.gitignore

How to keep your repo from filling up with unnecessary shizz

What is .gitignore

.gitignore is a text file which stores patterns by which to ignore files.

Benefits:

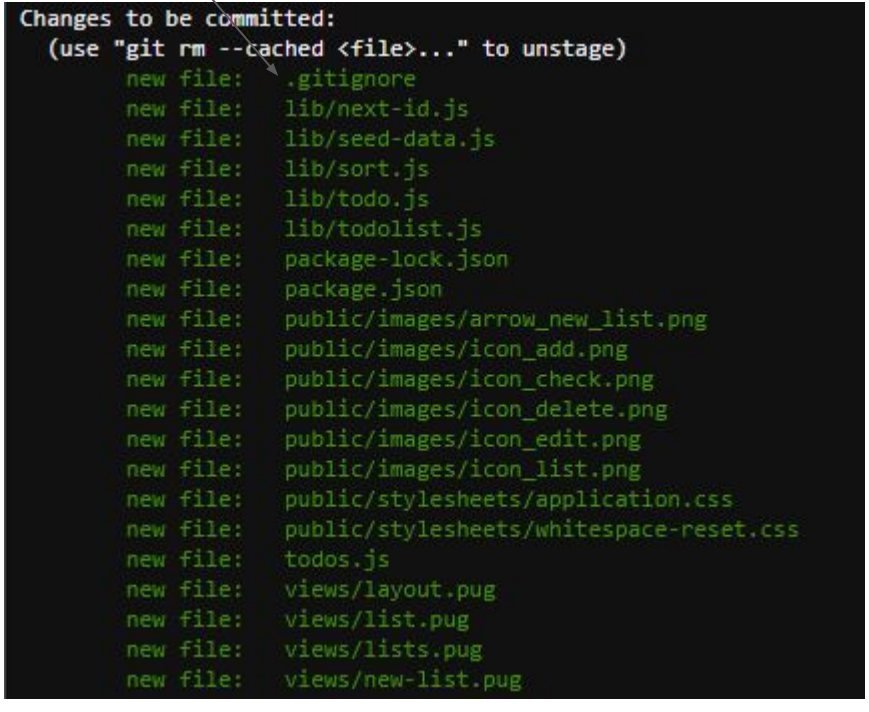
Using **.gitignore** saves resources:

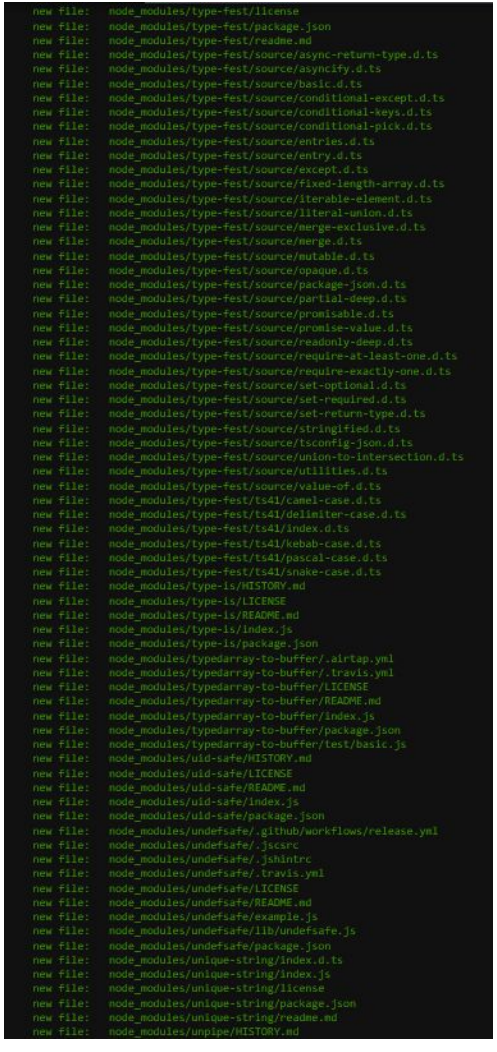
- Git repo tracking is more lightweight by not tracking dependencies and databases.
- Lightweight repos are more efficiently sent to and from remote.

Using **.gitignore** also makes tracking changes with **git status** much easier...

Same Project: Guess which one uses **.gitignore**

Hint:





Basics **.gitignore**

This can be in root directory and in any subdirectory as well, with the closest **.gitignore** file taking precedence.

this file follows the following rules:

- `#` for comments

- Blank lines ignored

- Standard glob patterns work

- `/` at pattern start avoids recursivity

- `/` at pattern end specifies directory

- `!` at pattern start ignores a pattern

Example Time

```
mkdir git_practice && cd git_practice

mkdir dir1

mkdir dir1/dir2

mkdir dir1/dir2/dir3

echo "Some Text" >> root-file.txt

echo "Some JavaScript" >> root-file.js

echo "Some Text" >> dir1/file1.txt

echo "Some JavaScript" >> dir1/file1.js

echo "Some Text" >> dir1/dir2/file2.txt

echo "Some JavaScript" >> dir1/dir2/file2.js

echo "Some Text" >> dir1/dir2/dir3/file3.txt

echo "Some JavaScript" >> dir1/dir2/dir3/file3.js

git init

touch .gitignore
```

This script or set of commands will set up a linux file structure to demonstrate how to use **.gitignore** using the examples on the following slides.

```
ec2-user:~/environment $ bash create_dir.bash
Initialized empty Git repository in /home/ec2-user/environment/git_practice/.git/
ec2-user:~/environment $ cd git_practice/
ec2-user:~/environment/git_practice (main) $ tree
.
├── dir1
│   ├── dir2
│   │   ├── dir3
│   │   │   ├── file3.js
│   │   │   └── file3.txt
│   │   ├── file2.js
│   │   └── file2.txt
│   ├── file1.js
│   └── file1.txt
├── root-file.js
└── root-file.txt

3 directories, 8 files
ec2-user:~/environment/git_practice (main) $
```

Note: **.gitignore** will be in root directory until otherwise specified

Asterisk as a Wildcard

```
ec2-user:~/environment/git_practice (main) $ echo "*.txt" >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  dir1/dir2/dir3/file3.js
A  dir1/dir2/file2.js
A  dir1/file1.js
A  root-file.js
ec2-user:~/environment/git_practice (main) $ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   dir1/dir2/dir3/file3.js
        new file:   dir1/dir2/file2.js
        new file:   dir1/file1.js
        new file:   root-file.js

ec2-user:~/environment/git_practice (main) $
```

- Wildcards are a feature of glob patterns.
- The pattern worked recursively by default, ignoring every **.js** file and tracking every other file
- To reset your project:
  - git rm --cached -rf * .gitignore
  - rm .gitignore

Leading “/” To Ignore Recursivity

```
ec2-user:~/environment/git_practice (main) $ echo "/*.txt" >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  dir1/dir2/dir3/file3.js
A  dir1/dir2/dir3/file3.txt
A  dir1/dir2/file2.js
A  dir1/dir2/file2.txt
A  dir1/file1.js
A  dir1/file1.txt
A  root-file.js
ec2-user:~/environment/git_practice (main) $ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   dir1/dir2/dir3/file3.js
        new file:   dir1/dir2/dir3/file3.txt
        new file:   dir1/dir2/file2.js
        new file:   dir1/dir2/file2.txt
        new file:   dir1/file1.js
        new file:   dir1/file1.txt
        new file:   root-file.js
```

- Appending the forward slash allows git to only focus on one level of the directory.
- The only file ignored was **root-file.txt**

Optional trailing “/” to specify directory

Without “/”

```
ec2-user:~/environment/git_practice (main) $ echo "dir2" >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  dir1/file1.js
A  dir1/file1.txt
A  root-file.js
A  root-file.txt
```

With “/”

```
ec2-user:~/environment/git_practice (main) $ echo "dir2/" >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  dir1/file1.js
A  dir1/file1.txt
A  root-file.js
A  root-file.txt
```

Ignore directory in root but not subdirectories

```
ec2-user:~/environment/git_practice (main) $ mkdir other_dir other_dir/dir1
ec2-user:~/environment/git_practice (main) $ echo "Some Text" >> other_dir/dir1/file1.txt
ec2-user:~/environment/git_practice (main) $ echo "Some JavaScript" >> other_dir/dir1/file1.js
ec2-user:~/environment/git_practice (main) $ echo "/dir1" >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  other_dir/dir1/file1.js
A  other_dir/dir1/file1.txt
A  root-file.js
A  root-file.txt
```

```
ec2-user:~/environment/git_practice (main) $ tree
.
├── dir1
│   ├── dir2
│   │   ├── dir3
│   │   │   ├── file3.js
│   │   │   └── file3.txt
│   │   ├── file2.js
│   │   └── file2.txt
│   ├── file1.js
│   └── file1.txt
├── other_dir
│   └── dir1
│       ├── file1.js
│       └── file1.txt
├── root-file.js
└── root-file.txt

5 directories, 10 files
```

- All subdirectories of the root **dir1** were ignored, but the **other_dir/dir1** files were tracked.
- This is because of the leading “/” indicating a relative path from the **.gitignore** file.

Ignore certain file only in a particular subdirectory

```
ec2-user:~/environment/git_practice (main) $ echo "/dir1/**/*.txt" >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  dir1/dir2/dir3/file3.js
A  dir1/dir2/file2.js
A  dir1/file1.js
A  root-file.js
A  root-file.txt
```

- The “**/**/***” syntax allows you to specify all subdirectories in a particular directory path.
- In this case, all “**.txt**” files have been ignored from **dir1** and it’s subdirectories.

Exceptions with “**!**”

```
ec2-user:~/environment/git_practice (main) $ echo "*" >> .gitignore
ec2-user:~/environment/git_practice (main) $ echo '!*.txt' >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  root-file.txt
ec2-user:~/environment/git_practice (main) $
```

- Un-ignoring files does not appear to work recursively, as the following patterns do not yield any positive results:
  - !./**/*.txt
  - !dir1/**/*.txt
  - !*/**/*.txt
  - !/**/*.txt
- EDIT: This did not work recursively because the first wildcard (*). If the first entry was instead “***.txt**”, it would work.
- Selecting by full path and specifying file patterns in that directory is the best way to utilize.

Nested **.gitignores**

```
ec2-user:~/environment/git_practice (main) $ echo "/dir1/**/*.txt" >> .gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  dir1/.gitignore
A  dir1/dir2/dir3/file3.js
A  dir1/dir2/file2.js
A  dir1/file1.js
A  root-file.js
A  root-file.txt
```

```
ec2-user:~/environment/git_practice (main) $ echo '!*.txt' >> dir1/.gitignore
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git status -s
A  .gitignore
A  dir1/.gitignore
A  dir1/dir2/dir3/file3.js
A  dir1/dir2/dir3/file3.txt
A  dir1/dir2/file2.js
A  dir1/dir2/file2.txt
A  dir1/file1.js
A  dir1/file1.txt
A  root-file.js
A  root-file.txt
```

The root **.gitiginore** file is overridden by the subdirectory's **.gitignore**

git diff

How to keep up with changes to your shizz

What is **git diff**?

git diff is a way to see the changes between what is in your working directory vs what has been committed or added to the staging area.

By itself, the command will examine unstaged changes.

By passing the option “**--staged**” or “**--cached**”, the command will examine staged changes.

Let’s go through this step by step with a new directory to see how this plays out.

Unmodified Repository

```
ec2-user:~/environment $ mkdir git_practice && cd git_practice
ec2-user:~/environment/git_practice $ git init
Initialized empty Git repository in /home/ec2-user/environment/git_practice/.git/
ec2-user:~/environment/git_practice (main) $ echo "Some Text" >> root.txt
ec2-user:~/environment/git_practice (main) $ git diff
ec2-user:~/environment/git_practice (main) $ git diff --staged
```

We created a new directory and created a repository in it without adding anything to the staging area.

Then we added a file **root.txt** and added **“Some Text”**.

- **git diff**: No commits to compare unstaged changes
- **git diff --staged**: No staged changes

First Staging

```
ec2-user:~/environment/git_practice (main) $ git add root.txt
ec2-user:~/environment/git_practice (main) $ git diff
ec2-user:~/environment/git_practice (main) $ git diff --staged
diff --git a/root.txt b/root.txt
new file mode 100644
index 0000000..5b1dd02
--- /dev/null
+++ b/root.txt
@@ -0,0 +1 @@
+Some Text
ec2-user:~/environment/git_practice (main) $
```

Now that **root.txt** is in the staging area, our **“--staged”** command is more helpful.

- **git diff**: Nothing since no unstaged changes
- **git diff --staged**: Staged changes from “null” to **root.txt**

First Commit

```
ec2-user:~/environment/git_practice (main) $ git commit -m "Initial Commit"
[main (root-commit) 3fcf1bb] Initial Commit

 1 file changed, 1 insertion(+)
 create mode 100644 root.txt
ec2-user:~/environment/git_practice (main) $ git diff
ec2-user:~/environment/git_practice (main) $ git diff --staged
```

After a fresh commit, there are no differences to report.

- **git diff**: Working Directory matches Commit.
- **git diff --staged**: Explains the difference between what

Additional Changes after Initial Commit

```
ec2-user:~/environment/git_practice (main) $ echo "More Text" >> root.txt
ec2-user:~/environment/git_practice (main) $ echo "Some JavaScript" >> root.js
ec2-user:~/environment/git_practice (main) $ git diff
diff --git a/root.txt b/root.txt
index 5b1dd02..0a9d763 100644
--- a/root.txt
+++ b/root.txt
@@ -1 +1,2 @@
 Some Text
+More Text
ec2-user:~/environment/git_practice (main) $ git diff --staged
```

Now we have made additional changes, so our base command will be helpful now.

- **git diff**: Only Tracks **root.txt** not root.js, since it is only comparing changes to previously committed files.
- **git diff --staged**: No staged changes

Back to the Staging Area

```
ec2-user:~/environment/git_practice (main) $ git add .
ec2-user:~/environment/git_practice (main) $ git diff
ec2-user:~/environment/git_practice (main) $ git diff --staged
diff --git a/root.js b/root.js
new file mode 100644
index 0000000..aeb60ff
--- /dev/null
+++ b/root.js
@@ -0,0 +1 @@
+Some JavaScript
diff --git a/root.txt b/root.txt
index 5b1dd02..0a9d763 100644
--- a/root.txt
+++ b/root.txt
@@ -1 +1,2 @@
 Some Text
+More Text
```

Now we have made additional changes, so our base command will be helpful now.

- **git diff**: All Changes are staged
- **git diff --staged**: This command picks up both **root.txt** and **root.js** since both files were explicitly added to the staging area.
    - In other words, git is now aware of **root.js**

Changes after Staging

Now that we have both staged changes and unstaged changes to a tracked file, both commands will give us feedback.

- **git diff**: Picks up additional changes to **root.js** after initial staging
- **git diff --staged**: This command picks up both **root.txt** and **root.js** since both files were explicitly added to the staging area.