About Readme:

A README is a text file that introduces and explains a project. It contains information that is commonly required to understand what the project is about.

This is also an easy way to answer questions that your audience will likely have regarding how to install and use your project.

Its usually writen before showing a project to other people or make it public.

It contains some information about the project like:

Name ,Description ,Installation ,Usage ,Support ,Roadmap ,Contributing ,Project status and etc.

MarkDown and how to write it:

Markdown is a way to style text on the web. You control the display of the document; formatting words as bold or italic, adding images, and creating lists are just a few of the things we can do with Markdown. Mostly, Markdown is just regular text with a few non-alphabetic characters thrown in, like # or *.

for example:

Headers:

This is an <h1> tag

This is an <h2> tag

This is an <h6> tag

Emphasis:

This text will be italic

This will also be italic

This text will be bold

__This will also be bold__

You **can** combine them

Images:

![GitHub Logo](/images/logo.png)

Format: ![Alt Text](url)

Links:

http://github.com - automatic!

[GitHub](http://github.com)

Blockquotes:

As Kanye West said:

> We're living the future so

> the present is our past.

Inline code:

I think you should use an

`<addr>` element here instead.

Git Flow ,Github flow and trunk based development:

Git:

Git flow works with different branches to manage easily each phase of the software development, it's suggested to be used when your software has the concept of "release".

Another good point of this flow is that fits perfectly when you work in team and one or more developers have to collaborate to the same feature.

The main branches in git flow are:

master

develop

features

hotfix

release

When you clone a GIT repository in your local folder you have immediately to create a branch from the master called develop, this branch will be the main branch for the development and where all the developers in a team will work to implement new features or bug fixing before the release.

Every time a developer needs to add a new feature he will create a new branch from develop that allow him to work properly in that feature without compromise the code for the other people in the team in the develop branch.

When the feature will be ready and tested it could be rebased inside the develop branch, our goal is to have always a stable version of develop branch because we merge the code only when the new feature is completed and it's working.

When all the features related to a new release are implemented in the develop branch it's time to branch the code to the release branch where there you'll start to test properly before the final deployment.

When you branch your code from develop to release you should avoid to add new features but you should only fix bugs inside the release branch code until to you create a stable release branch.

At the end, when you are ready to push live deploy live your project, you will tag the version inside the master branch so there you can have all the different versions that you release week by week.

Github:

GitHub Flow has some of the same elements as Git Flow, such as feature branches. But unlike Git Flow, GitHub Flow combines the mainline and release branches into a "master" and treats hotfixes just like feature branches.

This simplified model is better suited to continuous delivery models where changes can be quickly made and easily deployed, sometimes multiple times a day.

Trunk based:

In the trunk-based development model, all developers work on a single branch with open access to it. Often it's simply the master branch. They commit code to it and run it. It's super simple.

In some cases, they create short-lived feature branches. Once code on their branch compiles and passess all tests, they merge it straight to master. It ensures that development is truly continuous and prevents developers from creating merge conflicts that are difficult to resolve.

Semantic versioning:

Semantic versioning (also referred as SemVer) is a versioning system that has been on the rise over the last few years. It has always been a problem for software developers, release managers and consumers. Having a universal way of versioning the software development projects is the best way to track what is going on with the software as new plugins, addons, libraries and extensions are being built almost everyday.

Semantic Versioning is a 3-component number in the format of X.Y.Z, where :

X stands for a major version.

Y stands for a minor version.

Z stands for a patch.

So, SemVer is of the form Major.Minor.Patch

Below given are the scenarios when you should bump the value of X, Y and Z:

- Bump the value of X when breaking the existing API.
- Bump the value of Y when implementing new features in a backward-compatible way.
- Bump the value of Z when fixing bugs.

Conventional commits:

The commit message should be structured as follows:

<type>[optional scope]: <description>

[optional body]

[optional footer(s)]

The commit contains the following structural elements, to communicate intent to the consumers of your library:

fix: a commit of the type fix patches a bug in your codebase (this correlates with PATCH in semantic versioning).

feat: a commit of the type feat introduces a new feature to the codebase (this correlates with MINOR in semantic versioning).

BREAKING CHANGE: a commit that has a footer BREAKING CHANGE:, or appends a ! after the type/scope, introduces a breaking API change (correlating with MAJOR in semantic versioning). A BREAKING CHANGE can be part of commits of any type.

types other than fix: and feat: are allowed, for example @commitlint/config-conventional (based on the the Angular convention) recommends build:, chore:, ci:, docs:, style:, refactor:, perf:, test:, and others.

footers other than BREAKING CHANGE: <description> may be provided and follow a convention similar to git trailer format.

A scope may be provided to a commit's type, to provide additional contextual information and is contained within parenthesis, e.g., feat(parser): add ability to parse arrays.

Also some Rules for a great git commit message style:

- Separate subject from body with a blank line
- Do not end the subject line with a period
- Capitalize the subject line and each paragraph
- Use the imperative mood in the subject line
- Wrap lines at 72 characters
- Use the body to explain what and why you have done something. In most cases, you can leave out details about how a change has been made.