Increasing development productivity with repository management

kalis.me/increasing-development-productivity-repository-management

November 18, 2017

In my previous article I discussed the merits of having dotfiles installed on your system. We went through all the steps of getting your own dotfiles up and running, but there were still parts left untouched. We discussed what to do with your installed packages and your application preferences, but what happens to your code projects? Do you have to manually clone all your repos? Of course not. This is where repository management comes in.

Setting up repository management

The repository management structure we're going to be discussing brings improvements in three different aspects, which we will go through one by one.

1. Automating repository setup

In the same spirit as the earlier article about dotfiles, automating repository setup makes sure that you will have access to your code straight after setting up a new computer, without any manual labour.

The process for setting this up is similar to setting up package installation. We will have different lists containing links to remote git repositories, with optionally a custom name. These lists can be used for different categories (I have a list for work repos, and one for personal projects for example).

```
#! /usr/bin/env sh

DIR=$(dirname "$0")
cd "$DIR"

COMMENT=\#*
REPO_PATH=$(realpath ~/repos)

find * -name "*.list" | while read fn; do
    folder="${fn%.*}"
    mkdir -p "$REPO_PATH/$folder"
    while read repo; do
        if [[ $repo == $COMMENT ]];
        then continue; else
            pushd "$REPO_PATH/$folder"
            git clone $repo
            popd
        fi
    done < "$fn"
done
```

In the script, we loop through all list files, and read every repository. A folder is created in `~/repos/` for every list (e.g. `~/repos/personal`, `~/repos/work`), and every instance in the list is cloned into their corresponding folder. So my dotfiles repository will be cloned into `~/repos/personal/dotfiles`. You can add any number of list files, with any number of repositories inside.

By adding this setup script to your dotfiles, you can be sure that all your repositories are there whenever you set up a new computer.

2. Easy repository navigation

With your repositories added to your dotfiles, and installed in a common location, with a common structure (`~/repos/$category_folder/$repository_folder`), it is already a lot easier to navigate to them. But manually navigating can still get tedious if you tend to switch repos often. This is where the `repo` shell function comes in.

```
function repo
    set -l repo_base ~/repos
    set -l repo_path (find "$repo_base" -mindepth 2 -maxdepth 2 -type d -name "*$argv*" | head -n 1)
    if not test "$argv"; or not test "$repo_path"
        cd "$repo_base"
    else
        echo "found $repo_path"
        cd "$repo_path"
    end
end
```

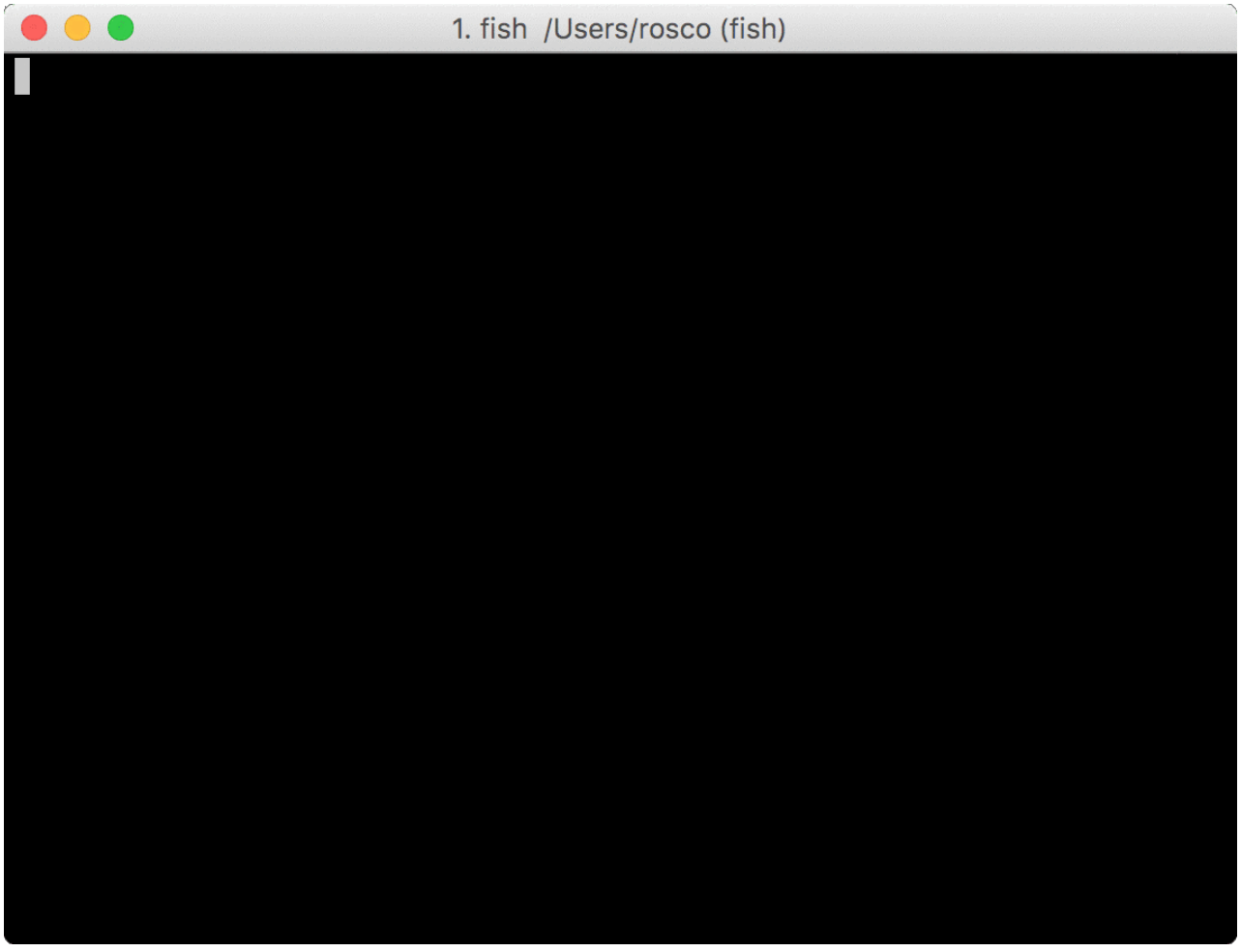
I created the `repo.fish` function, and added it to the Fish configuration inside my dotfiles. The function takes one argument, for which it searches for directories inside all subfolders of the root repo folder (`~/repos`), and returns the first result, if there is one. So `repo dotfiles` or even `repo dot` will result in navigating to `~/repos/personal/dotfiles` , and `repo kalis` will navigate to `~/repos/personal/kalis.me` .

This function is written specifically for the Fish shell, but since it mainly uses other command line tools such as `find` and `head` , porting it to a different shell flavour won't be too difficult.

To complete the navigational experience, the function needs some custom shell completions, of course. Luckily, Fish makes creating custom completions as easy as adding a `repo.fish` file inside the `completions/` folder.

```
for repo in (find ~/repos -mindepth 2 -maxdepth 2 -type d)
    complete -f -c repo -a (basename "$repo")
end
```

This loops through all existing repository folders, and adds them to the completions for the `repo` function using the built-in `complete` command. Entering `repo d` could then result in these completions:



3. Bulk repository actions

We have seen that organising your repositories in the ways above can make it very easy to navigate between repos, but there are more advantages to such a setup. Since we know where all repositories reside, we can easily loop through them all, and perform bulk actions on them.

```
function forrepos --description 'Evaluates $argv for all repo folders'
    for d in (find ~/repos -mindepth 2 -maxdepth 2 ! -path . -type d)
        pushd $d
        set repo (basename $d)
        echo $repo
        eval (abbrex $argv)
        popd > /dev/null
    end
end
```

This function loops through all repository folders, navigates into them, and evaluates the passed arguments. Note that this function uses my own `abbrex` to expand fish abbreviations. This makes it possible to use abbreviations as well as actual functions, since `eval` doesn't automatically evaluate these abbreviations otherwise. The `forrepos` function makes it especially easy to execute git commands for all repos as I have fish abbreviations set for most common git commands (e.g. `gs` maps to `git status`, `gf` to `git fetch`, and `gm` to `git merge FETCH_HEAD --ff-only`).

Executing `forrepos gs -sb` will grant a quick overview of the status of all repositories, and `forrepos gf` will fetch changes from all remote repos.

Conclusion

As we have seen in the steps above, the advantages to setting up repository management, with the corresponding utility functions, come in three parts.

The first advantage lies in the automation of system setup, which builds upon the concepts introduced in my dotfiles article.

Because the automatic repo setup clones the repos into specific folders, they are easily found or iterated over. This makes it possible to quickly navigate between code repositories on your file system. While this may seem trivial at first, if you navigate between repositories often - like many software developers do - it will definitely pay off in the long run to automate this.

Adding the `forrepos` function to your workflow will make it possible to quickly check the status of your projects by running `forrepos gs -sb` or periodically update your repositories by running `forrepos gf` and `forrepos gm`.

Integrating these steps into my everyday development workflow has helped increase my productivity by quite a margin, and I hope they can help you too.

If you enjoyed reading this article, please consider sharing it on Facebook, Twitter or LinkedIn, or giving my dotfiles repository, which includes my repository management setup, a ☆ on GitHub.

Let me know if you integrated repository management into your own development workflow in the comments below!