FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Deep Town Guild Support System
Student:	Daniel Hampl
Supervisor:	Ing. Marek Suchánek
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2019/20

Instructions

Based on a conceptual-model based analysis, design and implement a modular system using web services and apps to support guild activities within the mobile game Deep Town. The system must be able to gather and appropriately transform necessary data from the game, store it using a well-defined data model and provide it to other services using documented and secured APIs. The system will then contain at least two such services for a presentation of stored data:
- Web application with responsive design (for desktops as well as mobile devices) with guild search and details, leaderboards, and other useful content.
- Discord chatbot that allows to query information directly in chat.
Substantiate your choice of used technology (libraries, frameworks, etc.). Test the system using unit and integration tests. Document your system to allow community development of new applications for your system. In conclusion, evaluate the benefits of your solution in comparison to similar existing solutions.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague November 25, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Deep Town Guild Support System

Daniel Hampl

Department of Software Engineering

Supervisor: Ing. Marek Suchánek

May 12, 2019

Acknowledgements

First of all, I would like to thank my supervisor Ing. Marek Suchánek, who has been here for me, day and night, seven days a week, answering my questions and providing me with vital information and sources, guiding me through my work on this thesis. I would also like to thank Rockbite and especially Davit Yeghiazaryan for granting me the means to create this project.

Declaration

In Prague on May 12, 2019 . . . .

Citation of this thesis

Hampl, Daniel. *Deep Town Guild Support System.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato bakalářská práce popisuje management gild v mobilní hře Deep Town a analyzuje nynější způsoby sběru dat o hráčích z této hry. Následně je navrhnut efektivnější způsob sběru dat, který je poté využit pro tvorbu modulárního systému, který sbírá, ukládá a dodává zmíněná data uživatelům. Nakonec je tento systém otestován pomocí unit a integračních testů pro zajištění jeho stability.

Klíčová slova DeepTown, Deep Town, Deep Town Admin Tools, DTAT, DeepTownAdminTools, hra, data, API, mobil, Python, Discord bot

Abstract

This thesis describes guild management in the mobile game Deep Town and analyses current ways of gathering data about players from this game. Afterwards, a more efficient way to gather said data is designed, and it is then used to form a modular system, which gathers, stores and provides said data to users. In the end, the system is tested using the unit and integration tests, thus ensuring its stability.

Keywords DeepTown, Deep Town, Deep Town Admin Tools, DTAT, DeepTownAdminTools, game, data, API, phone, Python, Discord bot

Contents

List of Figures

Introduction

I have been playing a mobile game called Deep Town for over two years now. It is a game, where players mine resources, which are then used to build an infrastructure. After a player builds a sufficient infrastructure, they can choose to join a guild, and participate in various guild events. To participate in these events, players need to gather a specific amount of various items, for which is the guild rewarded.

In order to maximise the guild efficiency guild leaders need to know the processing potential of each guild member. It is possible to view player details, but only a single player at a time, without any way of copying the data apart from taking a picture, or rewriting the data shown. Thus, there is no comprehensive way to view the entire guild's information at once without wasting a considerable amount of time.

Most guilds also make a periodic participation leaderboard, for which someone needs to rewrite player nicknames and their donation score. Afterwards, it needs to be compared with the previous time this was done, as players can only view donations since the start of the game. Thus making it quite time-consuming and dull job, which hardly anybody would want to do.

Over the time I spent playing this game, I became an administrator of one of the top guilds, and I have personally met with these problems. Thus, I have decided to choose this topic for my Bachelor thesis, and make a comprehensive tool, allowing players to gain easier access to information like that.

Goals

The goal of this thesis is to make an easily extensible modular system, that allows players to view information from game Deep Town. This system has to be thoroughly tested to guarantee system integrity and ease of future development.

This system will have two options for the representation of acquired data. The first one will be a responsive website, allowing access to necessary data through a mobile phone or PC. The second option will be a Discord bot, as most of the guild organisation in this game is primarily through this service (Discord).

This system should also be well documented to allow smooth future development. There should also be some comprehensive documentation within the Discord bot to allow its use without any hindrance.

Structure

This thesis follows the traditional software engineering approach. In the beginning, Chapter 1 will be focussing on the analysis of Deep Town itself, followed by current solutions related to our problem as well as similar cases in other games. Afterwards, the analysis chapter will conclude with a description of the functional and non-functional requirements for our system.

Chapter 2 will be a follow up on the requirements, that have arisen from the analysis, evaluating the advantages and disadvantages of different technologies needed.

Chapter 3 will be concerning the system design starting with the system architecture, followed by GUI and database, and concluded by the business logic.

Chapter 4 will describe the implementation of the whole system including the choice of frameworks and technologies based on the second chapter.

Chapter 5 will describe the testing of the whole system with unit and integration tests.

In the end, this thesis will be concluded by evaluating the completion of set goals and listing options for future improvement, that arose during development.

Chapter 1

Analysis

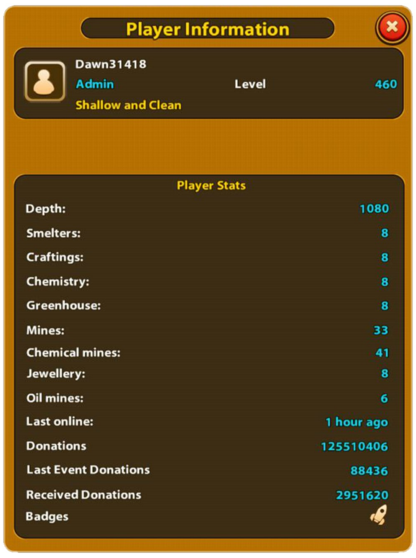
1.1 Deep Town

Deep Town is a mobile game, where every player acts as AI (artificial intelligence), whose purpose of existence is resource gathering, construction and replication. At the start, each player appears on a planet isolated from everyone else. Afterwards, they need to gather resources and build essential infrastructure, in order to be able to communicate with other AIs (players). Afterwards, they gain the ability to join or form a guild.

Each guild is a collective of up to 50 players, one of them being admin and in most cases few moderators selected by the admin. Players in one guild can request and donate resources to each other. That is done by one player requesting a specific amount of resources and others within the guild can then donate to reach the set goal. Every week there is also a guild event which requires players to donate four types of items in order to gain experience points and various other rewards.

Guild leaders usually want to monitor their players' activity, which can be done using the player information card [Figure 1.1], where total donations and last time a player was online are listed among other things. Recently even the last event donations were added, which saved much time to many guild leaders, but there is still no easy way to store the displayed data, which leads to players rewriting the displayed data to an excel sheet, in order to have a notion about the overall player activity. What makes it even harder is the fact, that every player's data are displayed separately, thus when anyone wants to copy the last event's donations, they would have to display each player, copy a number, and then move on to another player.

Most of the top guilds also monitor the processing capability of each member in order to set achievable goals for each guild event and to effectively distribute labour. That is either done by rewriting building levels of each player to some



Player Information

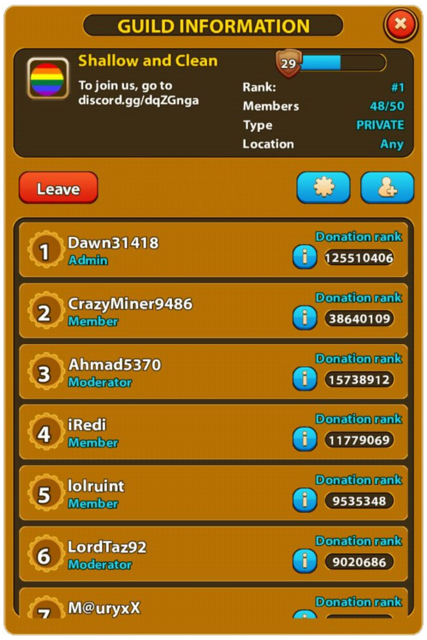
Dawn31418
Admin
Shallow and Clean

Level	460

Player Stats

Depth:	1080
Smelters:	8
Craftings:	8
Chemistry:	8
Greenhouse:	8
Mines:	33
Chemical mines:	41
Jewellery:	8
Oil mines:	6
Last online:	1 hour ago
Donations	125510406
Last Event Donations	88436
Received Donations	2951620
Badges	

Figure 1.1: Player information



GUILD INFORMATION

Shallow and Clean
29
To join us, go to discord.gg/dqZGnga

Rank:	#1
Members	48/50
Type	PRIVATE
Location	Any

Leave

1	Dawn31418	Admin	Donation rank 125510406
2	CrazyMiner9486	Member	Donation rank 38640109
3	Ahmad5370	Moderator	Donation rank 15738912
4	iRedi	Member	Donation rank 11779069
5	lolruint	Member	Donation rank 9535348
6	LordTaz92	Moderator	Donation rank 9020686
7	M@uryxX		Donation rank

Figure 1.2: Player list

excel sheet or having the members fill it in, which could be an aggravating task, as the building levels change and, the sheet needs to be updated.

1.2 Current solutions

Currently, there are two ways people are getting donation data from Deep Town. First is rewriting the data either from a game screen directly or from its screenshots, since it takes a considerable amount of time to rewrite it all, and it is better to take the records at the same time for every player to ensure data precision. That can be done either from the player information card [Figure 1.1] or directly from the player list [Figure 1.2], where only overall donations and names are listed. The second option is sniffing of the game API which requires a fair bit of knowledge regarding IT thus being the less used option among the guilds.

For gathering rest of the player data, guilds usually rewrite the data from the game screen, either having a single person do it, or forcing every member to fill in their data. In some cases, guilds also used sniffing the game API as the means to gather player data, but this was less common than in the case of gathering only donations. Most likely the reason for that is the fact, that the game API does not show building levels while listing all players. Therefore, it is needed to list details for each player separately.

When we take a look at data gathering in other games, for example at League of Legends or Defense of the Ancients otherwise known as LOL or DOTA respectively, they provide a public API in order to offer their community access to all the interesting data they store. Many sites, for example Lolskill [1] or Dotabuff [2], run on these public APIs providing access to match history, player information and lots of other useful data.

In the case of browser games, where there is no public API, for example, Ikariam the data gathering can be done using JavaScript document object model. This method was used in the Ikalogs project, where authors used a chrome plugin to gather player's data in order to store them in their database and later on display on their website. [3]

1.3 Requirements

1.3.1 Functional

The system needs to offer the option to search for a guild and display its players and their information containing their name, level and building levels. There also has to be an option to show donations given/received over a set period. Especially during guild events, as these donations are vital to the organisation of every guild.

As most of the guilds organise on Discord, there has to be a Discord bot providing access to the stored data. The same access has to be ensured even with the website, which has to be another access point, in order to offer the functionality to those guilds that do not organise on Discord.

1.3.2 Non-functional

Since Deep Town is a mobile game, the website needs to be responsive to ensure good usability on mobile devices as well as desktops.

The whole system needs to be well secured to ensure server stability and trustworthiness of the provided service. Moreover, to ensure software quality, this system needs to be well tested, documented, and it needs to be written following appropriate conventions.

Chapter 2

Technology

As part of our system, we will need a database to store data, a server to access the database and provide data to two front end interfaces website and Discord bot.

2.1 Database

First, let us take a look at the comparison of traditional SQL databases and their NoSQL counterparts. This comparison will be based on an ACM article ”Comparing NoSQL MongoDB to an SQL DB” [4], where authors base their comparison on relatively small databases. In the aforementioned article, authors are using probably the most widely known NoSQL database MongoDB and Microsoft SQL Server Express database as a representative of SQL databases.

In conclusion of their research, the authors found, both databases indifferent in speed while inserting data. While updating database data using the primary key was much faster on the MongoDB, than the SQL database, this being most likely due to default indexing of the primary key in MongoDB. On the other hand, updating using non-primary keys such as string formatted name the SQL database outperformed the NoSQL database by a similar margin. If we take a look at more complex queries, MongoDB was much faster than its SQL counterpart up to a point, where results could not even be displayed in one graph.

However, while using an aggregate function (average), the SQL database outperformed MongoDB by about 23 times. MongoDB also has other strong sides, such as high scalability, not requiring a rigid schema as SQL databases do and the option to be easily turned into a distributed database, but NoSQL databases are still new and cannot compare to the support, that most SQL databases have as SQL can still be considered an industry standard.

Let us take a look at two representatives of SQL databases. They are both free, open-source and widely used, thus offering ample support from their respective communities and ensuring prompt patch for any potential bugs. The first one being a small and simple SQLite database and the second one being rather larger and more complex PostgreSQL database. According to an article published on DigitalOcean talking, among other things, about these two databases [5], the SQLite database is often used for testing or as a disk access replacement, it is quite fast but can only be accessed for writing once at a time, which is caused by direct access to system storage. Hence it is more suitable for smaller, less complex systems, testing and development.

On the other hand, PostgreSQL can offer multiple writing sessions at once, thanks to the server process encompassing the data storage. It offers permission management unlike the SQLite, which is only a file on system storage. It also offers a wide variety of data types including binary encoded JSON or universally unique identifier. The PostgreSQL is also slightly harder to set up and more demanding on memory, as it allocates 10MB for each request session.

2.2 Server

In this section, I will be evaluating different programming languages, that might be used for building a web server. Out of all of the programming languages at my disposal, I have chosen C++, Java, Python and NodeJS as candidates to be used in this system. I have chosen these languages mainly for their large user base, which will most likely come in hand while resolving any problems encountered during the implementation phase.

2.2.1 C++

Personally, I am fond of C and C++ since it allows the programmer to decide everything for themselves. Fully understand their code, and the inner workings of the final program, where the memory is stored, when it is cleared, how it is cleared and the form in which it is stored. However, this might not always be an advantage, as it can be an arduous task, to have to take care of every minor part of your system yourself, as opposed to letting the compiler or interpreter take care of cleaning memory and other miscellaneous matters.

According to Martin Reddy in his book API Designs for C++ [6], most web services use scripting languages like PHP, Perl or Python, however, if we take a look at larger scaled web services, like Facebook, C++ is used to achieve higher performance with the use of tool called HipHop, which converts PHP code into C++.

Thus, in conclusion, C++ is more useful, when used for larger systems, where performance is vital, and the speed of delivery is less important.

2.2.2 Java

Java is a higher levelled programming language if compared to C++, it still allows the programmer to control a lot of its behaviour, but it offers the option to take care of many miscellaneous matters, for example cleaning memory with the garbage collector. It also offers various integrated development environments, such as Eclipse or Netbeans providing the option to autogenerate chunks of code, and thus saving much time for the programmer using it.

On the other hand, according to L. Prechtel and his empirical comparison [7], Java has a higher overall memory consumption and lower performance, which is caused by it running on top of a virtual machine.

Therefore, it could be said that Java is an excellent middle ground for developing web services, but Java still lacks, if precise memory manipulation or highest performance is needed.

2.2.3 Python

Python is a high-level programming language, offering dynamic typing, and thus relatively easy to use and often recommended as a first programming language to learn. According to D. Sarkar [8], Python cannot achieve the same performance efficiency as C or C++. However, it can match up to Java (or even outperform it) [7], especially with the use of various optimisations Python offers.

Python also has a wide opensource community offering over 80,000 packages, which you can freely install and use, thus offering a tool for almost anything you might need.

On the other hand, Python also has a few vital flaws. One of them is the Global Interpreter Lock, otherwise know as GIL, which ensures, that each instance of Python can run only one thread at once, even on multi-core processors. That can be resolved by running more instances of Python for example with the use of Apache server, to manage these multiple instances.

Another flaw is the incompatibility between versions, not only 2.7.x and 3.x but even between various 3.x Python versions, where once flawless code might stop working after updating to a newer version of Python.

In conclusion, Python is an excellent choice for any system, where the goal is the speed of development with reasonable performance and possible future optimisations.

2.2.4 JavaScript

JavaScript has been according to GitHub the most used programming language for the last five years [9], thus offering high community support and a wide variety of packages. Using JavaScript for server-side development also brings the benefit of having the same programming language on the front-end as well as the back-end. [10]

Another great thing about JavaScript is the native support of JSON formatted data, which is one of the most widely used formats for transferring data through web services. As such it can save time and memory while dealing with web services.

On the other hand, unlike Python or many other interpreted languages for that matter, JavaScript can only run a single thread in every instance at a given time, thus needing to create multiple instances, to be able to use a multi-core processor to its limit, such as web workers. [11]

In conclusion, JavaScript is a great tool, for developing web services offering the option to have one team of developers with expertise in JavaScript developing both, front-end as well as back-end. It can offer fast development, and reasonable performance, thus being a great candidate for building various systems.

2.3 Website

There are many options for developing a website, which is connected to some sort of back-end server. These options are highly dependent on the technology used on the back-end server, for example, if the server is built using Java, then Java framework Vaadin might be useful, as it efficiently communicates with the rest of the system.

The front-end of the system can also be made traditionally using HTML, CSS and JavaScript, which is one of the most common approaches for small websites. [12]

In larger projects, websites are often developed with the use of various JavaScript frameworks, such as Angular, or React, often offering cleaner and smoother approach.

2.4 Discord bot

When we take a look at options for developing a Discord bot, we can find a framework of some quality, for pretty much any programming language commonly used for development, from C++ to Pharo.

From all these options I chose JavaScript and Python frameworks, as they are maintained up to date, thanks to wide communities both these languages have. Both of these frameworks offer approximately the same pool of functions. Hence, the difference between them is more about user's preference, than anything else. [13, 14]

Chapter 3

Design

After careful analysis of current solutions within Deep Town as well as other games, I was left with two viable options. The first one was, using the private API revealed through sniffing the game's traffic. However, this option was rejected after further analysis, as it leads to a ban of the account being used.

The second viable option was to try and go through the official channels and negotiate access to their API. After some negotiating with Rockbite (the company developing Deep Town) decided to create new public API in support of my project, which led to the following design.

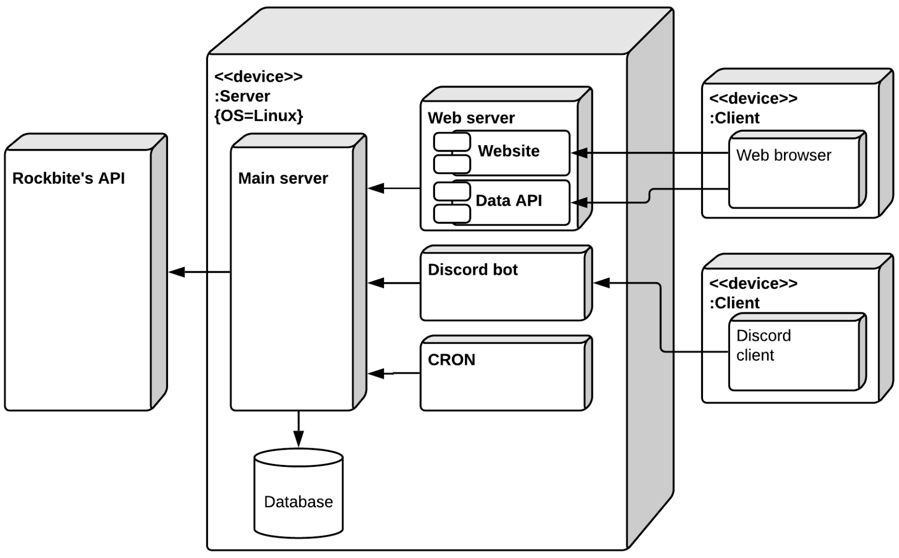


Figure 3.1: Architecture model

3.1 Architecture

The whole system will be comprised of 7 parts (as shown in Figure 3.1), the Rockbite's API, central server, database, web server, website, Discord bot and a CRON script. I have decided on this layout, as it will offer easy extensibility and interchangeability of all the system parts.

3.2 Rockbite's API

The Rockbite's API will be the cornerstone of this project, as it will supply the entire system with data. As of now, it has two methods. The first one [Listing 3.1], providing a list of guilds matching a string, and the second one [Listing 3.2], listing guild data, including a list of players and their information.

```json
{
    "status": "ok",
    "result": [
        {
            "guild_id": "58a28c884c1b9c33a06c38f0",
            "guild_name": "DigDeep",
            "level": 0
        },
        {
            "guild_id": "58ae885b134e2d6900f6aae9",
            "guild_name": "Deep Town",
            "level": 0
        },
        ...
    ],
    "server_time": "2019-04-06T20:22:39.979Z"
}
```

Listing 3.1: Rockbite's API – find guild

```
    {
    "status": "ok",
    "result": {
        "guild_id": "5919a8e24459ba06a0cfd66d",
        "guild_name": "Shallow and Clean",
        "guild_level": 29,
        "total_donations": 275660697.09599996,
        "total_level": 21062,
        "average_level": 438.7916666666667,
        "members": [
            {
                "user_id": "1002727971483461712O0",
                "user_name": "Feo",
                "donations": 882671,
                "received_donation": 654362,
                "last_event_donation": 0,
                "last_online": "2019-04-04T03:06:
                   01.040Z",
                "level": 459,
                "depth": 1080,
                "smelters_count": 8,
                "crafters_count": 8,
                "miners_count": 25,
                "oil_building_count": 2,
                "chemistry_mining_station_count": 12,
                "green_house_building_slot_count": 8,
                "chemistry_building_slot_count": 8,
                "jewellery_building_slot_count": 8
            },
            {
                "user_id": "G:800869535",
                ...
            },
            ...
        ]
    },
    "server_time": "2019-04-04T13:01:21.327Z"
}
```

Listing 3.2: Rockbite's API – guild by ID

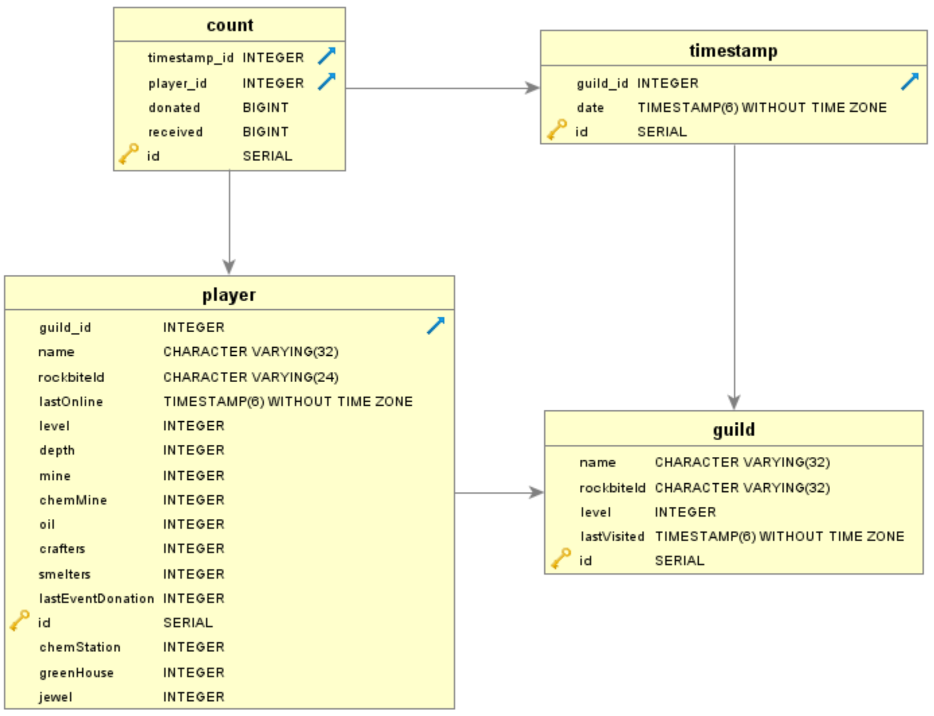


Figure 3.2: Database model

3.3 Database

The system database will need to be able to efficiently store all the data provided by Rockbite's API. It will have four tables namely `guild`, `player`, `timestamp` and `count`. All of these tables will have an integer with name `id` as a private key. A signed integer will be enough for over 11 years even even while using Rockbite's API up to its limit. [1]

3.3.1 Guild

The guild table will include a `name`, `rockbite_id`, `level` and `timestamp`. The guild name will be a maximum of 32 characters in length. Furthermore, it cannot be made unique, as multiple guilds in Deep Town can have the same name.

The `rockbite_id` will be a unique string of a maximum length of 32 characters. It will store the id provided by Rockbite to ensure uniqueness of every guild, which seems to be a string formatted hexadecimal 12-byte number. However, as IDs provided by Rockbite does not seem too reliable, as in the case of

[1] $$\frac{\text{max integer value}}{\text{max requests per day} \times \text{max players in guild} \times \text{days in year}} = \text{max years of use}$$
$$\frac{2147483647}{10000 \times 50 \times 365} = 11.77$$

a player they appear in multiple formats, thus I have decided to use string, which can be used to store almost anything, that might appear.

For storing the information about guild level usual integer will be used. As for the `timestamp`, it will store a date and time in the coordinated universal time zone. This information will later be used to discern, which guild is being used and thus needs to be updated.

3.3.2 Player

The player table will store information about players. It will have a foreign key, `guild_id`, forming a relationship of a player being a member of a guild.

The player table will also contain a player name, which multiple players can share. This name will be stored as a string with a maximum length of 32 characters. Since the player name is not unique, we will also need to store the id provided by Rockbite, which is provided as a string with multiple formats, thus raising the need for new id, mentioned at the start of this section.

Rest of the information provided by the Rockbite's API will be stored as shown in [Figure 3.2], the player `level`, `depth`, last events donations and levels of various buildings as an integer type and the information about players last online status as a timestamp in the UTC timezone.

3.3.3 Donations

To be able to store information about donations over a period we need to know total donations at the start and the and of said period. Each guild also has between 1 and 50 players. Thus I have decided to split the date and donation values of each player. Therefore being able to store the information about donations without wasting space.

The timestamp will have the aforementioned `id`, `guild_id` and `date`. The `guild_id` will be a foreign key, specifying the relationship between `timestamp` and `guild`, thus describing, which guild the concrete set of donations belongs to, and allowing faster querying. The `date` parameter will be the usual timestamp in the UTC timezone storing the time, at which the donation snapshot was taken.

The count table will store each player's donation values at the given time. These values will be stored as big integers, as some the players have reached the limit of integer in this regard. This table will also have two foreign keys. The first one being the `timestamp_id`, which will form a relationship between the `count` of a specific player and a `timestamp`. The second foreign key being the `player_id`, describing a relationship between the `count` and a `player`.

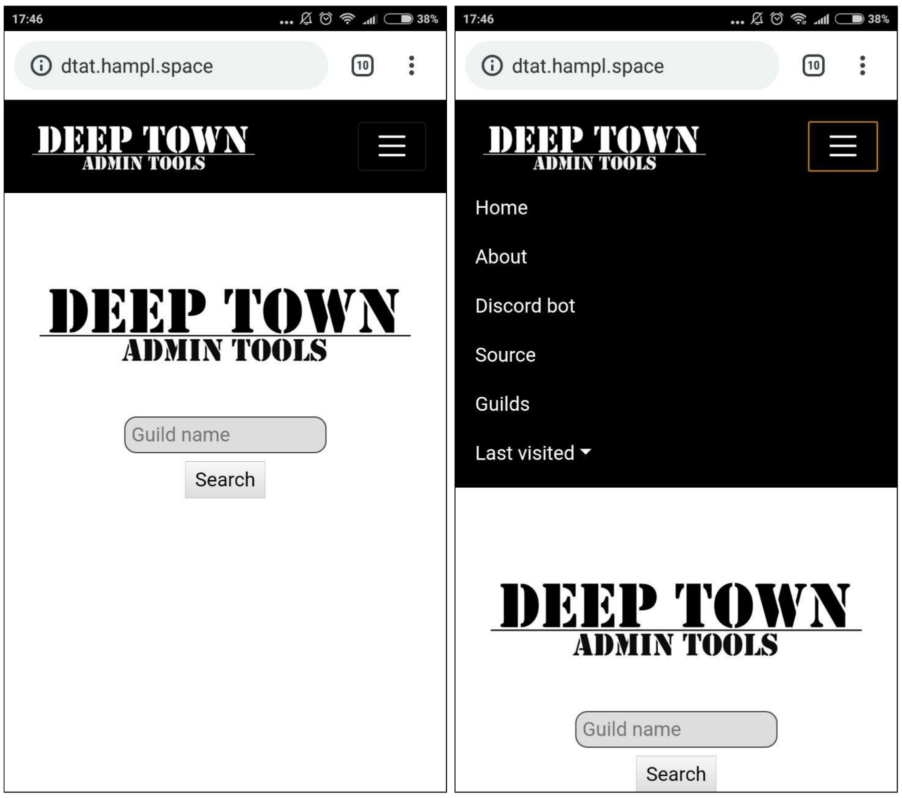


Figure 3.3: Home page – phone

3.4 User Interface

There will be two user interfaces. The first one is a responsive website for mobile devices as well as desktop computers with high resolution, providing access through a web browser. The second one is a Discord chatbot, which offers interaction and access to necessary data through the Discord client.

3.4.1 Website

Instead of wireframes, I have decided to create a prototype for the website and afterwards fill it with data using the public API, thus attached pictures were taken from the beta version of this new system.

All pages on this website will share the same navbar. With a click on the logo or Home link, the user will be redirected to the home page. In case of click on the about section, the user will be sent to a page with short introduction and disclaimer required by Rockbite, stating, that this is not an official project of theirs. The Discord bot link will lead to a page, where users can invite the DTAT (Deep Town Admin Tools) Discord bot to their Discord server.

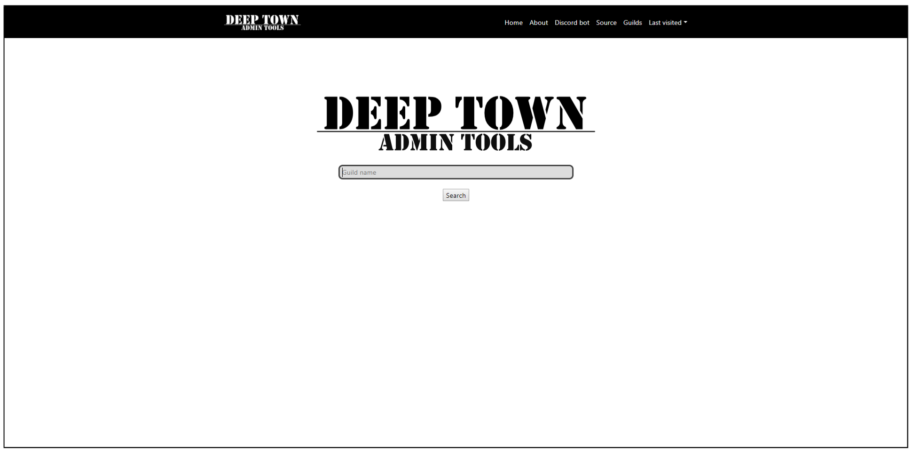


Figure 3.4: Home page – desktop

The source link will be pointing to the DTAT organisation on GitHub, where users can post issues or extend the system with their content. The guilds link will lead to the page, where all guilds within the system database will be listed. In case, that said user already visited one or more guilds on the current device the navigation bar will also include a last visited section, which will show a list of up to 8 guilds, to allow the user easier access.

Upon entering the homepage, the user focus will be moved to the search bar in the centre of the screen, which will allow the user to instantly search for their desired guild with either press of the search button or the enter key on users keyboard. The user can either decide to enter a string matching a guild or search without entering any. In the former case, guilds will be filtered, to match the said string and in the latter case, all guilds will be displayed in the list.

Once the user arrives at the guilds page, there will be a table of guilds with their names and level. The user can decide to list 10, 25, 50 or 100 rows at any given time and to search using name or level. This table can also be sorted in either descending or ascending order using both level and name. Once the user found their desired guild, they can access its detailed information with a click at the table row, which will redirect them to the guild page.

On the guild page, a secondary navigation bar appears. It will include options, to redirect to the donation page or export guild data to a CSV format, as some users still want to keep their guild's data in excel sheet for various purposes. Under the secondary navigation bar will be a guild name followed with a list of columns allowing to toggle said columns in order to allow better viewability and followed by a table with guild data. This table also has the option to change the number of results shown and to search the table contents using any of the displayed rows.

If the user decides to enter the donations page, there will be two tables displayed with a single row each containing a list of timestamps. There, the user will get the option to select one row from each table, and afterwards click at the count donations button. Afterwards, another table will be created at the bottom of the page. In this table, the name of each player and donations donated and received will be listed.

3.4.2 Discord bot

The Discord bot will have two main commands, every command will start with an exclamation mark, in order to faster discern, which message is meant for the bot.

The first main command being "!guild" or "!gld" for faster use. This command can be used either alone or followed by a guild name ("!guild any guild name"). If a guild name is included, it will list only guilds matching this name. Otherwise, it will list all the guilds within the system database. Since Discord limits every message's length to 2000 characters, these lists will need to be split into parts. In the case of the guild list, each part will have 20 guilds and allow for switching between parts with sending n or p to go to the next or previous part respectively.

Once the user finds their guild, they can send the number the guild has been assigned, which will in turn print a list of attributes which can be displayed and ask the user to list those, the ones to be displayed. The user will then send a message with numbers separated by a space, the first number signifying the column to be used for sorting and the rest of the numbers marking the columns to be displayed. Finally, the bot will send back one or more messages listing the guild data requested. This list might be split into parts depending on its length.

The second main command will list donations donated and received. It as two options "!received" and "!donations". The donations command will return a list of donations sorted by donations given, and the received command will return the same list, just sorted using the received donations.

The received command also has a short version to it ("!rec"), and not unlike the guild command, it can be followed by a guild name or used alone. Once invoked, it will list guilds in parts of up to 20 elements. After the user chooses one of these by sending a message with its number, the bot will respond with a list of timestamps offering the user to choose one. Once the user chooses and sends the required number, he will be given another option to select a timestamp. Afterwards, the bot will send the donations over the period defined by those two timestamps.

The donations command works the same way as the received command. The only difference is that the results are sorted by donations donated instead of received. Not unlike the received command it also has a shortened version "!don" in order to make it faster to use.

This Discord bot will also have the necessary "!help" command, which will list all the commands available or detailed information if followed by another command ("!help guild") without the usual prefix. There should also be miscellaneous commands, like "!web" which will return a link to the website, as some users might prefer that option over the chatbot interface.

3.5 Business logic

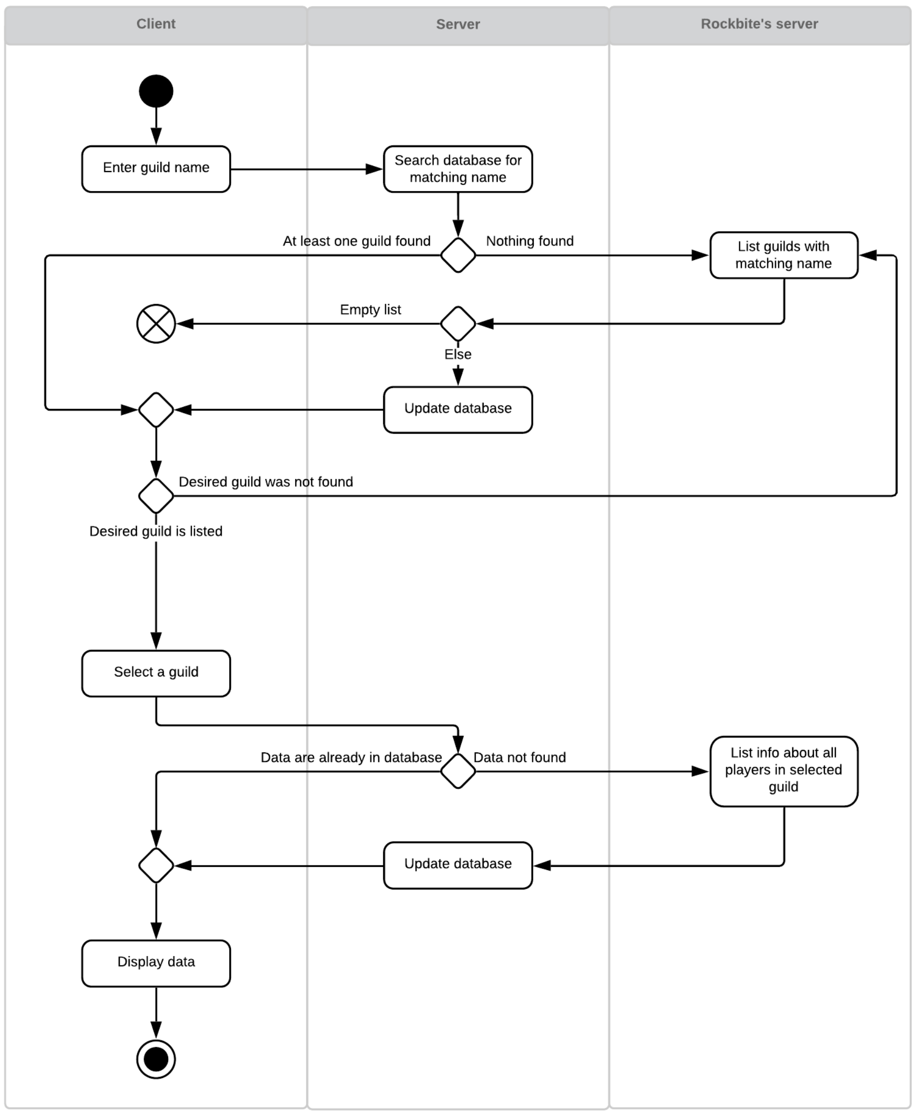


Figure 3.5: Retrieving data about a player

Main two features of this system are to display donations [Figure 3.6] and player [Figure 3.5] information. To display donations, the user first needs to choose a guild. This guild can either be already in the database or not yet loaded. In the case of the guild already being in the system database, it can be found either in the list of all guilds or searched by its name, which will show a list of guilds matching the name provided, from which the user can choose and move on to the next step.

However, if the guild is not yet in the system database, it first needs to be loaded. That is done with the use of the first API method provided by Rockbite [Listing 3.1]. Using this API method, we will gain the basic information about a few guilds matching the provided name, add them to the system database, if they are not already there, and list them for the user to choose one.

After the user displays the list of guilds, they can either choose to select one and continue further or ask for an update, which will load guilds from the Rockbite's API. This manual update is here for the cases when guilds with a certain name are already loaded in the database, but the user created a new guild with the same name and needs to load it to the system database.

After selecting a guild, the user will get to select two dates, including "now". If the user selects an interval delimited by now, it will update all guild data including current donation values for all players within the selected guild. In the case of the interval being selected from the past dates, the already saved values will be used, and no update will occur. Afterwards, donations over the selected period will be counted and returned for the player to see.

The second main feature is relatively similar to the first one described above. A guild first needs to be selected by using the same process. Afterwards, once the guild was selected, the database will be checked, if the guild has necessary data loaded, or only the basic information about the guild is present. In the former case, the guild data will be listed. In the latter case, the information will first need to be requested through the Rockbite's API, saved to the database, and afterwards served to the user.

In order, to be able to provide data about donations over past periods, this system will need to keep all guilds, whose donations or data has been listed within last month updated.

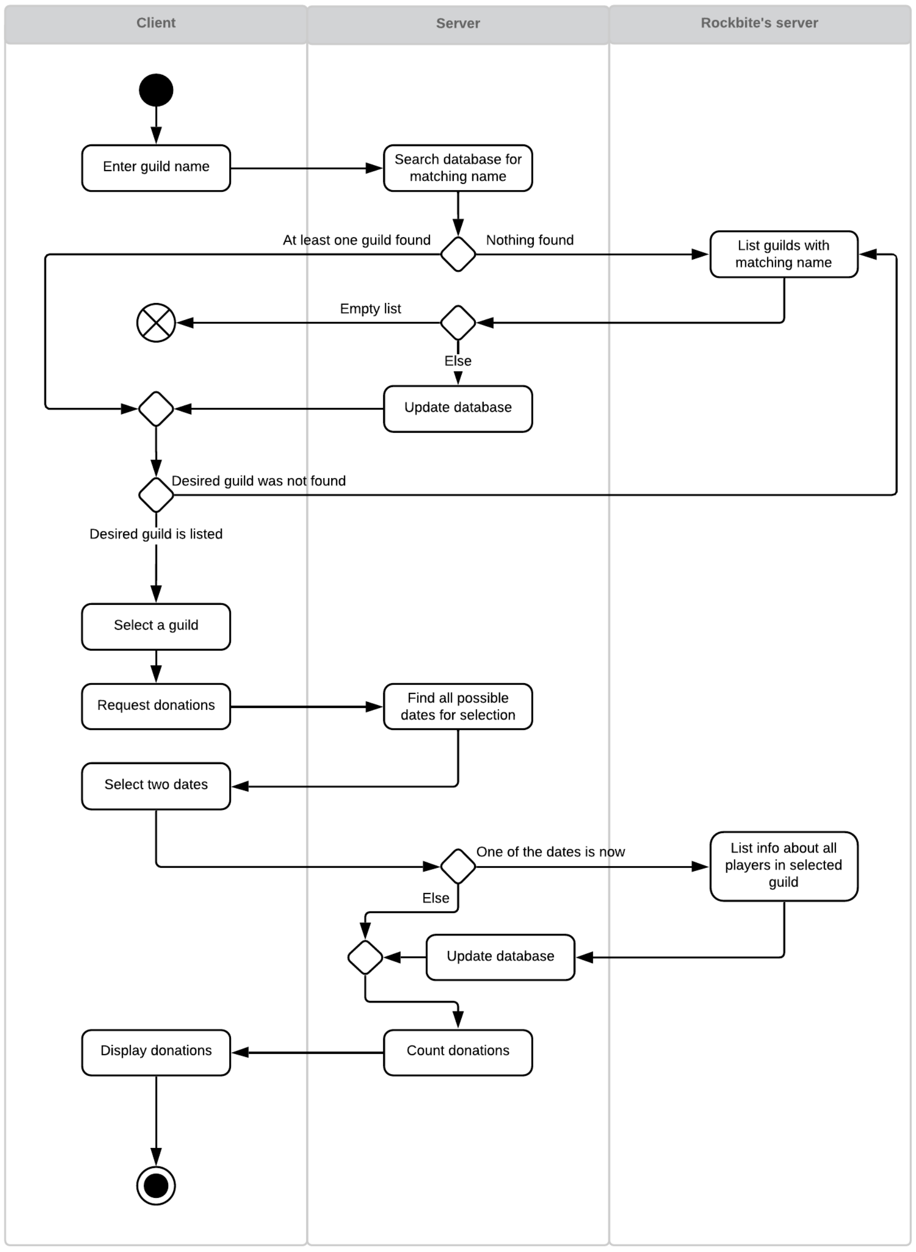


Figure 3.6: Retrieving data about a player's donations

Chapter 4

Implementation

After evaluating different technologies, I have decided to use Python for developing most of the DTAT system. The reason for this choice is mainly the inconsistency in the API created by Rockbite. Where, in some cases, information appears in different formats (integer, string). As such I have decided to use a dynamically typed programming language, to avoid any issues with formats provided by the API.

Moreover, Python enforces its indentation and thus offering a stable code style, which will be helpful in future development. In addition to that, I'm more experienced with Python than with JavaScript, which also helped with the selection process.

4.1 Main server

For the main server, I have decided to use Python 3.7, as it is currently the latest stable version. After settling on Python, I chose the Flask[15] framework. It does not have as many features, like Django[16]. However, I wouldn't have any use for most of these.

Thus, Flask was a perfect choice for me, as it gives me full control over the entire application. Moreover, it has minimal complexity and therefore is easily understood and extended, which will be vital in future development. It also has a large community and excellent documentation with all the necessary examples.

The SQLAlchemy integration for flask also offers an easily settable database connection to any SQL database you might want to use without any changes in code. Thus everyone, who would wish to run this system can choose their SQL database. I have chosen SQLite for testing and development, as it is easy to set up and PostgreSQL for the production deployment.

It is also possible to connect Flask to the MongoDB using the MongoAlchemy Flask extension[17], which is very similar to SQLAlchemy[18]. However, I have decided to stick with SQL databases for this project, as the MongoAlchemy Flask extension[19] has open and unanswered issues from a couple of years ago. Thus not being the most reliable framework.

For gathering data from the Rockbite's API, I chose the Requests framework, as it has quite detailed documentation and is quite easy to understand and use. However, when using it on my personal computer, i ran into an issue, where every request using URL took approximately 5 seconds longer than when done with IP address directly. It appears to be a known issue, and for some reason, it did not occur at the production server. Thus I decided to stick with the Requests framework instead of the urlib3, which allows a more direct approach but doesn't have as many options.

To ensure modularity of the main server, it is split into models, services and API methods. API methods are bundled into blueprint modules and afterwards imported into the Flask application. Thanks to this layout, it is possible to add a new API method, disable or modify a current one with relative ease and without any changes to other parts of the server.

Moreover, API methods are here only as an interface, and all business logic is within services. These services can be imported through the entire server, and are bundled according to their usage, thus offering the option to reuse already written code and therefore being in accordance with the DRY (Do not Repeat Yourself) principle.

The last part of this server are the models, defining the data structure of the main server as well as defining database layout. These models can be easily modified with the option to immediately update the database to the new layout using the Flask migrations module.

4.2 Web server

The web server, same as the main server, is written in Python 3.7 and uses the Flask framework. It has two main functions acting as a proxy server for the main servers public methods and to serve web pages for the DTAT website.

To request data from the main server, the Requests library is used for the aforementioned reasons. The URL for access to the main server is provided through a config file. Hence it is possible to easily change the port and URL of the main server.

4.3 Website

The website is written using HTML, CSS and Bootstrap. I chose these technologies because I'm new to website development, and I decided to learn the basics, before starting with advanced frameworks, such as Django, or React.

As for the smaller frameworks, I have used the DataTables[20] framework for managing tables, as it offers all the features needed and is relatively simple to use. Another framework I have used was Moment, which was used for formatting and sorting timestamps.

4.4 Discord bot

To create the Discord bot, I have used Python, to keep as much of the system in one language. However, it uses Python 3.6, because of the discord.py library, which is not yet updated for newer Python versions.

The Discord bot also uses the Requests framework for accessing the main server's API. With minor modifications, it can also be connected to the public API provided by the web server from any machine, thus offering the option for self-hosting a modified version of the Discord bot.

One of the problems, I have met with during the bot development, was the 2000 character limit for each message. Because of this limit, it is impossible to print an entire table at once. Therefore I have created a function, that accepts a starting text, an array of arrays with data that should be printed, formatting function and two optional parameters, list of visible columns and list of columns with a set width. Afterwards, the function finds an appropriate width for each column not marked as fixed and marked as visible, and prints the table formatted data in one or more messages.

4.5 Cron

To offer up-to-date data to all users, there is a need for periodic updates. Thus, I have decided, to provide daily updates for all active guilds[2], at the same time, as the weekly event starts and ends. Moreover, any user can request an update for the selected guild at any given time. This is only limited by up to one update per ten minutes, which is necessary, to provide services to all users, and not waste away the entire request limit on one guild.

These updates can be carried in two ways. First is the private API method, which updates all active guilds. However, due to the possibility of a failed update, which could occur in a case, when the main server is offline or is having some issues, I have decided to implement the second option.

[2]guild, that has been viewed in last month

The second option is Flask command, which can be accessed through command prompt, and will start its own session. Therefore, it is independent on the main server, which will ensure smooth update regardless of any problems with the rest of the system.

4.6 Security

To ensure the system security, the main server will be deployed as a local server, without the option to be accessed from outside. Afterwards, the web server will act as a proxy server relaying public methods for use by the website or any other tools developed by a third party. Moreover, the web server will be a local server as well, and Nginx will provide proxy between outside word and the web server. Using this method, it will be possible to limit the request frequency for each user, thus offering services for everyone, even with the use of a low spec server.

The Discord bot will be hosted on the same server, thus being able to access the local server. In case of the Discord bot being hosted on another machine a VPN can be used, or it can be connected to the public API. This option offers everyone to modify their version of the Discord bot and host it on their machine without the need for hosting the rest of the system.

I have decided on this approach, as it is fast to set up, there is no need for any tokens or login, which would lead to storing user information, and thus issues with GDPR (General Data Protection Regulation) would arise. This way I managed to avoid all these problems and the system security is still intact.

Chapter 5

Testing

In this chapter, I will be describing the testing of different parts of the DTAT system.

5.1 Main server

The main server was tested with, both, unit and integration tests, with the help of Python packages Mock[21] and Pytest[22]. I have chosen the mock package because of their comprehensive guide, which described all the features I needed and was easier to understand, than for example the Monkeypatch[23] module.

As for the Pytest package, I have decided to use it because it offers many built-in fixtures as well as the option to create custom fixtures with relative ease. It also automatically collects all tests and delivers needed fixtures to those functions, that require them. Moreover, the Pytest package is also recommended for testing in the flask documentation[15].

During the testing one issue occurred with the Mock package. I was trying to mock objects imported from a file, which contained a function with the same name as the said file, and the function was then imported to a module above. Because of this setup, I was unable to specify a correct path for mocking those objects, because the file was covered with the same name function contained in the said module. This issue was resolved by renaming files, which matched the description above.

5.2 Web server

The Web server was tested only briefly through manual testing because it only works as a proxy server and for serving website content. Moreover, for all those features, mostly third-party software was used, and business logic is minimal. Thus the need for complex testing did not arise.

5.3 User interface

Both user interfaces website and Discord bot were tested during the beta stage of development by a selected group of users. These users then noted bugs they found together with suggestions for future updates, of which some were implemented, such as the option to export data to CSV format.

In addition to that, selected parts of the Discord bot were also tested through the unit tests with the use of Pytest and Mock.

Conclusion

In the conclusion of this thesis, all the aims set were successfully met, and the DTAT (Deep Town Admin Tools) system was implemented and deployed for use by the general public. After careful consideration, Python was used to write most of this system, as well as various Python frameworks, such as Flask, SQLAlchemy or Requests.

To ensure the stability of this system, it was tested with integration and unit tests. Moreover, to ensure the ease of future development of the DTAT system, it was also documented, as well as written in accordance with the PEP8 convention.

As a part of this system, there are two interfaces created. One is a Discord bot allowing data access to any data required within a Discord client. This option also allows any player to directly print the required data into a selected Discord channel in the form of a leaderboard.

Another interface, which this system offers is a website, which allows more graphical interface, which is easier to use for some users. Furthermore, this website is responsive, thus offering comfortable access from mobile devices, as well as desktops.

This new system also significantly shortens the time needed for retrieving data about players and their donations, and also offers easier access to such data.

In the future, I would like to further extend this system with an event planner, which would show the most effective options to set up production with given parameters like stocks and building levels.

Bibliography

[1] Lolskill. http://www.lolskill.net, accessed on 2019-04-04.

[2] Dotabuff. https://www.dotabuff.com, accessed on 2019-04-04.

[3] ZigFreeD. Ikalogs. https://ikalogs.ru, 2012, accessed on 2019-03-04.

[4] Parker, Z.; Poe, S.; et al. Comparing NoSQL MongoDB to an SQL DB. In *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, New York, NY, USA: ACM, 2013, ISBN 978-1-4503-1901-0, pp. 5:1–5:6, doi:10.1145/2498328.2500047. Available from: http://doi.acm.org/10.1145/2498328.2500047

[5] ostezer; Drake, M. SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems, March 2019, accessed on 2019-04-04.

[6] Reddy, M. *API Design for C++*. Elsevier, 2011, 8 pp.

[7] Prechelt, L. An empirical comparison of seven programming languages. *Computer*, volume 33, no. 10, October 2000: pp. 23–29, ISSN 0018-9162, doi:10.1109/2.876288.

[8] Sarkar, D. *Text analytics with Python: a practical real-world approach to gaining actionable insights from your data.* Apress, [2016], 55-59 pp.

[9] Github. https://octoverse.github.com/projects#languages, 2019, accessed on 2019-04-04.

[10] Dayley, B.; Dayley, B.; et al. *Node.js, MongoDB and Angular web development*. Addison-Wesley, second edition, [2017], ISBN 0134655532.

[11] Peng, M. Multithreading Javascript. `https://medium.com/techtrument/multithreading-javascript-46156179cf9a`, September 2017, accessed on 2019-04-04.

[12] Robbins, J. *Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics*. O'Reilly Media, 2012, ISBN 9781449337551. Available from: `https://books.google.cz/books?id=A-tltyafYmEC`

[13] Rapptz. discord.py. `https://discordpy.readthedocs.io/en/latest/`, 2019, accessed on 2019-04-04.

[14] discord.js. discord.js. `https://discord.js.org`, 2018, accessed on 2019-04-04.

[15] Flask. `http://flask.pocoo.org`, accessed on 2019-04-04.

[16] Django. `https://www.djangoproject.com`, accessed on 2019-04-04.

[17] Flask MongoAlchemy. `https://pythonhosted.org/Flask-MongoAlchemy/`, accessed on 2019-04-04.

[18] SQLAlchemy. `https://www.sqlalchemy.org`, accessed on 2019-04-04.

[19] Flask-MongoAlchemy. `https://github.com/cobrateam/flask-mongoalchemy/issues`, accessed on 2019-04-04.

[20] DataTables. `https://datatables.net`, accessed on 2019-04-04.

[21] Mock. `https://docs.python.org/3/library/unittest.mock.html`, accessed on 2019-04-04.

[22] Pytest. `https://docs.pytest.org/en/latest/`, accessed on 2019-04-04.

[23] Monkeypatch. `https://docs.pytest.org/en/latest/reference.html#monkeypatch`, accessed on 2019-04-04.

APPENDIX **A**

Acronyms

DTAT Deep Town Admin Tools

API Application Programming Interface

CSV Comma Separated Values

SQL Structured Query Language

GDPR General Data Protection Regulation

UI User Interface

ACM Association for Computing Machinery

Appendix **B**

Contents of enclosed CD

```
readme.md ........................ the file with CD's contents description
├── main-server ......... the directory containing main server source codes
├── web-server .. the directory containing web server & website source codes
├── discord-bot .......... the directory containing Discord bot source codes
├── text ....................................... the thesis text directory
│   ├── thesis.pdf ........................ the thesis text in PDF format
│   └── source .................................. the thesis source codes
└── API ................................................. API endpoints
```