Univerza v Ljubljani

Fakulteta za elektrotehniko

David Hožič

OGRODJE ZA OGLAŠEVANJE NEZAMENLJIVIH ŽETONOV PO SOCIALNEM OMREŽJU DISCORD

Diplomsko delo
Univerzitetni študijski program prve stopnje Elektrotehnika

Mentor: doc. dr. Matevž Pustišek

Ljubljana, 2023

Univerza v Ljubljani
Fakulteta za elektrotehniko

Tržaška cesta 25, p.p. 2999
1000 LJUBLJANA, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 30
www.fe.uni-lj.si
e-mail: dekanat@fe.uni-lj.si

Spodaj podpisani študent, David Hožič, vpisna številka 64200069, avtor pisnega zaključnega dela študija z naslovom: Ogrodje za oglaševanje nezamenljivih žetonov po socialnem omrežju Discord,

IZJAVLJAM,

1. ◉ da je pisno zaključno delo študija rezultat mojega samostojnega dela;
   ○ da je pisno zaključno delo študija rezultat lastnega dela več kandidatov in izpolnjuje pogoje, ki jih Statut UL določa za skupna zaključna dela študija ter je v zahtevanem deležu rezultat mojega samostojnega dela;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil soglasje etične komisije;
5. da soglašam z uporabo elektronske oblike pisnega zaključnega dela študija za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. da dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.
8. da dovoljujem uporabo mojega rojstnega datuma v zapisu COBISS.

V: Ljubljani
Datum: 10. 8. 2023

Podpis študenta:

UNIVERZA V LJUBLJANI
FAKULTETA ZA ELEKTROTEHNIKO

David Hožič

OGRODJE ZA OGLAŠEVANJE NEZAMENLJIVIH ŽETONOV PO SOCIALNEM OMREŽJU DISCORD

Diplomsko delo

Univerzitetni študijski program prve stopnje Elektrotehnika

Mentor: doc. dr. Matevž Pustišek

Ljubljana, 2023

Zahvala

Rad bi se zahvalil mentorju za nudenje podpore in pomoč pri izdelavi diplomskega dela. Prav tako bi se rad zahvalil vsem ostalim profesorjem, ki so nas poučevali in nudili strokovno podporo v zadnjih treh letih. Nazadnje bi se rad zahvalil staršema, ki sta me ves čas podpirala.

Povzetek

Nezamenljivi žetoni (angl. *Non-fungible tokens*) so edinstvena digitalna sredstva, ki obstajajo na verigi blokov brez možnosti replikacije. Obstaja več pristopov za njihovo oglaševanje, pri čemer je eden izmed njih oglaševanje preko socialnega omrežja Discord z agresivnim pristopom. Diplomsko delo se osredotoča na proces oglaševanja in se nanaša na projekt Ogrodje za oglaševanje preko Discord omrežja (angl. *Discord Advertisement Framework*), ki je implementirano v programskem jeziku Python.

Najprej so v delu opisani nezamenljivi žetoni in pristopi k njihovemu oglaševanju. Nato je predstavljeno socialno omrežje Discord in pristop k oglaševanju na njem. Sledi predstavitev samega projekta diplomskega dela, katerega cilj je izdelava ogrodja za oglaševanje preko Discord omrežja, ki lahko deluje samodejno, brez posredovanja uporabnika, se ustrezno odziva na napake, nudi beleženje sporočil in je nastavljivo za različne načine delovanja.

V poglavju, ki se nanaša na projekt diplomskega dela, so predstavljeni zasnova in razvoj projekta, njegova dokumentacija ter avtomatizirano testiranje. Ogrodje se na najvišji ravni deli na jedro in grafični vmesnik, pri čemer lahko jedro deluje neprekinjeno na strežniku in je sposobno procesirati ukaze iz grafičnega vmesnika na daljavo. Oglaševalske podatke in parametre se v jedru nastavi preko Python datoteke, kar zahteva minimalno znanje Python jezika. Jedro je razdeljeno na več sektorjev za lažji razvoj in nadgradnjo. Grafični vmesnik je prav tako implementiran v jeziku Python. Opisan je razvoj grafičnega vmesnika, predstavljena je njegova struktura, na koncu pa je opisan tudi oddaljen dostop do jedra ogrodja. Objekte (račune, sporočila ipd.) je mogoče v grafičnem vmesniku definirati preko novega okna, ki se samodejno generira na podlagi podatkovnih tipov, prebranih iz izvorne kode funkcij in razredov v jedru ogrodja. Definirane objekte je mogoče shraniti v JSON datoteko, prav tako je omogočeno generiranje Python oglaševalske skripte, ki deluje v jedru ogrodja.

Po opisu razvoja in zasnove jedra ter grafičnega vmesnika ogrodja je opisan proces dokumentiranja. Dokumentacija je izdelana s sistemom Sphinx in se avtomatično gradi ter objavlja ob vsaki izdaji projekta preko platforme GitHub. Opis vseh javnih razredov in funkcij (programskega vmesnika) se samodejno generira iz same kode projekta.

Na koncu poglavja o projektu diplomskega dela je opisano še avtomatizirano testiranje kode, ki je implementirano z ogrodjem za avtomatizirano testiranje pytest. Ogrodje se na platformi GitHub avtomatično testira ob vsakem zahtevku za združitev veje, pri čemer se združitev zavrne, če katerikoli od testov ni uspešen. Z avtomatiziranim testiranjem se zmanjšajo možnosti za izdajo nove verzije ogrodja z napakami v delovanju.

Sklepam, da je ogrodje izjemno uporabno, ne le za oglaševanje NFT-jev, temveč tudi za oglaševanje katerekoli druge vsebine. Ker v času pisanja ni na voljo skoraj nobenega brezplačnega ogrodja za oglaševanje, ki bi bilo sposobno vsega, česar je sposobno to ogrodje, je smiselno sklepati, da je projekt izjemno uporabne narave.

Ključne besede: Python, grafični vmesnik, oddaljen dostop, shranjevanje v datoteko, dokumentacija, avtomatično testiranje, beleženje sporočil, zasnova in razvoj.

Abstract

First, the thesis describes non-fungible tokens and approaches to their advertisement. Next, it presents the social network Discord and the advertisement approach on this social network. It also explains the types of user accounts and channel types where advertisement can take place. The presentation then moves on to the project itself, which aims to create a framework for advertising on Discord, capable of operating automatically without user intervention, responding appropriately to errors, logging messages, and being configurable to function in multiple ways.

In the chapter related to the thesis's project, the design and development of the project, its documentation and automatic testing are all presented.

At the highest level, the framework is divided into a core and a graphical interface, where the core can run continuously on a server and is capable of remotely processing commands from the graphical interface. Advertisement data and parameters are set in the core via a Python script or program, requiring minimal knowledge of the Python language. The core is divided into several sectors to facilitate development and upgrades. The graphical interface is also implemented in Python. The development of the graphical interface is described, its structure is explained, and remote access to the core of the framework is discussed. Objects (accounts, messages, etc.) are defined in the graphical interface through a new window, automatically generated based on data types extracted from the source code of functions and classes in the core of the framework. The defined objects can be saved into a JSON file, and Python advertisement scripts that runs in the core of the framework can also be generated.

After describing the development and design of framework's core and graphical interface, the documentation process is explained. The documentation is created using the Sphinx system and is automatically built and published with each project release

through the GitHub platform. The description of all public classes and functions (the program's interface) is automatically generated from the project's source code.

Finally, the chapter on the thesis's project describes the process of automated testing (unit testing), which is implemented using the pytest testing framework. Upon a pull request on the GitHub platform, the framework is automatically tested, and the branch merge is rejected if any of the tests fail. Automated testing reduces the chances of a new version release being published with bugs being present.

In conclusion, the framework proves to be extremely useful not only for advertising NFTs but also for advertising any other content. Considering that, at the time of writing, there are almost no free advertising frameworks capable of what this framework can do, it is reasonable to conclude that the project is of significant practical value.

Keywords: Python, graphical interface, remote access, saving to file, documentation, automatic testing, message logging, design and development.

Kazalo

Slike

Bloki kode

Uporabljeni akronimi

Akronim	Pomen
API	Application Programming Interface
CSV	Comma-Separated Values
GPT	Generative Pre-trained Transformer
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
NFT	Non-Fungible Token
ORM	Object Relational Mapping
PIP	Preferred Installer Program
RAM	Random Access Memory
SQL	Structured Query Language

Uporabljeni pojmi

SLO	ENG
Asinhrona komunikacija	Async I/O
Avtomatsko odkrivanje	Automatic discovery
(GUI) Pripomočki	(GUI) Widgets
Sočasna večopravilnost	Concurrent multitasking
Izvorna koda	Source code
Nadzor verzij	Version control
Nezamenljivi žetoni	Non-fungible tokens
Omejevanje zahtev	Rate limiting
Ovoj API	API wrapper
Pridružna povezava	Invite link
Prožilec	Trigger
Pritrditev	Fixture
Robot	Bot
Veriga blokov	Blockchain
Vgrajeno sporočilo	Embedded message
Zahtevek za združitev vej	Pull request

1 Nezamenljivi žetoni

Nezamenljivi žetoni (angl. *Non fungible tokens*) so edinstvena digitalna sredstva, ki živijo na verigi blokov brez možnosti replikacije. Najpogosteje predstavljajo digitalna umetniška dela *[1]*.

Slika 1.1: Primer nezameljivega žetona (ustvaril DALL-E)

Nezamenljivost pomeni, da posameznega žetona ni mogoče preprosto zamenjati za drugega, kot se lahko menja sod olja za drug sod olja ali en evro za drug evro.

Najbližji primer nezamenljivega žetona v fizični obliki je bejzbolska kartica. Na primer, lahko imate zelo redko bejzbolsko kartico nekega igralca bejzbola in je ne morete enakovredno zamenjati za kakšno drugo, običajno bejzbolsko kartico.

2 Marketinški pristopi za promoviranje nezamenljivih žetonov

Obstaja več načinov oglaševanja, ki se jih lahko uporabi tudi za oglaševanje nezamenljivih žetonov. Ti so na primer uporaba Google Ads platforme (oglaševanje po Google iskalniku, YouTube aplikaciji), uporaba forumov in drugi načini. Eden izmed teh drugih načinov je morda še najbolj uveljavljen na tem področju, in sicer ga imenujemo agresivno oglaševanje (angl. *shilling*) *[2]*.

Pri agresivnem oglaševanju ni postavljanja promocijskega gradiva na spletno stran ali stojnice v javnost, temveč gre za nenehno objavljanje informacij o izdelku na socialna omrežja, kot so Instagram, Telegram, Discord ipd., omenjanje izdelka in deljenje povezav do njega povsod, kjer je to mogoče.

Te dni prodajalci nezamenljivih žetonov pogosto ne oglašujejo samostojno, temveč najamejo ljudi, ki to delo opravljajo namesto njih. Obstaja veliko ponudb za taka dela, ki so lahko zelo profitabilna, še posebej če imate orodje, ki lahko oglašuje namesto človeka, avtomatično, brez da bi bil ta prisoten.

To diplomsko delo se ne osredotoča na same nezamenljive žetone, temveč na proces oglaševanja, in sicer na socialnem omrežju Discord. Delo se navezuje na projekt Ogrodje za oglaševanje po Discordu (angl. *Discord Advertisement Framework*), ki je ogrodje za oglaševanje, napisano v programskem jeziku Python.

3 Discord

3.1 Kaj je Discord

Discord je bil ustvarjen leta 2015 s strani Discord Inc. (prej znan kot Hammer & Chisel), studia za razvoj iger, ki sta ga ustanovila Jason Citron in Stanislav Vishnevskiy. Platforma je bila zasnovana kot orodje za komunikacijo med igralci video iger. *[3]*

Zamisel za Discord je izvirala iz Citronove osebne izkušnje v vlogi igralca računalniških igric. Opazil je, da so mnoga obstoječa orodja za komunikacijo (Skype, TeamSpeak) za igralce zastarela in težko uporabna, zato je želel ustvariti uporabniku bolj prijazno platformo, ki bi igralcem omogočila enostavno komuniciranje med igranjem iger.

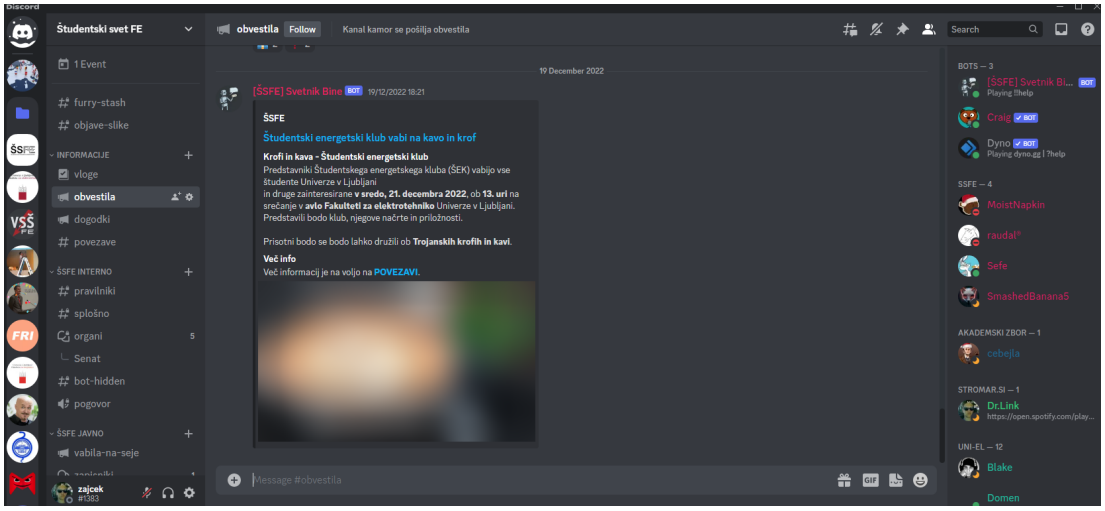
Discord se je od takrat razvil v več kot samo orodje za komunikacijo med igralci video iger in postal priljubljena platforma za skupnosti vseh vrst. Je priljubljena platforma, ki uporabnikom omogoča komunikacijo preko glasovnega, video in besedilnega klepeta. Pogosto se uporablja za različne namene, vključno z razpravljanjem o umetniških projektih, načrtovanjem družinskih izletov, iskanjem pomoči pri domačih nalogah. Prav tako ima dobro funkcijo iskanja nekoč objavljene vsebine, ki je uporabna na primer za iskanje primera dispozicije diplomske naloge, ki jo je nekdo objavil pred tremi meseci.

Čeprav lahko Discord skupnostim vseh velikosti služi kot dom, je še posebej priljubljen med manjšimi aktivnimi skupinami, ki med seboj pogosto komunicirajo. Večina skupnosti (strežnikov/cehov) je zasebnih in zahtevajo povabilo za vstop, kar prijateljem in skupnostim omogoča, da ostanejo povezani. Vendar pa obstajajo tudi večje, bolj javne skupnosti, osredotočene na določene teme, kot so priljubljene videoigre, ali pa, kot je to v primeru tega diplomskega dela, stvari, kot sta veriga blokov (angl. *blockchain*) in NFT. Uporablja se lahko tudi kot skupnost fakultete, kjer študenti lahko govorijo preko

glasovnih kanalov, delijo študijske materiale in postavljajo vprašanja o gradivu, ki ga ne razumejo.

Nekaj primerov Discord skupnosti, povezanih z Univerzo v Ljubljani:

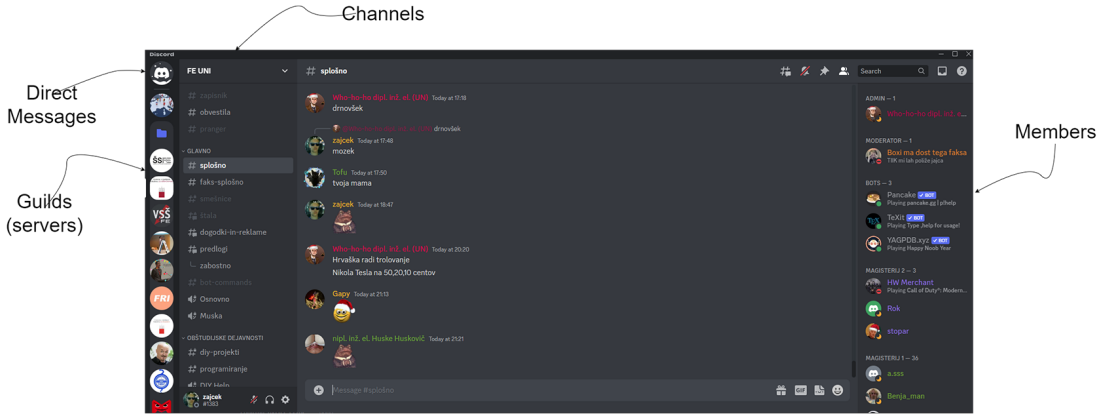
- Študentski svet FE (Slika 3.1),
- FE UNI,
- FE VSŠ,
- FRI UNI
- in druge.



Slika 3.1: ŠSFE Discord skupnost (strežnik oz. ceh)

3.2 Struktura omrežja Discord

Discord aplikacija je v osnovi sestavljena iz gumba za direktna (osebna) sporočila, seznama cehov/strežnikov, seznama kanalov in seznama uporabnikov (uporabniških računov), ki so pridruženi v ceh. *[4]*.



Slika 3.2: Struktura Discord aplikacije

3.2.1 Uporabniški računi

Obstajata dve vrsti računov, ki sta lahko v cehu:

1. Uporabniški računi
2. Robotski (avtomatizirani) računi

Discordovi pogoji uporabe prepovedujejo avtomatiziranje uporabniških računov[1].

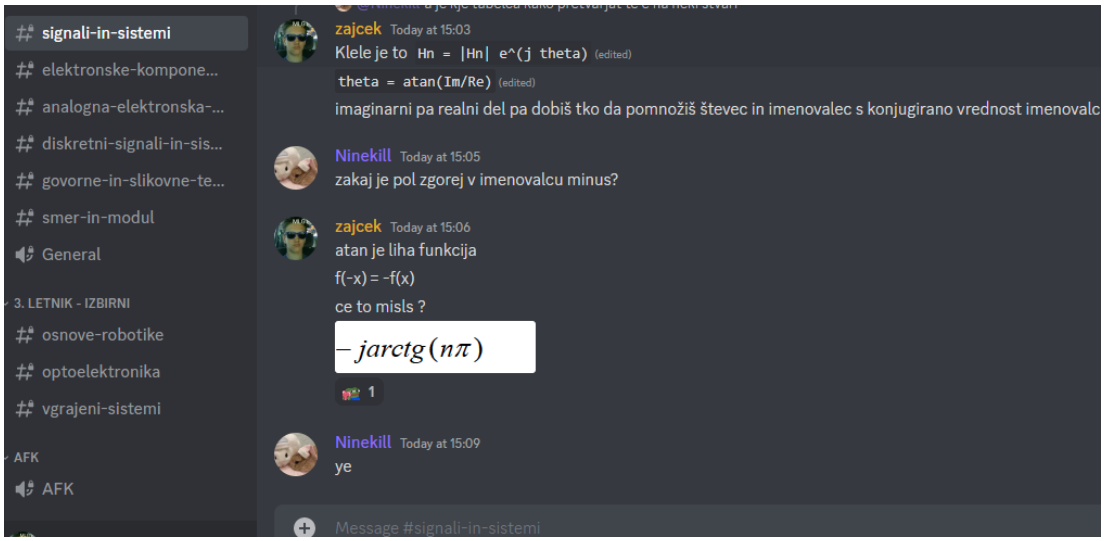
[1] „Automated user accounts (self-bots)“: https://support.discord.com/hc/en-us/articles/115002192352-Automated-user-accounts-self-bots-.

3.2.2 Kanali

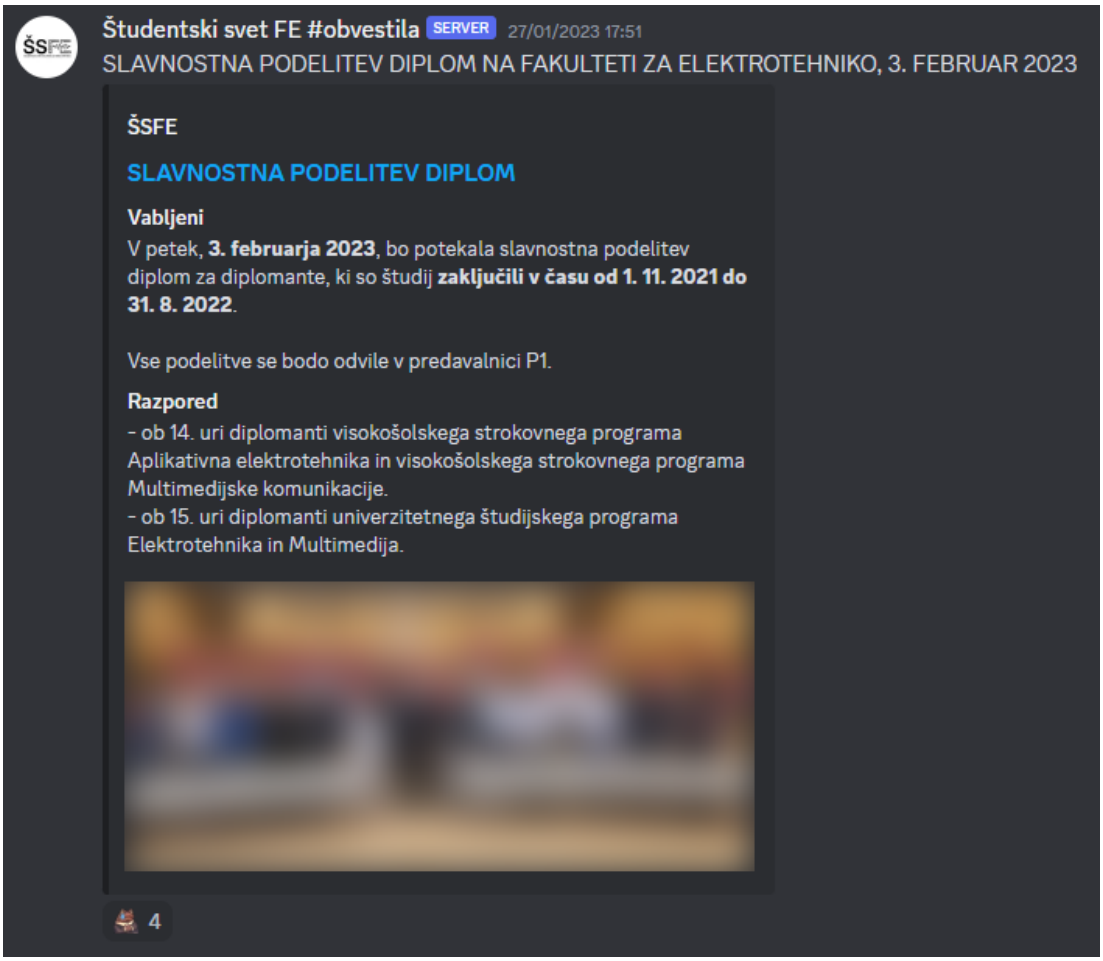
Discord ima tri vrste kanalov:

1. Tekstovni kanali - kanali za pisanje besedila v cehu
2. Glasovni kanali - kanali za govor in predvajanje glasbe
3. Direktna sporočila - kanali za pogovor (tekstovno ali glasovno) med dvema uporabnikoma

Tekstovni kanali se nahajajo v cehih in se jih lahko prepozna glede na simbol #, ki se nahaja pred imenom vsakega kanala. Sem lahko uporabnik pošilja navaden tekst, emotikone, binarne datoteke, nalepke ter v primeru robotskega (angl. *bot*) računa tudi tako imenovana vgrajena sporočila (angl. *Embedded messages* oz. *Embeds*), ki so malo bolj formatirana sporočila znotraj okrašene škatle (Slika 3.4).

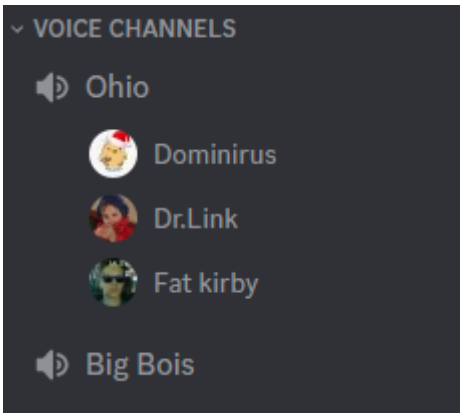


Slika 3.3: Discordov tekstovni kanal



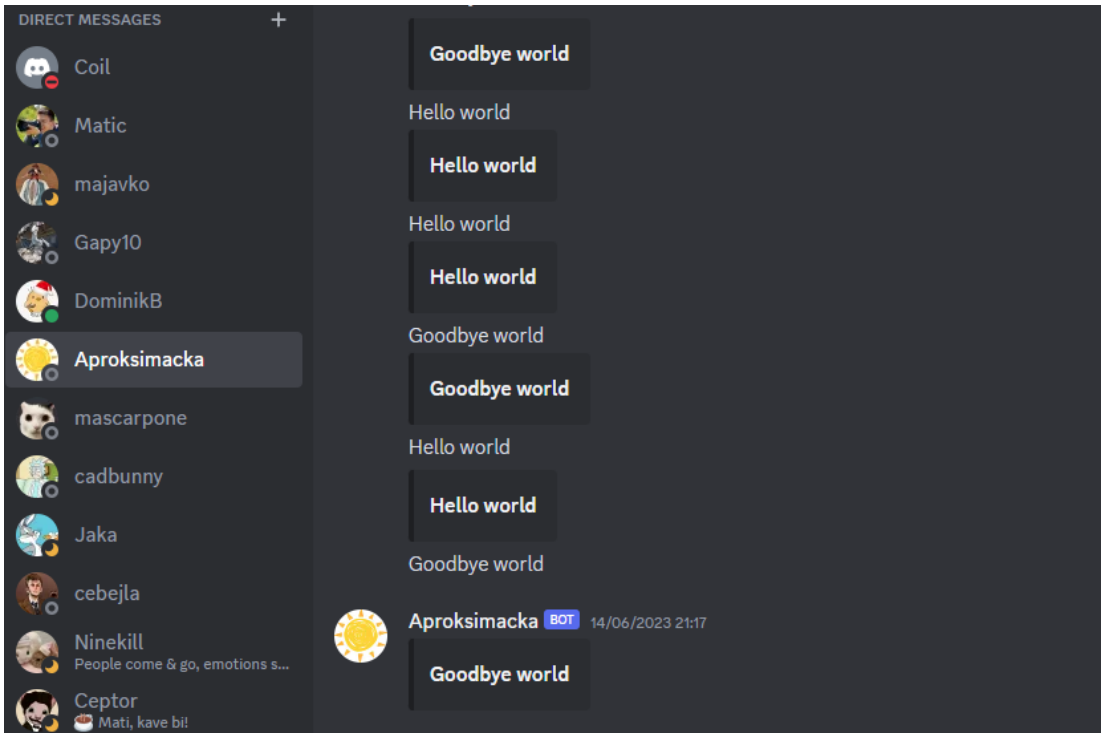
Slika 3.4: Vgrajeno sporočilo

Tako kot se tekstovni kanali lahko uporabljajo za pošiljanje tekstovnih sporočil, se analogno lahko v glasovne kanale pošilja glasovna sporočila oz. se lahko v njih pogovarja preko mikrofona ali pa predvaja glasbo. Za samo oglaševanje ti kanali niso tako aktualni, saj bi oglase lahko prejeli le uporabniki, ki so v času oglaševanja prisotni v kanalu.



Slika 3.5: Discordov glasovni kanal

Direktna oz. osebna sporočila so namenjena komunikaciji ena na ena med dvema uporabnikoma. Pošiljanje oglasov v ta sporočila bi sicer prineslo velik doseg uporabnikov, vendar je oglaševanje v direktna sporočila na vsiljiv oz. agresiven način v Discordovih pogojih uporabe prepovedano, kar pomeni, da lahko v tem primeru Discord ukine uporabnikov račun.



Slika 3.6: Discordova direktna sporočila

3.3 Oglaševanje po omrežju Discord

Po Discord omrežju se lahko oglašuje širok nabor tem, med katerimi so video igre, kreativni projekti, produkti, usluge ipd. Ne sme pa se oglaševati nelegalnih vsebin oz. vsebin, ki spodbujajo kršenje zakona, in vsebin, ki bi lahko povzročile škodo posameznikom *[5]*.

Oglašuje se lahko ročno ali pa avtomatično s primernim orodjem. Oglaševanje vključuje pisanje vsebine oglasa, ustvarjanje uporabniških računov, iskanje cehov, kamor se z uporabniškimi računi pridruži, in večkratno periodično pošiljanje sporočil v kanale cehov. Najpogosteje se za oglaševanje uporabljajo tekstovni kanali.

3.3.1 Generiranje vsebine

Oglaševalsko vsebino se lahko napiše na pamet iz glave ali pa se za to uporabi orodja, kot je na primer GPT *[6]*.

GPT-3 je najsodobnejši jezikovni model umetne inteligence, ki ga je razvilo podjetje OpenAI. Ima sposobnost generiranja besedila, ki se zdi človeško, in opravljanja različnih nalog, kot so prevajanje, povzemanje in odgovarjanje na vprašanja. GPT-3 je največji in najmočnejši jezikovni model do danes, s kapaciteto 175 milijard parametrov. Prislužil si je pozornost tako raziskovalcev kot industrijskih strokovnjakov zaradi svojih impresivnih sposobnosti in potencialnih aplikacij.

Za referenco, prejšnji odstavek je generiral GPT-3 sam.

Za generacijo oglaševalske vsebine lahko npr. GPT-ju podamo naslednjo zahtevo:

„Write advertisement for my White Rabbit NFT project which came out in 2023. We have over 9000 different NFT available for purchase. Talk about the limited Gray Rabbit NFT (only 50 available). Get additional information here www.mynft.domain“

In njegov rezultat bo:

„Attention all NFT collectors and art enthusiasts! We are excited to announce the launch of our White Rabbit NFT project, featuring over 9000 unique and one-of-a-kind digital art pieces. But that's not all - we have a special edition of limited Gray Rabbit NFTs available for purchase, with only 50 available worldwide. Don't miss out on the

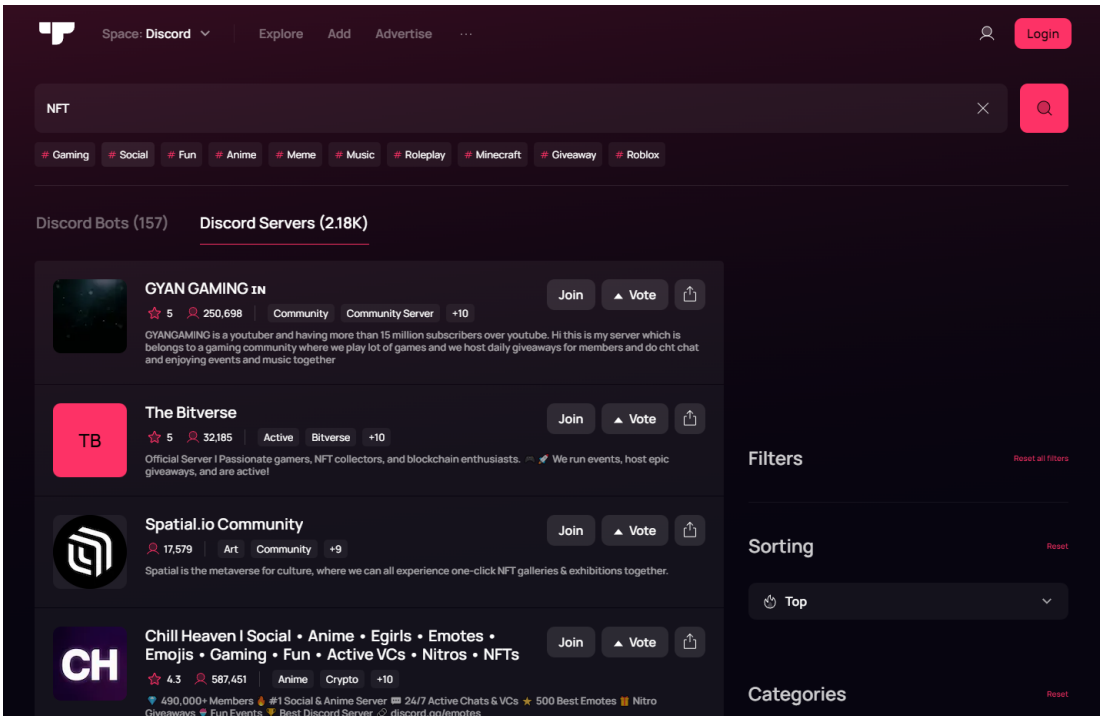
opportunity to own a piece of digital art history. These Gray Rabbit NFTs are sure to become highly sought after and increase in value over time. But act fast - there are only 50 available, and once they're gone, they're gone for good. For more information and to purchase your own Gray Rabbit NFT, visit www.mynft.domain. Don't wait - add one of these rare and valuable NFTs to your collection today!"

3.3.2 Iskanje cehov za oglaševanje

Ko je oglaševalska vsebina napisana, je potrebno najti cehe, kamor se bo to vsebino pošiljalo. Dober kraj za začetek je kar nek spletni iskalnik. Nabor cehov ponuja tudi Discord sam, ampak za to, da nek ceh pride na ta seznam, mora imeti zagotovljenih kar nekaj pogojev, med katerimi je tudi ta, da mora imeti vsaj 1000 članov. To je primerno, če želimo oglaševati v večje cehe, v primeru manjših cehov pa moramo te najti drugje.

Na srečo obstajajo tudi druge strani za iskanje cehov, kot je na primer spletna stran Top.GG. Na tej strani lahko v vgrajen iskalnik dodamo določene parametre, med katerimi je tudi imenska poizvedba, kjer lahko uporabimo akronim „NFT" in spletna stran nam bo vrnila cehe, povezane z NFT.

Tem cehom se lahko potem pridružimo in našo vsebino oglašujemo v primerne kanale. Cehi na temo NFT in kripto valut imajo ponavadi namenske kanale, ki so namenjeni oglaševanju, in lahko vanje oglašujemo brez posledic, medtem ko nas oglaševanje v drugih kanalih lahko privede do izključitve s strežnika.



Slika 3.7: Iskanje cehov na Top.GG *[7]*

4 Ogrodje za oglaševanje po Discordu

Slika 4.1: Logotip projekta (Narejen z DALL-E & Adobe Photoshop)

4.1 Namen projekta

Cilj projekta je izdelati ogrodje, ki lahko deluje 24 ur na dan in samodejno oglašuje vnaprej definirano ali dinamično vsebino, omogoča pregled poslanih sporočil in poročanje o uspešnosti preko beleženja zgodovine poslanih sporočil. Ker naj bi ogrodje delovalo brez prekinitev, je cilj izdelati ga na način, da bo delovalo kot demonski proces v ozadju, brez grafičnega vmesnika.

Konfiguracijo ogrodja se bo izvedlo preko Python datoteke oz. skripte, katero se bo neposredno pognalo v Python okolju. Ker ta način konfiguracije zahteva nekaj dela in ni najbolj enostaven, je cilj izdelati tudi grafični vmesnik, ki bo deloval kot dodaten nivo nad samim jedrom ogrodja ter omogočal konfiguracijo in izvoz le-te na način, da se jo bo lahko uporabilo brez grafičnega vmesnika.

Za lažji pregled dogajanja na strežniku je cilj na grafičnem vmesniku implementirati možnost oddaljenega dostopa, ki bo omogočal direktno manipulacijo s sporočili in pre-

gled zgodovine poslanih sporočil za določitev uspešnosti oglaševanja.

4.2 Zasnova in razvoj

4.2.1 Zasnova in razvoj jedra

To poglavje govori o jedru samega ogrodja, kjer jedro obsega vse, kar ni del grafičnega vmesnika.

Jedro je zasnovano kot Python knjižnica/paket, ki se ga lahko namesti preko PIP-a (*Preferred Installer Program*), ki je vgrajen v Python in služi nalaganju Python paketov. Za uporabo jedra morajo uporabniki ustvariti oglaševalsko (konfiguracijsko) `.py` datoteko in definirati ustrezno konfiguracijo. To zahteva nekaj osnovnega znanja Python jezika, obstaja pa tudi možnost, da se to datoteko generira iz grafičnega vmesnika (*Zasnova in razvoj grafičnega vmesnika*).

AsyncIO

Jedro ogrodja je zasnovano za sočasno (angl. *concurrent*) večopravilnost, kar pomeni, da se lahko na videz več opravil izvaja naenkrat, v resnici pa se med njimi zelo hitro preklaplja. To je omogočeno s knjižnico `asyncio`. AsyncIO omogoča ustvarjanje `async` funkcij, ki vrnejo korutine (angl. *coroutine*). Te korutine lahko potem zaženemo v opravilih, med katerimi bo program preklopil vsakič, ko v trenutnem opravilu z `await` besedo na primer čakamo:

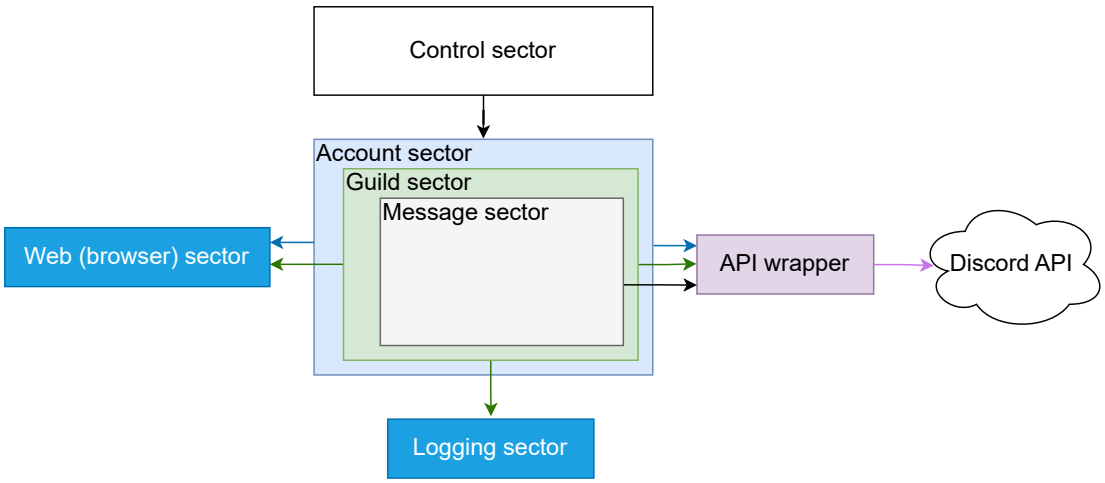
- konec neke asinhrone komunikacije (angl. *Async I/O*),
- da se trenutno opravilo zbudi iz spanja,
- da se nek semafor odklene[1],
- ipd.

[1] Semafor je mehanizem za sinhronizacijo opravil, kjer omejimo, koliko opravil lahko naenkrat dostopa do nekega zaščitenega dela kode oz. skupne surovine. https://docs.python.org/3/library/asyncio-sync.html.

Sektorji jedra ogrodja

Za lažjo implementacijo in kasnejši razvoj je jedro ogrodja razdeljeno na več sektorjev. Ti so:

- nadzorni sektor,
- sektor uporabniških računov,
- cehovski (strežniški) sektor,
- sporočilni sektor,
- sektor beleženja zgodovine sporočil,
- sektor (avtomatizacije) brskalnika,
- sektor za ovoj Discord API (angl. *Discord API wrapper sector*).



Slika 4.2: Sestava jedra ogrodja

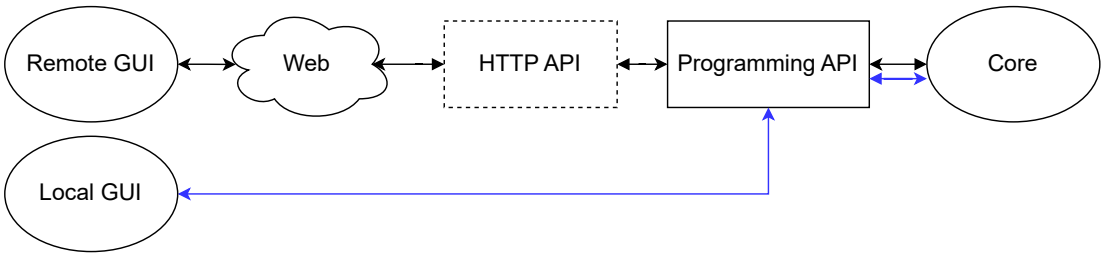
Nadzorni sektor

Nadzorni sektor skrbi za zagon samega ogrodja in njegovo zaustavitev. Skrbi tudi za procesiranje ukazov, ki jih ogrodje ponuja, preko lastnega programskega vmesnika ali preko HTTP vmesnika, pri čemer v programski vmesnik spadajo Python funkcije ogrodja in metode objektov za neposredno upravljanje ogrodja na isti napravi, HTTP vmesnik pa nudi podporo za upravljanje jedra na daljavo.

V tem sektorju se dodajajo novi uporabniški računi oz. se odstranjujejo tisti, v katerih je prišlo do napake. Prav tako se tu zgodi inicializacija sektorja beleženja sporočil, s katerim kasneje komunicira cehovski sektor.

Nadzorni sektor ima vedno vsaj eno opravilo (poleg opravil v ostalih sektorjih), in sicer je to tisto, ki skrbi za čiščenje uporabniških računov v primeru napak. Drugo opravilo se zažene le v primeru, da je vklopljeno shranjevanje objektov v datoteko. Ogrodje samo po sebi deluje tako, da ima vse objekte (račune, cehe, sporočila ipd.) shranjene neposredno v RAM. Že od samega začetka je ogrodje narejeno na način, da se želene objekte definira preko Python skripte in je zato shranjevanje v RAM ob taki definiciji neproblematično, problem pa je nastopil, ko je bilo dodano dinamično dodajanje in brisanje objektov, kar uporabnikom dejansko omogoča, da ogrodje uporabljajo dinamično. V tem primeru je bilo potrebno dodati neke vrste permanentno shrambo. Razmišljalo se je o več alternativah, ena izmed njih je bila, da bi se vse objekte shranjevalo v neko bazo podatkov, ki bi omogočala preslikavo bazičnih podatkov v objekte, kar bi z vidika robustnosti bila zelo dobra izbira, a bi to zahtevalo veliko prenovo vseh sektorjev, zato se je na koncu izbrala preprosta opcija shranjevanja objektov, ki preko `pickle` modula shrani vse račune ob vsakem normalnem izklopu ogrodja ali pa v vsakem primeru na dve minuti periodično. V prihodnosti so še vedno načrti za izboljšanje tega mehanizma in možnost uporabe prej omenjene podatkovne bazene ni izključena.

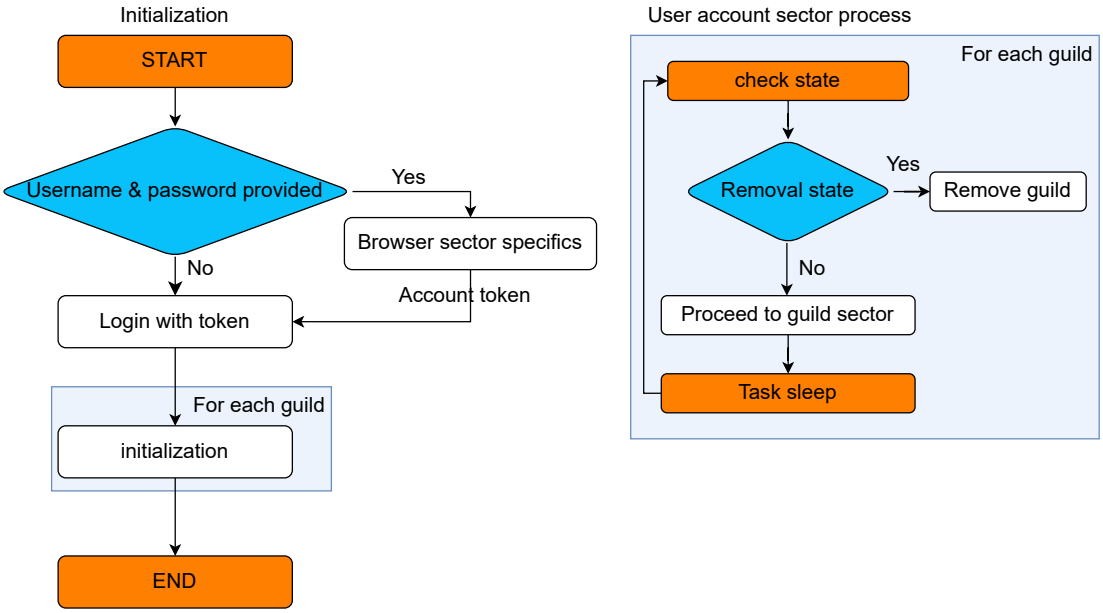
V nadzornem sektorju se poleg programskega vmesnika nahaja tudi HTTP vmesnik, ki služi kot podpora za oddaljen dostop grafičnega vmesnika do jedra. Deluje na knjižnici aiohttp, ki je asinhrona HTTP knjižnica. HTTP vmesnik je v resnici zelo preprost in deluje tako, da ob neki HTTP zahtevi ustvari novo `asyncio` opravilo, ki potem zahtevo posreduje programskemu vmesniku, kar pomeni, da je rezultat enak tistemu, ki bi ga dobili ob lokalnem delovanju na isti napravi. Vsi podatki se na HTTP vmesniku pretakajo v JSON formatu. Osnoven koncept je prikazan na spodnji sliki, kjer je z barvo puščic prikazan ločen potek.



Slika 4.3: Povezava do jedra

Sektor uporabniških računov

Sektor uporabniških računov je zadolžen za upravljanje z uporabniškimi računi. Za dodajanje novega uporabniškega računa morajo uporabniki ustvariti `daf.client.ACCOUNT`[2] objekt. V primeru, da je bil podan uporabniški žeton (angl. *token*), sektor takoj ustvari povezavo na sektor ovoja Discord API, če sta bila podana uporabniško ime in geslo, pa sektorju brskalnika poda zahtevo za prijavo preko brskalnika, iz katerega potem pridobi uporabniški žeton in zatem ustvari povezavo na sektor ovoja Discord API.



Slika 4.4: Delovanje sektorja uporabiških računov

[2] Vsebina včasih vsebuje reference na objekte, ki niso podrobno opisani v diplomskem delu, so pa na voljo v uradni spletni dokumentaciji projekta: https://daf.davidhozic.com/en/v2.9.x/

Cehovski sektor

Cehovski sektor je primarno zadolžen za upravljanje s cehi (strežniki).

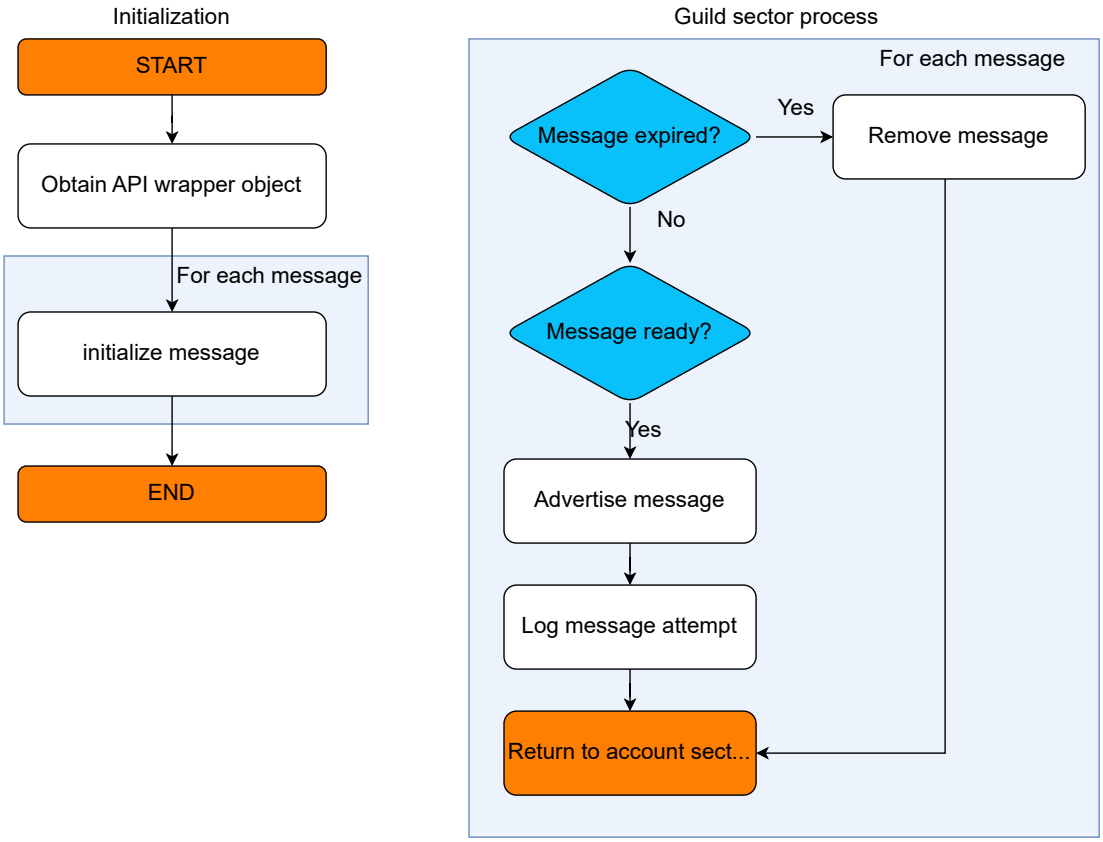
Sektorju pripadajo trije razredi:

- `GUILD`,
- `USER`,
- `AutoGUILD`.

`GUILD` in `USER` sta med seboj praktično enaka, edina razlika med njima je ta, da `USER` predstavlja osebe, katerim bomo pošiljali sporočila, `GUILD` pa predstavlja cehe s kanali.

`AutoGUILD` pa po drugi strani sam po sebi ne predstavlja točno specifičnega ceha, ampak več cehov, katerih ime se ujema s podanim RegEx vzorcem.

Sam cehovski sektor na začetku razvoja sploh ni bil potreben, a je bil vseeno dodan, preprosto zaradi boljše preglednosti ne samo notranje kode, ampak tudi kode za definiranje same oglaševalske skripte ob velikem številu sporočil. To je sicer posledično zahtevalo definicijo dodatnih vrstic v oglaševalski skripti, kar je hitro postalo opazno ob 90 različnih cehih. Vseeno se je ta izbira dobro izšla, saj je zdaj na cehovskem sektorju veliko funkcionalnosti, ki ne spadajo v ostale sektorje, kot je na primer avtomatično iskanje novih cehov in njihovo pridruževanje. Ta struktura nudi tudi veliko preglednosti v primeru beleženja sporočil (vsaj v primeru JSON datotek), kjer je vse razdeljeno po različnih cehih.



Slika 4.5: Delovanje cehovskega sektorja

Sporočilni sektor

Sporočilni sektor je zadolžen za pošiljanje dejanskih sporočil v posamezne kanale. V tem sektorju so na voljo trije glavni razredi za ustvarjanje različnih vrst sporočil:

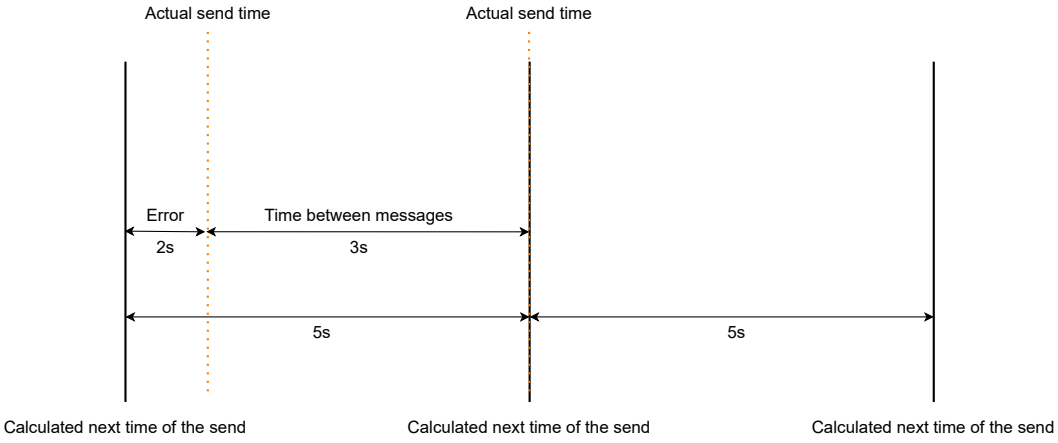
1. `TextMESSAGE` - pošiljanje tekstovnih sporočil v cehovske kanale,
2. `VoiceMESSAGE` - predvajanje zvočnih posnetkov v cehovskih kanalih,
3. `DirectMESSAGE` - pošiljanje tekstovnih sporočil v zasebna sporočila enega samega uporabnika.

`TextMESSAGE` in `DirectMESSAGE` sta si precej podobna, primarno gre v obeh primerih za tekstovna sporočila, razlika je v kanalih, ki jih `DirectMESSAGE` nima, temveč pošilja sporočila v direktna sporočila uporabnika. `VoiceMESSAGE` in `TextMESSAGE` sta si po vrsti podatkov sicer različna, vendar pa oba pošiljata sporočila v kanale, ki pripadajo nekemu cehu, in imata praktično enako inicializacijo.

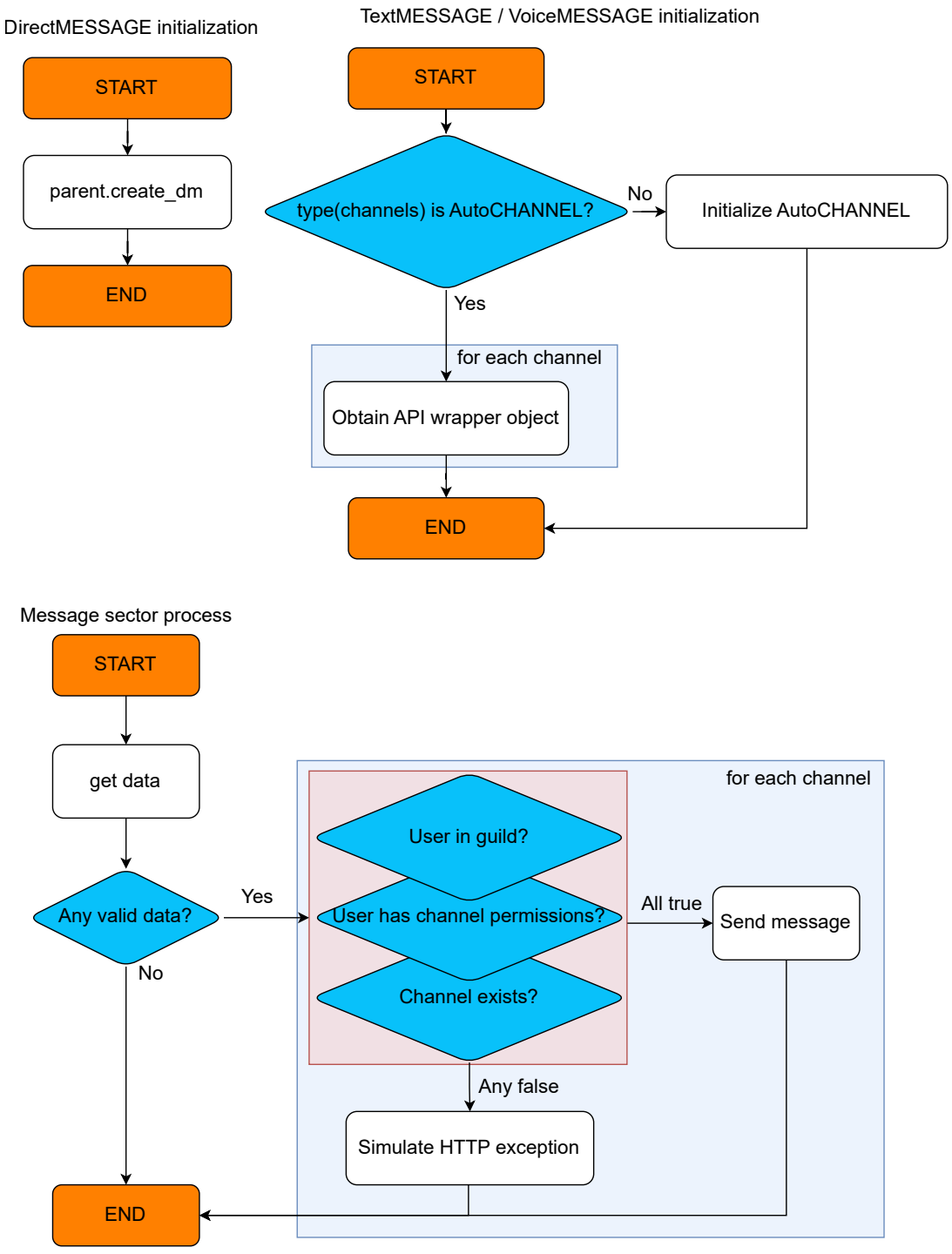
Pripravljenost sporočila za pošiljanje določa notranji atribut objekta, ki predstavlja točno specifičen čas naslednjega pošiljanja sporočila. V primeru, da je trenutni čas večji od tega atributa, je sporočilo pripravljeno za pošiljanje. Ob ponastavitvi „časovnika“ se ta atribut prišteje za konfigurirano periodo. Torej čas pošiljanja ni relativen na dejanski prejšnji čas pošiljanja, temveč je relativen na predvideni prejšnji čas pošiljanja. Taka vrsta računanja časa omogoča določeno toleranco pri pošiljanju sporočila, saj se zaradi raznih zakasnitev in omejitev zahtev (angl. *Rate limiting*) na Discord API dejansko sporočilo lahko pošlje kasneje kot predvideno. To je predvsem pomembno v primeru, ko imamo definiranih veliko sporočil v enem računu, kar je zagotovilo, da se sporočilo ne bo poslalo točno ob določenem času. Ker se čas prišteva od prejšnjega predvidenega časa pošiljanja, bo v primeru zamude sporočila razmak med tem in naslednjim sporočilom manjši točno za to časovno napako (če privzamemo, da ne bo ponovne zakasnitve).

Pred tem algoritmom je bil za določanje časa pošiljanja v rabi preprost časovnik, ki se je ponastavil po vsakem pošiljanju, a se je zaradi Discordove omejitve API zahtevkov in tudi drugih Discord API zakasnitev čas pošiljanja vedno pomikal malo naprej, kar je pomenilo, da če je uporabnik ogrodje konfiguriral, da se neko sporočilo pošlje vsak dan, in definiral čas začetka npr. naslednje jutro ob 10.00 (torej pošiljanje vsak dan ob tej uri), potem je po (sicer veliko) pošiljanjih namesto ob 10.00 uporabnik opazil, da se

sporočilo pošlje ob 10.01, 10.02 itd. Primer računanja časa in odprave časovne napake je prikazan na spodnji sliki.



Slika 4.6: Čas pošiljanja sporočila z upoštevanjem časa procesiranja



Slika 4.7: Delovanje sporočilnega sektorja

Sektor beleženja zgodovine sporočil

Sektor beleženja je zadolžen za beleženje poslanih sporočil oz. beleženje poskusov pošiljanja sporočil. Podatke, ki jih mora zabeležiti, dobi iz cehovskega sektorja. Beleži se tudi podatke o pridružitvi novih članov, če je to konfigurirano v cehovskem sektorju.

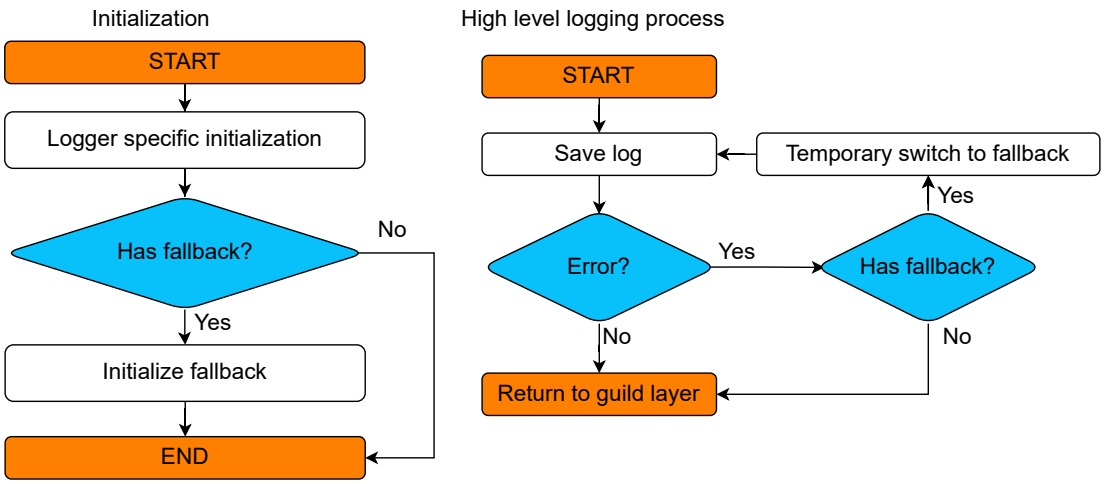
Omogoča beleženje v tri različne formate, kjer vsakemu pripada lasten objekt beleženja:

1. JSON - `LoggerJSON`
2. CSV (nekatera polja so JSON) - `LoggerCSV`
3. SQL - `LoggerSQL`

Ob inicializaciji jedra se v nadzornem sektorju poda želen objekt beleženja, ki se inicializira in shrani v sektor beleženja. Po svoji lastni inicializaciji se inicializira še njegov nadomestni (`fallback` parameter) objekt, ki se uporabi v primeru kakršnekoli napake pri beleženju.

Po vsakem poslanem sporočilu se iz cehovskega sektorja naredi zahteva, ki vsebuje podatke o cehu, poslanem sporočilu oz. poskusu pošiljanja ter podatke o uporabniškem računu, ki je sporočilo poslal. Sektor beleženja posreduje zahtevo izbranemu objektu beleženja, ki v primeru napake dvigne Python napako (angl. *exception*), na kar sektor beleženja reagira tako, da začasno zamenja objekt beleženja na njegov nadomestek in ponovno poskusi. Poskuša, dokler mu ne zmanjka nadomestkov ali pa je beleženje uspešno.

Pred JSON, CSV in SQL beleženjem se je vse beležilo v Markdown datoteke, kjer se je lahko podatke pregledovalo v berljivem formatu, vendar je bila ta vrsta beleženja kasneje zamenjana z JSON beleženjem.



Slika 4.8: Višji nivo beleženja

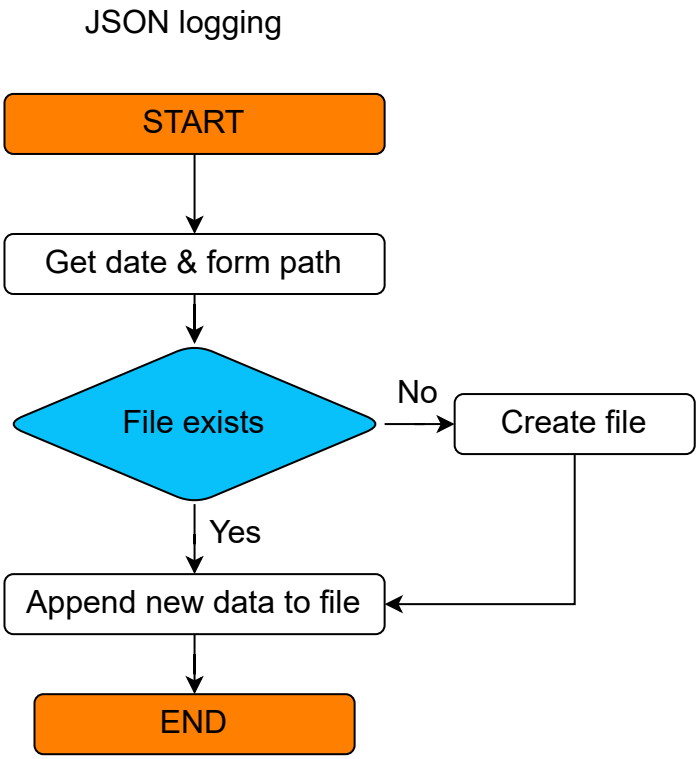
JSON beleženje

Kot že prej omenjeno, je JSON beleženje zamenjava za Markdown format beleženja. Razlog za zamenjavo je morebitna implementacija analitike, kar bi se v Markdown formatu težko implementiralo. V času pisanja je analitika na voljo le v primeru SQL beleženja.

JSON beleženje je implementirano z objektom beleženja `LoggerJSON`. Ta vrsta beleženja nima nobene specifične inicializacije, kliče se le inicializacijska metoda njegovega morebitnega nadomestka.

Ob zahtevi beleženja objekt `LoggerJSON` najprej pogleda trenuten datum, iz katerega tvori končno pot do datoteke od (v parametrih) konfigurirane osnovne poti. Končna pot je določena kot `Leto/Mesec/Dan/<Ime Ceha>.json`.

To pot, v primeru, da ne obstaja, ustvari in zatem z uporabo vgrajenega Python modula `json` podatke shrani v datoteko.



Slika 4.9: Process JSON beleženja

CSV beleženje

CSV beleženje deluje na enak način kot JSON beleženje. Edina razlika je v formatu, v tem primeru je to CSV. Lokacija datotek je enaka kot pri JSON beleženju. Za shranjevanje je uporabljen vgrajen Python modul `csv`.

Za sam pregled poslanih sporočil to ni najbolj primeren format, saj se vse shrani v eni datoteki, pri čemer za razliko od JSON formata ni večslojnih struktur.

SQL beleženje

SQL beleženje deluje precej drugače kot delujeta JSON beleženje in CSV beleženje, saj se podatki shranjujejo v podatkovno bazo, ki je v primeru uporabe SQLite dialekta lahko tudi datoteka.

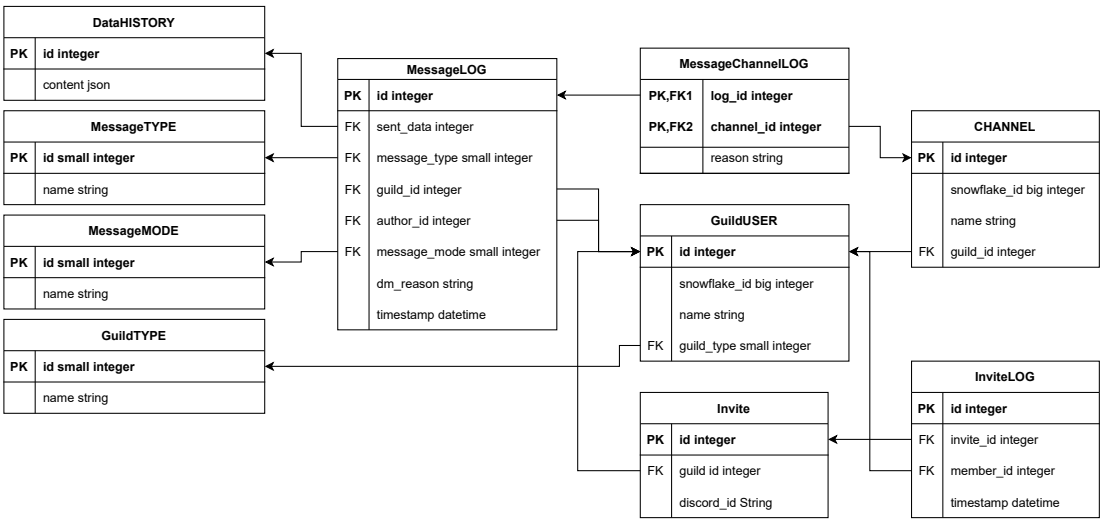
Beleženje je omogočeno v štirih SQL dialektih:

1. SQLite,
2. Microsoft SQL Server (T-SQL),
3. PostgreSQL,
4. MySQL/MariaDB.

Za čim bolj univerzalno implementacijo na vseh dialektih je bila pri razvoju uporabljena knjižnica `SQLAlchemy`.

Celoten sistem SQL beleženja je implementiran s pomočjo ORM, kar med drugim omogoča, da SQL tabele predstavimo s Python razredi, posamezne vnose v bazo podatkov oz. vrstice pa predstavimo z instancami teh razredov. Z ORM lahko skoraj v celoti skrijemo SQL in delamo neposredno s Python objekti, ki so lahko tudi gnezdene strukture, npr. vnos dveh ločenih tabel lahko predstavimo z dvema ločenima instancama, kjer je ena instanca gnezdena znotraj druge.

Ta vrsta beleženja je bila pravzaprav narejena v okviru zaključnega projekta pri predmetu Informacijski sistemi v 2. letniku. Ker smo morali pri predmetu izpolnjevati določene zahteve, je bilo veliko stvari pisanih neposredno v SQL jeziku, a vseeno je bila že takrat uporabljena knjižnica SQLAlchemy. Zaradi določenih SQL zahtev (funkcije, procedure, prožilci ipd.) je bila ta vrsta beleženja možna le ob uporabi Microsoft SQL Server dialekta. Kasneje se je postopoma celotno SQL kodo zamenjalo z ekvivalentno Python kodo, ki preko SQLAlchemy knjižnice dinamično generira potrebne SQL stavke, zaradi česar so bile odstranjene določene uporabne originalne funkcionalnosti, implementirane na sektorju same SQL baze, kot so npr. prožilci (angl. *trigger*), ki si jih lahko predstavljamo kot neke odzivne funkcije na dogodke. Je pa zaradi tega možno uporabljati bazo na več dialektih, dodatno pa je bilo veliko stvari lažje implementirati, saj se ni bilo potrebno zanašati na specifike dialekta.



Slika 4.10: SQL entitetno-relacijski diagram[Stran 31, 3]

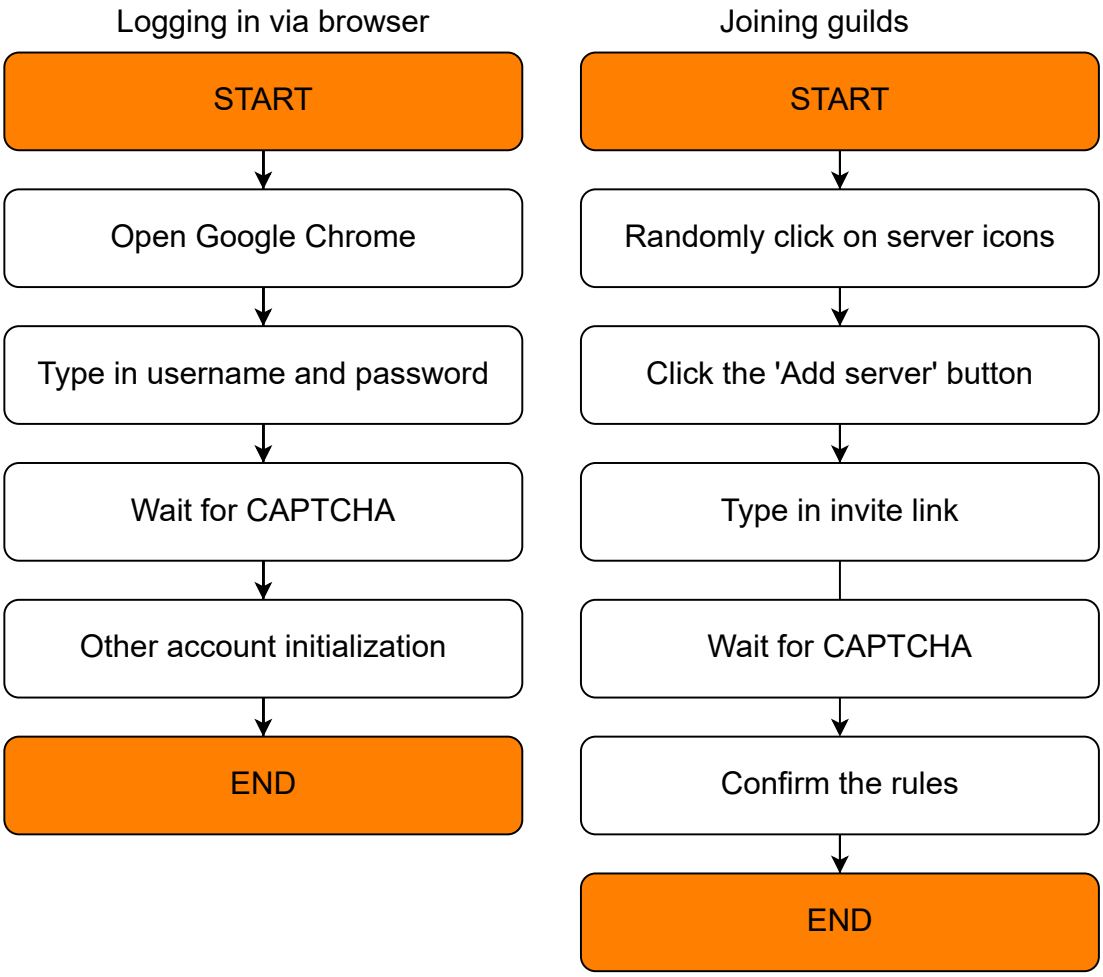
[3] Relacije (tabele) so opisane v uradni dokumentaciji: SQL Tables.

Sektor brskalnika

Velika večina ogrodja deluje na podlagi ovojnega API sektorja, kjer ta direktno komunicira z Discord API. Določenih stvari pa se neposredno z Discord API ne da narediti ali pa za izvedbo neke operacije (prepovedane v pogojih uporabe Discorda) obstaja velika možnost, da Discord suspendira uporabnikov račun.

Za ta namen je bil ustvarjen sektor brskalnika, kjer ogrodje namesto z Discord API komunicira z brskalnikom Google Chrome. To opravlja s knjižnico Selenium, ki je namenjena avtomatizaciji brskalnikov in se posledično uporablja tudi kot orodje za avtomatično testiranje spletnih grafičnih vmesnikov.

V ogrodju Selenium ni uporabljen za testiranje, temveč za avtomatično prijavljanje v Discord z uporabniškim imenom in geslom ter za pridruževanje novim cehom. Dejansko ta sektor posnema živega uporabnika.



Slika 4.11: Delovanje sektorja brskalnika

Ovojni Discord API sektor

Sektor, ki ovija Discord API, ni striktno del samega ogrodja, ampak je to knjižnica oz. ogrodje Pycord. PyCord je odprtokodno ogrodje, ki je nastalo iz kode starejšega discord.py. Ogrodje PyCord skoraj popolnoma zakrije Discord API z raznimi objekti, ki jih ogrodje interno uporablja.

Če bi si ogledali izvorno kodo (angl. *source code*) ogrodja, bi opazili, da je poleg `daf` paketa tudi paket z imenom `_discord`. To ni nič drugega kot PyCord ogrodje, le da je modificirano za možnost rabe na uporabniških računih (poleg avtomatiziranih robotskih računov).

4.2.2 Zasnova in razvoj grafičnega vmesnika

Ogrodje lahko v celoti deluje brez grafičnega vmesnika, a ta način zahteva pisanje konfiguracijskih `.py` datotek oz. Python skript, kar je marsikomu težje, sploh če se še nikoli niso srečali s Python jezikom.

V namen enostavnejše rabe ogrodja je izdelan grafični vmesnik, ki deluje ločeno od samega jedra ogrodja, z njim pa lahko komunicira lokalno preko programskega vmesnika ali pa na daljavo preko HTTP vmesnika.

Za dizajn vmesnika je izbran svetel dizajn, z modrimi odtenki za posamezne elemente (Slika 7.2).

Tkinter

Za izdelavo grafičnega vmesnika je bila uporabljena knjižnica ttkboostrap, ki je razširitev vgrajene Python knjižnice `tkinter`.

Tkinter knjižnica je v osnovi vmesnik na Tcl/Tk orodje za izdelavo grafičnih vmesnikov, doda pa tudi nekaj svojih nivojev, ki še dodatno razširijo delovanje knjižnice. Omogoča definicijo različnih pripomočkov (angl. *widgets*), ki se jih da dodatno razširiti in shraniti pod nove pripomočke, ki jih lahko večkrat uporabimo. Ti pripomočki so na primer `Combobox`, ki je neke vrste (angl.) „drop-down“ meni, `Spinbox` za vnašanje številskih vrednosti, `Button` za gumbe ipd. Posamezne pripomočke se da tudi znatno konfigurirati, kjer lahko spreminjamo stile, velikost, pisavo ipd *[8]*.

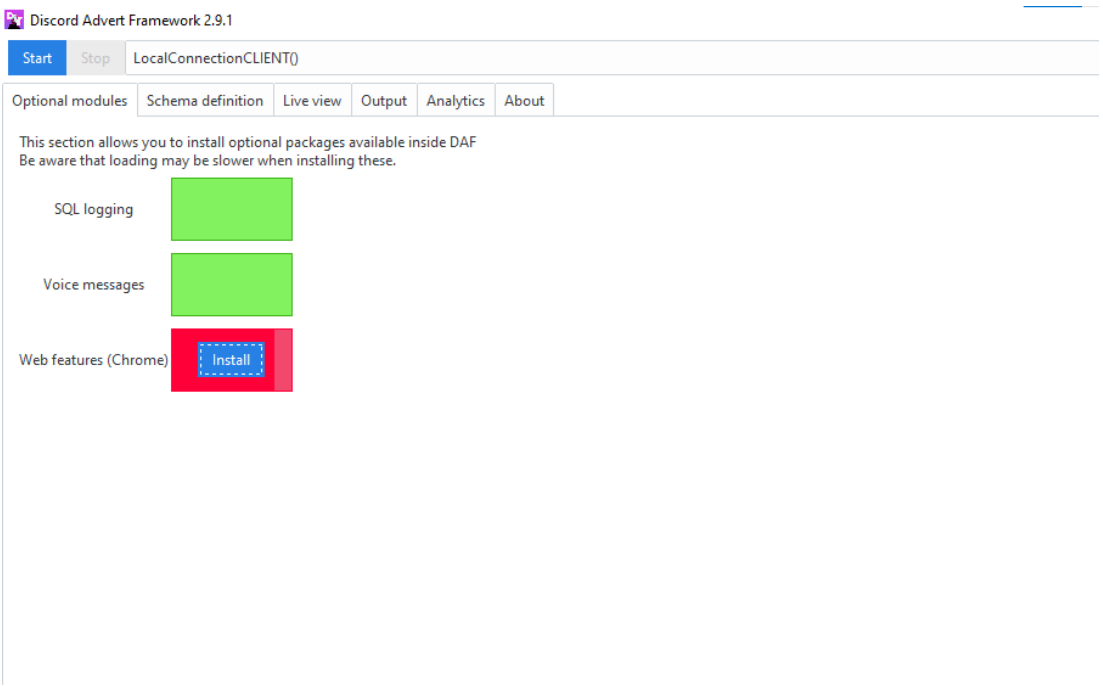
Pred izbiro Tkinter knjižnice je bila ena izmed možnosti tudi knjižnica PySide (QT), a na koncu se je vseeno obnesla Tkinter oz. ttkboostrap knjižnica, saj je že osnovni paket PySide6 knjižnice velik 70 MB, z dodatki pa skoraj 200 MB, medtem ko je Tkinter na veliko platformah že prisotna kar v sami Python distribuciji in ne zahteva nobene dodatne namestitve, torej je ttkboostrap edina dodatna zunanja knjižnica, ki jo je potrebno namestiti, in sicer se namesti kar sama, ob prvem zagonu grafičnega vmesnika.

Zavihki

Grafični vmesnik ogrodja je razdeljen na več zavihkov, kjer je vsak namenjen ločenim funkcionalnostim.

Optional modules zavihek

Optional modules zavihek omogoča namestitev dodatnih modulov, ki jih osnovni paket ogrodja pogreša (zaradi hitrejšega zagona). Sestavljen je iz statusnih panelov, ki v primeru, da so obarvani rdeče (modul ni nameščen), vsebujejo še gumb za namestitev. Gumb bo namestil potrebne pakete, potem pa bo uporabniku sporočeno, da mora za spremembe ponovno odpreti vmesnik. Po ponovnem zagonu bo statusni panel za posamezen modul obarvan zeleno.



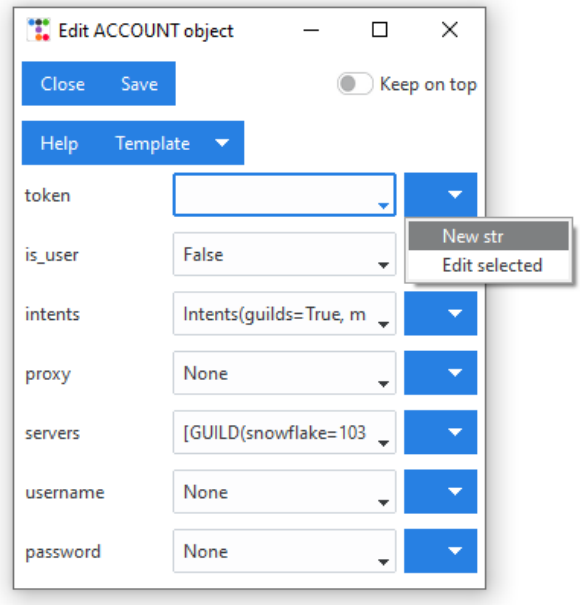
Slika 4.12: Izgled *Optional Modules* zavihka

Schema definition zavihek

Schema definition omogoča definicijo uporabniških računov (in v njih cehov, sporočil ipd.), definicijo upravljalnika za beleženje, izbiro globine izpisov na konzoli in konfiguracijo povezave do jedra ogrodja. Omogoča tudi shrambo teh definicij v JSON datoteko, branje definicij iz JSON datoteke in generacijo ekvivalentne Python datoteke (konfiguracijske datoteke jedra), ki požene le jedro orodja (brez grafičnega vmesnika). Pravzaprav je ta zavihek namenjen definiciji neke predloge, ki jo lahko potem uvozimo v jedro ogrodja. Izgled je prikazan na Slika 7.2.

Omogoča tudi dinamično branje in dodajanje objektov v že zagnanem vmesniku preko gumbov, ki vsebujejo besedo *live*.

Uporabniške račune (in ostale objekte) se lahko definira s klikom na opcijski meni *Object options* in opcijo *New ACCOUNT*. Ob kliku se odpre novo okno, ki je avtomatično in dinamično generirano iz podatkov o podatkovnih tipih, ki jih sprejme razred ob definiciji. Za vsak parameter se generirajo oznaka, opcijski meni in opcijski gumb, v katerem lahko urejamo izbrano vrednost oz. definiramo novo.



Slika 4.13: Definicija uporabiškega računa

Shranjevanje sheme (predloge) v datoteko, nalaganje sheme iz datoteke in generiranje

ekvivalentne Python datoteke je možno preko opcijskega menija *Schema*. Datoteka, kamor se shrani shema, je datoteka formata JSON in vsebuje definirane račune, objekte za beleženje sporočil, objekte za povezovanje z jedrom ipd. Vsi objekti znotraj grafičnega vmesnika pravzaprav niso pravi Python objekti, ampak so dodaten nivo abstrakcije, ki je sestavljen iz samega podatkovnega tipa (razreda) definiranega objekta in pa parametrov, ki so shranjeni pod slovar (`dict`). Pretvorba v JSON poteka rekurzivno, tako da se za vsak objekt v JSON naredi nov pod-slovar, kjer so zapisani podatkovni tip (kot besedilo) in parametri objekta.

Nalaganje sheme (predloge) iz JSON datoteke je možno preko *Schema* menija in poteka rekurzivno, tako da se za vsak vnos najprej na podlagi celotne poti do razreda naloži (angl. *import*) Python modul, potem pa iz modula še podatkovni tip (razred). Zatem se ustvari abstraktni objekt na enak način, kot je bil ustvarjen pred shranjevanjem v JSON shemo.

Preko *Schema* menija je možno ustvariti tudi ekvivalentno Python datoteko, ki bo oglaševala na enak način kot v grafičnem vmesniku, brez dejanskega grafičnega vmesnika. Ob kliku na gumb *Generate script* se definira Python koda, ki na vrhu definira vse potrebno in zatem zažene ogrodje. Primer skripte je prikazan v Blok kode 7.1.

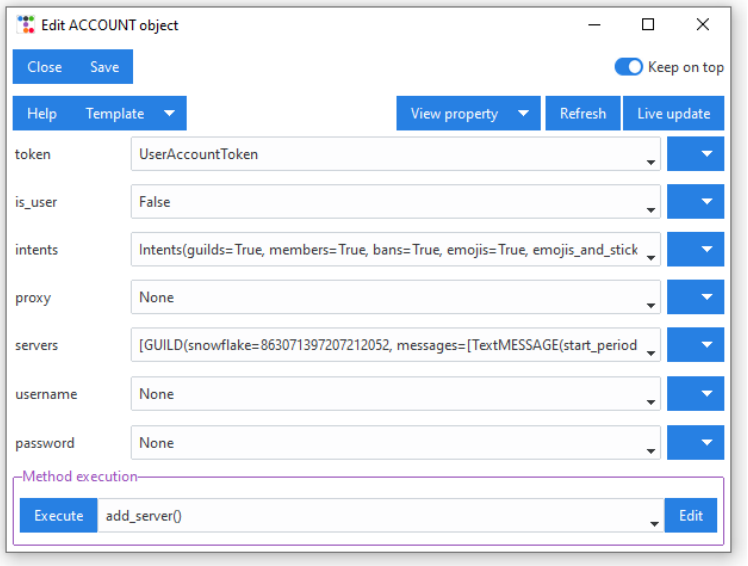
Live view zavihek

Medtem ko je *Schema definition zavihek* namenjen definiciji vnaprej definirane sheme oz. predloge objektov, *Live view* zavihek omogoča direktno manipulacijo z objekti, ki so dodani v delujoče jedro ogrodja.

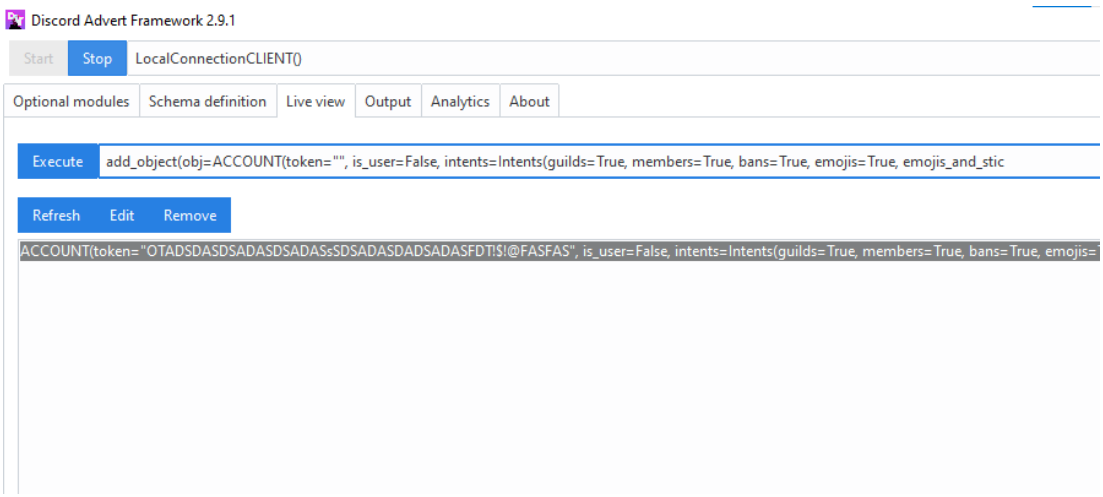
Na začetku zavihka se nahaja opcijski meni, v katerem je *add_object* funkcija, znotraj katere lahko definiramo nov račun. Ob kliku na gumb *Execute* bo definiran račun takoj dodan v ogrodje in začel z oglaševanjem.

Pod opcijskim menijem se nahajajo trije gumbi. *Refresh* posodobi spodnji seznam z uporabniškimi računi, ki oglašujejo, *Edit* gumb odpre okno za definiranje računov, kjer se vanj naložijo obstoječe vrednosti iz uporabniškega računa, ki ga urejamo. Okno poleg gumbov oz. pripomočkov, ki jih ima pri urejanju v *Schema definition zavihku*, vsebuje tudi dva dodatna gumba. Ta gumba sta *Refresh* gumb, ki v okno naloži osvežene vrednosti iz dejanskega objekta, dodanega v ogrodje, in *Live update* gumb, ki dejanski objekt v ogrodju na novo inicializira z vrednostmi, definiranimi v oknu. Na dnu okna

je znotraj vijoličnega okvirja možno izvajanje metod (funkcij) na objektu.



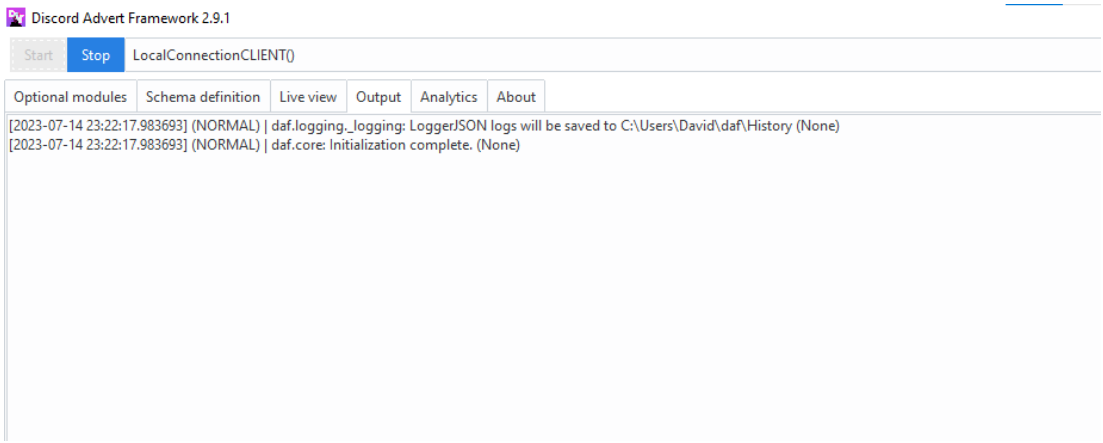
Slika 4.14: Prikaz parametrov in metod delujočega računa



Slika 4.15: *Live view* zavihek

Output zavihek

Vse, kar se nahaja v *Output* zavihku, je seznam izpisov, ki se izpišejo na standardnem izhodu STDOUT. Uporabi se ga lahko za bolj podroben pregled, kaj se dogaja z jedrom ogrodja.



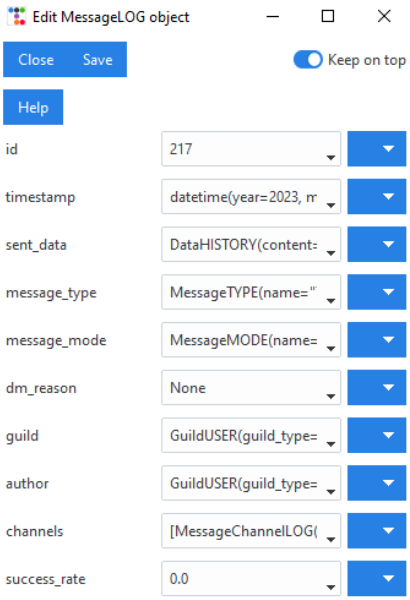
Slika 4.16: *Output tab* zavihek

Analytics zavihek

Analytics zavihek omogoča analizo poslanih sporočil in njihovo statistiko. Prav tako omogoča analizo pridruževanj preko sledenja cehovskih pridružnih povezav (angl. *Invite links*). Izgled je prikazan na Slika 7.1.

Za pridobitev vnosov se uporabi gumb *Get logs*, ki na podlagi parametrov, definiranih v zgornjem opcijskem meniju, vrne v spodnji seznam filtrirane elemente. Te elemente se lahko vsakega posebej pregleda z gumbom *View log*, ki odpre okno za urejanje objektov.

Za pridobitev statistike se uporabi gumb *Calculate*, ki na podlagi opcijskega menija nad gumbom v spodnjo tabelo vrne podatke.



Slika 4.17: Prikaz vnosa o poslanem sporočilu.

Povezava grafičnega vmesnika z jedrom ogrodja

Grafični vmesnik lahko s stališča lokacije delovanja deluje na dva načina. Prvi je lokalni način, kjer grafični vmesnik jedro ogrodja zažene na istem računalniku. Drugi način je oddaljeni režim delovanja, kjer se grafični vmesnik poveže na HTTP strežnik, ki deluje znotraj jedra ogrodja, in nanj pošilja zahtevke, ki se v jedru potem preslikajo na programski vmesnik. Koncept je prikazan na Slika 4.3.

V primeru oddaljenega dostopa se podatki serializirajo v JSON reprezentacijo, kjer so navadne vrednosti neposredno serializirane v JSON format, večina objektov pa v slovar (`dict`), kjer so zapisani pot do podatkovnega tipa (razreda) objekta in njegovi atributi. Obstaja nekaj izjem pri serializaciji objektov, ena izmed teh je `datetime` tip objekta, ki se serializira v besedilo po standardu ISO 8601. Pretvorbo v končno JSON reprezentacijo opravlja vgrajena knjižnica `json`, medtem ko pretvorbo objektov v slovar opravlja funkcija *`daf.convert.convert_object_to_semi_dict()`*. Serializacijo in deserializacijo opravljata grafični vmesnik in jedro oba enako. Včasih se pri pošiljanju podatkov iz grafičnega vmesnika na jedro sploh ne serializira, temveč se pošlje le referenco (identifikator) objekta, kjer se na strežniku (jedru) objekt pridobi iz spomina preko reference.

daf.convert.**convert_object_to_semi_dict**(*to_convert: Any*, *only_ref: bool = False*) → Mapping

Converts an object into dict.

Parametri

- **to_convert** (*Any*) – The object to convert.
- **only_ref** (*bool*) – If True, the object will be replaced with a ObjectReference instance containing only the object_id.

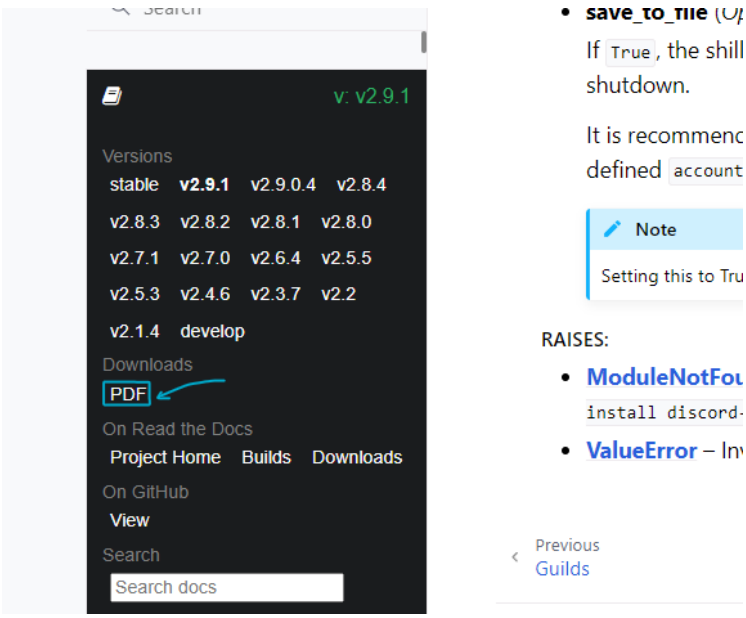
Za konfiguracijo oddaljenega dostopa je potrebno na vrhu vmesnika izbrati `RemoteConnectionCLIENT` in nastaviti parametre. Prav tako je potrebno ustrezno konfigurirati jedro[1].

[1] Več o konfiguraciji je na voljo v dokumentaciji ogrodja - Remote control (core).

4.2.3 Dokumentacija

Za projekt obstaja obsežna dokumentacija, ki se samodejno zgradi ob vsaki novi verziji ogrodja in zatem objavi na spletni strani[1].

Na voljo je tako v spletni (HTML) kot tudi lokalni obliki (PDF), ki se jo lahko prenese tako, da se na spletni strani dokumentacije kurzor premakne na levi spodnji del strani nad verzijo dokumentacije in zatem klikne *PDF*.



Slika 4.18: Prenos PDF dokumentacije

Sphinx

Sistem, uporabljen za grajenje dokumentacije projekta, se imenuje Sphinx. Sphinx je popularno orodje med Python razvijalci za generiranje dokumentacije v več formatih. Razvijalcem omogoča ustvarjanje profesionalne dokumentacije za lastne projekte, kar je nuja pri javnih projektih.

Sphinx omogoča enostavno dokumentiranje z berljivo sintakso (reStructuredText) z veliko funkcionalnostmi, kjer je ena izmed njih možnost branja t. i. *docstring* besedila iz izvorne kode (angl. *source code*) projekta in vključevanje te vsebine v dokumentacijo.

[1] Na voljo na: https://daf.davidhozic.com/en/v2.9.x/.

Je zelo konfigurabilno orodje, kjer se konfiguracijo izvede preko `.py` datoteke, kamor lahko dodajamo tudi svojo Python kodo.

Primarno Sphinx podpira reStructuredText za pisanje dokumentov, podpira pa tudi ostale formate, npr. Markdown preko dodatnih razširitev. Enačbe se lahko piše v jeziku LaTeX.

reStructuredText

reStructuredText je jezik, na katerem deluje Sphinx in je priljubljen *markup* jezik za dokumentacijo projektov.

Znotraj sintakse reStructuredTexta so na voljo različne vloge in direktive, ki se uporabljajo za dodajanje oblikovanja in strukture dokumentom. Vloge se uporabljajo za aplikacijo oblikovanja na določene besede in stavke v isti vrstici, direktive pa so uporabljene za dodajanje nove vsebine v dokument ali za aplikacijo oblikovanja na večvrstično vsebino *[9]*.

Blok kode 4.1: reStructuredText direktiva

```
.. figure:: img/rickroll.png
    :scale: 50%

    Rickroll image

.. math::
    :label: Derivative of an integral with parameter

    \frac{d}{dy}(F(y))=\int^{g_2(y)}_{g_1(y)}f_y dx +
    (f(g_2(y), y)\cdot g_2(y)'{dy} - f(g_1(y), y)\cdot g_1(y)')
```

Blok kode 4.2: reStructuredText vloga

```
:math:`\int 1 dx = x + C`.
If the above isn't hard enough, the
:eq:`Derivative of an integral with parameter`
is a bit harder.
```

Organizacija dokumentacije projekta

Projekt je v celoti dokumentiran s Sphinx sistemom. Na prvem nivoju je dokumentacija razdeljena na:

1. Vodnik - voden opis, kako uporabljati ogrodje,
2. API referenco - opis vseh razredov in funkcij programskega vmesnika, ki jih lahko uporabniki uporabijo v primeru, da pišejo svojo kodo, ki uporablja jedro ogrodja.

Vodnik je pisan ročno v `.rst` datotekah, ki so nastanjene v `/project root/docs/source/guide` mapi. Dodatno se deli še na vodnik za GUI in vodnik za jedro.

API referenca je avtomatično generirana iz komentarjev v izvorni kodi ogrodja in jih dodatno deli pod različne kategorije.

V nekaterih direktorijih so prisotne datoteke `dep_local.json`. To so predgradne konfiguracijske datoteke, ki dajejo informacijo o tem, iz kje in kam naj se kopirajo dodatne datoteke (ki so skupne drugim delom dokumentacije) in katere `.py` skripte naj se izvedejo po kopiranju. Na primer `/project root/docs/source/dep_local.json` datoteka ima sledečo vsebino:

Blok kode 4.3: Predgradna konfiguracijska datoteka

```
{
    "copy": [
        {"from": "../images/*", "to": "./DEP/"},
        {"from": "../../Examples/**", "to":"./DEP/"}
    ],
    "scripts": ["./scripts/generate_autodoc.py"]
}
```

Na podlagi zgornje definicije se bo bodo v ./DEP mape skopirale slike iz nekega zgornjega direktorja. Prav tako se bodo kopirali primeri uporabe jedra ogrodja. Na koncu se bo izvedla skripta `generate_autodoc.py`, ki bo na podlagi *doc_category()* Python dekoraterja generirala `autofunction` in `autoclass` Sphinx direktive, ki bodo ob gradnji dokumentacije prebrale vsebino *docstring*-ov posameznih razredov in funkcij ter jo vstavile v dokument.

`daf.misc.doc.`**`doc_category`**(*cat: str*, *manual: bool | None = False*, *path: str | None = None*, *api_type: Literal['Program', 'HTTP'] = 'Program'*)

Used to mark the object for documentation. Objects marked with this decorator function will have `sphinx.ext.autodoc` directives generated automatically.

Parametri

- **`cat`** (*str*) – The name of the category to put this in.
- **`manual`** (*Optional[bool]*) – Generate `function` directives instead of `autofunction`. Should be used when dealing with overloads.
- **`path`** (*Optional[str]*) – Custom path to the object.
- **`api_type`** (*Literal["Program", "HTTP"]*) – The type of API, the documented item belongs to. Defaults to ‚Program‘

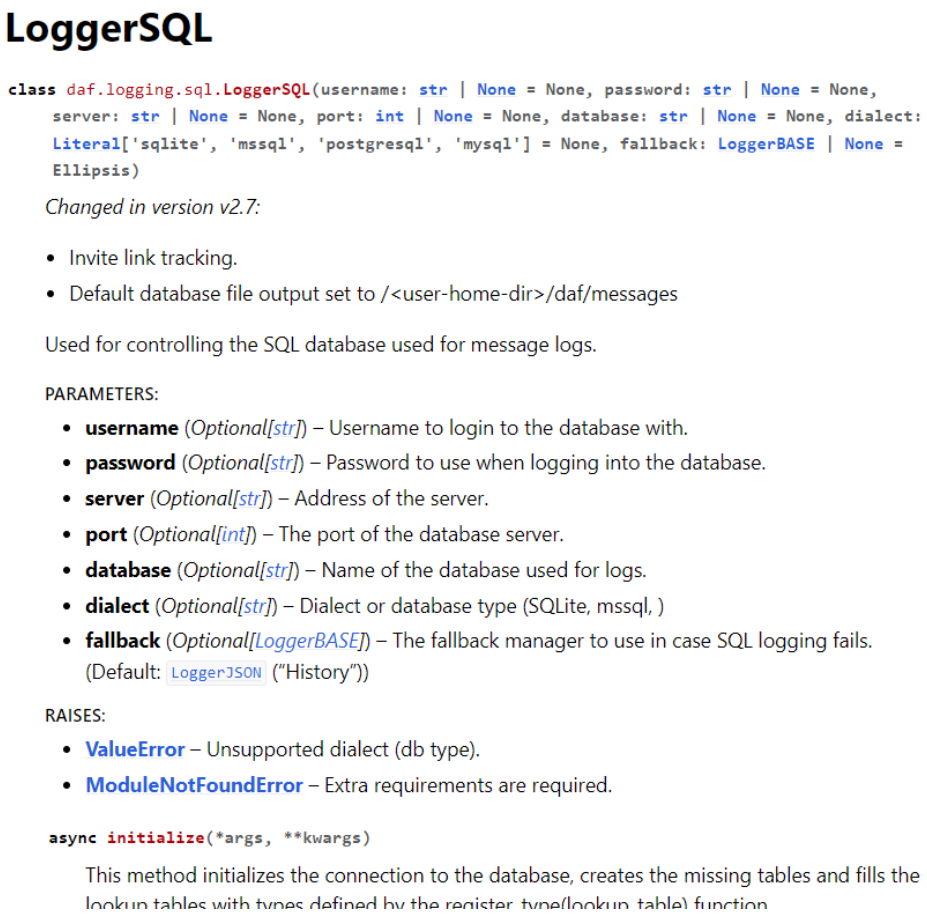
Blok kode 4.4: Uporaba *doc_category()* dekoratorja.

```
@doc.doc_category("Logging reference", path="logging.sql")
class LoggerSQL(logging.LoggerBASE):
    ...
```

Iz *doc_category()* generirane `autofunction`/`autoclass` direktive so del Sphinx-ove vgrajene razširitve `sphinx.ext.autodoc`. Razširitev vključi pakete in izbrska *docstring*-e funkcij in razredov, zatem pa ustvari lep opis o funkciji oz. razredu. V primeru, da je v `autoclass` direktivi uporabljena `:members:` opcija, bo `autodoc` razširitev vključila tudi dokumentirane metode in atribute, ki so del razreda.

Blok kode 4.5: Iz *doc_category()* generirana direktiva

```
.. autoclass:: daf.logging.sql.LoggerSQL
    :members:
```



LoggerSQL

```
class daf.logging.sql.LoggerSQL(username: str | None = None, password: str | None = None,
    server: str | None = None, port: int | None = None, database: str | None = None, dialect:
    Literal['sqlite', 'mssql', 'postgresql', 'mysql'] = None, fallback: LoggerBASE | None =
    Ellipsis)
```

Changed in version v2.7:

- Invite link tracking.
- Default database file output set to /<user-home-dir>/daf/messages

Used for controlling the SQL database used for message logs.

PARAMETERS:

- **username** (*Optional[str]*) – Username to login to the database with.
- **password** (*Optional[str]*) – Password to use when logging into the database.
- **server** (*Optional[str]*) – Address of the server.
- **port** (*Optional[int]*) – The port of the database server.
- **database** (*Optional[str]*) – Name of the database used for logs.
- **dialect** (*Optional[str]*) – Dialect or database type (SQLite, mssql,)
- **fallback** (*Optional[LoggerBASE]*) – The fallback manager to use in case SQL logging fails. (Default: `LoggerJSON` ("History"))

RAISES:

- **ValueError** – Unsupported dialect (db type).
- **ModuleNotFoundError** – Extra requirements are required.

```
async initialize(*args, **kwargs)
```

This method initializes the connection to the database, creates the missing tables and fills the lookup tables with types defined by the register_type(lookup_table) function

Slika 4.19: Rezultat autoclass direktive

Iz Slika 4.19 lahko vidimo, da ima `LoggerSQL` dodatno vsebino, ki je ni imel v `autoclass` direktivi. Ta vsebina je bila vzeta iz same izvorne kode razreda.

Dokumentacija projekta je gostovana na spletni strani Read the Docs (RTD). RTD je spletna platforma za dokumentacijo, ki razvijalcem programske opreme zagotavlja enostaven način za gostovanje, objavljanje in vzdrževanje dokumentacije za njihove projekte. Je odprtokodna platforma, zgrajena na že prej omenjenem Sphinx-u. Poleg gostovanja dokumentacije ponuja RTD tudi nadzor verzij (angl. *version* control) in določeno avtomatizacijo. RTD je za projekt konfiguriran tako, da za vsako izdajo nove verzije projekta avtomatično zgradi dokumentacijo, aktivira verzijo in jo nastavi kot privzeto. Na tak način je dokumentacija pripravljena za uporabo praktično takoj ob izdaji. Prav tako se dokumentacija zgradi ob vsakem zahtevku za združitev vej (angl. Pull request) na GitHub platformi.

4.2.4 Avtomatično testiranje

Za zagotavljanje, da ob novih verzijah projekta ne pride do napak, je za preverjanje delovanja implementirano avtomatično testiranje.

Vsi avtomatični testi so pisani znotraj ogrodja za testiranje z imenom pytest.

pytest - ogrodje za testiranje

Kot že ime namiguje, je pytest ogrodje za testiranje na Python platformi.

Avtomatične teste se pri pytestu implementira s Python funkcijami, katerih ime se začne s „test“. Testi lahko sprejmejo tudi parametre, kjer so ti lahko tudi pritrditve (angl. *fixture*), ki jih lahko lahko uporabimo kot inicializacijske funkcije *[10]*. V pritrditvi lahko npr. povežemo podatkovno bazo, konektor na bazo vrnemo iz pritrditve, in v primeru, da je naš test definiran kot

```
@pytest.mark.asyncio
async def test_example_test(example_fixture):
    ...
```

bo prejel vrednost, ki jo je pritrditev vrnila. Pritrditev ima lahko različno dolgo življenjsko dobo/obseg (npr. globalen obseg, obseg modula, obseg funkcije), kar pomeni, da bo lahko več testov prejelo isto vrednost, ki jo je pritrditev vrnila, dokler se življenjska doba ne izteče. Pritrditev je lahko tudi Python generator[1], kar nam omogoča inicializacijo testov in čiščenje na koncu na sledeč način:

[1] https://wiki.python.org/moin/Generators

Blok kode 4.6: pytest pritrditev z inicializacijo in čiščenjem

```
@pytest_asyncio.fixture(scope="session")
def example_fixture(some_other_fixture):
    # Initialization
    database = DataBaseConnector()
    database.connect("discord.svet.fe.uni-lj.si/api/database")

    yield database  # Value that the tests receive

    # Cleanup
    database.disconnect()
    database.cleanup()
```

Preverjanje, ali je test uspel, se izvede s stavkom `assert`, ki dvigne `AssertionError` napako, če vrednost v `assert` stavku ni enaka `True`. V primeru, da je dvignjen AssertionError, pytest zabeleži test kot neuspel in izpiše napako. Kako podroben bo izpis, se lahko nastavi ob zaganjanju testa, npr. `pytest -vv`, kjer `-vv` nastavi podrobnost. Kot primer si poglejmo, kaj bo izpisal, če v `assert` stavek kot vhod damo primerjavo dveh seznamov.

Blok kode 4.7: Primerjava dveh seznamov, ki nista enaka

```
assert [1, 2, 3] == [1, 2, 3, 4, 5, 6]
```

Iz zgornjega testa je očitno, da to ne drži in da bo test dvignil napako, ampak v assertu nimamo nobenega besedila, da bi se izpisalo, kaj je šlo narobe, tako da bi pričakovali, da pytest vrne samo informacijo, da test ni uspel. Izkaže se, da nam bo pytest izpisal, točno kateri elementi se v seznamu razlikujejo.

Blok kode 4.8: pytest izpis ob neuspelem testu pri primerjavi dveh seznamov.

```
==================== test session starts ====================
platform win32 -- Python 3.8.10, pytest-7.2.0, pluggy
cachedir: .pytest_cache
rootdir: C:\dev\git\discord-advertisement-framework
plugins: asyncio-0.20.3, typeguard-2.13.3
asyncio: mode=strict
collected 1 item

test.py::test_test FAILED                                  [100%]

========================= FAILURES ==========================
_____________________ test_test ____________________

    def test_test():
>       assert [1, 2, 3] == [1, 2, 3, 4, 5, 6]
E       assert [1, 2, 3] == [1, 2, 3, 4, 5, 6]
E         Right contains 3 more items, first extra item: 4
E         Full diff:
E         - [1, 2, 3, 4, 5, 6]
E         + [1, 2, 3]

test.py:6: AssertionError
```

Testiranje ogrodja

Testi so v ogrodju razdeljeni po posameznih nivojih in funkcionalnostih. Skoraj vsi testi delujejo sinhrono, tako da se v testu kliče notranje funkcije posameznih objektov, ki bi jih ogrodje klicalo v primeru navadnega delovanja. Na tak način so izvedeni, saj je testiranje v navadnem (asinhronem) načinu, kjer se vse zgodi v `asyncio` opravilih precej težje, saj bi namreč morali loviti ogrodje ob točno določenih časih, da bi dejansko testirali to, kar želimo. Kljub temu obstajata dva testa, ki ogrodje poženeta v navadnem načinu, in sicer sta to testa, ki testirata, če je perioda pošiljanja prava, in vzporedno preverjata tudi delovanje dinamičnega pridobivanja podatkov. Kot sem že prej omenil, je pri teh dveh testih potrebno uloviti pravi čas, zato se včasih pojavijo problemi z Discordovim omejevanjem hitrosti na API klice, kar lahko povzroči, da bo pri pošiljanju sporočila ovojni API nivo rabil več časa, da naredi zahtevo na API, saj bo čakal, da se omejitev izteče. V tem primeru bo pytest izpisal, da test ni uspel in ga je potrebno ponoviti. Vsi testi se nahajajo v mapi `./testing`, relativno na dom projekta.

Avtomatičnih testov običajno ne zaganjam ročno na osebnem računalniku (razen tistih, ki preverjajo delovanje neke nove funkcionalnosti), temveč se na GitHub platformi avtomatično zaženejo ob vsakem zahtevku za združitev vej (*Pull request*), ko hočem funkcionalnost s stranske GIT veje prenesti na glavno. Dokler se vsi testi ne izvedejo, GitHub ne bo pustil, da se funkcionalnost prenese na glavno vejo.

5 Rezultat in zaključek

V okviru diplomskega dela „Ogrodje za oglaševanje nezamenljivih žetonov po socialnem omrežju Discord“, je bil cilj izdelati ogrodje, ki bi lahko neodvisno oglaševalo v več cehih in pošiljalo različna sporočila, vse na enem samem računu. V planu ni bilo nobene večje razširitve kot oglaševanje NFT, vendar vseeno ogrodje v končni fazi omogoča univerzalno oglaševanje tako NFT kot katerekoli druge vsebine in ne samo periodično, omogoča tudi oglaševanje enkratne vsebine. Poleg vsebine je mogoče definirati tudi začeten čas pošiljanja, število pošiljanj in periodo pošiljanja. Ogrodje omogoča tudi pošiljanje na več računih hkrati, namesto na enem računu, kot je to bilo na začetku predvideno. Odporno je na napake pošiljanja, kot so izbrisani kanali, cehi, pomanjkanje pravic, in se nanje ustrezno odziva, kjer uporabniku preko opozorilnih izpisov omogoča spremljanje delovanja. Za sledenje zgodovine pošiljanj je podprto beleženje poskusov pošiljanj v več formatih, kjer je eden izmed teh tudi SQL beleženje, kar omogoča analitiko oziroma statistiko poslanih sporočil in tudi sledenje pridružnih povezav (angl. *invite link*) za posamezen ceh in s tem spremljanje dosega uporabnikov z določenimi sporočili. Za iskanje novih cehov je na podlagi brskalnika Google Chrome implementirano iskanje in pridruževanje se novim cehom, kjer se za iskanje lahko nastavi poljubne parametre in s tem konfigurira, točno katerim cehom se bo ogrodje pridružilo. Vse zgoraj omenjeno je omogočeno v jedru ogrodja preko Python skripte oziroma konzole, je pa na vrhu izdelan grafičen vmesnik (Slika 7.2) za lažje upravljanje z ogrodjem, definicijo objektov (računov, cehov, sporočil ipd.) ter prikaz statistike in vsebine poslanih sporočil. Omogočen je tudi oddaljen dostop do jedra ogrodja, kjer lahko jedro in grafični vmesnik delujeta samostojno in komunicirata na daljavo preko HTTP vmesnika, kar pomeni, da lahko jedro deluje na nekem oddaljenem strežniku 24/7, grafični vmesnik pa uporabniki zaženejo ob poljubnem času in se potem povežejo na jedro ogrodja. V *prilogah* se nahajajo primeri definicij in uporabe.

Z delovanjem ogrodja sem v končni fazi zadovoljen. Zdi se mi, da sem izpolnil vse želene zahteve, ki jih zahteva samodejno oglaševanje po Discordu, seveda pa se da stvari vedno izboljšati, ne glede na to, kako dobro so narejene. Discord se tudi kar hitro razvija in dodaja nove funkcionalnosti, ki bi se jih dalo v ogrodju še dodatno podpreti (npr. obvestilni kanali). Pri pridruževanju v nove cehe se včasih pojavi CAPTCHA test, ki ga mora rešiti uporabnik sam, in ena izmed izboljšav tu bi bila lahko samodejno reševanje CAPTCHA testov. Prav tako bi se lahko ob vzponu orodij, kot je ChatGPT, izdelalo samodejno odgovarjanje na vprašanja v direktnih/privatnih sporočilih, ki bi jih drugi uporabniki poslali uporabniku, na katerem deluje ogrodje.

Zaključim lahko, da je projekt izjemno uporaben za oglaševanje po Discordu, saj je zaradi Discordove strukture na platformi prisotno ogromno povezanih cehov, kamor lahko oglašujemo in s tem orodjem to lahko počnemo avtomatično in preprosto. Prav tako v času začetka izdelave ogrodja in tudi ob času pisanja tega diplomskega dela ni na vidiku nobenega bolj razširjenega in za uporabo preprostega ogrodja (vsaj ne brezplačnega), kar daje ogrodju še dodatno vrednost.

6 Literatura

[1]

investopedia (RAKESH SHARMA), „Non-Fungible Token (NFT): What It Means and How It Works“, Dostopno: https://www.investopedia.com/non-fungible-tokens-nft-5115211 [Dostopano: 01. 2023]

[2]

NFTing, „What is NFT Shilling“, Dostopno: https://nfting.medium.com/what-is-nft-shilling-192bec98429c [Dostopano: 07. 2023]

[3]

Discord, „Discord Company“, Dostopno: https://discord.com/company [Dostopano: 07. 2023]

[4]

Discord, „Discord Interface“, Dostopno: https://support.discord.com/hc/en-us/categories/200404398 [Dostopano: 01. 2023]

[5]

Discord, „Discord community guidelines“, Dostopno: https://discord.com/guidelines [Dostopano: 07. 2023]

[6]

GPT-3 (Open-AI), „Uporaba GPT za generiranje oglaševalske vsebine“, Dostopno: https://openai.com/blog/gpt-3-apps [Dostopano: 07. 2023]

[7]

Top.GG, „Top.GG webpage“, Dostopno: https://top.gg/ [Dostopano:

07. 2023]

[8]

Python Software Foundation, „tkinter — Python interface to Tcl/Tk“, Dostopno: https://docs.python.org/3/library/tkinter.html#module-tkinter [Dostopano: 05. 2023]

[9]

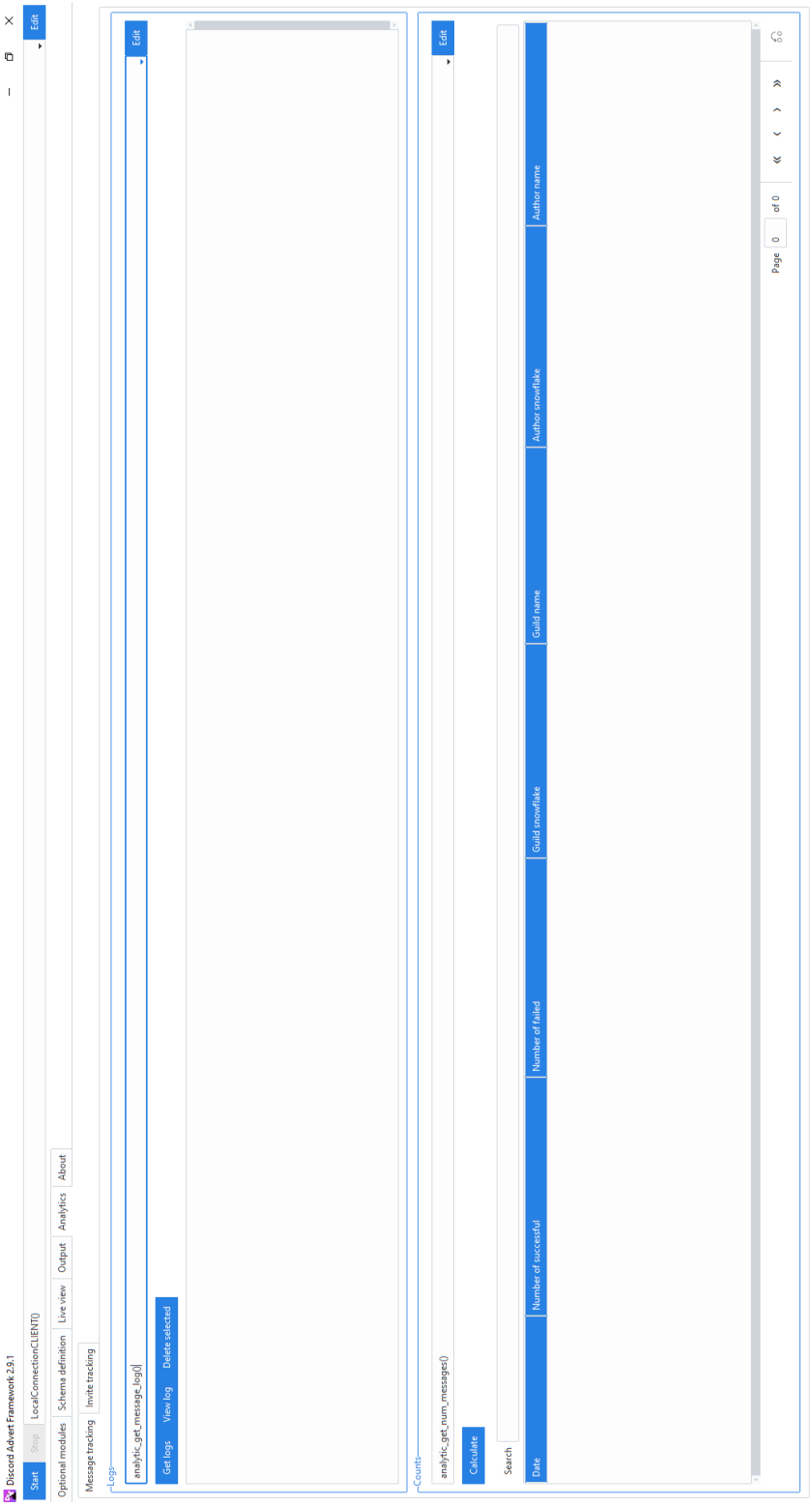
Docutils, „Docutils: Documentation Utilities“, Dostopno: https://docutils.sourceforge.io/ [Dostopano: 01. 2023]

[10]

Shaheer Shahid Malik, „Why Use Pytest to Test Your Software Project?“, Dostopno: https://medium.com/codex/why-use-pytest-to-test-your-software-project-7758ac9970ba [Dostopano: 01. 2023]

7 Priloge

7.1 Dodatne slike



Slika 7.1: Zavihek *Analytics*

Slika 7.2: Grafični vmesnik (*Schema definition* zavihek)

7.2 Primeri konfiguracije ogrodja

7.2.1 Pošiljanje sporočila z naključno periodo

Pošiljanje tekstovnega sporočila s periodo naključno med 1 uro in 2 urama, z začetkom pošiljanja 13.07.2023 00:00:00, kjer se sporočilo pošlje petkrat.

Blok kode 7.1: Pošiljanje sporočila z naključno periodo - jedro ogrodja

```
# Import the necessary items
from daf.logging._logging import LoggerJSON

from daf.message.text_based import TextMESSAGE
from daf.guild import GUILD
from datetime import timedelta
from datetime import datetime
from daf.client import ACCOUNT
from daf.logging.tracing import TraceLEVELS
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\David\\daf\\History",
)

# Define remote control context


# Defined accounts
accounts = [
    ACCOUNT(
        token="OTMHH72GFA7213JSDH2131HJb",
        is_user=True,
```

```
        servers=[
            GUILD(
                snowflake=123456789,
                messages=[
                    TextMESSAGE(
                        start_period=timedelta(
                            hours=1.0,
                        ),
                        end_period=timedelta(
                            hours=2.0,
                        ),
                        data="We are excited to announce the␣
↪launch of our White Rabbit NFT project!",
                        channels=[
                            231321312312312312312123,
                            98766523123124312323323277,
                        ],
                        start_in=datetime(
                            year=2023,
                            month=7,
                            day=13,
                        ),
                        remove_after=5,
                    ),
                ],
            ),
        ],
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
```

```
    debug=TraceLEVELS.NORMAL,
    save_to_file=False
)
```

Blok kode 7.2: Pošiljanje sporočila z naključno periodo - GUI shema

```
{
  "loggers": {
    "all": [
      {
        "type": "daf.logging._logging.LoggerJSON",
        "data": {
          "path": "C:\\Users\\David\\daf\\History"
        }
      },
      {
        "type": "daf.logging.sql.mgr.LoggerSQL",
        "data": {
          "database": "C:\\Users\\David\\daf\\messages",
          "dialect": "sqlite"
        }
      },
      {
        "type": "daf.logging._logging.LoggerCSV",
        "data": {
          "path": "C:\\Users\\David\\daf\\History",
          "delimiter": ";"
        }
      }
    ],
    "selected_index": 0
  },
  "tracing": 3,
```

```
 "accounts": [
   {
     "type": "daf.client.ACCOUNT",
     "data": {
       "token": "OTMHH72GFA7213JSDH2131HJb",
       "is_user": true,
       "servers": [
         {
           "type": "daf.guild.GUILD",
           "data": {
             "snowflake": 123456789,
             "messages": [
               {
                 "type": "daf.message.text_based.TextMESSAGE",
                 "data": {
                   "start_period": {
                     "type": "datetime.timedelta",
                     "data": {
                       "hours": 1.0
                     }
                   },
                   "end_period": {
                     "type": "datetime.timedelta",
                     "data": {
                       "hours": 2.0
                     }
                   },
                   "data": "We are excited to announce the␣
↪launch of our White Rabbit NFT project!",
                   "channels": [
                     231321312312312312312123,
                     987665231231243123232327 7
                   ],
                   "start_in": {
```

```
                    "type": "datetime.datetime",
                    "data": {
                      "year": 2023,
                      "month": 7,
                      "day": 13
                    }
                  },
                  "remove_after": 5
                }
              }
            ]
          }
        }
      ]
    }
  ],
  "connection": {
    "all": [
      {
        "type": "daf_gui.connector.LocalConnectionCLIENT",
        "data": {}
      },
      {
        "type": "daf_gui.connector.RemoteConnectionCLIENT",
        "data": {
          "host": "http://"
        }
      }
    ],
    "selected_index": 0
  }
}
```

7.2.2 Avtomatsko odkrivanje (angl. *discovery*) cehov in kanalov

Pošiljanje s fiksno periodo dveh ur in avtomatično odkrivanje pridruženih cehov in kanalov na podlagi RegEx vzorca.

Blok kode 7.3: Avtomatsko odkrivanje cehov in kanalov - jedro ogrodja

```
# Import the necessary items
from daf.logging._logging import LoggerJSON

from daf.message.text_based import TextMESSAGE
from daf.guild import AutoGUILD
from daf.message.base import AutoCHANNEL
from datetime import timedelta
from daf.client import ACCOUNT
from daf.logging.tracing import TraceLEVELS
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\David\\daf\\History",
)

# Define remote control context


# Defined accounts
accounts = [
    ACCOUNT(
        token="OTMHH72GFA7213JSDH2131HJb",
        is_user=True,
        servers=[
            AutoGUILD(
```

```
                include_pattern=".*",
                exclude_pattern="David's Dungeon",
                messages=[
                    TextMESSAGE(
                        start_period=None,
                        end_period=timedelta(
                            hours=2.0,
                        ),
                        data=[
                            "We are excited to announce...!",
                        ],
                        channels=AutoCHANNEL(
                            include_pattern=
↪"shill|advert|promo|projects",
                            exclude_pattern="vanilla-
↪projects|ssfe-obvestila",
                        ),
                    ),
                ],
                logging=True,
            ),
        ],
    ),
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL,
    save_to_file=False
)
```

Blok kode 7.4: Avtomatsko odkrivanje cehov in kanalov - GUI shema

```
{
  "loggers": {
    "all": [
      {
        "type": "daf.logging._logging.LoggerJSON",
        "data": {
          "path": "C:\\Users\\David\\daf\\History"
        }
      },
      {
        "type": "daf.logging.sql.mgr.LoggerSQL",
        "data": {
          "database": "C:\\Users\\David\\daf\\messages",
          "dialect": "sqlite"
        }
      },
      {
        "type": "daf.logging._logging.LoggerCSV",
        "data": {
          "path": "C:\\Users\\David\\daf\\History",
          "delimiter": ";"
        }
      }
    ],
    "selected_index": 0
  },
  "tracing": 3,
  "accounts": [
    {
      "type": "daf.client.ACCOUNT",
      "data": {
```

```
        "token": "OTMHH72GFA7213JSDH2131HJb",
        "is_user": true,
        "servers": [
          {
            "type": "daf.guild.AutoGUILD",
            "data": {
              "include_pattern": ".*",
              "exclude_pattern": "David's Dungeon",
              "messages": [
                {
                  "type": "daf.message.text_based.TextMESSAGE",
                  "data": {
                    "start_period": null,
                    "end_period": {
                      "type": "datetime.timedelta",
                      "data": {
                        "hours": 2.0
                      }
                    },
                    "data": [
                      "We are excited to announce the launch of␣
↪our White Rabbit NFT project!"
                    ],
                    "channels": {
                      "type": "daf.message.base.AutoCHANNEL",
                      "data": {
                        "include_pattern":
↪"shill|advert|promo|projects",
                        "exclude_pattern": "vanilla-
↪projects|ssfe-obvestila"
                      }
                    }
                  }
                }
```

```
            ],
            "logging": true
          }
        }
      ]
    }
  },
  "connection": {
    "all": [
      {
        "type": "daf_gui.connector.LocalConnectionCLIENT",
        "data": {}
      },
      {
        "type": "daf_gui.connector.RemoteConnectionCLIENT",
        "data": {
          "host": "http://"
        }
      }
    ],
    "selected_index": 0
  }
}
```

7.2.3 Oddaljen dostop

HTTP strežnik in GUI shema za povezovanje na ta strežnik.

Blok kode 7.5: Oddaljen dostop - jedro ogrodja

```
# Import the necessary items
from daf.logging._logging import LoggerJSON
from daf.remote import RemoteAccessCLIENT
```

```
from daf.logging.tracing import TraceLEVELS
import daf

# Define the logger
logger = LoggerJSON(
    path="C:\\Users\\David\\daf\\History",
)

# Define remote control context
remote_client = RemoteAccessCLIENT(
    host="0.0.0.0",
    port=80,
    username="ime",
    password="geslo",
)

# Defined accounts
accounts = [
]

# Run the framework (blocking)
daf.run(
    accounts=accounts,
    logger=logger,
    debug=TraceLEVELS.NORMAL,
    remote_client=remote_client,
    save_to_file=False
)
```

Blok kode 7.6: Oddaljen dostop - GUI shema

```
{
  "loggers": {
```

```
  "all": [
    {
      "type": "daf.logging._logging.LoggerJSON",
      "data": {
        "path": "C:\\Users\\David\\daf\\History"
      }
    },
    {
      "type": "daf.logging.sql.mgr.LoggerSQL",
      "data": {
        "database": "C:\\Users\\David\\daf\\messages",
        "dialect": "sqlite"
      }
    },
    {
      "type": "daf.logging._logging.LoggerCSV",
      "data": {
        "path": "C:\\Users\\David\\daf\\History",
        "delimiter": ";"
      }
    }
  ],
  "selected_index": 0
},
"tracing": 3,
"accounts": [],
"connection": {
  "all": [
    {
      "type": "daf_gui.connector.LocalConnectionCLIENT",
      "data": {}
    },
    {
      "type": "daf_gui.connector.RemoteConnectionCLIENT",
```

```
        "data": {
          "host": "https://discord.svet.fe.uni-lj.si",
          "username": "ime",
          "password": "geslo"
        }
      }
    ],
    "selected_index": 1
  }
}
```