimrc` every time you devise a clever new keymapping or autocommand will eventually produce a configuration file that is several thousand lines long ... and utterly unmaintainable.

So, in the next article in this series we'll explore Vim's simple plug-in architecture, which allows you to factor out parts of your `.vimrc` and isolate them in separate modules. We'll look at how that plug-in system works by developing a standalone module that ameliorates some of the horrors of working with XML.

Related topics

- Start learning about Vimscript, the embedded language for extending the Vim editor, with the first article in this series: "Scripting the Vim editor, Part 1: Variables, values, and expressions" (developerWorks, May 2009).
- See the following resources to continue learning about the Vim editor and its many commands:
  - The Vim homepage
  - The online book A Byte of Vim
  - Various hardcopy books on Vim
  - Vim's own manual
  - Steve Oualline's Vim Cookbook
- For more extensive examples of Vim scripting, see:
  - The Vim Tips wiki
  - The Vimscript archive
- Start at the Vim distributions downloads page to upgrade to the latest version of Vim for your platform.
- In the developerWorks Linux zone, find hundreds of articles, tutorials, discussion forums, and a wealth other resources for Linux developers and administrators.
- Evaluate IBM products in the way that suits you best: Download a product trial, try a product online, use a product in a cloud environment, or spend a few hours in the SOA Sandbox learning how to implement Service Oriented Architecture efficiently.