Creating a GitHub Dotfiles Repository

Centralizing shell, git, and editor configurations for portable, version-controlled development environments

Ronald 'Ryy' G. Thomas

2026-02-11

Table of contents

1 Introduction 2
- 1.1 Motivations . . . 3
- 1.2 Objectives . . . 3

2 Prerequisites and Setup 4

3 What is a Dotfiles Repository? 4

4 Repository Structure and Organization 5
- 4.1 Creating the Initial Repository . . . 5

5 Essential Configuration Files 6
- 5.1 Shell Configuration . . . 6
- 5.2 Git Configuration . . . 6
- 5.3 Cross-Platform Compatibility . . . 7

6 Automated Installation Scripts 8
- 6.1 Basic Script Structure . . . 8
- 6.2 Safe File Linking . . . 9
- 6.3 Installation Logic . . . 9

7 Advanced Features and Security 10
- 7.1 Package Management Integration . . . 10
- 7.2 Secure Configuration Management . . . 10
- 7.3 Team Collaboration . . . 11
- 7.4 Things to Watch Out For . . . 11

8 What Did We Learn? 12
- 8.1 Lessons Learnt . . . 12
- 8.2 Limitations . . . 13
- 8.3 Opportunities for Improvement . . . 13

9 Wrapping Up 14

10 See Also 14
10.1 Related Posts . . . 14
10.2 Key Resources . . . 14

11 Reproducibility 14
11.1 System Requirements . . . 14
11.2 Environment Verification . . . 15
11.3 Security Review Checklist . . . 15

12 Let's Connect 15

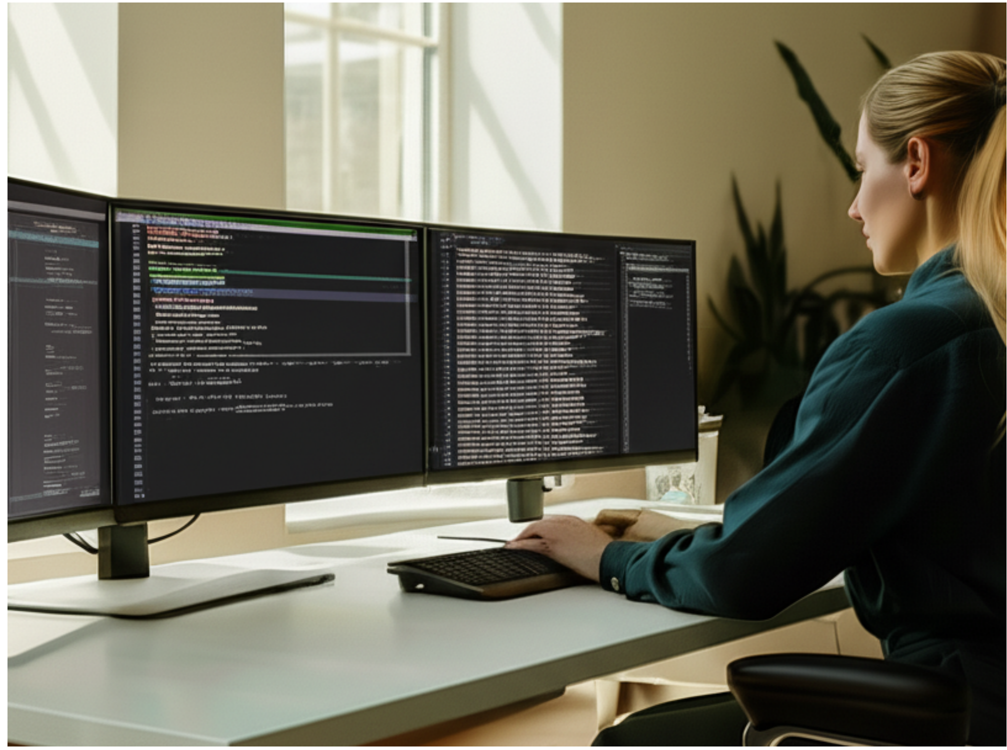

Figure 1: Dotfiles repository workflow for managing development environment configurations

A dotfiles repository transforms scattered configuration files into a single, version-controlled system.

1 Introduction

I did not really know how fragile my development environment was until the day I sat down at a freshly imaged workstation and realized that none of my shell aliases, git shortcuts, or editor settings had survived the transition. What followed was an afternoon of hunting through old backups and re-typing configurations from memory, a process I had repeated more times than I care to admit.

The solution turned out to be straightforward: a GitHub repository dedicated to storing, versioning, and automatically deploying the configuration files that define a development environment. These

are the so-called "dotfiles," named for the leading period that hides them from standard directory listings on Unix-like systems. Once centralized in version control, they can be cloned onto any machine and installed with a single script.

This post walks through the process of creating such a repository from scratch, including the directory layout, the core configuration files, a safe installation script, and the security considerations that come with publishing personal configurations to a public repository.

More formally, this post documents the workstation-level Infrastructure as Code (IaC) tier of the Workflow Construct described in post 52. A dotfiles repository is the keystone artefact of that tier: it is the versioned, declarative source from which a clean machine is rebuilt by running `install.sh`. The construct's other single-laptop layers (Shell, Editor, Vim plugins, zsh aliases) are downstream of this repository in the sense that they are all reproduced from its contents.

1.1 Motivations

- I was tired of spending hours reconfiguring shell aliases, git shortcuts, and editor settings every time I moved to a new machine or re-imaged a workstation.
- My configurations had diverged between my office desktop, my laptop, and a remote server, and I could never remember which machine held the most current version of a given file.
- I wanted a single `git clone && ./install.sh` command that could reproduce my working environment on any macOS or Linux system.
- Onboarding a collaborator onto a shared coding style required sending them a patchwork of files by email, which was error-prone and unversioned.
- I needed a reliable backup strategy for configurations that had accumulated years of incremental refinement.

1.2 Objectives

1. Build a well-organized dotfiles repository with directories for shell, git, editor, and system configurations.
2. Write an automated installation script that creates symbolic links safely, backing up any existing files before overwriting.
3. Implement cross-platform detection so the same repository works on both macOS and Linux.
4. Establish security practices that prevent credentials, API keys, and private SSH keys from leaking into a public repository.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

Version control is the foundation of reproducible configuration management.

2 Prerequisites and Setup

Before creating a dotfiles repository, ensure you have the following tools available:

- **Git** version 2.0 or higher
- A **GitHub account** with an SSH key configured
- **Command line access** (Terminal on macOS, any shell on Linux)
- A **text editor** for writing configuration files

This guide targets macOS and Linux environments. Platform-specific differences are handled through conditional logic in the configuration files themselves.

Security prerequisites. Before placing any configuration file in a public repository, understand the distinction between safe and unsafe content:

```
# Safe for public dotfiles repositories
bashrc, zshrc            # Shell configuration
gitconfig                # Git settings (no creds)
vimrc, tmux.conf         # Editor and terminal
aliases, functions       # Custom commands
Brewfile                 # Package manager lists

# NEVER include in public repositories
id_rsa, id_ed25519       # Private SSH keys
aws_credentials          # Cloud credentials
env_files                # Secrets in env vars
netrc                    # Authentication tokens
```

3 What is a Dotfiles Repository?

A dotfiles repository is a version-controlled directory that stores the configuration files governing your shell, editor, git, and other command-line tools. On Unix-like systems, these files are tradition-

ally prefixed with a dot (`.bashrc`, `.gitconfig`, `.vimrc`), which hides them from standard `ls` output. The repository collects them in one place, tracks changes over time, and provides an installation mechanism to deploy them to any machine.

The analogy is straightforward: just as `renv.lock` captures the exact R package versions needed to reproduce an analysis, a dotfiles repository captures the exact environment configurations needed to reproduce a development setup. The difference is that dotfiles span the entire operating system rather than a single project.

Modern repositories typically store configuration files without their leading dots (e.g., `vimrc` instead of `.vimrc`). This improves visibility in file browsers, produces cleaner GitHub displays, and makes the symbolic link mapping in the installation script explicit.

4 Repository Structure and Organization

The foundation of any successful dotfiles setup is a directory layout that separates configurations by category. A clean structure makes it straightforward to find, edit, and extend individual files as your environment evolves.

```
dotfiles/
├── README.md
├── install.sh
├── Makefile
├── .gitignore
├── shell/
│   ├── bashrc
│   ├── zshrc
│   ├── aliases
│   └── functions
├── git/
│   └── gitconfig
├── editors/
│   └── vimrc
├── system/
│   ├── inputrc
│   └── editorconfig
└── packages/
    ├── Brewfile
    └── apt-packages.txt
```

Notice that the files are stored without their leading dots. The installation script adds the dot prefix when creating symbolic links in `$HOME`.

4.1 Creating the Initial Repository

Start by creating the directory structure locally and pushing it to GitHub:

```
mkdir ~/dotfiles && cd ~/dotfiles

git init

mkdir -p shell git editors system packages

echo "# My Dotfiles" > README.md
echo "Development environment configurations" \
  >> README.md
```

5 Essential Configuration Files

With the repository structure in place, the next step is populating it with the files that define your development environment. Each file should be portable across machines, well-documented with comments explaining non-obvious choices, and conservative in its defaults.

5.1 Shell Configuration

The shell configuration file is the most frequently used dotfile. Here is a minimal `zshrc` that establishes history management, path modifications, and modular sourcing of additional files:

```
# shell/zshrc
HISTSIZE=10000
SAVEHIST=10000
setopt SHARE_HISTORY
setopt HIST_IGNORE_DUPS

export PATH="$HOME/.local/bin:$PATH"
export PATH="/opt/homebrew/bin:$PATH"

export EDITOR="vim"
export BROWSER="open"

[ -f ~/.aliases ] && source ~/.aliases
[ -f ~/.functions ] && source ~/.functions
```

The modular approach of sourcing `~/.aliases` and `~/.functions` separately keeps the main `zshrc` concise while allowing topic-specific files to grow independently.

5.2 Git Configuration

The `gitconfig` file controls git's behavior across all repositories on the machine. Store it without credentials; configure the email address per repository or via `git config --global`:

```
# git/gitconfig
[user]
    name = Your Name

[core]
    editor = vim
    autocrlf = input
    excludesfile = ~/.gitignore_global

[alias]
    st = status
    co = checkout
    br = branch
    cm = commit -m
    lg = log --oneline --graph --decorate
    unstage = reset HEAD --

[push]
    default = simple

[pull]
    rebase = true
```

5.3 Cross-Platform Compatibility

Different operating systems require platform-specific paths and command flags. Use conditional logic so the same dotfiles work on both macOS and Linux without manual editing:

```
case "$OSTYPE" in
  darwin*)
    export BREW_PREFIX="/opt/homebrew"
    alias ls="ls -G"
    ;;
  linux*)
    export BREW_PREFIX="/home/linuxbrew/.linuxbrew"
    alias ls="ls --color=auto"
    ;;
esac

[ -f "$BREW_PREFIX/bin/brew" ] \
  && eval "$("$BREW_PREFIX/bin/brew" shellenv)"
```

This pattern of using `$OSTYPE` for platform detection is reliable across Bash and Zsh and avoids the need for separate configuration files per operating system.

[Cinnamon] Soft mood and latex workflow

Workflow

Figure 2: Soft ambiance image representing careful configuration and organization

Careful organization of configuration files pays dividends across every machine you touch.

6 Automated Installation Scripts

The power of a dotfiles repository lies in its installation script, which transforms a bare machine into a fully configured development environment with a single command. A well-designed script creates symbolic links from the repository to the expected locations in `$HOME`, backing up any existing files before overwriting them.

6.1 Basic Script Structure

Start with strict error handling and logging functions:

```
#!/bin/bash
set -e

log() {
    echo "[INFO] $1"
}
```

```
warn() {
    echo "[WARN] $1"
}

error() {
    echo "[ERROR] $1"
}
```

6.2 Safe File Linking

The core function creates symbolic links while preserving existing configurations:

```
link_file() {
    local src="$1"
    local dest="$2"

    if [ -e "$dest" ]; then
        warn "$dest exists, creating backup"
        mv "$dest" \
          "${dest}.backup.$(date +%Y%m%d_%H%M%S)"
    fi

    ln -sf "$src" "$dest"
    log "Linked $src -> $dest"
}
```

The timestamped backup ensures that no existing configuration is lost, even if the script is run multiple times.

6.3 Installation Logic

The main function ties everything together:

```
install_dotfiles() {
    local dir
    dir="$(cd "$(dirname "$0")" && pwd)"

    log "Installing dotfiles from $dir"

    link_file "$dir/shell/zshrc" "$HOME/.zshrc"
    link_file "$dir/shell/aliases" "$HOME/.aliases"
    link_file \
      "$dir/shell/functions" "$HOME/.functions"

    link_file \
```

```
        "$dir/git/gitconfig" "$HOME/.gitconfig"

    link_file "$dir/editors/vimrc" "$HOME/.vimrc"

    log "Dotfiles installation complete!"
    log "Restart your shell or run: source ~/.zshrc"
}

install_dotfiles
```

The script is deliberately simple. Each `link_file` call is an explicit mapping from the repository path to the expected location in `$HOME`, making the deployment transparent and easy to audit.

7 Advanced Features and Security

With a functional repository and installation script in place, the next step is hardening the setup against the security risks inherent in publishing configuration files to a public repository.

7.1 Package Management Integration

Automating software installation alongside configuration deployment creates a complete environment setup:

```
# Brewfile for macOS package management
brew "git"
brew "vim"
brew "tmux"
brew "python@3.11"

cask "visual-studio-code"
cask "iterm2"
cask "docker"
```

Run the Brewfile during installation:

```
brew bundle --file=packages/Brewfile
```

7.2 Secure Configuration Management

Never commit secrets to version control. Instead, use a local environment file that is excluded from git:

```
# In .zshrc, source a local-only file
if [ -f ~/.env.local ]; then
```

```
    export $(grep -v '^#' ~/.env.local | xargs)
fi

# ~/.env.local (NOT tracked by git)
# GITHUB_TOKEN=your_token_here
# AWS_ACCESS_KEY_ID=your_key_here
```

For teams that need encrypted secrets in git, `git-crypt` provides transparent encryption:

```
git-crypt init
git-crypt add-gpg-user your-gpg-key-id
echo "secrets/* filter=git-crypt diff=git-crypt" \
  >> .gitattributes
```

7.3 Team Collaboration

A Makefile provides a standardized interface for common operations, making the repository accessible to team members who may not be familiar with the internal structure:

```
.PHONY: install update backup test

install:
    @echo "Installing dotfiles..."
    ./install.sh

update:
    @echo "Updating dotfiles..."
    git pull origin main
    ./install.sh

backup:
    @echo "Backing up current configurations..."
    ./scripts/backup.sh

test:
    @echo "Testing dotfiles configuration..."
    ./scripts/test.sh
```

7.4 Things to Watch Out For

1. **Forgetting `.gitignore` patterns.** I accidentally committed an `.env.local` file early on because I had not yet added the ignore rule. Add sensitive patterns to `.gitignore` before the first commit.
2. **Stale backups accumulating.** The `link_file` function creates timestamped backups on every run. Periodically clean `$HOME/*.backup.*` files to avoid clutter.

3. **Platform-specific breakage.** A Homebrew path that works on macOS will fail silently on Linux. Always test conditional blocks on both platforms before pushing.
4. **Symlink confusion in editors.** Some editors follow symlinks differently. If you edit `~/.zshrc` and the changes do not appear in the repository, confirm that your editor resolves symlinks correctly.
5. **Public repository exposure.** Running `git log -p` on a public dotfiles repository reveals the full history, including anything committed and later removed. Use `git-filter-repo` to scrub leaked secrets from history.

Figure 3: UCSD Geisel Library representing structured knowledge management

Systematic organization of development configurations mirrors the structured approach of academic research.

8 What Did We Learn?

8.1 Lessons Learnt

Conceptual Understanding:

- A dotfiles repository is fundamentally a backup and deployment system for the implicit knowledge embedded in configuration files that accumulate over years of daily use.
- The symbolic link model preserves a single source of truth in the repository while allowing tools to read files from their expected locations in `$HOME`.
- Storing files without leading dots is a modern convention that improves visibility and makes the link mapping explicit in the installation script.
- Security is not an afterthought; the `.gitignore` and the separation of secrets into untracked `.env.local` files must be established before any content is committed.

Technical Skills:

- Writing a portable `install.sh` that handles backup, linking, and platform detection in under 50 lines of Bash.

- Using `$OSTYPE` case statements for cross-platform conditional logic that works across both Bash and Zsh.
- Structuring a Brewfile to automate macOS application installation alongside command-line tool provisioning.
- Configuring `git-crypt` for transparent encryption of sensitive files within a public repository.

Gotchas and Pitfalls:

- Running the install script without first adding `.gitignore` rules can expose secrets in the very first commit.
- Homebrew paths differ between Intel macOS (`/usr/local`) and Apple Silicon (`/opt/homebrew`), requiring explicit detection.
- The `set -e` flag in installation scripts will abort on the first error, which is desirable but can mask the root cause if logging is insufficient.
- Automated backups create a growing collection of timestamped files in `$HOME` that require periodic manual cleanup.

8.2 Limitations

- Public repositories expose all committed content, including deleted files that remain in git history, which creates a permanent security risk if credentials are ever committed by mistake.
- SSH keys must be generated per machine and cannot be safely distributed through a dotfiles repository, limiting the scope of automation.
- Cross-platform scripts require testing on every target platform; a configuration that works on macOS may fail on Ubuntu, Fedora, or Arch Linux due to differences in installed tools and paths.
- Personal preferences embedded in dotfiles may conflict with team or organizational coding standards, requiring separate personal and shared repositories.
- The installation script assumes a standard `$HOME` directory structure and may not work correctly in non-standard environments such as containers, NixOS, or enterprise-managed workstations.

8.3 Opportunities for Improvement

1. Add a CI/CD pipeline using GitHub Actions that tests the installation script on both macOS and Ubuntu runners after every push.
2. Integrate with Docker or Podman to create a containerized development environment that combines the dotfiles with a complete software stack.
3. Implement a `--dry-run` flag in the installation script that reports what changes would be made without creating any links or backups.
4. Add a Makefile target that runs ShellCheck on all Bash scripts to catch portability and syntax issues automatically.
5. Create a `diff` command that compares deployed configurations in `$HOME` against the repository versions to detect local drift.
6. Explore Ansible or Chezmoi as declarative alternatives to hand-written installation scripts for managing complex multi-machine setups.

9 Wrapping Up

A GitHub dotfiles repository transforms the error-prone ritual of manual environment configuration into a version-controlled, automated process. The core idea is simple: store your configuration files in a git repository, write an installation script that creates symbolic links, and never configure a machine by hand again.

The main takeaways from this exercise:

- New machine setup drops from 4-6 hours of manual configuration to roughly 15-30 minutes with an automated install script.
- A single repository provides complete version history of every configuration change, with the ability to roll back problematic updates.
- Cross-platform conditional logic in shell files allows the same repository to serve macOS and Linux without forking into separate branches.
- Security requires active discipline: secrets must be separated into untracked files from the start, not removed after the fact.

I would encourage starting small, with just your shell configuration and git aliases, and growing the repository incrementally as you identify more files worth tracking. The community repositories listed below provide excellent models for advanced techniques.

10 See Also

10.1 Related Posts

- Configure the Command Line for Data Science Development – Terminal and Zsh shell configuration
- Set Up Git for Version Control – Git installation and initial configuration

10.2 Key Resources

- Dotfiles Community Guide – Curated showcase of popular dotfiles repositories and frameworks
- Holman, Z. (2014). "Dotfiles Are Meant to Be Forked" – Influential essay on dotfiles philosophy
- Athalye, A. "Managing Your Dotfiles" – Practical guide from the creator of Dotbot
- OWASP Secrets Management Cheat Sheet – Security best practices for credential handling
- Atlassian: "The Best Way to Store Your Dotfiles" – Alternative approach using bare git repositories

11 Reproducibility

11.1 System Requirements

- **Git**: Version 2.0 or higher

- **Shell**: Bash 4.0+ or Zsh 5.0+
- **GitHub Account**: For repository hosting
- **Platform**: macOS 12+ or Ubuntu 20.04+

11.2 Environment Verification

```
git --version
$SHELL --version
uname -a
echo $HOME
```

11.3 Security Review Checklist

Before publishing, verify:

- No SSH private keys (`id_rsa`, `id_ed25519`)
- No API tokens or credentials
- No hardcoded passwords or personal information
- `.gitignore` includes all sensitive file patterns
- Environment variables externalized to `.env.local`

12 Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.