Kickstart-Guide

A quick reference guide for developers

Ahmet Emirhan Göktaş

Table of contents

1. Kickstart-Guide 3
1.1 Feedback 3
2. Guides 4
2.1 Git 4
3. About the Kickstart-Guide 13
3.1 Using the Guide 13
3.2 License 13

1. Kickstart-Guide

Welcome to the Kickstart-Guide! This guide will help you get started with various tools. Currently, the guide includes the following tools:

- Git

Source code for the guide is available on GitHub.

1.1 Feedback

If you have any feedback or suggestions for improvement, please feel free to open an issue on the GitHub repository. Your feedback is greatly appreciated!

2. Guides

2.1 Git

2.1.1 Git

Git is a distributed version control system that allows multiple developers to collaborate on a project. It provides a way to track changes, and merge code changes. Git is widely used in the software development industry, and is an essential tool for any developer. There is already a lot of documentation available for Git and one of the best resources is the GitHub Git Guide. This guide will go over the basics briefly and then dive into more advanced topics.

Table of Contents:

1. Installing Git
2. Basics
3. Gitflow
4. References

2.1.2 Install

You can check if Git is installed on your system by running the following command:

```
git version
```

The output should be similar to this:

```
git version 2.43.1
```

If you get an error like `git: command not found`, you can fallow the instructions below to install Git.

Installing on Windows

Installing `git` on Windows is straightforward

1. Download the installer from the official website.
2. Run the installer and follow the instructions.
3. After the installation is complete, you can open terminal and run `git version` to check if the installation was successful.

> **Tip**
>
> In Windows 10 or later, you can also use WSL which allows you to run a Linux on top of Windows. This allows you to be in a Linux environment while still using Windows.

> **Note**
>
> At the `Adjusting your PATH environment` choose `Git from the command line and also from 3rd-party software`. This will allow you to use `git` from the windows terminal. This article assumes that you have done this. Otherwise you will have to use the Git Bash terminal that comes with the installer for the commands in this guide.

Installing on macOS

Usially `git` is already installed on macOS. But if it for some reason is not installed, you can install it either with Homebrew or by downloading the installer.

USING HOMEBREW

Homebrew is a package manager for macOS. If you have Homebrew installed, you can install `git` easily.

1. Open terminal and run the following command to install `git`:

   ```
   brew install git
   ```

2. After the installation is complete, you can run `git version` to verify that the installation was successful.

USING THE INSTALLER

1. Download the latest installer.
2. Run the installer and follow the instructions.
3. After the installation is complete. You can verify the installation by running `git version` in the terminal.

Installing on Linux

You can install `git` on Linux using the package manager of your distribution.

DEBIAN/UBUNTU OR DERIVATIVES

```
sudo apt-get update
sudo apt-get install git
```

FEDORA OR DERIVATIVES

```
sudo dnf install git-all
```

ARCH LINUX OR DERIVATIVES

```
sudo pacman -S git
```

After the installation is complete, you can verify the installation by running `git version` in the terminal.

Configuration

Installing git allows you to clone repositories and track changes in your projects. However, you need to configure git with your credentials before you can push changes to a remote repository. This can be done by running the following commands in the terminal:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

You can also omit the `--global` flag to set the configuration for a single repository. For that you need to navigate to the repository and run the same commands without the `--global` flag.

From this point on you can use git to track changes in your project. Git will ask you your credentials when you try to push your changes to a remote repository. You can also use SSH keys to authenticate with the remote repository. Which is not only more secure but also more convenient.

> **Note**
>
> If you are using GitHub, you have to use SSH keys to authenticate with the remote repository. GitHub has removed support for password authentication on August 13, 2021.

SSH KEYS

To create an SSH key, you can run the following command in the terminal:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

This will create a new SSH key, using the provided email as a label. You can safely accept the default file location when asked. You can also add a passphrase to your SSH key for an extra layer of security. Which is recommended. Without a passphrase the ssh key will be stored in your computer as plain text.

> **Tip**
>
> If ssh-keygen asks you to overwrite the file this means that you already have created an SSH key in the past. You can either overwrite the file or provide a new file name. To create a custom-named SSH key you can type the default file location and then the name of the new file.

After generating the SSH key, the next step is adding it to your remote repository. Instructions for GitHub and for GitLab are available in their respective documentation.

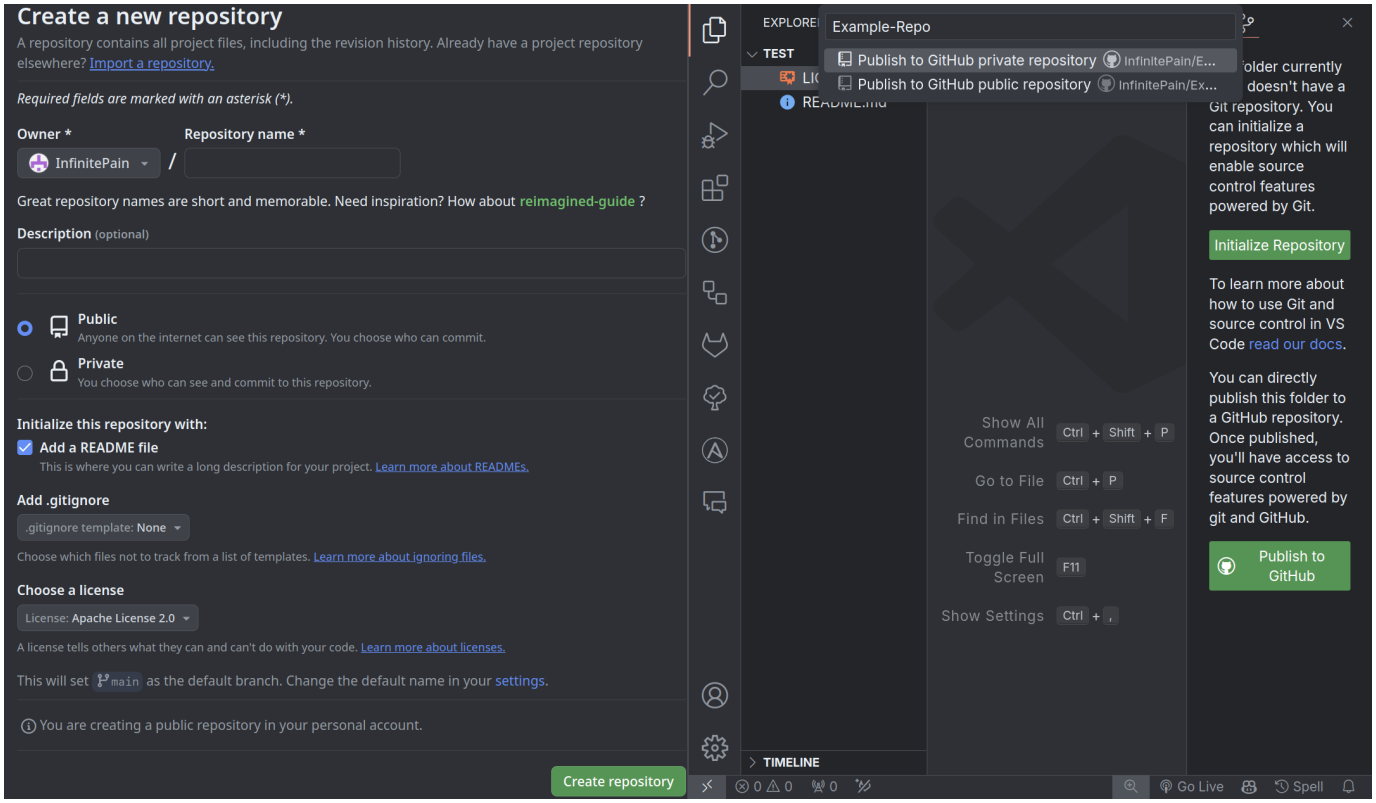
You can also add the SSH key to your SSH agent which will allow you to use the key without having to enter the passphrase every time you use it. For that GitHub has a good guide on how to add SSH key to the ssh-agent.

2.1.3 Basics

In this guide, we will mostly use the command line to interact with Git. But all of these can also be done using a GUI tool like GitHub Desktop, GitKraken or even VS Code.

Creating a Repository

There is couple of ways you can do this. You can create a new repository on GitHub with some initial files and then clone it to your local machine. Or you can create a new repository on your local machine and then publish it to GitHub. In VS Code press `^ Ctrl` + `⇧ Shift` + `P` or `⌘ Cmd` + `⇧ Shift` + `P` to open the command palette and type `Publish to GitHub` to create a new repository on GitHub.



Clone

Cloning a repository means that you are making a copy of the repository on your local machine. This can be done by running the following command in the terminal:

```
git clone <repository-url>
```

> **Tip**
>
> If you want to use your SSH key for authentication, clone the repository using the SSH path. If you have already cloned the repository using HTTPS but want to use SSH for authentication, you can change the remote URL using the following command:
>
> ```
> git remote set-url origin <new-repository-url>
> ```

Status

The `git status` command shows the status of the working directory and the staging area. It won't modify anything in your repository, it just shows you the current status like which files are staged, unstaged, or untracked.

Add

The `git add` command adds changes in the working directory to the staging area effectively telling Git that you want to include updates to a particular file in the next commit.

To add specific files, you can run the following command:

```
git add <file>
```

To add all files, you can run the following command:

```
git add .
```

> **Tip**
>
> When you delete a file from a repository, you still need to run `git add` otherwise git won't be aware of the deletion.

Commit

The `git commit` command is used to save changes to the local repository. It is like a snapshot of your repository at a particular point in time. But the changes are not yet on the remote repository.

To commit changes, you can run the following command:

```
git commit -m "commit message"
```

It's highly recommended to read about Conventional Commits. It's a specification for adding human and machine readable meaning to commit messages.

Push

The `git push` command is used to upload local repository content to a remote repository. It is used to make the commits you have made on your local repository online. It's recommended to pull the latest changes from the remote repository before pushing your changes. By doing so, you can resolve any conflicts before pushing your changes and reduce the chances of merge conflicts.

To push changes, you can run the following command:

```
git push
```

Pull

The `git pull` command is used to update the local version of a repository from a remote. It can be done by running the following command:

```
git pull
```

2.1.4 Gitflow

Gitflow is a branching model that is used for managing the development of a project. You have probably seen it in various open-source projects. It's bascially a set of rules that define how branches are created and merged in order to manage the development of a project. It is consist of two components, the main branches and the supporting branches.

The Main Branches

The main branches are the branches that are used to manage the release of the project. They are:

1. `main` branch: This is the branch that contains the official release of the project.
2. `develop` branch: This is the branch that contains the latest development of the project and used to create the release of the project.It is where all the new features come together and are tested before they are released.

The Supporting Branches

The supporting branches are the branches that are used to develop new features and fix bugs. They are:

1. `feature` branch: This is the branch that is used to develop new features. It is branched off from the `develop` branch and merged back into the `develop` branch when the feature is complete.
2. `release` branch: This is the branch that is used to prepare the release of the project. The purpose of this branch is to stop the development of new features and fix bugs. It is branched off from the `develop` branch and merged back into the `main` branch when the release is complete.
3. `hotfix` branch: This is the branch that is used to fix bugs in the release. It is branched off from the `main` branch and merged back into the `main` branch when the bug is fixed.

The Workflow

Let's think of a scenario where we have a project that is being developed. The project has the following branches:

1. `main` branch
2. `develop` branch

CREATING/FINISHING A FEATURE

We want to develop a new feature for the project. First, we create a new branch called `feature-my-feature` from the `develop` branch.

```
# Create a new branch called feature-my-feature from the develop branch
git checkout -b feature-my-feature develop
```

After we have finished developing the feature, we merge the `feature-my-feature` branch back into the `develop` branch.

```
# Merge the feature-my-feature branch into the develop branch
git checkout develop
git merge --no-ff feature-my-feature
git branch -d feature-my-feature
```

> **Tip**
>
> The last command is optional but recommended. It deletes the `feature-my-feature` branch after it has been merged into the `develop` branch. Don't worry, the changes are still there in the `develop` branch. We just want to keep our branches clean.

CREATING/FINISHING A RELEASE

... After some time, we reach a point where we want to release the project. At this stage, it's crucial to consider how we version our releases. This is where Semantic Versioning (SemVer) comes into play. SemVer is a versioning scheme for software that aims to convey meaning about the underlying changes in a release through the version number itself. It is formatted as `MAJOR.MINOR.PATCH`, where:

- **MAJOR** versions indicate incompatible API changes,
- **MINOR** versions add functionality in a backwards-compatible manner, and
- **PATCH** versions include backwards-compatible bug fixes.

For more detailed information on SemVer and its rules, visit semver.org.

Now, to prepare our project for release, we create a new branch called `release-1.0` from the `develop` branch. The version number `1.0` should follow the SemVer guidelines, indicating this is our first stable release with a set of completed features.

```
# Create a new branch called release-1.0 from the develop branch
git checkout -b release-1.0 develop
./bump-version.sh 1.0
git commit -a -m "chore(release): bump version number to 1.0"
```

The `bump-version.sh` script is an imaginary script that we use to bump the version number of the project. After we have finished preparing the release, we merge the `release-1.0` branch back into the `main` branch.

```
git checkout main
git merge --no-ff release-1.0
# Tag the release
git tag -a 1.0
# Also Merge develop branch to keep it up to date with the release
git checkout develop
git merge --no-ff release-1.0
git branch -d release-1.0
```

> **Tip**
>
> The last command is optional but recommended. It deletes the `release-1.0` branch after it has been merged into the `main` branch. Don't worry, the changes are still there in the `main` branch. We just want to keep our branches clean.

FIXING A BUG IN THE RELEASE

After releasing version 1.0 of our project, we discovered a bug that needs to be addressed immediately. It's also important to adhere to SemVer principles. If the bug fix is backward compatible and doesn’t introduce new features, it should increment the PATCH version. For instance, if we're fixing a bug in version `1.0`, the hotfix version would be `1.0.1`.

```
# Create a new branch called hotfix-1.0.1 from the main branch
git checkout -b hotfix-1.0.1 main
./bump-version.sh 1.0.1
git commit -a -m "chore(release): bump version number to 1.0.1"
```

After we have fixed the bug, we merge the `hotfix-1.0.1` branch back into the `main` branch.

```
git checkout main
git merge --no-ff hotfix-1.0.1
# Tag the release
git tag -a 1.0.1
# Also Merge develop branch to keep it up to date with the main
git checkout develop
git merge --no-ff hotfix-1.0.1
git branch -d hotfix-1.0.1
```

Tip

The last command is optional but recommended. It deletes the `hotfix-1.0.1` branch after it has been merged into the `main` branch. Don't worry, the changes are still there in the `main` branch. We just want to keep our branches clean.

2.1.5 References

- GitHub Git Guide
- GitHub Authentication Documentation
- Gitflow Medium
- Gitflow Nvie

3. About the Kickstart-Guide

The Kickstart-Guide is a direct, practical resource for developers at all stages of their journey. It's crafted to offer quick access to information. This guide stands as a ready reference to support your work, learning, and exploration in the field of development.

Who Is It For?

- **Newcomers:** Introducing foundational concepts and tools to those new to development.
- **Experienced Developers:** Providing quick refreshers and advanced insights.
- **Teams:** Serving as a common knowledge base to ensure consistency in practices.

3.1 Using the Guide

Dive into any section that matches your current needs. The guide is designed for you to quickly find what you're looking for, understand it, and apply it to your projects or learning path.

3.2 License

Shared under the Apache License 2.0, the Kickstart-Guide encourages open use and contribution. You can find the full license in the LICENSE.

https://infinitepain.github.io/Kickstart-Guide/dev