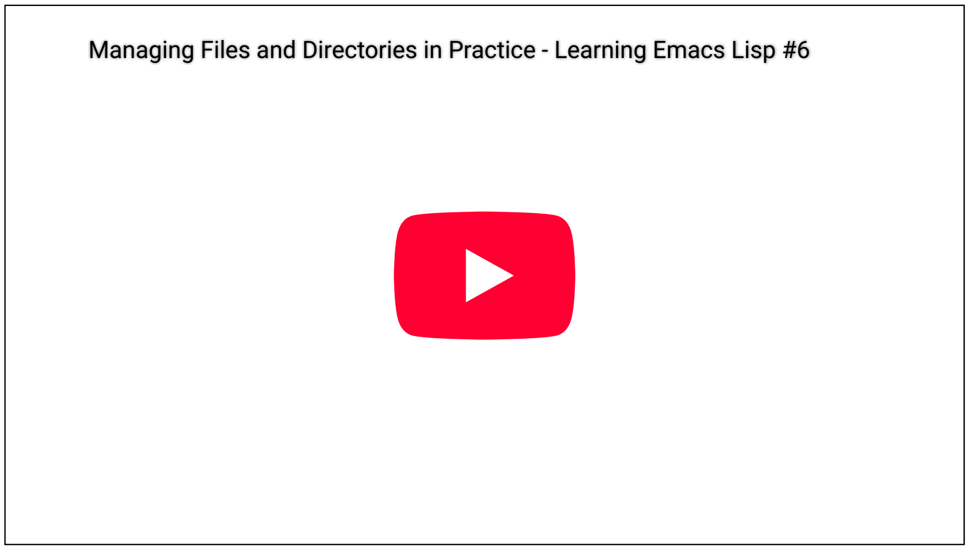
Managing Files and Directories in Practice


Managing Files and Directories in Practice - Learning Emacs Lisp #6

¶ What will we cover?

Today we're going to talk about the many ways you can interact with files and directories using Emacs Lisp!

There are a few different ways to do this with user-facing Emacs functionality (`find-file`, `eshell`, Dired, etc), but at some point you will want to automate some of these tasks with Emacs Lisp code.

Today I'll show you a wide range of the functions you have at your disposal and then use them in real code examples!

The final code from this episode can be found on GitHub.

¶ Explaining symbolic links

But first, what is a symbolic link?

It is an entry in the file system that points to a file or directory somewhere else in the file system. It is used to make it appear like a directory exists at `~/.emacs.d` when it actually exists at `~/.dotfiles/.emacs.d`!

¶ Our project: dotcrafter.el

As we talked about in previous videos in this series, we are working on a dotfiles management package for Emacs.

In this video we're going to add functionality to create symbolic links for your configuration files into the real locations where they belong in the home directory!

Some context:

- We've been focused on dotfiles generated from Org Mode files
- You'll probably also have configuration files that don't belong in Org files!
- We need a way to automatically link those configuration files into the home directory
- We will implement code to "mirror" your configuration files into the home directory using symbolic links

By the end of this episode, we'll have a fully working dotfiles management package!

We'll commit improvements made in this video to the GitHub repository: https://github.com/daviwil/dotcrafter.el

The starting point of the examples in this episode is this version of dotcrafter.el from the repository.

> If you find this guide helpful, please consider supporting System Crafters via the links on the How to Help page!

¶ Getting the current directory

Emacs will resolve most file paths relative to the current directory which is determine by the variable `default-directory`. This buffer-local variable will be different for each buffer you open.

For file buffers, it will contain the directory where the buffer's file lives:

```
default-directory
```

What about the `*scratch*` buffer? It *probably* returns the directory where Emacs was launched from!

You can also change `default-directory` if necessary!

Emacs Manual: File Names

¶ Manipulating file paths

When you want to automate file operations in Emacs, you'll often need to grab different parts of a path so that you can build new paths. There are a few functions for this purpose!

In Emacs, file paths are considered to have two parts:

- The directory part
- The non-directory part, i.e. the file name and its extension

The functions you will want to use for this purpose are all prefixed with `file-name`!

- `file-name-directory`: Get the directory part of a file path
- `file-name-nondirectory`: Get the filename (non-directory) part of a file path
- `file-name-extension`: Get the extension of the file (without the leading period .)
- `file-name-sans-extension`: Get the path without the file extension
- `file-name-base`: Get the base file name without path or extension
- `file-name-as-directory`: Turn the file name into a directory name

NOTE: The file paths you pass to these functions do not have to exist!

```
(file-name-directory (buffer-file-name))
```

```
(file-name-nondirectory (buffer-file-name))
```

```
(file-name-extension (buffer-file-name))
```

```
(file-name-sans-extension (buffer-file-name))
```

```
(file-name-base (buffer-file-name))

(file-name-as-directory (buffer-file-name))

(file-name-as-directory
 (file-name-sans-extension (buffer-file-name)))
```

Emacs Lisp Manual: File Names

¶ Resolving file paths

It is a good idea to resolve file paths any time you use them to ensure they are being used for the location you expect!

- `file-name-absolute-p` will tell you whether a file name is "absolute": it contains a complete file system path
- `file-relative-name` with give you the path of a file relative to another path
- `expand-file-name` will return an absolute path for a file under a specified directory

```
(file-name-absolute-p (buffer-file-name))      ;; t
(file-name-absolute-p "Emacs-Lisp-06.org")     ;; nil
(file-name-absolute-p "dir/Emacs-Lisp-06.org") ;; nil

(file-relative-name (buffer-file-name) "~/Notes")     ;; Streams/Emacs-Lisp-06.org
(file-relative-name (buffer-file-name) "~/.dotfiles")
    ;; ../Notes/Streams/Emacs-Lisp-06.org

(expand-file-name "Emacs-Lisp-06.org")
    ;; /home/daviwil/Notes/Streams/Emacs-Lisp-06.org

;; The file doesn't have to exist!
(expand-file-name "Emacs-Lisp-06.org" "~/.dotfiles")
    ;; /home/daviwil/.dotfiles/Emacs-Lisp-06.org
```

What about resolving paths containing environment variables?

```
(expand-file-name "$HOME/.emacs.d")
(substitute-in-file-name "$HOME/.emacs.d")
```

Emacs Lisp Manual: Absolute and Relative File Names Emacs Lisp Manual: Functions that Expand Filenames

¶ Example: Resolving the destination path of a configuration file

We can use a few of the functions we just discussed to find where a file inside of the dotfiles folder should be linked in the home directory!

Here's what we need to do:

- Resolve the relative path of a file under the dotfiles folder relative to the dotfiles folder
- Resolve that relative path against the home directory (or more specifically the output directory)

For example:

```
~/.dotfiles/.files/

~/.dotfiles/.files/.local/share/applications/Emacs.desktop

  Resolve to ⟶   .local/share/applications/Emacs.desktop
  Resolve to ⟶ ~/.local/share/applications/Emacs.desktop
```

We're also going to define a variable that holds the specific subpath of the dotfiles folder where these linked configuration files should live so that they're easier to manage.

```
(defcustom dotcrafter-dotfiles-folder "~/.dotfiles"
  "The folder where dotfiles and org-mode configuration files are stored."
  :type 'string
  :group 'dotfiles)

(defcustom dotcrafter-output-directory "~"
  "The directory where dotcrafter.el will write out your dotfiles.
This is typically set to the home directory but can be changed
for testing purposes."
  :type 'string
  :group 'dotfiles)

(defcustom dotcrafter-config-files-directory ".files"
  "The directory path inside of `dotcrafter-dotfiles-folder' where
configuration files that should be symbolically linked are stored."
  :type 'string
  :group 'dotfiles)

(setq dotcrafter-dotfiles-folder "~/Projects/Code/dotcrafter.el/example")
(setq dotcrafter-output-directory "~/Projects/Code/dotcrafter.el/demo-output")

(defun dotcrafter--resolve-config-files-path ()
  (expand-file-name dotcrafter-config-files-directory
                    dotcrafter-dotfiles-folder))

(defun example--resolve-config-file-target (config-file)
  (expand-file-name
   (file-relative-name
    (expand-file-name config-file)
    (dotcrafter--resolve-config-files-path))
   dotcrafter-output-directory))

(example--resolve-config-file-target "~/Projects/Code/dotcrafter/example/.files/.emacs.d/init.el")
```

¶ Checking if files and directories exist

The `file-exists-p` function returns `t` if a file or directory exists or `nil` otherwise:

```
(file-exists-p "~/.dotfiles/.emacs.d")  ;; t
(file-exists-p "~/.dotfiles/foobar")    ;; nil
```

There are a few more functions that you can use to check if the user has access to a file, whether its writable or executable, etc:

- `file-readable-p`
- `file-executable-p`
- `file-writable-p`

Emacs Lisp Manual: Testing Accessibility

¶ Creating directories

You can easily create a directory with the `make-directory` command.

The first parameter is the path to the directory to be created and the second is an optional boolean (`t` or `nil`) which determines whether any missing parent directories in the path should also be created.

You can also set the second parameter to `t` to ensure that `make-directory` won't throw an error if the directory already exists!

```
(make-directory "~/.local/share/foobar")
(make-directory "~/.local/share/foobar")   ;; throws an error
(make-directory "~/.local/share/foobar" t) ;; no error!

(make-directory "~/.local/share/hello/system/crafters")   ;; error
(make-directory "~/.local/share/hello/system/crafters" t) ;; success!
```

Emacs Lisp Manual: Creating, Copying, and Deleting Directories

¶ Example: Creating expected directories before linking

When we begin creating symbolic links into the home directory, one thing we will need to be careful of is creating symbolic links too close to the home directory for commonly-used folders like `~/.config` or `~/.local/share`.

What we want to avoid is creating a symlink for these folders to our dotfiles folder and then having a bunch of unwanted files show up there that we must add to our `.gitignore`!

The solution here is to make sure that these directories already exist so that the algorithm we will write later won't try to create symbolic links instead. To accomplish this, we will create a new variable to hold the list of directories to be pre-created and then create those directories before we start the linking process:

```
(defcustom dotcrafter-ensure-output-directories '(".config" ".local/share")
  "List of directories in the output folder that should be created
before linking configuration files."
  :type  '(list string)
  :group 'dotfiles)

(defun example--ensure-output-directories ()
  ;; Ensure that the expected output directories are already
  ;; created so that links will be created inside
  (dolist (dir dotcrafter-ensure-output-directories)
    (make-directory (expand-file-name dir dotcrafter-output-directory) t)))

(example--ensure-output-directories)
```

¶ Listing files in directories

One thing you will probably want to do at some point is get a list of files in a given directory, possibly even for all child directories under that path as well.

The `directory-files` and `directory-files-recursively` functions are great for this purpose!

```
(directory-files "~/.dotfiles")
(directory-files "~/.dotfiles" t)           ;; Return full file paths
(directory-files "~/.dotfiles" t ".org")    ;; Get all file containing ".org"
(directory-files "~/.dotfiles" t "" t)      ;; Don't sort results
(directory-files "~/.dotfiles" t "" nil 3) ;; Maximum 3 results

(directory-files-recursively "~/.dotfiles" "\\.el$")
(directory-files-recursively dotcrafter-output-directory "")
(directory-files-recursively dotcrafter-output-directory "" t)

;; The fourth parameter can be a function that determines whether
;; a path can be traversed using any logic!
(directory-files-recursively "~/.emacs.d" "" nil
                             (lambda (dir)
                               (string-equal dir "~/.emacs.d/lisp")))

(directory-files-recursively "~/.config" "\\.scm" t nil nil) ;; Doesn't follow symlinks
(directory-files-recursively "~/.config" "\\.scm" t nil t)   ;; Follows symlinks!
```

Emacs Lisp Manual: Contents of Directories

¶ Example: Finding the list of all configuration files to be linked

As we talked about earlier, the goal of what we're doing today is to produce some code that will mirror a folder of configuration files in your dotfiles folder into the home folder using symbolic links.

We'll use the `directory-files-recursively` function to list all of the linkable files under the dotfiles path and then resolve them relative to the output path!

```
(defun example--find-all-files-to-link ()
  (let ((files-to-link
         (directory-files-recursively
          (dotcrafter--resolve-config-files-path)
          "")))
    (dolist (file files-to-link)
      (message "File: %s\n   - %s" file (example--resolve-config-file-target file)))))

(example--find-all-files-to-link)
```

¶ Copying, moving, and deleting files and directories

You can perform common file management tasks like copying, moving, and deleting files and directories with a few different Emacs Lisp functions.

Emacs Lisp Manual: Copying, Naming, and Renaming Files Emacs Lisp Manual: Creating, Copying, and Deleting Directories

¶ Copying

- `copy-file`: Copy the contents of one file to another

- `copy-directory`: Copy the contents of one directory to another, including all subdirectories

```
(copy-file "~/.emacs.d/init.el" "/tmp")  ;; Must end in a slash!
(copy-file "~/.emacs.d/init.el" "/tmp/")   ;; Copied to /tmp
(copy-file "~/.emacs.d/init.el" "/tmp/")   ;; Error, already exists!
(copy-file "~/.emacs.d/init.el" "/tmp/" t) ;; No error!
;; The remaining parameters are all about preserving file metadata

(copy-directory "~/.emacs.d/lisp" "/tmp")  ;; Must end in a slash!
(copy-directory "~/.emacs.d/lisp" "/tmp/") ;; Copied to /tmp/lisp

;; To copy the contents of the directory without the enclosing directory:
(copy-directory "~/.emacs.d/eshell" "/tmp/lisp" t t nil)
(copy-directory "~/.emacs.d/eshell" "/tmp/lisp" t t t)
```

¶ Renaming / Moving

- `rename-file`: Rename a file or directory

```
(rename-file "/tmp/init.el" "/tmp/init-new.el") ;; Rename file in same folder
(rename-file "/tmp/init-new.el" "~/.emacs.d/")  ;; Move file to different folder
(rename-file "~/.emacs.d/init-new.el" "~/.emacs.d/init.el")   ;; Error!
(rename-file "~/.emacs.d/init-new.el" "~/.emacs.d/init.el" t) ;; OK

;; It can also rename or move directories!
(rename-file "/tmp/lisp" "/tmp/lisp-two") ;; OK
(rename-file "/tmp/lisp-two" "/tmp/lisp") ;; OK
```

¶ Deleting

- `delete-file`: Delete a file, optionally moving it to the trash folder
- `delete-directory`: Deletes a directory, including files if desired

```
(delete-file "/tmp/lisp/dw-desktop.el")
(delete-file "~/.npmrc" t)

(delete-directory "/tmp/lisp")
(delete-directory "/tmp/lisp" t)
```

¶ Example: Migrating configuration files to the dotfiles folder

As we continue building our configurations, it's likely that we'll want to migrate a configuration folder into our dotfiles repository. Let's define a function that will make this really easy for the user:

- The user chooses a folder to move into their dotfiles configuration
- We ensure that the chosen file is located under the home directory (`dotcrafter-output-directory`)
- If so, move the file to the corresponding location under the config path

```
;; Run this to feed the demo!
(copy-directory "~/.dotfiles/.config/guix"
                (file-name-as-directory (expand-file-name ".config"
                                                          dotcrafter-output-directory)))
(copy-file "~/.dotfiles/.bash_profile"
           (file-name-as-directory dotcrafter-output-directory))

(defun dotcrafter-move-to-config-files (source-path)
  "Move a file from the output path to the configuration path."
  (interactive "FConfiguration path to move: ")
  (let* ((relative-path (file-relative-name (expand-file-name source-path)
                                            dotcrafter-output-directory))
         (dest-path (expand-file-name relative-path
                                      (dotcrafter--resolve-config-files-path)))
         ;; Strip any trailing slash so that we can treat the directory as file
         (dest-path (if (string-suffix-p "/" dest-path)
                        (substring dest-path 0 -1)
                      dest-path)))
         ;; Make sure that the path is under the output directory and that it
         ;; doesn't already exist
         (when (string-prefix-p ".." relative-path)
           (error "Copied path is not inside of config output directory: %s" dotcrafter-output-dire
         (when (file-exists-p dest-path)
           (error "Can't copy path because it already exists in the configuration directory: %s" de

         ;; Ensure that parent directories exist and then move the file!
```

```
          (make-directory (file-name-directory dest-path) t)
          (rename-file source-path dest-path)))

      ;; TODO: Link this path back into the dotcrafter-output-directory
```

¶ Creating symbolic links

Using symbolic links, we're able to keep our configuration files in a local Git repository and then make them appear in our home folder.

Creating symbolic links is very easy in Emacs with the `make-symbolic-link` function:

```
(make-symbolic-link "~/.dotfiles/.config/guix" "~/.config/guix")   ;; Error if exists
(make-symbolic-link "~/.dotfiles/.config/guix" "~/.config/guix" t) ;; No error!
```

However, this doesn't work exactly the same on Windows! You might need to run Emacs with elevation for it to work.

You can also check if a file is a symbolic link using `file-symlink-p` and get the path it points to using `file-truename`:

```
(file-symlink-p "~/.emacs.d")         ;; .dotfiles/.emacs.d
(file-symlink-p "~/.emacs.d/init.el") ;; nil
(file-truename "~/.emacs.d/init.el")  ;; /home/daviwil/.dotfiles/.emacs.d/init.el
```

Emacs Lisp Manual: Copying, Naming, and Renaming Files

¶ Example: Creating symbolic links for all configuration files

Here's where everything in this episode finally comes together!

We're going to implement a more elaborate algorithm that will create symbolic links at the optimal level in the home directory so that we don't need to create a link for every single file.

If you've ever used GNU Stow, this will look pretty familiar!

¶ The Process

This is what we'll do:

- Loop over all files in `dotcrafter-config-files-directory`
- For each file, break the path into pieces for each directory up to the filename
- For each piece of the file's path, check if the folder exists
- If it exists, check if it's a symbolic link that points to the matching directory in the config folder
- If it doesn't exist, create the symlink there

Here's a clearer depicton of what this means:

```
~/.dotfiles/.files/.local/share/applications/Emacs.desktop
                 ~/.local/share/applications/Emacs.desktop
                  L .local exists? YES
                         L share exists? YES
                               L applications exists? NO, create link!
```

¶ The Code

Let's walk through the code line by line before running it!

- `dotcrafter-link-config-files`: The user-facing function that links the whole config directory
- `dotcrafter-link-config-file`: The "internal" function that handles linking a single file

```
(defun dotcrafter--link-config-file (config-file)
  ;; Get the "path parts", basically the name of each directory and file in the
  ;; path of config-file
  (let* ((path-parts
          (split-string (file-relative-name (expand-file-name config-file)
                                            (dotcrafter--resolve-config-files-path))
                        "/" t))
         (current-path nil))
    ;; Check each "part" of the path to find the right place to create the symlink.
    ;; Whenever path-parts is nil, stop looping!
    (while path-parts
      ;; Create the current path using the first part and remove it from the
      ;; front of the list for future iterations
      (setq current-path (if current-path
                             (concat current-path "/" (car path-parts))
                           (car path-parts)))
      (setq path-parts (cdr path-parts))

      ;; Figure out whether the current source path can be linked to the target path
      (let ((source-path (expand-file-name (concat dotcrafter-config-files-directory "/" current-pa
                                            dotcrafter-dotfiles-folder))
            (target-path (expand-file-name current-path dotcrafter-output-directory)))
        ;; If the file or directory exists, is it a symbolic link?
        (if (file-symlink-p target-path)
            ;; If the symbolic link exists, does it point to the source-path?
            (if (string-equal source-path (file-truename target-path))
                ;; Clear path-parts to stop looping
                (setq path-parts '())
              (error "Path already exists with different symlink! %s" target-path))
          ;; If the target path is an existing directory, we need to keep
          ;; looping, otherwise we can create a symlink here!
          ;; Otherwise, the file is probably a directory so keep looping
          (when (not (file-directory-p target-path))
            ;; Create a symbolic link to the source-path and
            ;; clear the path-parts so that we stop looping
            (make-symbolic-link source-path target-path)
            (setq path-parts '())))))))

(defun dotcrafter-link-config-files ()
  (interactive)
  (let ((config-files
         (directory-files-recursively
          (dotcrafter--resolve-config-files-path)
          "")))
    ;; Ensure that the expected output directories are already
    ;; created so that links will be created inside
    (dolist (dir dotcrafter-ensure-output-directories)
      (make-directory (expand-file-name dir dotcrafter-output-directory) t))

    ;; Link all of the source config files to the output path
    (dolist (file config-files)
      (dotcrafter--link-config-file file))))
```

¶ The final code in action

One last piece will bring together everything we've done in the past few episodes is this function:

```
(defun dotcrafter-update-dotfiles ()
  "Generate and link configuration files to the output directory.

This command handles the full process of \"tangling\" Org Mode
files containing configuration blocks and creating symbolic links
to those configuration files in the output directory, typically
the user's home directory."
  (interactive)
  (dotcrafter-tangle-org-files)
  (dotcrafter-link-config-files)
  (dotcrafter--update-gitignore))
```

This will tangle all of our Org configuration files, link all output files to the home directory, and update the `.gitignore` to ignore any of the generated files in the repo.

Let's try it all out!

```
emacs -Q --batch -l demo.el
```

We can also run this function multiple times and it will work just fine!

¶ What's next?

Now that we've got a functioning package, it's time to take things to the next level by creating major and minor modes for it!

In the next episode, I'll show you how to create a minor mode to gracefully handle automatic Org file tangling.

In the following episodes, we'll create a major mode that provides a user interface for the package and then start polishing it up to be published on MELPA!