Wichtiges Zeugs

* **xcrun simctl io booted recordVideo app.mov** im Terminal für Videoaufname von app!
* tree -L 1/2/3/…. baustruktur im Terminal
* git push/pull im terminal
* open ./ öffnet pfad wo man gerade ist
* Vi name der file -> öffnet file im editor, dann shift a zum einfügen von sachen, esc zum einfügen un ganz am ende der Line :wq zum bestätigen
* whitch name 🡪 sucht nach file!

**nfctlog:**

**$** nfctlog --help

Usage: nfctlog [OPTIONS] COMMAND [ARGS]...

Options:

-v, --verbosity INTEGER

-p, --port TEXT Serial port eg: COM24 (only used on windows

systems).

--help Show this message and exit.

Commands:

poll

rdt1p Reads one page from memory from protected memory area.

rdt1reg Reads one register at given address Examples: Both read

one...

readnfc

readsw

repl Start an interactive shell.

reset resets the kronegger reader to default settings

resetswstate Resets a started tag and restores after calibration...

rffield Controls RF field

select Sends ISO-14443A select command, logis tag UID if...

trans Sends bytes given as data over the air, logis the response.

trd1p Reads one page from memory.

trd2p Reads two pages from memory.

twr1p Writes one page of data (16 byte) to memory, starting at...

unlockdynamic Unlocks dynamic lock area

writesw Writes .cod file to chip.

wrt1p Writes one page of data (16 byte) to memory, starting at...

wrt1reg Reads one reg

nfctlog resetswstate 🡪 patch zurücksetzen

nfctlog rdt1p -f 🡪patch auslesen u mit control + c stoppen

nfctlog resetswstate -s 17 für neue version

id des patches: mfctlog readnfc bei 12: b``-> 5 0er zum Schluss

Terminal:

Open file in Terminal: vim direction 🡪 vim Fastlane/fastfile

**Git-Help:**

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)

clone Clone a repository into a new directory

init Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)

add Add file contents to the index

mv Move or rename a file, a directory, or a symlink

reset Reset current HEAD to the specified state

rm Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)

bisect Use binary search to find the commit that introduced a bug

grep Print lines matching a pattern

log Show commit logs

show Show various types of objects

status Show the working tree status

grow, mark and tweak your common history

branch List, create, or delete branches

checkout Switch branches or restore working tree files

commit Record changes to the repository

diff Show changes between commits, commit and working tree, etc

merge Join two or more development histories together

rebase Reapply commits on top of another base tip

tag Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)

fetch Download objects and refs from another repository

pull Fetch from and integrate with another repository or a local branch

push Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some

concept guides. See 'git help <command>' or 'git help <concept>'

to read about a specific subcommand or concept.

**Git allg.**

Git sieht Daten nicht in dieser Weise. Stattdessen betrachtet Git seine Daten eher als eine Reihe von Snapshots eines Mini-Dateisystems. Jedes Mal, wenn Du committest (d.h. den gegenwärtigen Status Deines Projektes als eine Version in Git speicherst), sichert Git den Zustand sämtlicher Dateien in diesem Moment („Snapshot“) und speichert eine Referenz auf diesen Snapshot. Um dies möglichst effizient und schnell tun zu können, kopiert Git unveränderte Dateien nicht, sondern legt lediglich eine Verknüpfung zu der vorherigen Version der Datei an. Git betrachtet Daten also wie in Bild 1-5 dargestellt.



[**Die drei Zustände**](#Die-drei-Zustände)

Jetzt aufgepasst. Es folgt die wichtigste Information, die Du Dir merken musst, wenn Du Git kennen lernen willst und Fallstricke vermeiden willst. Git definiert drei Haupt-Zustände, in denen sich eine Datei befinden kann: committed, modified („geändert“) und staged („vorgemerkt“). „Committed“ bedeutet, dass die Daten in der lokalen Datenbank gesichert sind. „Modified“ bedeutet, dass die Datei geändert, diese Änderung aber noch nicht committed wurde. „Staged“ bedeutet, dass Du eine geänderte Datei in ihrem gegenwärtigen Zustand für den nächsten Commit vorgemerkt hast.

Das führt uns zu den drei Hauptbereichen eines Git Projektes: das Git Verzeichnis, das Arbeitsverzeichnis und die Staging Area.



Bild 1-6. Arbeitsverzeichnis, Staging Area (xxx) und Git Verzeichnis

Das Git Verzeichnis ist der Ort, an dem Git Metadaten und die lokale Datenbank für Dein Projekt speichert. Dies ist der wichtigste Teil von Git, und dieser Teil wird kopiert, wenn Du ein Repository von einem anderen Rechner klonst.

Dein Arbeitsverzeichnis ist ein Checkout („Abbild“ xxx) einer spezifischen Version des Projektes. Diese Dateien werden aus der komprimierten Datenbank geholt und auf der Festplatte in einer Form gespeichert, die Du bearbeiten und modifizieren kannst.

Die Staging Area ist einfach eine Datei (normalerweise im Git Verzeichnis), in der vorgemerkt wird, welche Änderungen Dein nächster Commit umfassen soll. Sie wird manchmal auch als „Index“ bezeichnet, aber der Begriff „Staging Area“ ist der gängigere.

Der grundlegend Git Arbeitsprozess sieht in etwa so aus:

1. Du bearbeitest Dateien in Deinem Arbeitsverzeichnis.
2. Du markierst Dateien für den nächsten Commit, indem Du Snapshots zur Staging Area hinzufügst.
3. Du legst den Commit an, wodurch die in der Staging Area vorgemerkten Snapshots dauerhaft im Git Verzeichnis (d.h. der lokalen Datenbank) gespeichert werden.

Wenn eine bestimmte Version einer Datei im Git Verzeichnis liegt, gilt sie als „committed“. Wenn sie geändert und in der Staging Area vorgemerkt ist, gilt sie als „staged“. Und wenn sie geändert, aber noch nicht zur Staging Area hinzugefügt wurde, gilt sie als „modified“. In Kapitel 2 wirst Du mehr über diese Zustände lernen und darüber, wie Du sie sinnvoll einsetzen und wie Du den Zwischenschritt der Staging Area auch einfach überspringen kannst.

### [Deine Einstellungen überprüfen](#Deine-Einstellungen-überprüfen)

Wenn Du Deine Einstellungen überprüfen willst, kannst Du mit dem Befehl git config --list alle Einstellungen anzuzeigen, die Git an dieser Stelle (z.B. innerhalb eines bestimmten Projektes) bekannt sind:

Außerdem kannst Du mit dem Befehl git config {key} prüfen, welchen Wert Git für einen bestimmten Variablennamen verwendet:

## Hilfe finden

Falls Du jemals Hilfe in der Anwendung von Git benötigst, gibt es drei Möglichkeiten, die entsprechende Seite aus der Dokumentation (manpage) für jeden Git Befehl anzuzeigen:

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

Beispielsweise erhältst Du die Hilfeseite für den git config Befehl so:

$ git help config

**For-Schleife swift**

**for** zähler **in** 1...3{

print(zähler)

}

**for** zähler **in** stride(from: 0, to: 100, by: 20){

print(zähler)

}

**for** zähler **in** stride(from: 0, through: 100, by: 20){

print(zähler)

}

**let** array = [1,2,3,4,5,6,7,8,9,10]

**for** (Index,eintrag) **in** array.enumerated(){

print("Der Index \(Index) hat den Eintrag: \(eintrag)")

}

**let** dictionary = ["Mike": 23, "Joe": 27, "Lisa": 25]

**for** (name, alter) **in** dictionary{

print("\(name) ist \(alter) Jahre alt")

**MVC**

**OOP**

**Wann benutze ich Klassen, Stukturen oder Enums?**

* Klassen für referense-Types und Structs und Enums für value-Typs
* Structs um einfachte Typen mit properties zu erstellen
* Enums für die Gruppierung von zusammengehörigen Werten geeignet
* Structs und Klassen werden eher kombiniert
* Struct eignen sich ohne Verfärbung als einfache Typen leichter einsetzten
* Klassen wenn Verfärbung notwendig um bessere Hierarchie zu erstellen, um übersichtliche und Klaren Code zu schreiben. Damit nicht eine gigantische Klasse erstellt werden muss, die alles enthält
* Klassen zum Kopieren von Werten geeignet
* Structs zum referenzieren auf werte

Value Type Instanzen werden kopiert, wenn sie zugewiesen werden; Refernece Type Instanzen werde nur mehrfach referenziert.

Eine Instanz ist ein „Objekt“ eines bestimmten Typens.

Structs haben einen speziellen Initialisier namens „Menberwise Initialisier“.

Dieser enthält automatisch alle Properties des Structs als Parameter.

In Enumerationen können mit Cases mehrere zusammengehörige Werte gruppiert werden und einfach darauf zugegriffen werden.

**Docker:**

docker-compose -f local.yml up