

Distributed Systems Tutorial

Inhaltsverzeichnis

1. Projektthemenauswahl.....	1
Chat Anwendung.....	1
Auktion System	2
Multiplayer online game	2
2. Umgang mit Git/GitHub.com.....	3
1. Teil - Erstellung eines Repository über GitHub, welches dann lokal verändert und dann wieder in GitHub hochgeladen wird	3
2. Teil - Ein Git Repository wird lokal erstellt, welches dann auf GitHub gehostet wird.....	7
3. Teil - Git Branching	9

1. Projektthemenauswahl

In dem ersten Teil des Tutorials werden mögliche Standardbeispiele klassischer Anwendungen von verteilten Systemen vorgestellt. Dies soll die Projektthemenauswahl im Modul Distributed Systems erleichtern, bei denen der Fokus auf die Durchführbarkeit und Demonstrierbarkeit von Algorithmen verteilter Systeme liegt.

Chat Anwendung

Bei dem Beispiel der Chat Anwendung soll gewährleistet werden, dass die Übertragung von Nachrichten in Echtzeit ermöglicht wird.

In der nachfolgenden Tabelle werden Implementierungsmöglichkeiten und deren Herausforderungen aufgezeigt, mit der eine solche Anwendung umgesetzt werden kann.

Umsetzungsmöglichkeiten	Herausforderungen
Architecture Model Client – Server bei der die Teilnehmer die Clients und die Systemkomponente die Server darstellen	Das System kann entweder aus unterschiedlichen Komponenten wie Clients und Servern (Bsp.: WhatsApp) oder aus gleichwertigen Komponenten bestehen wie in Peer to Peer Systemen
Dynamic Discovery Neuer Chat Teilnehmer/Systemkomponente entdeckt den Server, der die Chat Anwendung anbietet und tritt dem Chat/System bei	<ul style="list-style-type: none">• Neue Teilnehmer sollen einem Chat ohne Neukonfiguration oder Neustart der bestehenden Chat Teilnehmer beitreten können• Neue Systemkomponenten sollten ohne Neukonfiguration oder Neustart des Systems in das System gelangen
Leader election (voting) LCR-Algorithmus: Wird verwendet, um eine Systemkomponente zu bestimmen, die für koordinierende Aufgaben zuständig ist	<ul style="list-style-type: none">• Neue Chat Teilnehmer müssen möglicherweise von einer zuständigen Komponente akzeptiert/genehmigt werden• Systemkomponente müssen möglicherweise von einem Koordinator auf deren Funktionsfähigkeit überprüft werden
Reliable ordered multicast Um einen zuverlässigen Nachrichtenaustausch zu garantieren ist ein Multicast erforderlich	<ul style="list-style-type: none">• Soll die Zuverlässigkeit bei Nachrichtenaustausch garantiert werden?

Damit die Nachrichten in der richtigen Reihenfolge empfangen werden, ist zumindest eine FIFO-Ordnung von Nöten	<ul style="list-style-type: none"> Für verschiedene Aspekte des Systems können unterschiedliche Mechanismen für den Austausch von Nachrichten erforderlich sein (z. B. können die Anforderungen an die Reihenfolge der Nachrichten auf Chatebene und auf Systemebene unterschiedlich sein)
Fault Tolerance Erkennung und Behandlung von Absturz Fehlern von Client, Leader und Server	Was passiert, wenn Chat Teilnehmer/Systemkomponenten nicht mehr verfügbar/erreichbar sind?

Auktion System

Bei dem Beispiel eines Auktion Systems soll gewährleistet werden, dass die Erstellung und Abgabe von Geboten auf Auktionen in Echtzeit durchgeführt werden können

Herausforderungen:

- Bieter sollten eine Auktion betreten und verlassen können, ohne die Auktion zu unterbrechen
- Neue Systemkomponenten sollten in das System eingeführt werden können, ohne den Systembetrieb zu stören
- Sollen alle Teilnehmer alle Gebote erhalten?
- Was passiert, wenn ein Leader ausfällt? Wie kann die Konsistenz sichergestellt werden, da das letzte Höchstgebot pro Auktion benötigt wird, damit das System ordnungsgemäß weiter funktioniert?

Multiplayer online game

Bei dem Beispiel eines multiplayer online game soll gewährleistet werden, dass das Spiel von mehreren Spielern gleichzeitig gespielt werden kann.

Herausforderungen:

- Wo soll die Spielwelt gespeichert werden?
- Wie kann ein neuer Spieler ein Spiel (oder eine Gruppe von Spielern) finden und beitreten?
- Wie können neue Systemkomponenten in das System integriert werden, ohne das Spiel zu unterbrechen?
- Wie kann eine einheitliche Darstellung der Spielwelt gewährleistet werden?
- Wie lässt sich eine konsistente Sicht auf das System (globaler Zustand) über viele Server hinweg sicherstellen?
- Ist die Reihenfolge der von den Akteuren durchgeführten Aktionen wichtig? Welche Garantien sind dabei erforderlich?

Dies waren drei Standardbeispiele klassischer Anwendungen von verteilten Systemen. Aufgezeigt wurden die Herausforderungen, die zu bewältigen sind.

2. Umgang mit Git/GitHub.com

Im folgenden Tutorial wird zunächst auf die zwei Begriffe Git und GitHub eingegangen, um diese besser voneinander abgrenzen zu können. Im weiteren Verlauf wird der Umgang mit Git und GitHub anhand eines Beispiel Projekts näher erläutert.

Git:

Ist eine Versionskontroll- Software für Entwickler. Dies ermöglicht es Entwicklern, den Überblick zu behalten und zu einer früheren Phase zurückzukehren, wenn sie beschließen, einige der vorgenommenen Änderungen rückgängig zu machen. Mit der Versionsverwaltung können die verschiedenen Versionen/Phasen in einem sog. Repository zusammengefügt werden, mit dem Ihr als Team gemeinsam arbeiten könnt. Auch wenn es ein zentrales Repository für ein Projekt gibt, können sich die Teammitglieder eine lokale Kopie des Repository auf Ihr Arbeitsgerät ziehen, um auch offline daran arbeiten zu können. Die unterschiedlichen Versionen in den verschiedenen Repositories kann man dann bei Bedarf zusammenführen (mergen).

GitHub:

GitHub ist ein Hoster von Git und stellt eine Code Hosting Plattform dar. Hiermit wird ermöglicht seine Git Projekte online in der Cloud oder im Web zu hosten. GitHub ermöglicht wie mit Git Versionen von Software- Projekten zu verwalten, zu älteren Versionen zurückzukehren, branchen, mergen und generell im Team gleichzeitig an Projekten zu arbeiten. Der Unterschied zu Git besteht darin, das Git auf der lokalen Maschine läuft und GitHub in der Cloud bzw. im Web Git hosted.

1. Teil - Erstellung eines Repository über GitHub, welches dann lokal verändert und dann wieder in GitHub hochgeladen wird

Um mit GitHub arbeiten zu können ist ein Account zwingend notwendig. Darüber hinaus muss Git installiert werden, um es auf deiner lokalen Maschine verwenden zu können. Bei der Auswahl des Code Editors könnt ihr frei nach euren Präferenzen wählen. Der verwendete Code Editor im Beispiel Projekt ist Visuell Studio Code, welcher über einen integrierten Terminal verfügt.

1. Repository anlegen

Legt ein neues Repository auf GitHub an, welches auf public gestellt ist und eine README file enthält.

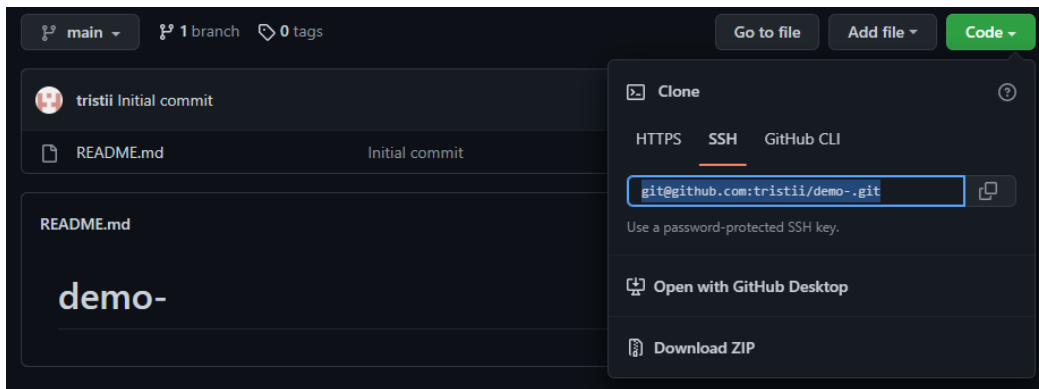
2. SSH Key anlegen und registrieren

Um auf Ihr GitHub lokal Repository zuzugreifen, klonen Sie den HTTPS-Link und geben dann beim Hochladen (Pushen) von Änderungen Nutzernamen und Passwort ein. Um bei häufigem Pushen nicht ständig die Anmeldedaten anzugeben zu müssen, gibt es die Möglichkeit SSH-Schlüssel für die passwortlose Anmeldung einzurichten.

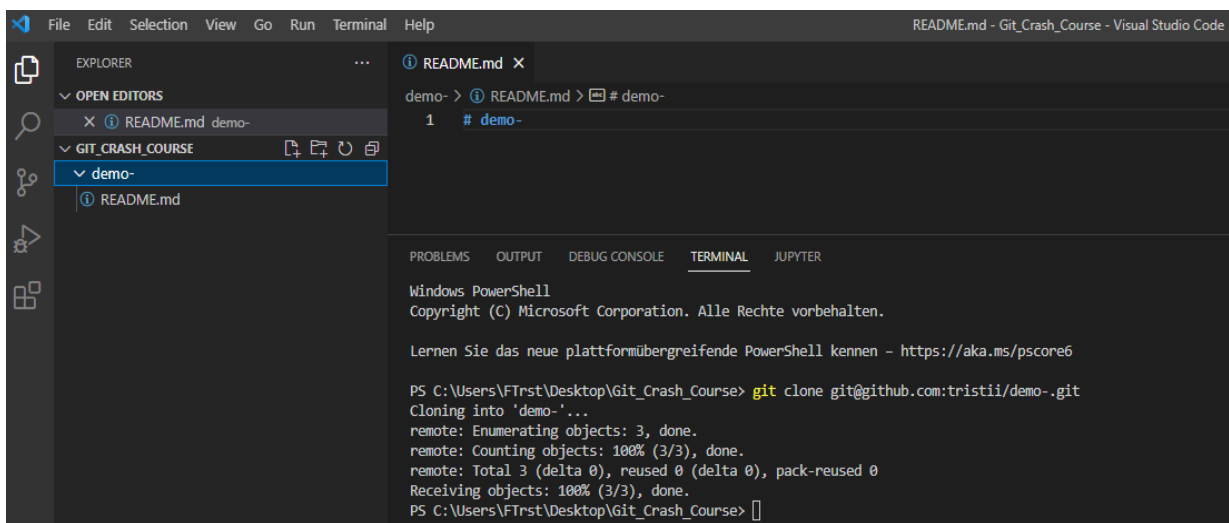
Hierbei klonen Sie das Repository über den alternativen SSH-Link und die Authentifizierung übernimmt dann ein Schlüsselpaar. Das Schlüsselpaar aus privatem/geheimem und öffentlichem Schlüssel erstellen Sie lokal. Der öffentliche Schlüssel wird bei GitHub hinterlegt. Nähere Infos bzgl. des Erzeugens eines SSH Schlüssels und das Hinzufügen zu einem ssh- agent können dem folgenden GitHub-Link entnommen werden: <https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent>

3. Repository klonen

Nun wollen wir das Repository auf unsere lokale Maschine klonen, um damit auch lokal daran zu arbeiten. Hierfür muss zunächst im Terminal zu dem gewünschten Zielverzeichnis navigiert werden, indem das Repository abgelegt werden soll. Daraufhin wird der SSH- Link des GitHub Repository kopiert.



Im Terminal wird nun der Befehl **git clone** und der **kopierte SSH- Link** eingefügt. Nach betätigen der Eingabetaste sollte das Repository als Ordner im Zielverzeichnis abgelegt sein. In diesem Beispiel in VS-Code wird das „demo-“ Repository direkt angezeigt da sich der Editor bereits im Zielverzeichnis befindet.



4. Git Commit

Um in das Repository zu gelangen, muss noch mit **cd** (change direction) und der Name des Repos (**demo-**) zu dem Zielverzeichnis navigiert werden. Nun editieren wir unsere README file und speichern die Änderung lokal ab. Über **git status** kann man sich eine Übersicht über alle lokalen Änderungen verschaffen.

```
demo- > ⓘ README.md > Ⓜ # demo-
1  # demo-
2
3  Hello World

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

PS C:\Users\FTrst\Desktop\Git_Crash_Course> cd demo-
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo-> git status
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo-> |
```

Die Datei mit der Änderung muss zunächst mit dem **git add + Dateiname** Befehl in den Staging-Bereich (Ort, wo die Dateien landen, wenn sie mit dem add Befehl vorgemerkt werden) verschoben werden. Mit dem **git commit -m** "beliebige Nachricht" kann ein Commit mit der geänderten Datei aus dem Stage erstellt werden

```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo-> git add README.md
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo-> git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md

PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo-> git commit -m "change README file"
[main 29898ab] change README file
1 file changed, 1 insertion(+), 1 deletion(-)
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo-> git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

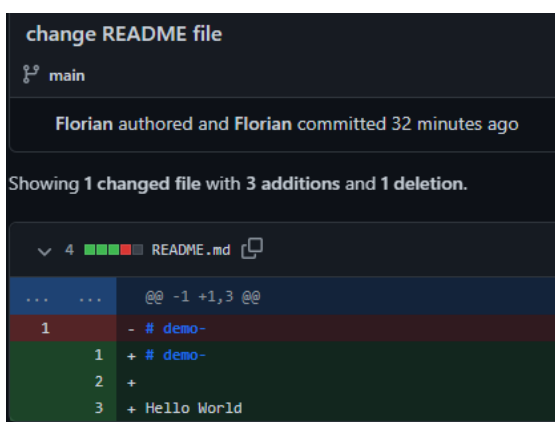
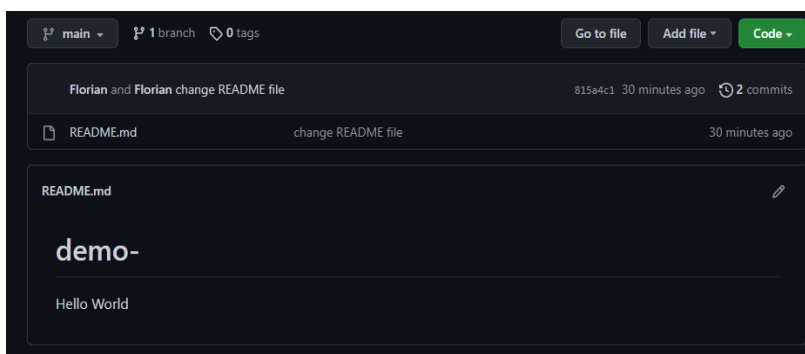
5. Git push

Bis jetzt wurde der Code nur lokal abgespeichert. Um dieses Commit auf GitHub zu veröffentlichen, muss mit dem Befehl **git push** der Commit live zu unserem Remote Repository übermittelt werden, wo das Projekt gehostet ist. Zwei weitere Argumente müssen dem Befehl **git push** angehängt werden. 1. **origin**, steht hierbei für den Speicherort unseres Git Repository und 2. **main**, welches den Branch darstellt, an den gesendet (push) werden soll.

```
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo-> git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use "git push" to publish your local commits)

nothing to commit, working tree clean
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo-> git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 272 bytes | 90.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:tristii/demo-.git
 48dcf90..815a4c1  main -> main
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo-> █
```

Die Änderungen, die lokal erstellt und auf GitHub hochgeladen/veröffentlicht wurden, sind nun auf GitHub live sichtbar (2 commits).

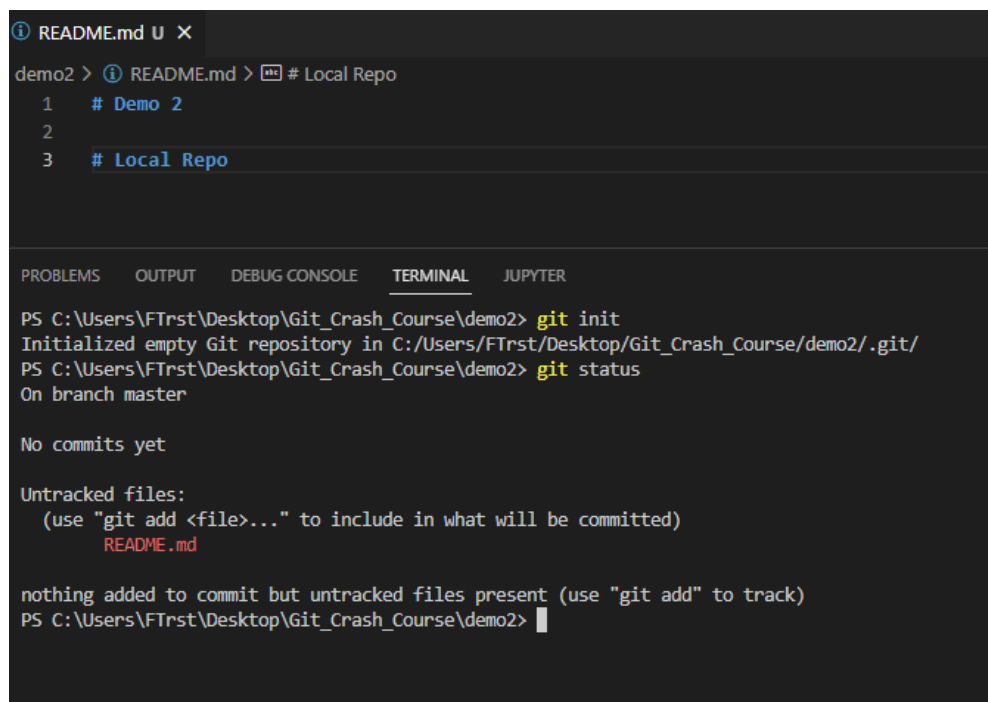


Dies war die Demonstration, wie ein Git Repository über GitHub erstellt und dann lokal Änderungen vorgenommen worden sind, die dann wiederum an GitHub übermittelt wurden.

2. Teil - Ein Git Repository wird lokal erstellt, welches dann auf GitHub gehostet wird

1. Repository anlegen

Erstelle ein Repository lokal in einem beliebigen Code Editor und füge eine **README.md file** hinzu, in die du Text einbettest. Navigiere im Terminal zu dem Verzeichnis, indem das Repository liegt. Mit **git init** wird das Repository dann als ein leeres Git Repository initialisiert. **git status** zeigt auf das Dateien innerhalb des Git Repository unbekannt sind (**untracked files**). Diese Datei wurde im Repository erstellt aber nicht mit **git add** hinzugefügt.



The screenshot shows a code editor with a file named **README.md** open. The file content is:

```
demo2 > ① README.md > [ms] # Local Repo
1 # Demo 2
2
3 # Local Repo
```

Below the editor is a terminal window with the following output:

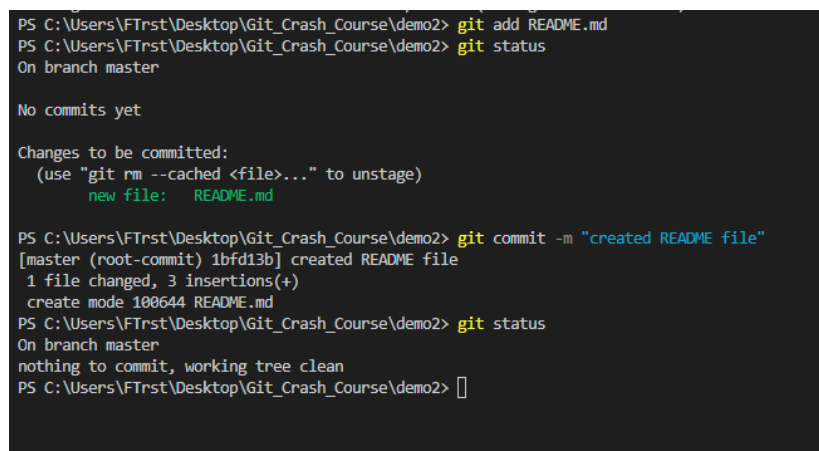
```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git init
Initialized empty Git repository in C:/Users/FTrst/Desktop/Git_Crash_Course/demo2/.git/
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
  README.md

nothing added to commit but untracked files present (use "git add" to track)
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2>
```

Füge die README File mit **git add + Dateiname** hinzu. Danach wird ersichtlich über **git status**, dass die Datei bereit ist committed zu werden, welches mit dem Befehl **git commit -m "beliebige Nachricht"** erfolgt.



The screenshot shows a terminal window with the following output:

```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git add README.md
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git status
On branch master

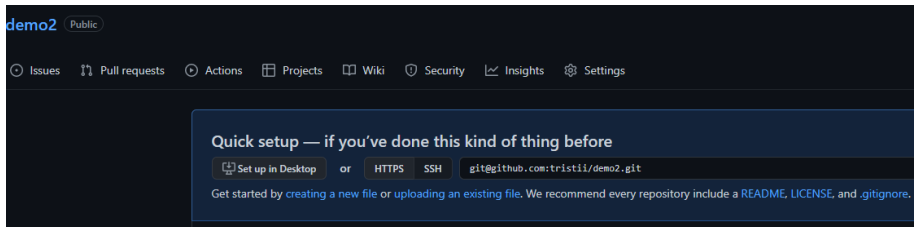
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
  new file:   README.md

PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git commit -m "created README file"
[master (root-commit) 1bfd13b] created README file
1 file changed, 3 insertions(+)
create mode 100644 README.md
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git status
On branch master
nothing to commit, working tree clean
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2>
```


2. Verbindung zu einem Remote Repository herstellen

Um den Commit auf ein Remote Repository zu pushen, muss zunächst eine Verbindung zu solch einem hergestellt werden, da das Git Repository lokal erstellt wurde. Der einfachste Weg eine Verbindung zu einem Remote Repository aufzubauen ist ein leeres GitHub Repository mit dem gleichen Namen des lokalen Repository zu benennen.



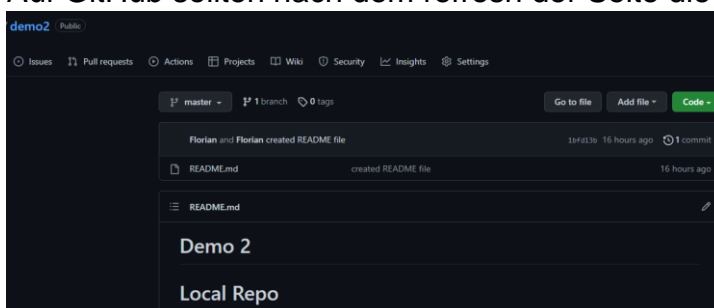
Daraufhin gebt Ihr im Terminal mit dem Befehl `git remote add origin git@github.com:tristii/demo2.git` und den **SSH- Link** des GitHub Repository ein und bestätigt eure Eingabe mit Enter. Nun ist euer lokales Git Repository mit dem remote Repository verbunden. Um die Connection zu überprüfen können sie mit dem Befehl `git remote -v` alle remote Repositories anzeigen lassen, die mit diesem lokalen Repository verbunden sind.

```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git remote add origin git@github.com:tristii/demo2.git
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git status
On branch master
nothing to commit, working tree clean
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git remote -v
origin  git@github.com:tristii/demo2.git (fetch)
origin  git@github.com:tristii/demo2.git (push)
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> |
```

Jetzt kann über den Befehl `git push origin master` die Änderungen auf das remote Repository hochgeladen werden.

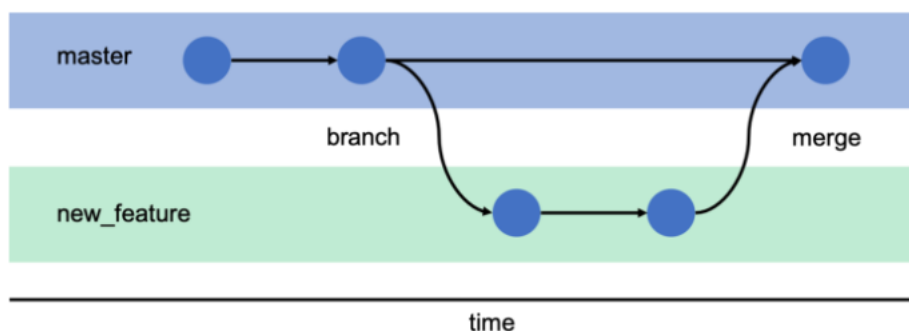
```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git push origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 247 bytes | 61.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:tristii/demo2.git
 * [new branch]      master -> master
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> |
```

Auf GitHub sollten nach dem refresh der Seite die commits zu sehen sein.



3. Teil - Git Branching

Im dritten Teil geht es um Git Branching. Bis jetzt wurden alle commits auf dem Master Branch ausgeführt. Dieser stellt eine festgelegte Namensbezeichnung für den Main bzw den default Branch in einem Repository dar. Sobald mehrere Personen in einem Projekt aktiv sind, sollten mehrere Branches genutzt werden. Ein Branch stellt hierbei Abzweigung zur sog. Master/Main Branch dar, welches eine komplett unabhängige Entwicklungslinie mit einer eigenen Historie darstellt. Jede kleine oder große Änderung entsteht in einem eigenen Branch und somit wird die Code Basis erstmal nicht beeinträchtigt. Beispiele für das Nutzen einer weiter Branch sind die Implementation eines neuen Features oder eines Hotfix. Sobald ein Feature ausführlich getestet wurde, ist es jederzeit möglich einen Branch mit der Hauptlinie bzw. dem Master/Main Branch zusammenzuführen. Diesen Vorgang wird merge genannt.



Um die Funktionsweise von Branching besser zu verstehen, erstellen wir in unserem **Demo2 Repository** einen Feature Branch, der am Ende dann mit dem Master Branch zusammengeführt (merge) werden soll.

1. Branch

Mit dem Befehl **git branch**, sieht man auf welchem Branch sich aktuell befunden wird. Zunächst muss mit **git checkout -b + Name der neuen Branch**, ein neuer Branch erstellt werden, auf den direkt gewechselt wird. Mit **git checkout master** können zwischen den verschiedenen Branches gewechselt werden, in meinem Fall zurück zum Master Branch.

```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git branch
* master
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git checkout -b feature
Switched to a new branch 'feature'
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git branch
* feature
  master
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git checkout master
Switched to branch 'master'
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> |
```

2. Änderungen auf dem Branch „Feature“

Modifiziere die README file, wenn du dich auf dem Branch **feature** befindest. Danach lassen sich mit **git status** eine Übersicht über mögliche Änderungen anzeigen. Die Änderungen müssen nun gespeichert werden. Dafür wird zunächst mit **git add README.md** die Änderungen zur Staging-Area hinzugefügt werden, bevor dann über **git commit -m "beliebige Nachricht"** die Datei committet. Jetzt ist die Änderung in Git gespeichert, aber nur auf dem Branch „Feature“.

```
README.md X
demo2 > README.md > # Local Repo > ## Feature Development
1 # Demo 2
2
3 # Local Repo
4
5 ## Feature Development
6
7 1. Requirement Analysis
```

```
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo2> git checkout feature
Switched to branch 'feature'
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo2> git status
On branch feature
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo2> git add README.md
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo2> git commit -m "updated README"
[feature 66cca39] updated README
 1 file changed, 5 insertions(+), 1 deletion(-)
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo2> 
```

3. Merge

Wenn zum „Master“ Branch mit **git branch checkout master** gewechselt wird, erkennt man, dass die Änderung hier nicht vorgenommen wurde, sondern nur im Branch „Feature“

```
README.md X
demo2 > README.md > # Local Repo
1 # Demo 2
2
3 # Local Repo

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo2> git checkout master
Switched to branch 'master'
PS C:\Users\FTTrst\Desktop\Git_Crash_Course\demo2> 
```

Mit **git diff + Name der Branch** können sich vorgenommene Änderungen anzeigen lassen, indem die beiden Versionen miteinander verglichen werden. Dies kann hilfreich sein, um nochmals zu überprüfen was danach zusammengeführt werden soll.

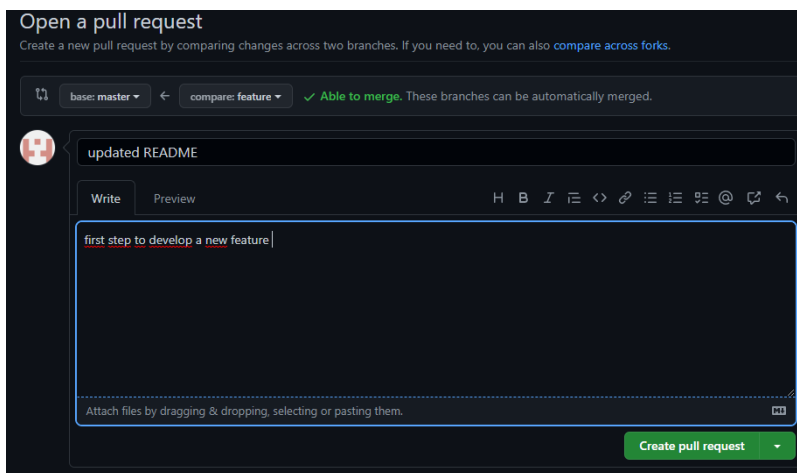
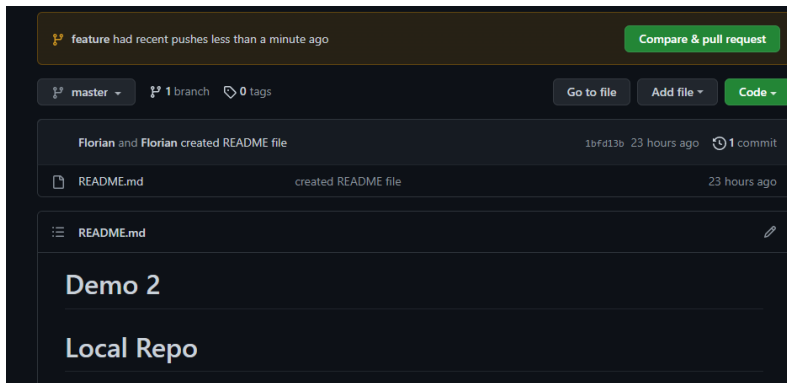
```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git diff feature
diff --git a/README.md b/README.md
index 07bd88c..e272dbd 100644
--- a/README.md
+++ b/README.md
@@ -1,7 +1,3 @@
 # Demo 2

-# Local Repo
-
-## Feature Development
-
-1. Requirement Analysis
\ No newline at end of file
+# Local Repo
\ No newline at end of file
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> 
```

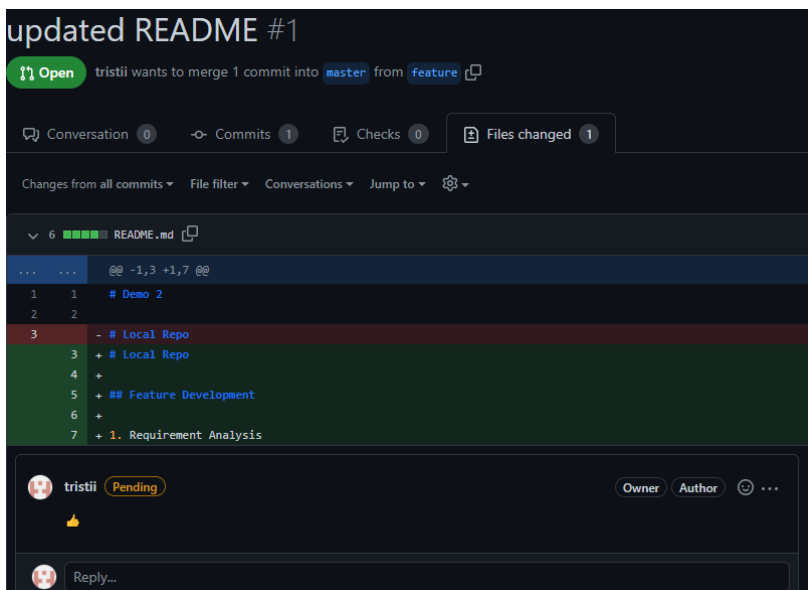
Bevor diese Änderung jetzt zusammengeführt wird, werden die Änderungen auf der Branch „Feature“ GitHub zur Verfügung gestellt. Dies stellt die gewöhnliche und realitätsnahe Vorgehensweise dar, um effizientes und transparentes Arbeiten im Team an einem Projekt zu ermöglichen. Dafür muss zunächst auf den **Branch „Feature“ gewechselt** werden und dann mit **git push origin + Name der Branch** die Änderungen GitHub zur Verfügung gestellt werden. Nun können wir einen sog. PR (Pull Request) machen. Grundsätzlich ist es eine Anfrage, ob Ihr Code in einen anderen Branch gezogen werden soll. In diesem Beispiel haben wir einen Feature Branch und wir wollen diesen Code in den Master Branch ziehen. Diesen Pull request werden wir auf der GitHub Oberfläche durchführen.

```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git checkout feature
Switched to branch 'feature'
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git status
On branch feature
nothing to commit, working tree clean
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git push origin feature
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 314 bytes | 104.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'feature' on GitHub by visiting:
remote:   https://github.com/tristii/demo2/pull/new/feature
remote:
To github.com:tristii/demo2.git
 * [new branch]   feature -> feature
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> 
```

Nachfolgend kann über den Button **compare & pull request** ein PR erstellt werden. Im Beispiel wurde eine Kurze Beschreibung hinzugefügt.



Nach dem Erstellen kann man sich über **files changed** die Änderungen anzeigen und ggf. Kommentare zu einzelnen Code Zeilen hinzufügen.



Wenn alles so weit passt kann über den Reiter **Conversation** über den Button **merge pull request** die Änderungen der Feature Branch mit der Master Branch zusammengeführt werden. Die Änderungen sollten nun im Master Branch auf GitHub zu sehen sein. Um die Änderungen auch lokal auf dem Master Branch sehen zu können müssen diese zunächst von GitHub auf die lokale Umgebung **mit git pull origin master** gezogen werden. Nun sollte der lokale Master Branch aktualisiert worden sein.

```
1 README.md X
demo2 > 1 README.md > 2 # Local Repo > 3 ## Feature Development
1  # Demo 2
2
3  # Local Repo
4
5  ## Feature Development
6
7  1. Requirement Analysis

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER powershell
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git checkout master
Switched to branch 'master'
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git pull origin master
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 622 bytes | 103.00 KiB/s, done.
From github.com:tristii/demo2
* branch      master      -> FETCH_HEAD
1bfd13b..8c19226 master    -> origin/master
Updating 1bfd13b..8c19226
Fast-forward
 README.md | 6 +++++
 1 file changed, 5 insertions(+), 1 deletion(-)
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2>
```

Wenn ein Branch nicht mehr genutzt wird bzw. mit dem Master Branch zusammengeführt (merged) worden ist, ist es üblich den Branch zu löschen, da er meist nicht mehr wiederverwendet wird. Um den Branch lokal zu löschen, wird der Befehl **git branch -d <branch>** und für das Löschen remote muss der Befehl **git push origin --delete <branch>** ausgeführt werden.

```
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git branch
feature
* master
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git branch -d feature
Deleted branch feature (was 66cca39).
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2> git push origin --delete feature
To github.com:tristii/demo2.git
- [deleted]      feature
PS C:\Users\FTrst\Desktop\Git_Crash_Course\demo2>
```

Mit diesem Teil endet das Tutorial, welches die Basics im Umgang mit Git/GitHub widerspiegelt. Verwendet wurden die grundlegenden Git Befehle, die als Einstieg in die Versionskontrollsoftware essenziell sind.