

# Afgangs Projekt – IT-Teknolog

2024

*Navn:* Jonas Mansoor Larsen

*Skole:* UCN, Sofiendals vej. IT-teknolog

*Vejleder:* Henning Thomsen

*Anslag:* 36.848



Indledning .....	5
Problemformulering .....	5
Krav .....	6
Afgrænsning .....	6
Planlægning .....	7
Retrospektivt blik på planlægning .....	7
Teknisk analyse .....	8
Design .....	8
Valg af GitHub .....	8
Valg af Python .....	8
Valg af Requests library .....	8
Valg af Fpdf .....	8
Tanker om Interface .....	9
Mock-Up af GUI: .....	9
Valg af custom tkinter .....	9
Tanker om mulige ekstra features: .....	9
Ønsket flow i systemet .....	10
Opsamling af designprocess .....	10
Implementering .....	11
Opsætning af GitHub .....	11
Generering af PAT .....	11
API .....	13
Hvad er en API? .....	14
HTTP-forespørgsler .....	14
Struktur af en HTTP request .....	15
HTTP metoder .....	15
HTTP Headers .....	15
Body .....	16
Struktur af en HTTP response .....	16
REST .....	16
Uniform interface .....	17
Client-server decoupling .....	17

Statelessness.....	17
Cacheability .....	17
Layered system architecture .....	17
Hvordan virker REST API'er? .....	17
Postman .....	18
JSON.....	18
System moduler .....	18
Initialisering af applikationen.....	20
API klient.....	21
Dataformatter .....	22
FormatReponse(self, response) .....	22
FetchRepos(self, response) .....	22
FormatCommits(self, response, chosen_repo, chosen_branch) .....	22
Txtbox modul.....	24
Test af systemet .....	25
Manuel test af header opsætning og parametre .....	25
RESTClient.....	25
I main: .....	26
Resultat: .....	26
Manuel test af multiple rendering af dropdowns: .....	26
Fix af multiple rendering.....	26
Resultat: .....	27
Bug or “by-design?” .....	27
Opsummering af test .....	28
Delkonklusion, fra design til løsning .....	28
Diskussion .....	29
Tidsforbrug og effektivitet.....	29
Frigørelse af ressourcer.....	29
Svagheder .....	29
Perspektivering .....	29
Avancerede filtrerings muligheder.....	31
Dokument opsætning .....	31

Forbedret GUI.....	31
Automatisering.....	31
Diverse .....	31
Opsamling af perspektivering .....	31
Konklusion .....	30
Referencer .....	32

## Indledning

I en udviklingsafdeling er effektivitet altafgørende for at opretholde en konkurrencedygtig fordel. Hver gang der frigives en ny version af et softwareprodukt, er det nødvendigt at dokumentere ændringer i form af release- eller change-notes. Disse noter informerer brugere, udviklere og andre parter om nye funktioner, fejlrettelser og diverse ændringer, der er implementeret siden den tidligere version. Imidlertid kan den manuelle proces med at udarbejde disse noter være både tidskrævende og uoverskuelig i forhold til, at huske på hvad der er lavet, slettet og flyttet.

Udviklere kan bruge meget tid på at gennemgå commit history for kodeændringer og issues for at generere nøjagtige release-notes. Denne proces kan tage værdifuld tid væk fra andre kritiske opgaver såsom kodning, debugging og planlægning. Derudover er der risiko for menneskelige fejl, som kan føre til ufuldstændige eller ukorrekte release-notes, hvilket kan skabe forvirring og ineffektivitet blandt holdet og slutbrugere.

Formålet med det projekt er at udvikle en løsning, der kan minimere manuelt arbejde i forhold til genereringen af release- og change-notes, og dermed reducere den tid og de ressourcer, der bruges på denne opgave. Implementeringen af denne proces kan sikre en mere konsistent og nøjagtig dokumentation af ændringer, hvilket i sidste ende vil øge effektiviteten i udviklingsafdelingen. Denne løsning agter at være integrerbar med eksisterende infrastruktur.

## Problemformulering

Hovedspørgsmål:

"Hvordan kan man optimere tidsforbruget i en udviklingsafdeling ved at minimere manuel håndtering af dokumentationsprocesser, specifikt håndtering af release- og change notes"

Underspørgsmål:

Hvordan påvirker manuel håndtering af release notes effektiviteten i udviklingsafdelingen?

## Krav

### Ikke funktionelle krav

- Accessibility

Alle udviklere skal have mulighed for at lave ændringer i disse notes samt adgang til programmet. Programmets formål er at spare en hel del tid for den almene udvikler såvel som holdet

- Scalability

Det skal være skalerbart, ift. at det skal kunne bruges på alle repos man har adgang til, samt ønskes det at der skal kunne være plads til tilføjelser, såsom translation og andre moduler.

- Skal kunne integreres i den nuværende infrastruktur

Meget vigtigt at det kan integreres sømløst til den nuværende infrastruktur, det skal kunne opsættes uden nogle former for nedetid

### Funktionelle krav:

- Skal kunne generere release/change notes fra specifikke repos og branches
- UI/GUI til at bestille notes
- Notes skal udelukkende kunne bearbejdes internt
- Oversætning af release notes uden 3<sup>rd</sup> party såsom Google translate

## Afgrænsning

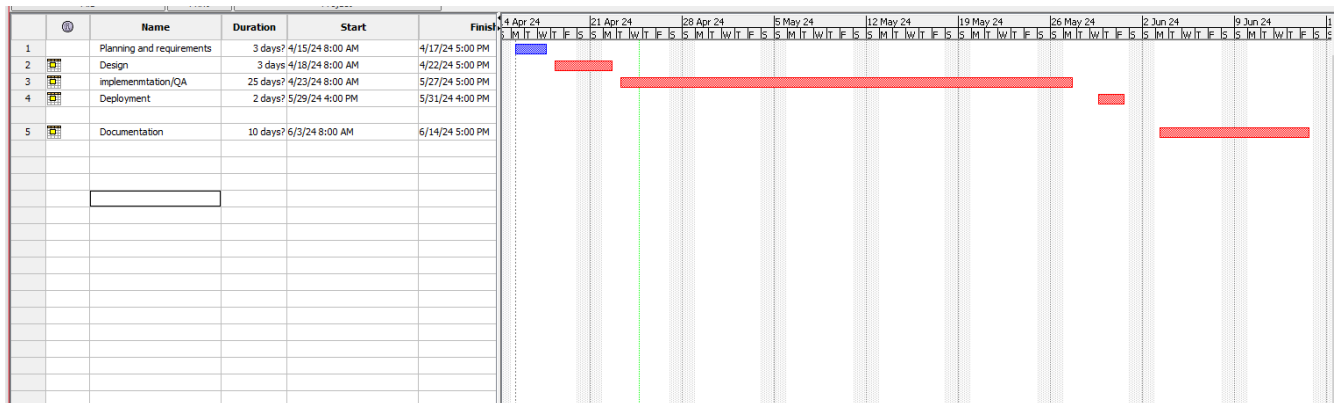
Auto-Test: Grundet systemets lille størrelse, blev det vurderet at det ikke var grund til auto-test cases. Det ville være for tidskrævende at opsætte et framework til det formål og hertil at udvikle disse test-cases

Sikkerhed: Da systemet formentligt ville køre på et internt netværk, ville der ikke være de største sikkerhedsrisici, ydermere er den data som systemet henter lige pt, frit tilgængelige alligevel. Den kan kun hente data fra public repos lige nu, den kan ikke pushe noget data i nuværende state. Derfor er sikkerhed ikke blevet prioriteret

Maintenance i SDLC: Da det er et eksamensprojekt og ikke et produkt der udvikles for at komme på markedet, er der ikke umiddelbar grund til at gå igennem til en maintenance fase i SDLC da det er et post-releasestadie

## Planlægning

Planlægningen i det projekt, har været baseret ud fra vandfaldsmodellen. Grunden til det er bl.a at projektets udvikling har båret præg af SDLC-modellen (Software Development Life-Cycle) Dette giver en meget lineær struktur til den generelle udvikling og fremgang, som passer godt til vandfaldsmodellen, som er karakteriseret af denne sekventielle tilgang til projektstyringen og udviklingen, samt klart definerede målsætninger, hertil med minimale chancer for betydelige ændringer.



Ift SDLC, kan man tyde hvilke faser der ikke er medtaget grundet projektets omfang og projektgruppens størrelse. Der er bl.a taget beslutning om at kombinere Udvikling og testning, da det giver mest mening med én person. Så der er løbende blevet testet manuelt på nye features.

### Retrospektivt blik på planlægning

- Planning and requirements: Denne fase foreløb på fornuftig vis, ved at indhente nødvendig informationer om den teoretiske del af systemet, der blev opsat krav til systemet og planlagt herefter.
- Design: Denne fase var fuld af research og refleksion over tanker til systemet. Dog var et af hovedprincipperne heri at det skulle være simpelt at bruge og overskueligt at kigge på. Der blev fundet frameworks som var nemt integrerbare og der blev dannet tanker om et eventuelt UI
- Implementation/QA: Denne fase gik meget med opsætning og læsning af dokumentation mens der sideløbende blev udviklet primitive moduler til test af systemets funktionaliteter for senere at udbygge disse efter best practice.
- Deployment: Denne fase blev nedprioriteret til fordel for mere tid til de andre opgaver, efter at man fandt for mange komplikationer i opsætningen
- Documentation: Denne fase brugte mere tid end forventet på diagrammering og lignende.

# Teknisk analyse

## Design

For at danne det bedste overblik og struktur over udviklingen af projektet, skulle der igennem en designfase, tages beslutninger omkring frameworks og scope ud fra kravene. Denne proces startede med en god omgang research omkring funktionaliteten i forskellige frameworks.

## Valg af GitHub

Det første valg der skulle foretages mht design, omhandlede hvor en virksomhed eller brugers codebase, ville være opbevaret. Dette er en svær ting at skrænke sig ind på, med de mange forskellige muligheder der er. F.eks. Azure Repos, GitHub, GitLab og mange flere. Derfor vil det projekt, tage udgangspunkt i GitHub, da det har en meget veludført API dokumentation, det er nemt tilgængeligt for private brugere og har et enormt community. Slutteligt er det også den platform som der er brugt i tidligere projekter og i mange virksomheder.

## Valg af Python

Når man udvikler et stykke software, er det selvfølgelig yderst vigtigt at beslutte sig, hvilket sprog, man vil programmere i. Valget omkring at udvikle i python, blev truffet hovedsageligt på baggrund af simpliciteten af sproget og den store mængde af biblioteker til formålet, ydermere er det sproget, som ville kræve mindst mulig ramp-up time i form af research, da det er sproget som har været brugt i tidligere projekter. Man kunne sagtens argumentere for at det ville give mere mening at udvikle systemet i JavaScript. Det er fordi JavaScript er bygget hen mod denne slags udvikling og har mulighed for asynkron kodning og derfor en stor dynamik i applikationen. Dog ville det tage for meget tid fra projektet at skulle sætte sig fuldt og fast ind i det sprog og man ville ende med at få et mindre færdigt produkt.

## Valg af Requests library

For at håndtere API requests i Python, skal man bruge et tredjeparts bibliotek. Requests biblioteket, er et af de mest brugervenlige og ligetil, man kan finde. Du kan sende en request og se response på kun 3 linjers kode. Til det projekt, virkede det tilstrækkeligt med de funktionaliteter det indeholdt. Der var muligheder for at sende requests, parse af response, HTTP status codes mfl.

## Valg af Fpdf

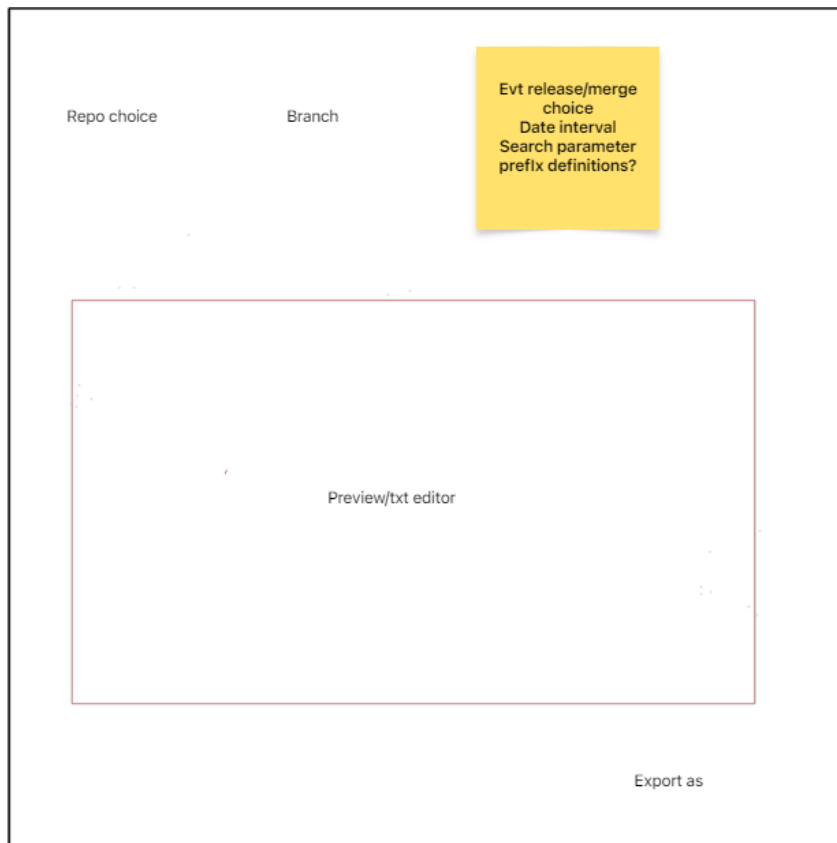
For at kunne output disse commit til et format som kunne bruges til change og/eller release notes, skulle disse kunne formateres til PDF format. Fpdf er et gratis PDF-konverterings bibliotek, oprindeligt lavet til PHP men nu også til Python. Dette bibliotek havde en okay dokumentation men med en meget basal og let forståelig sideopsætning, var det at foretrække fremfor alternativerne.



## Tanker om Interface

Der var flere forskellige løsninger i forhold til hvordan en evt bruger, skulle kunne tilgå disse commit og formatere dem. Dog var det eneste der gav ordentlig mening et GUI (Graphical User Interface). Dette interface skulle helst være nemt at bruge, specielt for første gangs brugere. Der blev tegnet et mock-up af hvordan det skulle se ud. Hertil var der stor fokus på simplicitet. Der er ingen grund til at rode det med overflødige funktioner. Der blev fokuseret på kernesystemet og så er der altid mulighed for udvidelser efterfølgende.

### *Mock-Up af GUI:*



Der skulle være mulighed for at vælge repository og tilhørende branch. Hertil skulle man også have en text editor i sin frame, for at kunne følge med i hvad man opsamlede. Der skulle være en export knap, med mulighed for at vælge formater som TXT og PDF. Der er på mocket også en gul sticky note med filtrerings ideer til at gennemse commits.

### *Valg af custom tkinter*

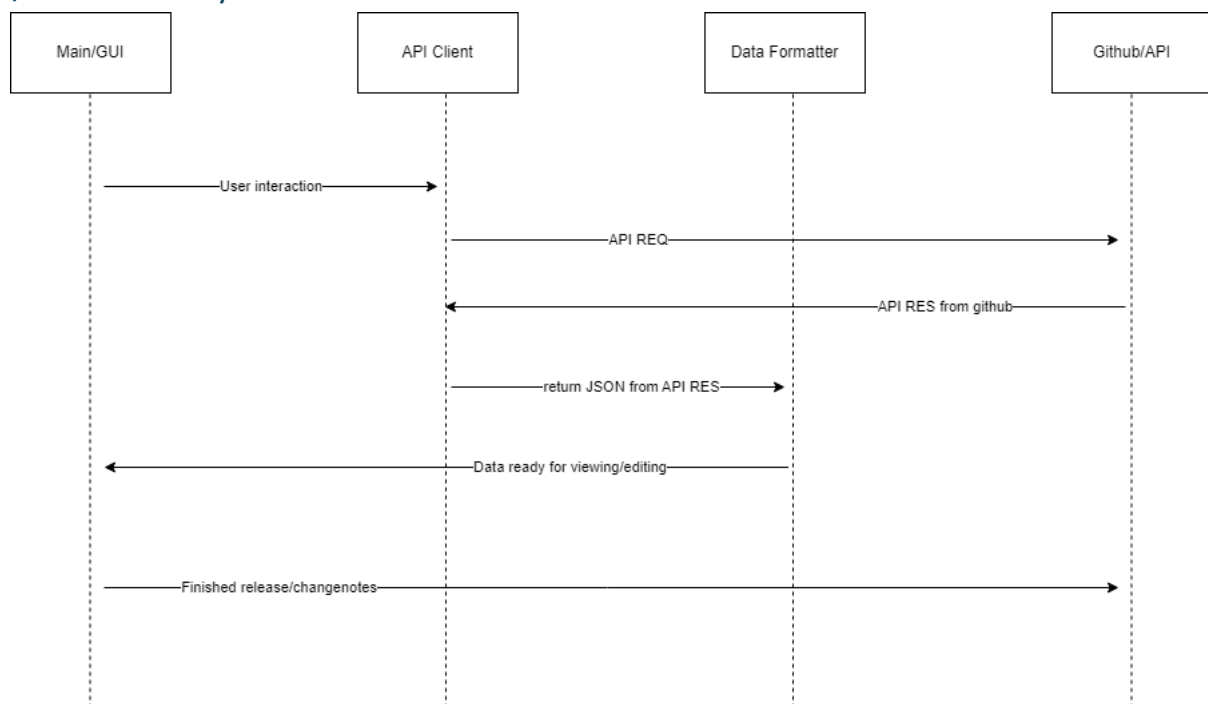
For at lavet et GUI i Python, er der et par forskellige muligheder. Man har tidligere erfaring med Tkinter, dog blev det ofte meget uforudset i forhold til placering af diverse elementer og andre små fejl. Der er udkommet en videreudvikling af det bibliotek, ved navn CustomTkinter. Dette bibliotek er stadig early development men de funktioner som skulle bruges til det system, er fint implementeret og derfor er det valgt til det projekt

### Tanker om mulige ekstra features:

- Opsamling af merges/commits løbende for at opretholde en dynamisk changelog

- Baser changelog/release notes udra repo, periode, feature/branch (filtre til at søge)
- Keyword searches til at vide hvad der skal inputtes (ADDED, REMOVED, FIXED)
- Opstilling af dokumentet på en fin vis
- Sammenligning af nuværende changelog og ældre changelog eller kombination af disse

### Ønsket flow i systemet



Dette diagram er lavet for at give et overblik over den ønskede kommunikation i systemet.

### Opsamling af designprocess


Formålet med det system er at hjælpe udviklingsafdelinger med at spare tid på en ellers meget tidskrævende opgave. Med denne ønskede opsætning, kan der spares meget tid for den almene udvikler, ved næsten automatisk at indhente commit-historik for et givent repository og herunder også branch. Der vil være mulighed for at ændre i dokumentet og hertil at gemme det, så det er klar til en mulig release eller kan sendes videre til dem der er ansvarlige for at klargøre dem. Denne løsning vil sørge for at alle ændringer vil fremkomme i dokumentet.

## Implementering

### Opsætning af GitHub

For at holde styr på projektets codebase og samtidig at have en codebase at teste systemet på, skulle der oprette et GitHub repository. Det er en meget simpel opsætning, som kun kræver at man har lavet en bruger inde på deres platform. Når man har oprettet sig som bruger, er det næste at sørge for at installere git på sin computer. Når disse steps er ordnet, skal man ind på Github og trykke på "new repo". Man giver sit repo et navn og definere synligheden af det, herefter kan man gå ind i en mappe på sin lokale computer, hvor man har noget kode, som man vil have op i et repo. Via din foretrukne terminal, kan du køre et par git kommandoer, som vises til dig når du vælger at oprette et repo (GitHub, 2024)

Quick setup — if you've done this kind of thing before


 Set up in Desktop

 or 

HTTPS

SSH

git@github.com:Jonas-ML/sdfsd.git



Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

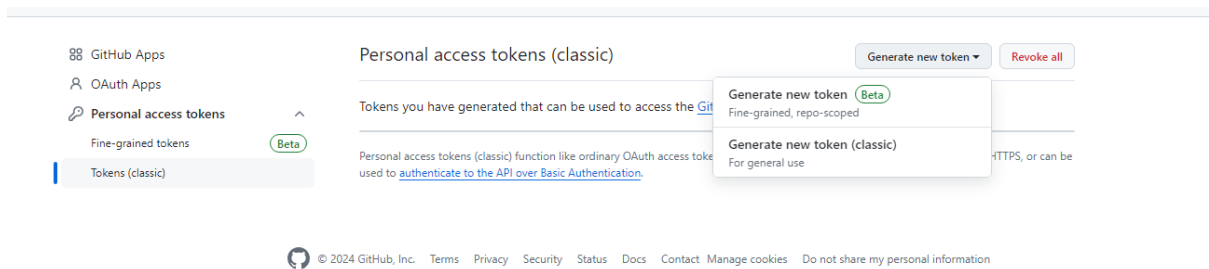
```
echo "# sdfsd" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:Jonas-ML/sdfsd.git
git push -u origin main
```



Der er mange forskellige måder at oprette et repository i Github. Man vælger bare den stil, der bedst passer en. Vil man gå 100 procent GUI? Eller er man ikke bange for at bruge lidt terminal. Man styrer det helt selv. Ydermere er der github integrationer i IDE's som VScode, som gør det endnu nemmere at både opsætte og maintaine.

### Generering af PAT

For at tilgå visse "developer features" på GitHub, skal man lave det der hedder PAT(Personal Access Token) som er en ekstra sikkerhedsforanstaltning. I hensyn til projektet, skal du bruge en for at bruge visse API endpoints men også for at komme udenom deres API kald kapacitet på 60 kald i timen. Med denne PAT, har man en kvote på 5000 kald i timen. Måden du opsætter denne på, er inde i "settings tab" og herefter i "developer settings" herinde kan du vælge 2 typer af PATS til oprettelse.



Når man har valgt hvilken en af de 2, skal man specificere navn, expiration date og en masse rettigheder, alt efter hvad man vil have den har adgang til at gøre med de valgte repositories.

## New fine-grained personal access token Beta

Create a fine-grained, repository-scoped token suitable for personal API use and for using Git over HTTPS.

### Token name \*

A unique name for this token. May be visible to resource owners or users with possession of the token.

### Expiration \*

 The token will expire on Wed, Jul 10 2024

### Description

What is this token for?

### Resource owner

## Repository access

### ☒ Public Repositories (read-only)

### ☐ All repositories

This applies to all current *and* future repositories you own.  
Also includes public repositories (read-only).

### ☐ Only select repositories

Select at least one repository. Max 50 repositories.  
Also includes public repositories (read-only).

## API

I det projekt at det GitHubs egen API der har ageret server, mens der er blevet udviklet en let API klient til client-side, som håndterer kommunikationen.

Der har ikke været behov for mere hjælp end hvad deres dokumentation har kunne give, da den er meget uddybende med eksempler på Requests og responses. Herunder ses et eksempel:

### List commits [↗](#)

**Signature verification object**

The response will include a `verification` object that describes the result of verifying the commit's signature. The following fields are included in the `verification` object:

Code samples for "List commits"

**Request example**

```
GET /repos/{owner}/{repo}/commits
```

**per\_page** integer

The number of results per page (max 100). For more information, see "[Using pagination in the REST API](#)."

Default: `30`

**page** integer

The page number of the results to fetch. For more information, see "[Using pagination in the REST API](#)."

Default: `1`

HTTP response status codes for "List commits"

Status code	Description
200	OK
400	Bad Request
404	Resource not found
409	Conflict
500	Internal Error

```
GET /repos/{owner}/{repo}/commits
```

cURL   JavaScript   GitHub CLI

```
// Octokit.js
// https://github.com/octokit/core.js#readme
const octokit = new Octokit({
  auth: 'YOUR-TOKEN'
})

await octokit.request('GET /repos/{owner}/{repo}/commits', {
  owner: 'OWNER',
  repo: 'REPO',
  headers: {
    'X-GitHub-Api-Version': '2022-11-28'
  }
})
```

**Response**

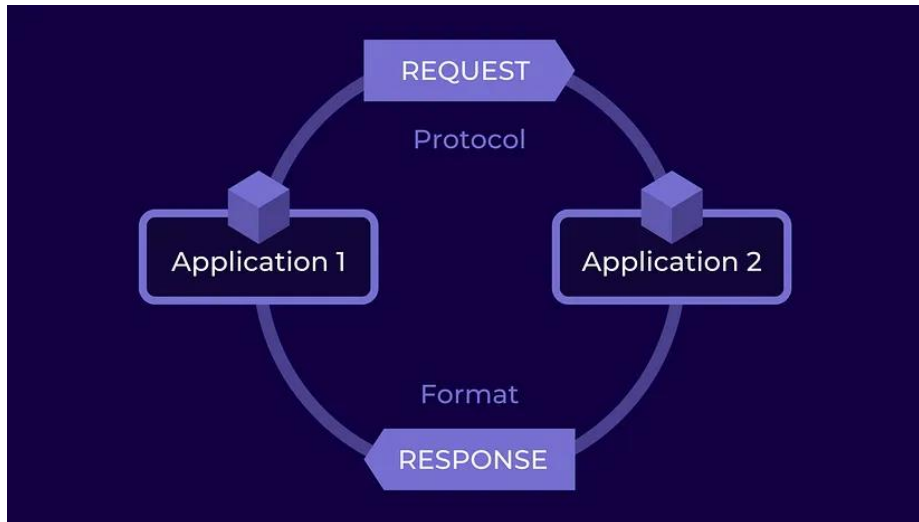
Example response   **Response schema**

Status: 200

```
{
  "type": "array",
  "items": {
    "title": "Commit",
    "description": "Commit",
    "type": "object",
```

## Hvad er en API?

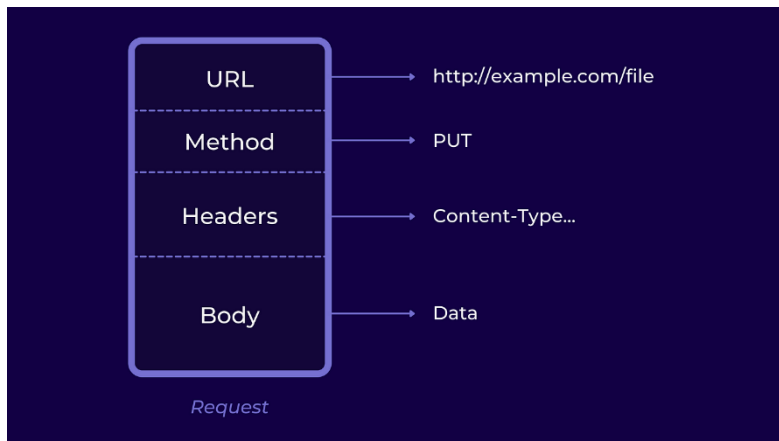
En API eller Application Programming Interface er ligesom en software "mellemand" der skaber mulighed for at 2 eller flere uafhængige applikationer kan udveksle data. Denne mellemand har så en række standarder og regler som skal følges for denne dataudveksling, nogle specifikke standarder som en API oftest bruger er HTTP og REST. (Frye, 2015)



## HTTP-forespørgsler

HTTP-forespørgsler udgør en enorm mængde af internettet i dag. De tillader kommunikation over internettet mellem servere og klienter. Hver gang du tilgår en ressource på en hjemmeside eller lignende, sker der er en HTTP-forespørgsel (også kaldet request) hvor dette data udveksles. Klienten sender en HTTP-request angående en bestemt ressource og serveren sender s en response tilbage med tilhørende data. Dette er også det bagvedliggende i de fleste API'er, ihvertfald dem som er rettet mod internettet. HTTP er en protokol for hvordan disse forespørgsler håndteres. De mest almindelige typer af forespørgsler er således: POST, GET, PUT, DELETE. Grunden til disse er de mest almindelige er at de danner fundamentet for CRUD (Create, Read, Update, Delete), Som er de grundlæggende typer af operationer man oftest ønsker at kunne bruge i applikationer. (Suciu, 2021)

## Struktur af en HTTP request



Dette er strukturen på en HTTP request. Den har en URL som er det endpoint den skal kommunikere med, hertil specificeres en metode/operation. Udover det, indeholder den hertil en header som oftest specificerer metadata info, som content-type, authorization og andre parametre. Det sidste lag er body og det er oftest her, hvor det data der er forespurgt, bliver opbevaret.

### HTTP metoder

Der er en god mængde af metoder til HTTP forespørgsler, dog fokuseres der i det afsnit, kun på dem som er fundamentale for REST. Som nævnt tidligere er det også dem der følger CRUD princippet

- GET – bliver brugt til at hente eller læse fra en ressource via internettet. Den response man modtager, vil indeholde den ressource som er koblet til det pågældende URL, dette vil oftest være i JSON eller HTML format men kan også være i de fleste formater man kunne få brug for, inklusiv eksekverbare filer.
- POST – bruges til at oprette nye ressourcer på serveren, den kan meget af det samme som en GET request, bare istedet for at der er fra server – klient, vil dette være klient – server
- PUT – bruges til at opdatere en allerede eksisterende ressource, istedet for at sende data, sender den instruktioner for opdatering af det pågældende data
- DELETE – bruges til at slette ressourcer fra serveren

### HTTP Headers

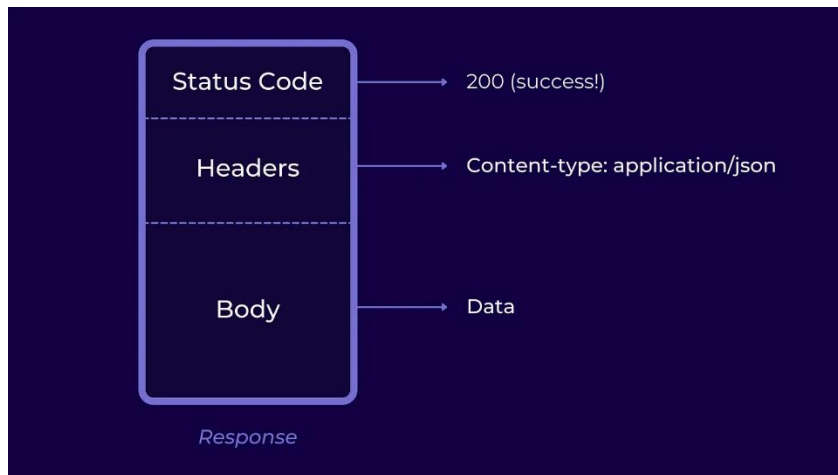
En header i en request eller response, fortæller meget om hvilket data der skal forventes, hvilke sikkerhedsforanstaltninger der skal opretholdes, afsender/modtager, timestamps, cookies og meget andet generel information. F.eks. den typisk header Content-Type, specificere hvilket format ressourcen er i og hjælper her serveren med at bearbejde den. Accept er modparten hertil, den specificerer hvilket format data som den vil acceptere. Der

er et hav af forskellige headers man kan definere og det kan kræve lidt research at finde frem til dem der er nyttige for en selv. (Suciu, 2021)

### Body

Det er her hvor det data man sender eller modtager, bliver opbevaret. Der er næsten ingen grænser for hvilket slags data du kan bruge heri

### Struktur af en HTTP response



Den tydelige forskel på request og response i disse eksempler er brugen af status codes i response. Disse HTTP status codes fortæller meget information på få numre.

- 100-199: Informational responses, de kan sige noget om hvorvidt en request er modtaget
- 200-299: Successful responses, de siger noget om at pågældende request er modtaget, bearbejdet og returneret med en HTTP response
- 300-399: Redirects
- 400-499: Client errors
- 500-599: Server errors

### REST

En REST API (RESTful) er en API der gør brug af en arkitektur stil ved navn "Representational state transfer". Denne arkitektur er yderst kendt for sin fleksibilitet, effektivitet og skalerbarhed. For at udvikle en REST API, er der 6 overordnede punkter, som skal følges for at overholde standarden. (IBM, 2024)



### Uniform interface

Alle API requests for den samme ressource, skal være ens. API'en skal sørge for at det samme stykke data, kun har en enkelt URI (Uniform Resource Identifier). Ydermere skal de forespurgte ressourcer i pågældende response, ikke være for store, dog skal det indeholde alt nyttigt data for klienten der sender denne request.

### Client-server decoupling

For at opretholde en REST API, er det yderst vigtigt at klient og server er totalt uafhængige af hinanden. Det eneste klienten skal vide om det den requester, er dennes URI. Den skal ikke kunne interagere med serveren på andre måder. På den anden side, må serveren ej heller modificere klient-applikationen mere end at respondere med forespurgt data.

### Statelessness

Dette er et punkt, man ofte hører når man drøfter emnet. Statelessness, betyder at hver forespørgsel, skal indeholde alt data nødvendigt for at serveren kan processere denne. Det betyder at der aldrig behøver at være sessioner fra klient til server. Det øger skalerbarheden og effektiviteten gevaldigt, når en server ikke behøver at gemme klient informationer og tilstande

### Cacheability

Når der er mulighed for det, skal en ressource kunne caches på enten klient eller server-side. En response fra server, kan indeholde informationer om tilladelser af caching. Caching forbedrer performance, da serverende istedet for at svare disse requests på ny ved at frembringe ressourcen, bare kan trække denne op fra sit cache. Der er mange forskellige måder at cache på. Det kan være for at spare server-side eller klient-side, det kan ydermere også være en kombination.

### Layered system architecture

I denne arkitektur, kan der være mange lag som en request og response skal igennem. Det er derfor vigtigt at både server og klient, ikke ved hvornår de kommunikerer med et reelt endpoint eller mellemmand

### Code on demand

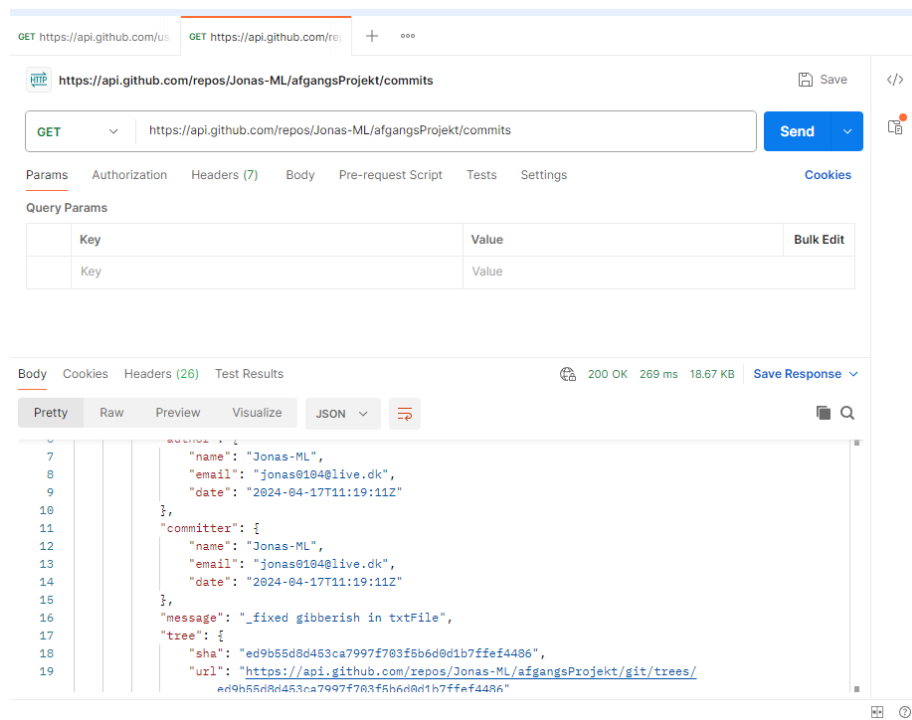
Dette er et valgfrit step i arkitekturen som indebærer funktionaliteten for at klienten, kan hente eksekverbar kode ned fra serveren af. Der kan selvfølgelig være nogle sikkerhedsrisici involveret i det og derfor er det en valgfri implementering

### Hvordan virker REST API'er?

Rest API'er kommunikerer gennem HTTP-forespørgsler, som set i tidligere afsnit, er det baseret ud fra nogle basale databasefunktioner som Create, Read, Update, Delete. Bedre kendt som CRUD (referer til HTTP afsnit)

## Postman

Postman er en applikation som agter at støtte i test, udvikling og dokumentation af API'er (Application Programming Interfaces). Fra Postman, kan du sende alle typer af HTTP-forespørgsler inklusiv diverse payloads, parametre og headers, til de API-endpoints som du specificerer, den understøtter et hav af forskellige dataformater, bl.a. Form-data, binær og JSON, som er det format der er brugt til forespørgsler i det projekt.



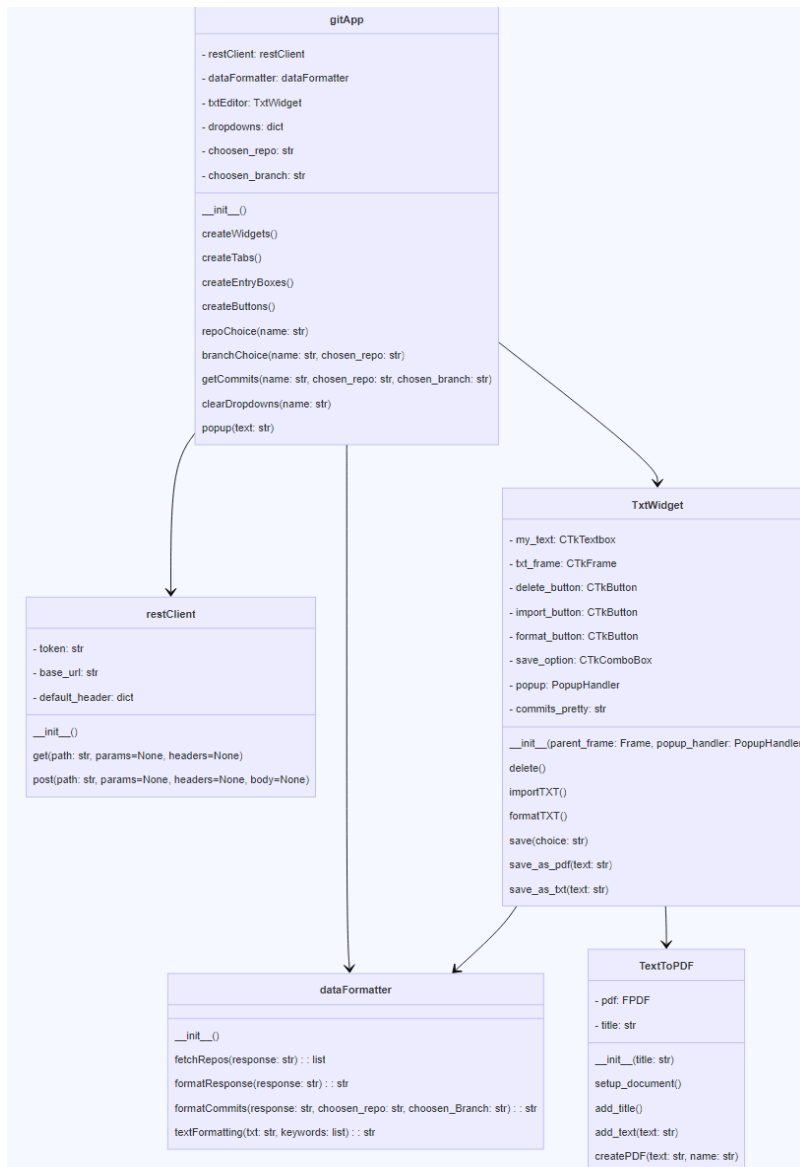
I det projekt, er Postman blevet brugt til at teste og verificere Github API endpoints for at finde den rette struktur mht. standard forespørgsler såvel som parametriserede-forespørgsler ved hjælp af Github's egen API dokumentation.

## JSON

JavaScript Object Notation er et populært dataudvekslingsformat. Det bliver oftest brugt til at transmittre struktureret data mellem klient og server i webapplikation og herunder særdeles API 'er. JSON er yders nemt at læse for både mennesker og maskiner, det er derfor nemt at komme i gang med. Det er ligeglad med hvilket programmeringssprog det skal indgå i og så understøtter det også de mest almindelige datastrukturer. (JSON ORG, 2024)

## System moduler

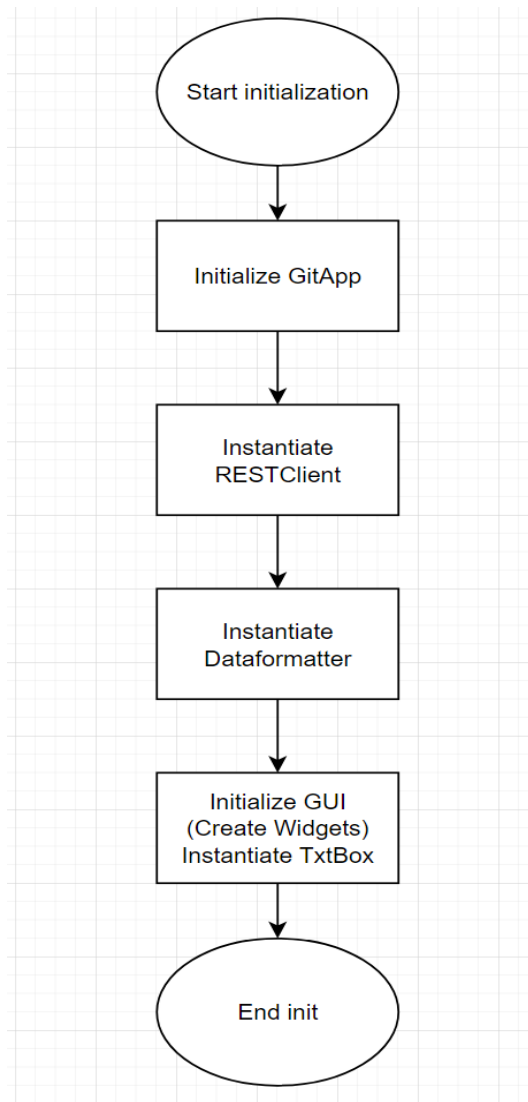
For at opretholde en god udviklingspraksis er systemet blevet modulariseret efter bedste evne. Derfor er systemet udviklet objekt orienteret, da det giver bedst mulighed for det, ydermere giver det mulighed for at abstrahere de mere komplekse elementer og en fornuftig genbrugelighed i disse moduler. Genbrugeligheden i det, gør at systemet er nemmere at udvikle videre på og hertil nemmere at skalere. Alle disse punkter gavner en højere robusthed i systemet.



Dette diagram giver et overblik over modulerne i systemet

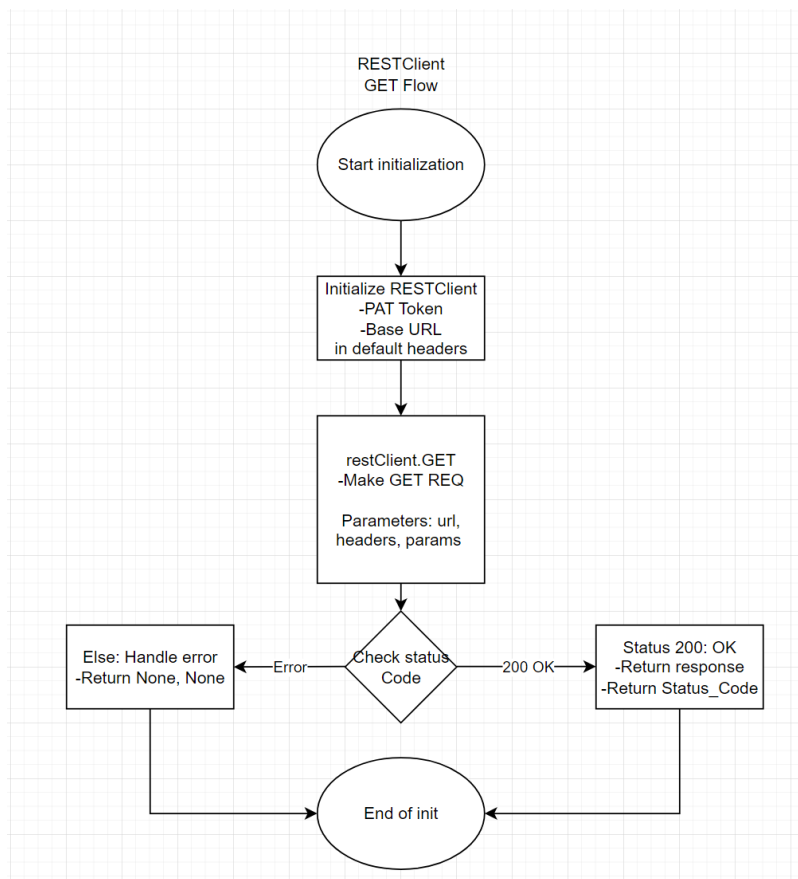
- GitApp initialisere restClient, txtWidget og dataformatter og er afhængige af disse for at fungere
- RestClient er API klientmodulet, der sørger for kommunikation mellem main applikationen og GitHubs API
- Dataformatter modulet er det der sørger for formatering af diverse JSON response fra restClient og formatering af commits og text til txtWidget
- TxtWidget er modulet der sørger for at text-editor virker med dennes funktionaliteter, dette modul er afhængig af dataformatter modulet og TextToPDF modulet for at fungere.
- TextToPDF modulet, sørger for generering af PDF-filer baseret på teksten fra text-editor

## Initialisering af applikationen



Initialiseringsprocessen starter når man åbner main programmet, her bliver der lavet instanser af API-Klienten, Dataformatter-modulet og så bliver GUI initialiseret. Som en del af GUI, bliver txtBox også instansieret herunder.

## API klient



Når man skal kommunikere med en ekstern API, skal man opsætte sine requests med headers og evt authorization i form af tokens. Der kan også være visse parametre der skal påsættes alt efter behov. Dette er mange variabler at tage hensyn til hver gang man skal forespørge forskellige ressourcer. Her er en API klient en meget fornuftig instans at have. Da man hertil kan konfigurere den til at håndtere alle disse detaljer og hermed abstrahere denne del af logikken væk fra den primære applikation.

Dette er API-klienten som håndterer det mest af kommunikationen til Githubs API. Der er kun funktioner til GET og POST da der ikke har været behov for andet. POST metoden er ej heller rigtigt blevet brugt endnu. Klienten instantiere sig selv med en authorization token i header, som er den PAT der blev lavet og er opbevaret i en txt fil på lokal maskine, den har også sit base url, så man er fri for at skrive en hel url når man laver en request.

Man kan se i eksemplet at der er et par parametre at tage hensyn til i funktionen. Den modtager en url til endpoint, parameter til definition af branch og eventuelle ekstra headers der skal på, udover de default der er specificeret i init. Funktionen returnerer så en statuskode og selve response objektet. I tilfælde af exception, returnere den none for at undgå et crash (shoonefeld, 2016)

## Dataformatter

Dette er modulet til at formatere data og ekstrahere nyttig information. Dette modul er blevet udbygget lige så stille hen ad vejen når der har været problemer med formatering af data.

INIT-metoden bliver passeret som det er lige pt. Det grundes at metoderne heri, kun opererer med data der er passeret som argumenter fra de andre moduler.

### *FormatReponse(self, response)*

Den første metode der blev defineret, var formatResponse. Denne blev lavet da den response man fik printet i terminalen, blev printet i en lang string og var derfor tæt på ulæselig. Denne funktion tager et response objekt og konverterer det til en JSON – formateret streng.

### *FetchRepos(self, response)*

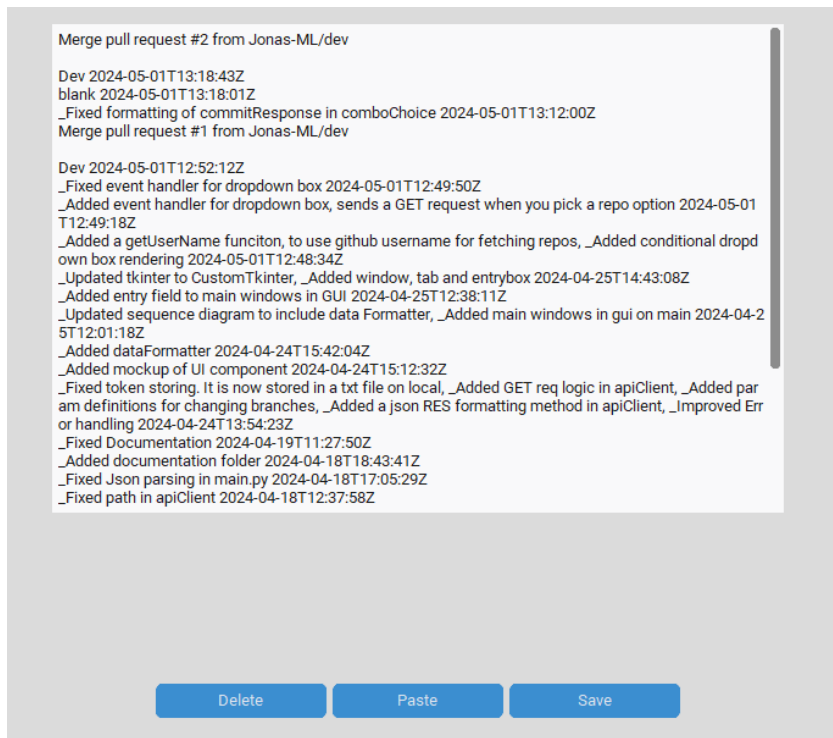
Denne metode blev lavet for at ekstrahere repository navne ud fra response objektet.

```
1  {
2      "id": 787883394,
3      "node_id": "R_kgD0LvYlgg",
4      "name": "afgangsProjekt",
5      "full_name": "Jonas-ML/afgangsProjekt",
6      "private": false,
7      "owner": {
8          "login": "Jonas-ML",
9          "id": 116717120,
10         "node_id": "U_kgD0BvT2QA",
11         "avatar_url": "https://avatars.githubusercontent.com/u/116717120?v=4",
12         "gravatar_id": "",
13         "url": "https://api.github.com/users/Jonas-ML",
14         "html_url": "https://github.com/Jonas-ML",
15         "followers_url": "https://api.github.com/users/Jonas-ML/followers",
16         "following_url": "https://api.github.com/users/Jonas-ML/following{/other_user}",
17         "gists_url": "https://api.github.com/users/Jonas-ML/gists{/gist_id}",
18         "starred_url": "https://api.github.com/users/Jonas-ML/starred{/owner}/{/repo}",
19         "subscriptions_url": "https://api.github.com/users/Jonas-ML/subscriptions",
20         "organizations_url": "https://api.github.com/users/Jonas-ML/orgs",
21         "repos_url": "https://api.github.com/users/Jonas-ML/repos",
22         "events_url": "https://api.github.com/users/Jonas-ML/events{/privacy}",
23         "received_events_url": "https://api.github.com/users/Jonas-ML/received_events",
24         "type": "User",
25         "site_admin": false
26     },
27     "html_url": "https://github.com/Jonas-ML/afgangsProjekt"
28 }
```

Som man kan se fra response i Postman, er repositories adskilt med keys som id, node\_id, name og lidt andre. I denne funktion, ledes der efter "name" key i response. Det er nemlig denne value der skal bruges for at finde branches og commits for det pågældende repository og det gør dermed at man bare kan bruge samme funktion til flere aspekter. Hvis du skal søge efter en branch, er det nemlig også "name" key som det er adskilt efter.

### *FormatCommits(self, response, chosen\_repo, chosen\_branch)*

Denne metode blev defineret i sammenhæng med den indbyggede text-editor i GUI. Da man fik commits printet i meget rodet form, således:

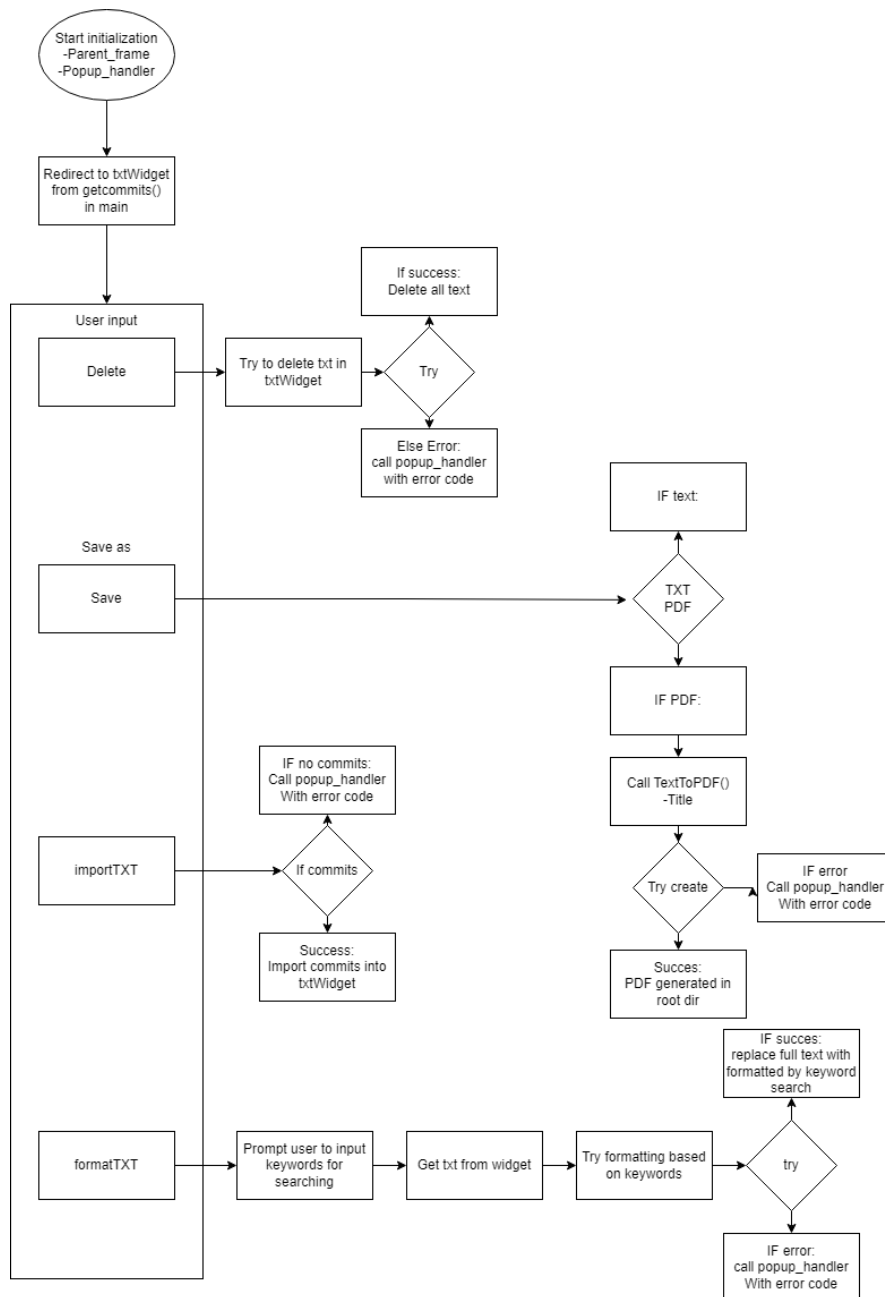


Denne metode giver mulighed for at læseligt format og inkluderer endda informationer om hvilket repo og branch som pågældende commit er taget fra:

```
=====
Repository: afgangProjekt on Branch: dev
Commit: changed name of formatmethods -> JSON_Formatter
Date: 2024-05-31T14:33:18Z
=====
Repository: afgangProjekt on Branch: dev
Commit: added Exception handling in dataformatter
Date: 2024-05-31T14:32:16Z
=====
Repository: afgangProjekt on Branch: dev
Commit: Added a popup window for user feedback and error handling
Date: 2024-05-31T14:26:48Z
=====
```

Denne metode bliver kaldt efter `FormatReponse()` og tager argumenter, som valgt repo, branch og de rå commits: `commits_pretty = dataFormatter().formatCommits(commits_raw, choosen_repo, choosen_Branch`

## Txtbox modul



Textbox-modulet begyndte som en intergreret del af main-applikationen. Dog blev det hurtigt meget omfangsrigt og blev derfor vurderet til at passe bedst i sit eget modul. Som ses af diagrammet gør det modul brug af TextToPDF, som agerer undermodul, da TextToPDF ikke skal bruges andre steder i applikationen. I Text-editor er der mulighed for at slette, gemme til txt og PDF, der er også mulighed for at formatere sine commit ud fra de keywords man specificere. Hertil hvis man har lavet en fejl eller vil have alle commit ind som de begyndte, er der en import knap som henter disse commit ind fra den variable de er opbevaret i.



## Test af systemet

Testning er en kritisk fase i udviklingsprocessen, da det sikrer, at systemet fungerer som forventet og lever op til de specificerede krav og ønsker. I det projekt blev der anvendt manuelle tests for at validere funktionaliteten af de forskellige komponenter i systemet. Her er en gennemgang af tests og resultaterne.

### Manuel test af header opsætning og parametre

For at kunne bruge systemet i den ønskede tilstand, skulle der være mulighed for at ændre headers som man havde lyst til, samt at den altid skulle inkorporere visse default headers til autorisering, der skulle også være mulighed for at ændre parametre til ændring af branches i et repository.

- Opsætning af Headers: Authorisationstoken blev tilføjet som en standardheader, mens yderligere headers kunne tilføjes efter behov.
- Ændring af Parametre: Parametre blev brugt til at specificere branches i et repository, hvilket forbedrer systemets brugervenlighed.
- Verifikation: Headers og parametre blev printet ud for at bekræfte deres korrekte opsætning.

### RESTClient

I API klienten, blev disse opsat som vist herunder og sat til at blive printet ud for at bekræfte strukturen:

```
class restClient: # init of object with baseURL and token
    def __init__(self):
        self.token = token
        self.base_url = 'https://api.github.com'
        self.default_header = {"Authorization" : token}

    def get(self, path, params=None, headers=None):
        url = self.base_url + "/" + path
        final_headers = self.default_header.copy() # Makes a copy of the dict to get a
        if headers:
            final_headers.update(headers)

        print(params)
        print(final_headers)

        try:
            response = requests.get(url, headers=final_headers, params=params)
            print(url)
            status_code = response.status_code
```

Målet var at denne AUTH token altid skulle pådattes headers, dog skulle der være mulighed for i MAIN at pådutte flere headers, uden at overwrite. Ydermere skulle vi gerne via "params" ændre i branches, for at forbedre usability

### *I main:*

```
a = restClient()
response, status_code = a.get("repos/Jonas-ML/afgangsProjekt/commits", params={"sha":"dev"}, headers={"test": "test"})

if status_code == 200:
    res = a.formatResponse(response)
else:
    print("ERROR CODE:", status_code)
```

### Output:

```
{'sha': 'dev'}
{'Authorization': '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r', 'test': 'test'}
https://api.github.com/repos/Jonas-ML/afgangsProjekt/commits
```

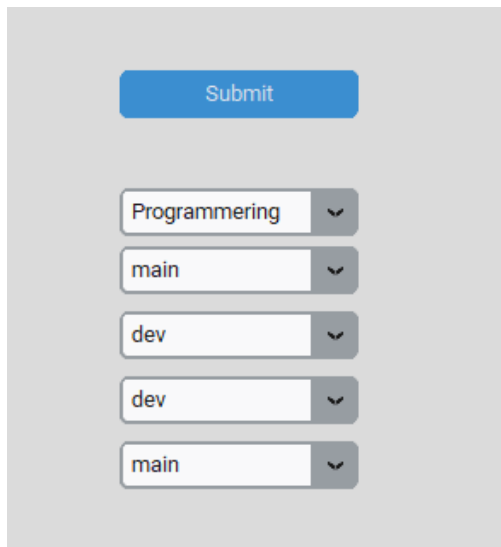
- Auth token
- Params
- URL

### *Resultat:*

Testene viste, at autorisationstoken blev korrekt anvendt, og yderligere headers blev tilføjet uden at overskrive default headers. Parametre for ændring af branches fungerede som forventet, hvilket bekræftede systemets fleksibilitet

### Manuel test af multiple rendering af dropdowns:

Under udviklingen blev der observeret et problem med multiple rendering af dropdowns, når man trykkede flere gange på submit-knappen eller valgte forskellige repositories.



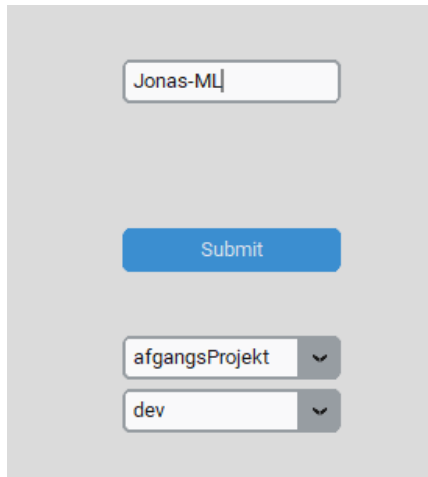
The screenshot shows a web form on a light gray background. At the top is a blue rectangular button with the text "Submit" in white. Below the button are five vertically stacked dropdown menus. Each menu has a white input field and a dark gray arrow pointing downwards on the right side. The first dropdown menu is labeled "Programmering". The second, third, and fourth dropdown menus are labeled "main", "dev", and "dev" respectively. The fifth dropdown menu is labeled "main".

### *Fix af multiple rendering*

Problemet blev fixet ved at man definerede en dropdown dictionary, til at holde på de initialiserede dropdowns via name variabelen, så de kun er lavet en gang pr username

### Resultat:

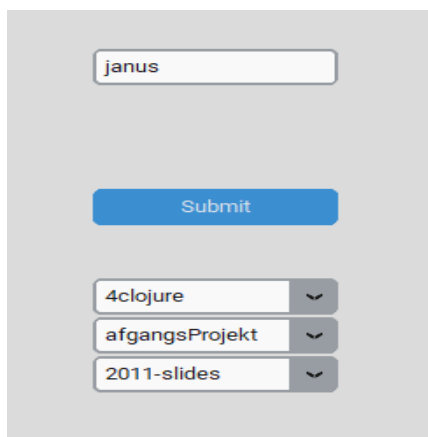
Efter implementeringen af løsningen blev problemet med multiple rendering løst. Systemet genererede nu korrekt kun én dropdown per brugernavn, hvilket forbedrede brugeroplevelsen og systemets stabilitet.



A screenshot of a web form on a light gray background. At the top is a text input field containing 'Jonas-ML'. Below it is a blue 'Submit' button. Under the button are two dropdown menus. The first dropdown menu has 'afgangsProjekt' selected, and the second dropdown menu has 'dev' selected. Both dropdown menus have a small downward arrow icon on the right side.

### Bug or “by-design?”

I den nuværende version af systemet, er det muligt at gå tilbage til login siden og skrive forskellige usernames og hente forskellige commit fra de pågældendes branches og repositories. Dette skyldes at den dictionary der holder styr på de initialiserede dropdowns, gør det på username basis. Så når man skriver et andet username, kommer man udenom den blokade. Dog kan man argumentere for at det kan være en nyttig funktion i systemet, da der kan være behov for aggregering af commit fra flere parter.



A screenshot of a web form on a light gray background. At the top is a text input field containing 'janus'. Below it is a blue 'Submit' button. Under the button are three dropdown menus. The first dropdown menu has '4closure' selected, the second dropdown menu has 'afgangsProjekt' selected, and the third dropdown menu has '2011-slides' selected. All three dropdown menus have a small downward arrow icon on the right side.

### Opsummering af test

Som der ses ud fra nogle af de fundne bugs i systemet, er test en vital proces i udviklingen af et system. Man vil gerne levere et system til en slutbruger som er så fejlfrit som muligt og det er umuligt uden at teste funktionerne i systemet. I erhvervsmæssige praksis, er test en enorm opgave som kræver en bred stab med forskellige kompetencer og tanker. Man kan finde bugs på de mest tilfældige og uforudsete måder og derfor er man nødt til at tænke "ud af boksen" for at teste systemet så grundigt som muligt. I det projekt var ønsket at test afsnittet skulle fylde en større del. Dog er hver "mindre" test dog ikke blevet bogført, men der har været mange imellem og langt de fleste er blevet fixet.

### Delkonklusion, fra design til løsning

Ude fra en sammenfattende vurdering af tanker og ønsker fra designprocessen, sammenholdt mod den realiserede løsning. Må man erkende at systemet er endt i en fornuftig og tilstand, dog mere minimal end først forventet. Der er opnået det ønskede flow i programmet og en god kommunikation mellem disse moduler. Det grafiske user-interface er blevet meget lig det mock-up der blev tegnet i designprocessen. Ift. De opstillede funktionaliteter i designprocessen, er 3/5 blev realiseret. Man kan formatere efter keyword searching, man kan basere sin commit fetching på branches, får vist sine commit med informationer som repo/branch, dato osv og PDF-dokumentet hertil bliver opstillet på en fin læselig vis. Der vil dog stadig være plads til mange forbedringer fremadrettet. Udfra de opstillede funktionelle krav, mangler der mulighed for oversætning af disse notes via et translationsmodul. Dette modul blev nedprioriteret til fordel for de andre moduler da det hurtigt ville blive for omfangsrig en udvikling da de fleste "nemme" opsætning er baseret udfra lige præcis google-translate og andre tredjeparter, som helst skulle undgås.

## Diskussion

### Tidsforbrug og effektivitet

En af de mest markante forbedringer ved dette system er besparelse af tidsforbrug og omkostninger forbundet med generering af release- og change-notes. Dette system hjælper udvikleren ved at sammenfatte commit data og give mulighed for formatering af dette hvilket frigiver betydelig tid for udviklere, denne tid kan derfor bedre bruges på andre opgaver.

### Frigørelse af ressourcer

Ved at reducere tiden der skal bruges på denne type af dokumentations arbejde, frigøres der ressourcer til andre opgaver. Dette system kan bidrage til hurtigere udviklingscyklusser samt bedre produktkvalitet og kunderelationer.

### Svagheder

For lidt autonomi i systemet: I den nuværende iteration af systemet, kræver det stadig for meget manuel håndtering ift. den originale ide. Det skyldes bl.a. det større omfang som systemet fik, samt skiftende prioriteringer og nye ideer. Det kan være svært at overholde alle punkter af god udviklingspraksis samt at komme i mål med alle ideer.

Kun mulighed for public repos: Eftersom man kun kan søge i public repositories i GitHub i den nuværende iteration, kan dette system i nogle use-cases ikke bruges.

## Konklusion

Dette projekt har med succes udviklet et system, der reducerer tidsforbruget for en udvikler i at lave release- eller change-notes. Systemet anvender GitHub som platform og Python med visse relevante biblioteker som Requests og FPDF for at håndtere API kommunikation og formatering af tekst til PDF.

Ved at implementere en API-klient til kommunikation med GitHub's API, et dataformatteringsmodul til at strukturere JSON og et GUI bygget med CustomTkinter, er det lykkedes at skabe en meget brugervenlig løsning. Det grafiske user-interface, gør det nemt for udviklere at vælge repositories og branches for en given GitHub user og herfra at opsamle alle commits, som kan bruges til generering af læselige PDF-dokumenter.

Systemet blev testet igennem vha. manuel testing. Disse sikrede den korrekte opsætning af headers og parametre til API kommunikation, samt afhjalp det med at fixe bugs i relation til dropdown menuer og submit-knappen. Testafsnittet kunne godt have været mere omfattende, dog blev de fleste fejl rettet under udviklingen.

Ønsket til projektet var at der blev lavet et docker image af applikationen for at vise metoden i deployment-fasen af SDLC, dog var der mange problemer med denne løsning, ift at få opsat dependencies til at systemet GUI kunne køre fra en Docker container.

Udfra de opstillede krav, ønskerne fra design-fasen og det nuværende stadie af produktet, synes det at vise at projektet har opfyldt sine mål og leveret en løsning der effektivt reducerer tidsforbrug for udviklere eller andet relevant personale. Der er dog stadig rigt potentiale for yderligere forbedringer og udvidelser af systemet i form af automatisering for virkeligt at maksimere produktiviteten af dette værktøj

## Perspektivering

### Avancerede filtrerings muligheder

Tidsbaseret søgning: Muligheden for at filtrere commits efter dato eller periode, kan være nyttigt for at generere mere specifikke release eller change notes. Denne implementering, ville også kunne understøtte en sammenligning af 2 forskellige version, for at få en endnu mere strømlinet proces i generering af change-notes.

Filtrering efter bruger/team: Filtrering efter team eller bruger, ville også kunne hjælpe med at generere mere specifikke notes

### Dokument opsætning

Virksomhedsspecifik dokumentopsætning: I et eventuelt slutprodukt, ville det være ønsket at der var mulighed for at opsætte PDF dokumentet, efter virksomhedens interne behov. Det kunne være brug for logo eller andre elementer der er specifikke til virksomhedens dokument standarder

Skabeloner: Når man laver change/release-notes, følger de oftest en standard som er en opdeling af "ADDED", "FIXED", "REMOVED" osv. Ønsket i dette system, var at man kunne vælge at filtrere efter disse og få en automatisk genereret PDF med de ønskede afsnit og de relevante commits herunder. Ved hjælp af periode filtrering, ville man også kunne opsætte og generere versions specifikke notes meget nemmere.

### Forbedret GUI

Interaktivt/dynamisk dashboard: Udvikling af et mere interaktivt dashboard, hvor brugere nemmere kunne navigere i repo's, branches og andre elementer. Det ville også være en fordel med en form for data visualisering, som antal af commits, contributors, mfl. Dette ville også give systemet et større overblik. Man kunne også lave noget visualisering over hvor kode ændringer er forekommet.

### Automatisering

Automatiseret note generation: Integration af løsningen i et mere autonomt stadie i en CI/CD-pipeline, ville give virksomheden rige muligheder for automatisk at generere disse notes ved hver deployment. Dette ville reducere den manuelle arbejdskraft gevaldigt og samtidig, sørge for at noterne altid er helt up-to-date.

### Diverse

Mulighed for at tilgå private repositories: I et slutprodukt, ville det ønsket funktion være at kunne tilgå private repositories, da dette også ofte er en brug funktion når der opbevares kode i GitHub repositories

### Opsamling af perspektivering

Disse udvidelser kan øge værdien af systemet betydeligt, ved at gøre det mere fleksibelt, brugervenligt og autonomt. Dette ville gøre systemet et værdifuldt værktøj i de fleste

udviklingsafdelinger. Ved fortsættelse af udvikling og forbedringer af systemet, kunne man sikre at det forbliver relevant og nyttigt.

## Referencer

Frye, M.-K. (2015). Hentet fra mulesoft:

<https://www.mulesoft.com/resources/api/what-is-an-api>

GitHub. (2024). Hentet fra GitHub: <https://docs.github.com/en/get-started>

IBM. (9. april 2024). *IBM*. Hentet fra <https://www.ibm.com/topics/rest-apis>

JSON ORG. (2024). Hentet fra JSON: <https://www.json.org/json-en.html>

shoonefeld, J. v. (2016). *realpython*. Hentet fra <https://realpython.com/api-integration-in-python/>

Singht, R. (2023). *Tutorialspoint*. Hentet fra <https://www.tutorialspoint.com/8-tips-for-object-oriented-programming-in-python>

Suciu, D. (2021). *Medium*. Hentet fra <https://medium.com/api-world/api-architecture-the-http-protocol-and-its-importance-aeba0fe46f91>