ТЕХНОЛОГИЧНО УЧИЛИЩЕ
ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

Тема: Приложение за автоматично качване и тагване на изображения във фотобанка.

Дипломант:

Стела Тихомирова Касабова

Дипломен ръководител:

инж. Мартин Стоянов

СОФИЯ

2 0 2 2

УВОД

В днешно време всеки може да се занимава с фотография – нужен е само телефон с камера. Превърнало се е от интересно хоби в сложна и трудоемка професия. С набирането на популярност се появят и нови опции за продаване на снимки – фотобанки, където потребители могат да качат своите фотографи или да купят правата за ползване на нечии чужди.

Създадени са приложения, чиято цел е да улеснят и забързат процеса на продаване - качването на множество снимки наведнъж не е приятно, особено ако става за дума за стотици или хиляди файлове. За съжаление част от тези решения на проблема просто създават нов – добавяне на ключови думи. Те помагат на потребителите да намират снимки по-бързо или да са сигурни, че работата им има по-голям шанс да бъде намерена. Не всички приложения предлагат добавянето им към снимките под формата на метаданни, което улеснява самото качване, но има немалък шанс да навреди на доходите, идващи от продаването във фотобанки.

Целта на дипломната работа е да предостави интерфейс в който да може да се избере директория от машината на потребителя, след което да визуализира снимките и да покаже предложените ключови думи. Потребителят има възможността да архивира директорията и да качи снимките, както и техните тагове под формата на метаданни, към FTP сървър.

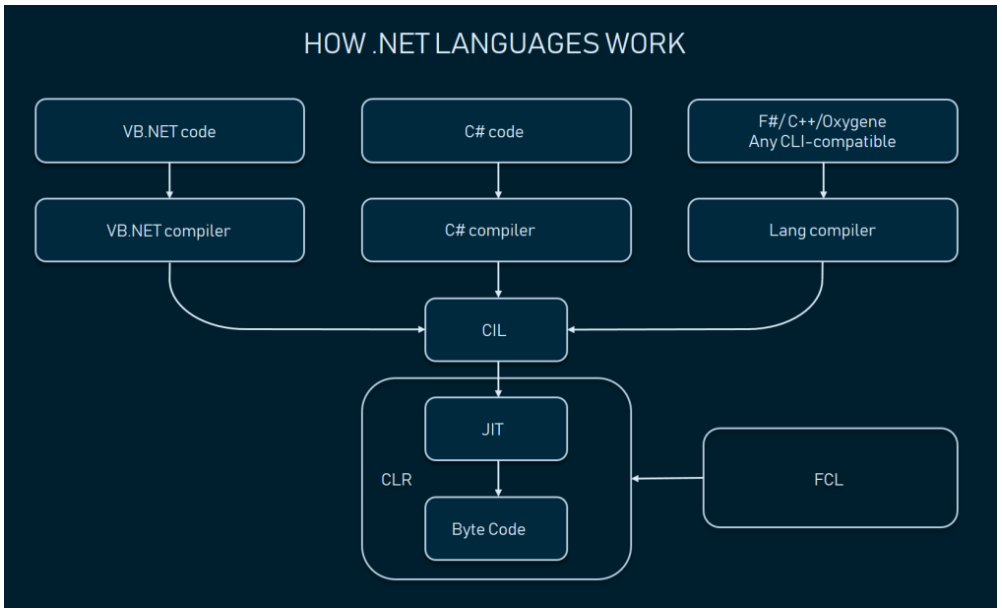
1. ПЪРВА ГЛАВА - Проучване

1.1. Технологии за разработване на desktop приложения

Desktop приложенията са важна част от ежедневието на всеки един човек. Широкото им разпространение води и до много различни начини за тяхното създаване. Петте най-често използвани езици за програмиране, с чиято помощ се разработват тези приложения са: C#, C++, Python и Java.

1.1.1. C#

C# е обектно ориентиран език, разработен от Microsoft. Програмите разчитат на .NET за изпълнението си. В архитектурата на .NET, чиято схема е показана на фиг. 1.1., има 3 главни компонента. Написаният код се свежда до Common Intermediate Language (CIL), което запазва всичко под формата на assembly. След това се преминава към Common Language Runtime (CLR), което се грижи за изпълнението на кода , както и за разпределяне на ресурсите, използвани от програмата. За да се изпълнят инструкциите от CIL се използва Just-In-Time (JIT) компилатор, който е част от CLR процеса [12].



Фиг. 1.1. .NET архитектура

Често срещаните варианти за разработване на приложение, използвайки C#, са:

- WPF, или Windows Presentation Foundation, позволява създаването на приложение с markup и code-behind. За потребителския интерфейс се използва XAML, а за функционалностите – език, съвместим с .NET.
- Windows Forms, познато и като WinForms, е рамка за създаване на приложения с потребителски интерфейс за операционната система Windows.

И двете рамки са добри за създаване на приложения. WPF се фокусира повече върху добавянето на по-сложни функционалности, докато WinForms набляга на потребителския интерфейс.

1.1.2. C++

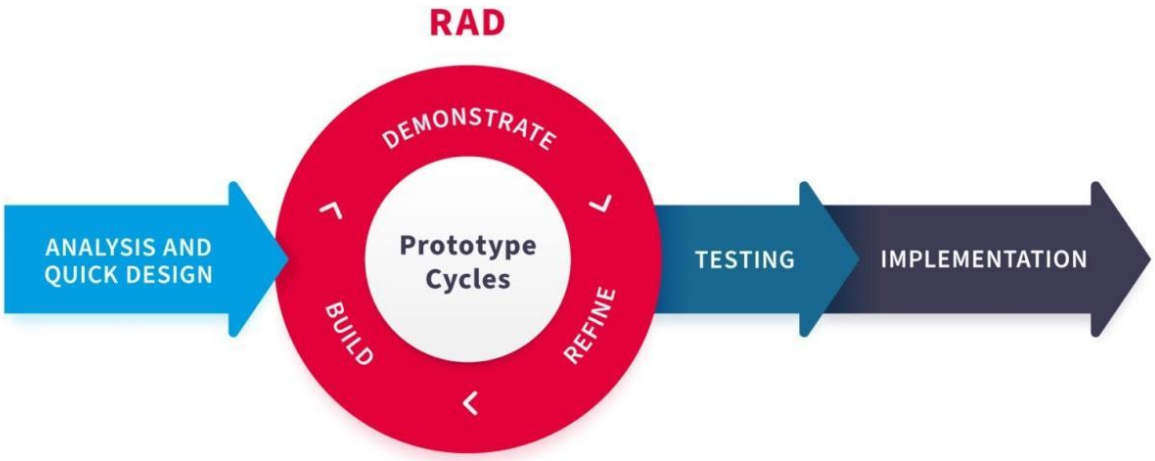
C++ е обектно ориентиран език за програмиране със статични типове, създаден като продължение на C. Езикът е достъпен на множество платформи, благодарение на големия брой компилатори. C++ се компилира директно към машинен код, което му позволява да е изключително бърз език, ако кодът е оптимизиран. Той е много широко разпространен и се използва в:

- Операционни системи
- Desktop приложения – голяма част от вградените приложения на Windows и Mac OS са написани на C++. Създаването на графичен потребителски интерфейс е възможно с помощта на рамки като Qt, wxWidgets и GTK+.

- Видео игри – Unreal Game Engine използва модифицирана версия на C++ и е една от най-мощните среди за разработка на игри.
- Embedded системи – програмите за Arduino микропроцесорите се пишат на C++.

1.1.3. Python

Python е обектно ориентиран език за програмиране с динамична семантика - означава, че всички създадени обекти са динамични – няма нужда да се обяви типът им при създаване. Това, както и фокусът върху структури от данни, прави Python подходящ за бързо разработване на приложения (Rapid Application Development, схемата на което е на фиг. 1.2.) [13].



Фиг. 1.2. Модел на Rapid Application Development

Този метод на работа позволява на програмистите да пропуснат компилацията и да минат директно към тестване и имплементация. При изпълнение на код се праща exception, ако възникне грешка. В случай, че това не е хванато от самата програма, на терминала се показва stack trace.

1.1.4. Java

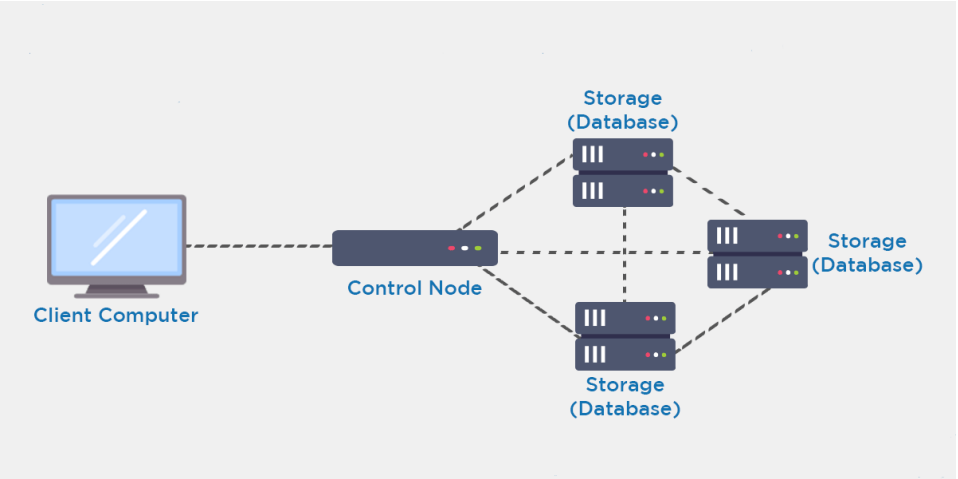
Java е обектно ориентиран език от високо ниво, разработен през 1995 година. Той има множество реализации, направени от GNU, Microsoft, IBM и Oracle, но в днешно време тази на Oracle е най-широко разпространена и използвана.

Подобно на C#, написаният код не се компилира до машинен код за процесора. Вместо това минава през виртуален процесор. Нарича се Java Virtual Machine (JVM) и свежда написаното на Java до байт код, които може да се разбере от машината. Езикът е подходящ за разработката на приложения – мобилни, web и desktop. За създаването на графичен потребителски интерфейс могат да се използват Java Foundation Classes/Swing и Abstract Window Toolkit API.

1.2. Съхраняване на файлове

1.2.1. Облачни хранилища

Облачните хранилища са модерния начин за съхраняване на дигитална информация. Потребителите могат да изпратят файловете си към сървър, където те се пазят, докато не са нужни за употреба. Повечето доставчици на тези услуги имат множество сървъри. Дори и да възникне проблем с един от сървърите, файловете могат да бъдат преместени на друго място, което предоставя постоянен достъп до информацията. Този метод на работа е представен на фиг. 1.3.



Фиг. 1.3. Комуникация с облачно хранилище

Има три подхода към облачните хранилища [14]:

- Public – подходящ за индивидуална употреба. В повечето случаи се плаща такса на доставчик за достъп до услугите. По този начин потребителите няма нужда да се занимават с поддръжката на системата.
- Private – обикновено се използва от компании. Тук няма доставчик, потребителят сам поддържа сървърите и се грижи за сигурността на данните.
- Hybrid – комбинира предишните подходи, подходящ за компании. Това позволява сигурността на private хранилище, но дава възможността обработката на информация да бъдат възложени на public хранилище с помощта на облачни изчислителни услуги.

Съществуват много публични облачни хранилища. Някои от най-разпространените са Microsoft OneDrive, Google Drive, Dropbox и Apple iCloud Drive.

1.2.2. Облачни изчислителни системи

Облачните изчислителни системи са разширение на идеята за облачни хранилища. Потребителите на тези системи имат достъп до

множество услуги срещу заплащане. Плащането се случва само когато услугите се използват, което им позволява да намалят потребителските разходи, тъй като няма нужда да се купуват и поддържат отделни машини за работа, а се ползват предоставените от системата като техен заместител.

Системите предлагат три вида услуги: Software as a Service (SaaS), Platform as a Service (PaaS) и Infrastructure as a Service (IaaS) [15].

- SaaS позволява на потребителите да използват софтуер, разработен от дистрибутора, без да се налага да го инсталират или да се тревожат за нужните му ресурси.
- PaaS дава достъп до облачни компоненти и рамка, върху която потребителите могат да разработват приложения. Всички части от структурата, върху която се работи, се контролират от доставчика.
- IaaS предоставя изчислителна инфраструктура, например сървъри, операционни системи и място за съхраняване на данни, чрез виртуализация. Това спестява на потребителите нуждата да купуват отделни машини, тъй като им се дава пълен контрол над наетите ресурси.

Има три главни облачни изчислителни системи, които се използват. Те са Amazon Web Services (AWS), Microsoft Azure и Google Cloud Platform (GCP).

1.3. Анализиране на изображения

Процесът за анализиране на изображения се свежда до извличане на полезна информация от подадена снимка или видео [16]. Човек би могъл да

се справи с това начинание без помощта на софтуер, но ако става дума за голямо количество файлове това се превръща в трудоемка задача.

Съществуват много различни начини за анализирането на изображения, някои от които са разпознаване на предмети, разделяне на сегменти, следене за движение и други [16]. Всеки един от тези методи се използва за различни цели. За сега няма метод, който да обхваща достатъчно функционалности за да се сравни с човешкото зрение, но използването на съществуващите варианти спестяват много време на потребителите.

Имплементациите на алгоритми за анализиране на изображения са многобройни, но някои от тях се справят по-добре от други. Облачните изчислителни системи предлагат едни от най-добрите услуги за извличане на информация от снимки под формата на Computer Vision API. Разработени са и алгоритми с отворен код, например OpenCV, но не се справят толкова добре колкото платените варианти.

1.4. Съществуващи реализации

1.4.1. Dropstock

Целта на Dropstock е да служи като посредник между потребителя и сървърите. Приложението е направено за тези, които вече използват услугите на Dropbox за да съхраняват снимките си. То работи като се свърже с Dropbox акаунт, след което потребителят трябва да обозначи папка, в която да действа Dropstock. Добавянето на файлове в тази папка ги праща на предварително избрани FTP сървъри. Има услуга, която

предлага ключови думи (тагове), описания и заглавия за снимките, но опцията за тяхното качване все още не съществува.

1.4.2. Wirestock

Това не е приложение, а по-скоро услуга. Подадените снимки не се качват в профила на потребителя, а в този на компанията, която е създала Wirestock. Тази функционалност има плюсове и минуси – по-вероятно е повече хора да намерят и купят снимките през вече известният профил на приложението, но по този начин се затруднява създаването на портфолио от потребителите. Тук, също като в Dropstock, няма опция за добавяне на метаданни.

1.4.3. StockSubmitter и Microstock Plus

StockSubmitter е един от най-добрите варианти за масово качване към фотобанки. Въпреки че приложението работи чудесно само, за още по-добри резултати създателите на StockSubmitter са добавили Microstock Plus. С помощта на този уеб базиран софтуер се дава възможност за организиране на файловете, следене на продажбите, свързани с профила на потребителя, както и добавяне на метаданни като ключови думи.

2. ВТОРА ГЛАВА - Изисквания и технологии

2.1. Функционални изисквания

- Приложението трябва да предостави интерфейс на потребителя, през който да се визуализират и разглеждат снимки от избрана директория.
- Заедно с файловете се зареждат ключови думи и описание, предоставени от Azure Computer Vision.
- Архивиране, което се запазва в Azure Data Lake.
- Качване към фотобанките чрез FTP.
- При пращането към сървър разширенията на снимките се променят на JPEG, тъй като повечето сайтове работят с този тип файлове.
- Потребителската активност се запазва в .log файл, ако няма такъв при инициализиране на програмата се създава нов.

2.2. Технологии на разработване

2.2.1. Език и библиотеки

Проектът е реализиран на езика Python. Моделът на Rapid Application Development е подходящ за приложението, тъй като функционалностите се нуждаят от често тестване. Това дава предимство на Python пред другите възможни избори. Освен това е един от най-използваните програмни езици и разполага с много библиотеки и източници на информация. За имплементиране на функционалните изисквания са използвани няколко библиотеки, най-важните от които са Iptcinfo3, azure и os.

Iptcinfo3 е библиотека с отворен код, разработена първоначално от Джон Картър за езика Perl, след което се адаптира да работи с Python от

Тамаш Гулачи [1]. Позволява създаването и модифицирането на метаданни за изображения. Една снимка може да има различни типове метаданни, като най-използваните са Exif и IPTC. Exif се фокусира на техническите детайли, като например тип камера или използвани настройки, докато IPTC набляга на съдържанието на самата снимка [6]. Тъй като е необходимо да се добавят ключови думи и описание към файл, данни от типа IPTC биха били най-подходящи. С помощта на тази библиотека изпращането на допълнения към съдържанието на снимките се улеснява.

Другите две библиотеки, azure и os, са използвани за архивирането на изображения. С помощта на azure се осъществява връзка с облачното хранилище и се позволява съхраняването на данни като blobs, познато и като binary large objects. Методите, взети от os, позволяват работа с файловете, които се намират на машината. Функционалности като променяне на разширения, обхождане на директории, редактиране на съдържанието и други са възможни благодарение на тази библиотека.

Освен тези библиотеки е използвана и рамката Tkinter. Това е един от най-разпространените инструменти за разработване на графичен интерфейс в Python. Разполага с множество функционалности за създаването на стабилно приложение. Произлиза от Tcl/Tk, добавяйки обектна ориентираност [2]. Това прави кода по-лесен за четене и разбиране, особено за програмисти вече свикнали с обектно ориентирани езици.

За създаването на приложението се използва PyCharm. Тази развойна среда е разработена от JetBrains и поддържа езиците Python, JavaScript, TypeScript, CSS и др. Предлага много полезни функционалности, които

улесняват писането на код, като автоматично оформление на кода, лесно навигиране до имплементациите на методи и класове, маркиране на грешки в реално време и предложения за автоматично довършване на код.

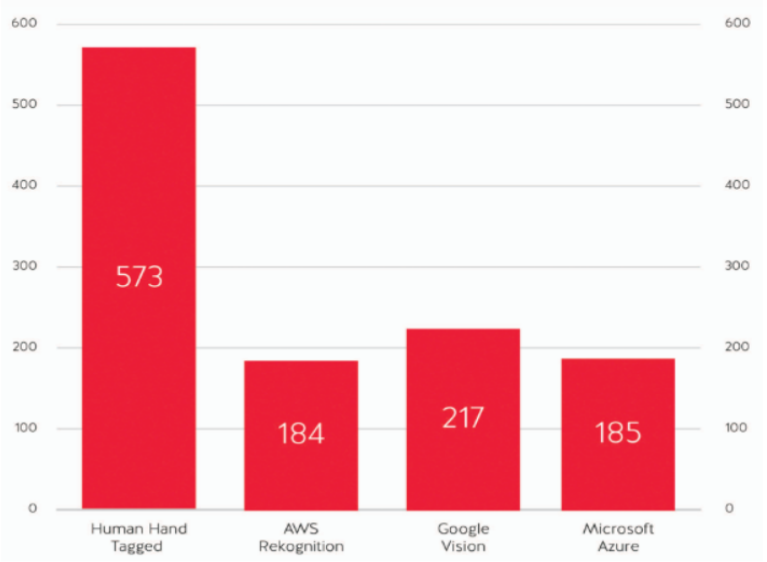
2.2.2. Azure

Тъй като за проекта е нужно не само съхраняване, но и анализиране на изображения, изборът на облачна изчислителна система пред облачно хранилище е по-разумен. Използването им е платено, но системите предоставят всички нужни услуги на едно място.

Опциите за облачна изчислителна система са три: Microsoft Azure, AWS и GCP. За да се избере една от системите трябва да се сравнят спрямо услугите, които ще се използват: съхраняване на файлове и извличане на информация от изображения. Цената за използване на 1GB от облачните хранилища в cool tier изглежда по следния начин [17, 18, 19]:

- Microsoft Azure – $0.01 per GB
- AWS – $0.023 per GB
- GCP - $0.005 ÷ $0.007 per GB (зависи от използвания сървър)

От това сравнение може да се установи, че Azure или GCP ще са най-евтините варианти. Финалното решение опира до сравнението на предложените функционалности на Vision AI.



Фиг. 2.1. Сравнение на AWS, GCP и Azure Vision AI

На Фиг. 2.1. е показано сравнение на това как са се представили с тагване Computer Vision услугите на AWS, GCP и Azure спрямо човек. Google Vision се справя най-добре със задачата, но изборът пак опира до цената на услугите. За един месец всеки един потребител има определен брой безплатни викания за анализиране на изображения. При GCP този брой е 1000, а при Azure – 5000 [20, 21]. Дори и да не е с точността на Google Vision, услугата на Azure е достатъчно добра, и в пъти по-изгодна, за да се окаже по-добрия избор за разработването на приложението.

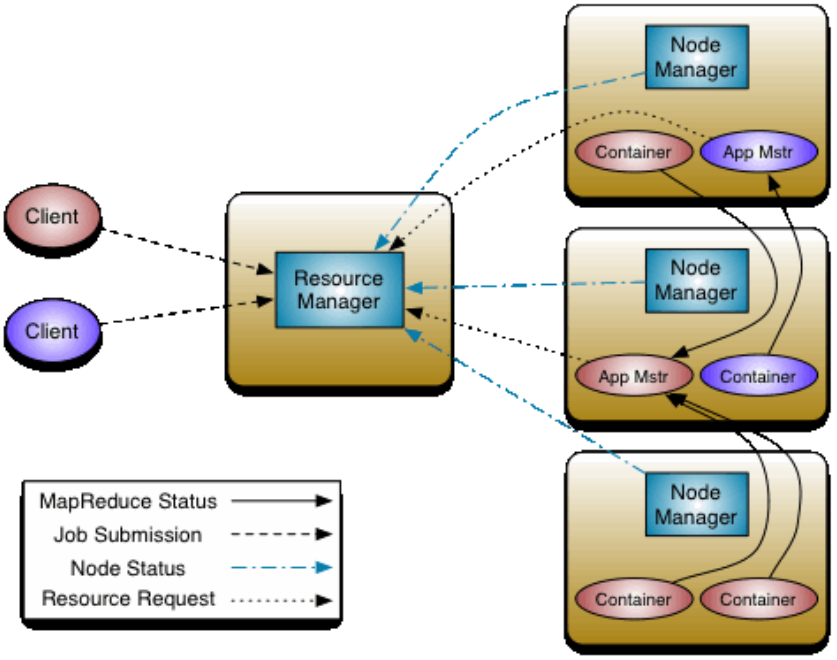
Azure Computer Vision е API, което има възможността да анализира снимки или видео записи и да връща данни за тях, включително и увереност в отговора [3]. Изключително полезно е за разработването на проекта, но идва с ограничения. При работата с изображения трябва да се има в предвид, че снимката не може да надвишава размер от 4MB. Височината и ширината на снимката трябва да имат стойност по-голяма от 50 пиксела.

Computer Vision разполага с много функционалности, някои от които са: намиране на обекти, генериране на ключови думи и описания, засичане на маркови продукти и други [8]. Разпознаването на обекти зависи от това дали предметите са достатъчно големи (поне 5% от снимката), дали са струпани заедно и дали се различават само по производителя. Ако някое от тези условия е изпълнено е възможно да доведе до неадекватен отговор от страна на Computer Vision [9].

Data Lake е облачна услуга, предлагана от Azure, която е базирана на моделът на работа на Apache Hadoop YARN (Yet Another Resource Negotiator) [10]. Действа с имплементирането на клъстери, които са групи от машини, действащи заедно за да съхраняват и анализират данни. В системата тези машини се наричат „nodes“/”възли”. Структурата на Hadoop YARN е показана на фиг. 2.2. и съдържа в себе си 6 части [11]:

- Client – подава заявки към компонентите.
- Container – тук се изпълняват заявките. Съдържа Container Launch Context (CLC), в което са записани нужните команди и ресурси за извършването на заявка.
- Application Master – взима CLC от Node Manager и комуникира с Resource Manager.
- Node Manager – следи действията, които се извършват във възложения му възел. Главната му функционалност е да се грижи за контейнерите. Работи заедно с Resource Manager, за да се контролира употребата на ресурси между възлите. Всички действия, извършени от Node Manager, се пазят в логове.

- Resource Manager – получава заявките от клиента и се грижи за разпределянето на свободните ресурси към клъстерите, които се нуждаят от тях.
- Application Manager – отговаря за следенето на група заявки, подадени на възел. Проверя дали данните са валидни, както и статуса на вече инициализирани действия. Може да откаже изпълнението на заявка ако няма достатъчно свободни ресурси.



Фиг. 2.2. Структура на работа на Apache Hadoop YARN

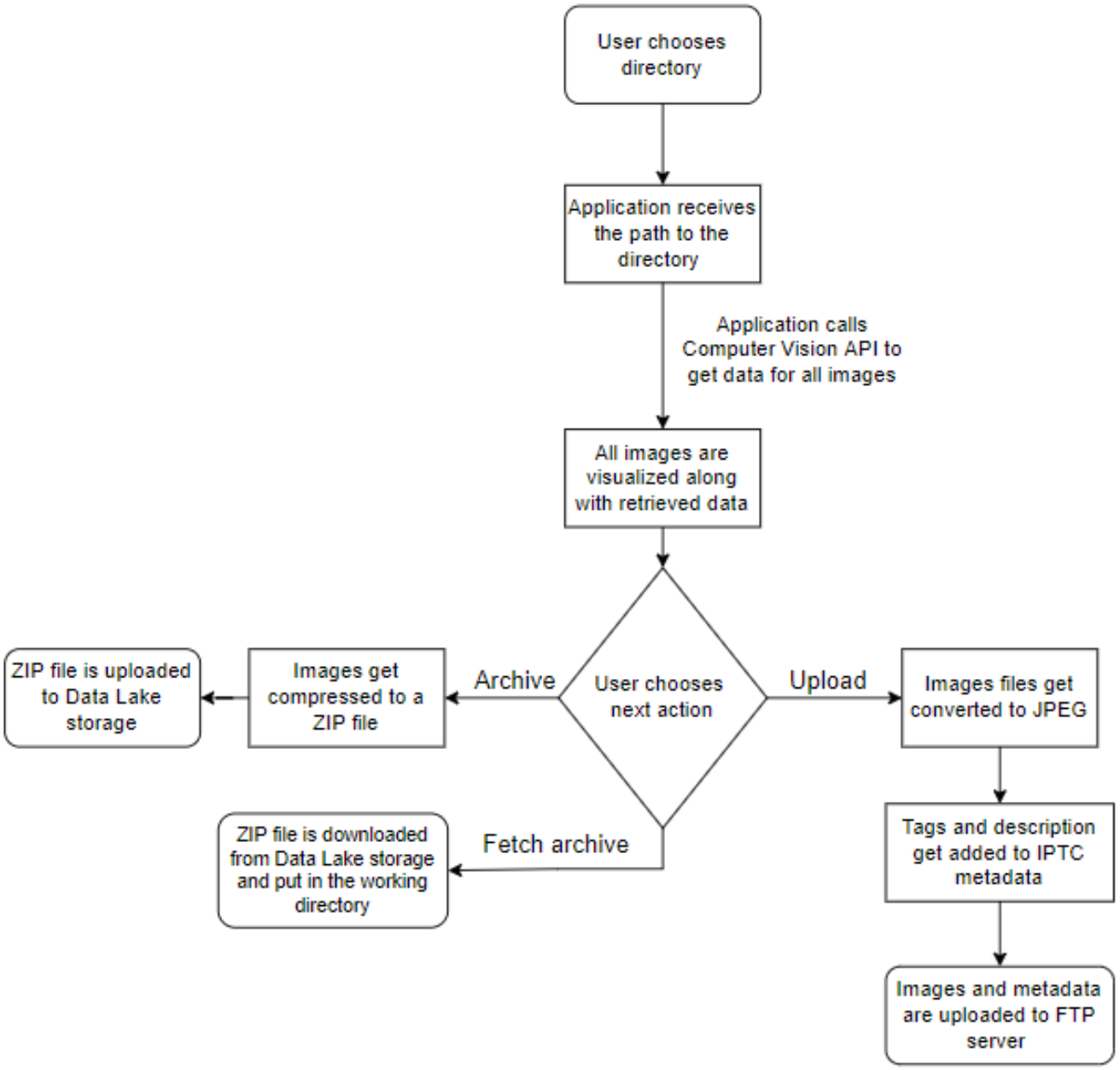
В Data Lake хранилищата може да има няколко типа контейнери: private, blob и container. Разликата между тях е нивото на достъп, с който разполагат анонимни лица. При private никаква информация не се предоставя на външни лица. При blob е позволено четене на обекти в контейнера, но достъпът до данните им е забранен. При container има достъп и до данните.

2.3. Описание на алгоритъма

При стартиране на приложението потребителят може да избере директория, от която да се визуализират снимките. Със заявка към Azure Computer Vision се получават ключовите думи и описанието. Всичко се зарежда в прозорец, като навигирането между фотографи се случва с бутони. След това може да се архивират файловете или да се качат към сървър.

Архивирането се осъществява като се обходи директорията и всички снимки се добавят в компресиран ZIP файл, който се изпраща към вече създадено Data Lake хранилище и blob контейнера, който е в него. След като веднъж снимките са изпратени, потребителят може да ги свали от облачното хранилище на компютъра си.

Ако потрбителят избере някоя от опциите за качване се прави опит за свързване със сървъра, който при успех позволява да изпратим снимките и създадените към тях файлове, съдържащи метаданните. Тъй като част от фотобанките работят само с JPEG файлове, включително и двете, използвани в проекта, преди самото качване директорията със снимките се обхожда и всички файлове на изображения се модифицират да са с JPEG разширение.



Фигура 2.3. Блок схема на принципа на работа

3. ТРЕТА ГЛАВА – Имплементация на приложение за автоматично тагване и качване във фотобанка

3.1. Структура на проекта

Приложението може да се раздели на четири главни части:

- Графичен потребителски интерфейс
- Извличане на метаданни
- Архивиране
- Качване към FTP сървър

3.2. Графичен потребителски интерфейс

За реализирането на тази част от приложението е използвана Python рамката Tkinter. Първата стъпка е инициализиране и пускане на прозорец, което се случва в началото на файла mainWindow.py:

```
root = Tk()
root.title('Application')
root.state("zoomed")


dir_button = Button(root, text="Choose directory", command=lambda: getImages(filedialog.askdirectory()))
dir_button.place(anchor=CENTER, relx=.5, rely=.5)

root.mainloop()
```

Фиг. 3.1. Създаване на Tk основа

В този код се създава инстанция на Tk, което служи като основа за поставяне на елементи. След инициализацията се добавя бутон за избиране на директория, в която да работи програмата. Пътят до тази директория се подава на функцията getImages(path_param), която би могла да се раздели на две части:

```python
def getImages(path_param):
    logging.basicConfig(level=logging.DEBUG, filename='logs.log', format='%(asctime)s %(levelname)s:%(message)s')
    logger = logging.getLogger(__name__)
    logger.info("Visualizing directory: " + path_param)

    image_paths = []
    images = []
    tag_arr = []

    extensions = ('.JPG', '.jpg', '.JPEG', '.jpeg')
    for file in os.listdir(path_param):
        if file.endswith(extensions):
            path = os.path.join(path_param, file)
            # Azure API only works with images 4MB and under
            if os.path.getsize(path) / (1024 * 1024) < 4:
                image_paths.append(path)

    # Resizes photos for consistent image display
    for image in image_paths:
        temp_image = PIL.Image.open(image)
        temp_image = temp_image.resize((600, 600), PIL.Image.ANTIALIAS)
        temp_resized = ImageTk.PhotoImage(temp_image)
        images.append(temp_resized)
```

Фиг. 3.2. Визуализиране на изображенията

Първата част на функцията обикаля подадения път и намира всички изображения. Преди да се започне с визуализацията на файловете се взима инстанция на файла logs.log. Ако не съществува или не може да бъде намерен приложението ще създаде нов лог и ще започне да записва потребителската активност там.

Заради ограниченията на Computer Vision само снимките по-малки от 4MB могат да се подават за получаването на метаданните, съответно само те ще бъдат показани на потребителя. Тъй като повечето фотографи са с прекалено голям размер, за да се визуализират удобно, преди добавянето на екрана се смаляват до размер 500x500 пиксела. Тази промяна не е трайна, при качването или архивирането изображенията са с оригиналния си размер.

След нужните проверки и промени снимките са добавят като обекти в масив, който ще се използва за визуализацията. Програмата не работи директно с файловете преди да се стигне до добавянето на метаданни или качването в FTP сървърите. По този начин се избягва случайно повреждане или нежелани промени.

```
# Adds frames for structured display of elements
image_frame = Frame(root, width=700, height=700)
image_frame.place(anchor=CENTER, relx=.4, rely=.5)
tag_frame = Frame(root)
tag_frame.place(anchor=CENTER, relx=.65, rely=.5)

tag_arr = getTagLabels(image_paths[0], tag_frame)
for g in range(0, len(tag_arr)):
    tag_arr[g].grid(row=g, column=0)

image_label = Label(image_frame, image=images[0])
image_label.grid(row=0, column=0, columnspan=3)

caption_label = Label(image_frame, text=getCaption(image_paths[0]))
caption_label.grid(row=1, column=1)
```

Фиг. 3.3. Рамки за поставяне на елементи

Във втората част се добавят рамки, за да може елементите да не се припокриват или да си пречат. Първата снимка от масива се слага в съответната рамка заедно с ключовите думи и описанието, чието извличане ще бъде описано в точка 3.3. Функцията getTagLabels се намира във файла getImageData.py и връща всички ключови думи под формата на етикети, които могат да бъдат добавени на екрана.

3.2.1. Функционалност на бутоните за сменяне между снимките

```
def cycleImages(image_number, caption_label, image_label, image_frame, images, image_paths, tag_frame, tag_arr):
    # Delete old tags
    for i in range(0, len(tag_arr)):
        tag_arr[i].grid_forget()

    image_label.grid_forget()
    image_label = Label(image_frame, image=images[image_number - 1])
    image_label.grid(row=0, column=0, columnspan=3)

    caption_label.grid_forget()
    caption_label = Label(image_frame, text=getCaption(image_paths[image_number - 1]))
    caption_label.grid(row=1, column=1)

    # Display new tags
    tag_arr = getTagLabels(image_paths[image_number - 1], tag_frame)
    for j in range(0, len(tag_arr)):
        tag_arr[j].grid(row=j, column=0)
```

Фиг. 3.4. Смяна на изображения, ключови думи и описания

Извикването на cycleImages, която се намира във файла buttonFunctions.py, заличава предишната информация от екрана. След като това се изпълни следващия обект от масива се показва на екрана и се зареждат новите данни. По този начин потребителите могат да разгледат цялата колекция от снимки заедно с ключовите думи и описанието.

3.2.2. Прозорец за архивиране

```
def archive(path):
    archive_window = Toplevel(height=300, width=300)

    zip_name_label = Label(archive_window, text="What name should the directory be archived under? (Please separate "
                                                "words with underscores, not spaces)")
    zip_name_label.pack()
    zip_name_input = Entry(archive_window)
    zip_name_input.pack()

    submit_button = Button(archive_window, text="Submit", command=lambda: uploadBlob(path, zip_name_input.get()))
    submit_button.pack()
```

Фиг. 3.5. Прозорец за архивиране

Натискането на бутона с текст „Archive directory” извиква функцията archive(path), също намираща се във файла buttonFunctions.py. Приема път

към директория като параметър. Изкарва нов прозорец на екрана, където потребителят може да въведе името, под което да се архивира директорията, в която приложението работи в момента.

Бутонът “Fetch archive” извиква download(path). Функционалността е подобна на тази на archive(path). Единствената разлика е в текста, който се показва на новия прозорец и действието, което се извършва след натискането на submit_button.

3.3. Извличане на метаданни

Всички функции за извличане на данни се намират във файла getImageData.py. Преди функциите е създаден клас Tag, в чийто инстанции се записват имената на таговете, както и увереността на Computer Vision дали думата се отнася за снимката.

```
def getTags(path):
    decrypted_data = str(decryptKeys())
    decrypted_data.replace("'", '"')

    result = json.loads(decrypted_data)
    subscription_key = result["subscription"]

    with open('sas_keys.json', 'w') as file:
        file.write(decrypted_data)
        file.close()

    encryptKeys()

    endpoint = 'https://cvazureapi.cognitiveservices.azure.com/'
    analyze_url = endpoint + "vision/v3.2/analyze?"

    headers = {'Ocp-Apim-Subscription-Key': subscription_key, 'Content-Type': 'application/octet-stream'}
    params_tags = {'visualFeatures': 'Tags'}
    with open(path, 'rb') as image:
        data = image.read()
    # Connect to Computer Vision API and get tags
    response_tags = requests.post(analyze_url, headers=headers, params=params_tags, data=data)
    tags = response_tags.json()

    tag_arr = []
    for i in range(0, len(tags["tags"])):
        temp_tag = Tag(tags["tags"][i]["name"], tags["tags"][i]["confidence"])
        if temp_tag.confidence > 0.5:
            tag_arr.append(temp_tag)

    return tag_arr
```

Фиг. 3.6. Извличане на тагове

Във функцията getTags(path) се осъществява връзка с Azure Computer Vision с помощта на POST заявка. В резултатът от тази заявка могат да се намерят ключовите думи за снимката, чийто път е подаден като аргумент.

За да се подсигури точността на данните преди таговете да се върнат като резултат се проверя точността им. Azure Computer Vision поставя като стойност за увереността число между 0.99 и 0, което може и да се тълкува като процент. Ако няма поне 50% сигурност в тага той се пропуска.

```
def getCaption(path):
    decrypted_data = str(decryptKeys())
    decrypted_data.replace("'", '"')

    result = json.loads(decrypted_data)
    subscription_key = result["subscription"]

    with open('sas_keys.json', 'w') as f:
        f.write(decrypted_data)
        f.close()

    encryptKeys()

    endpoint = 'https://cvazureapi.cognitiveservices.azure.com/'
    analyze_url = endpoint + "vision/v3.2/analyze?"

    headers = {'Ocp-Apim-Subscription-Key': subscription_key, 'Content-Type': 'application/octet-stream'}
    params_tags = {'visualFeatures': 'Description'}
    with open(path, 'rb') as image:
        data = image.read()

    # Connect to Computer Vision API and get tags
    response_captions = requests.post(analyze_url, headers=headers, params=params_tags, data=data)
    caption = response_captions.json()
    return caption["description"]["captions"][0]["text"]
```

Фиг. 3.7. Извличане на описание

getCaption(path) е близко по логика на getTags(path), като главната разлика е какво се иска и как се връща крайният резултат. Когато се получи отговор от заявката той всъщност е масив, в който се съдържат няколко възможни описания за снимката. Тъй като са подредени по увереност, а не по азбучен ред, може да се вземе първият възможен резултат.

Използването на услугите на Azure става с помощта на ключ, който идва с абонамент. Един от начините за позволяване на достъп до профил, който вече има абонамент, е със Shared Access Signature, или накратко SAS ключ. За да се избегне злоупотреба от потребител с този ключ, както и с връзката към Data Lake хранилището, те са записани в JSON формат и са криптирани. Съхраняват се във файла sas_keys.json.

3.3.1. Криптиране и декриптиране на данни

Защитата на важните данни е направена с помощта на библиотеката Cryptography. От нея се извлича Fernet, което предоставя симетрично криптиране на данни. Това означава, че с помощта на специален ключ може да се манипулират данните, като при всяко криптиране и декриптиране ще получава един и същи резултат всеки път [5]. Служещите за това функции се намират в encrypt.py.

```
def encryptKeys():
    key = Fernet.generate_key()
    with open('key.key', 'wb') as file:
        file.write(key)
        file.close()

    with open('key.key', 'rb') as fkey:
        key = fkey.read()
        fkey.close()

    with open('sas_keys.json', 'rb') as file:
        data = file.read()
        file.close()

    fernet = Fernet(key)
    encrypted = fernet.encrypt(data)

    with open('sas_keys.json', 'wb') as file:
        file.write(encrypted)
        file.close()
```

Фиг. 3.8. Криптиране на ключовете

encryptKeys() създава нов уникален ключ всеки път когато се извика, като въпросният ключ се записва във файла key.key. След като прочете данните от sas_keys.json криптира съдържанието и го записва на мястото на съществуващия текст.

```
def decryptKeys():
    with open('key.key', 'rb') as fkey:
        key = fkey.read()
        fkey.close()

    with open('sas_keys.json', 'rb') as file:
        encrypted_data = file.read()
        file.close()

    fernet = Fernet(key)
    return fernet.decrypt(encrypted_data).decode()
```

Фиг. 3.9. Декриптиране на ключовете

За декриптирането на файла се извиква decryptKeys(), което връща оригиналния текст във формата на String променлива.

3.4. Архивиране

Една функционалност, която липсва на някои от съществуващите реализации, е опцията за съхраняване на снимките. Възможността потребителите да подсигурят файловете си направо от приложението, което използват за качване във фотобанка, е по-удобна от това да търсят алтернативни варианти. С помощта на облачни хранилища това е възможно. Функцията, реализираща архивирането, е uploadBlob(path, zip_name) и се намира във файла blobFunctions.py.

```python
def uploadBlob(path, zip_name):
    decrypted_data = str(decryptKeys())
    decrypted_data.replace("'", '"')

    result = json.loads(decrypted_data)
    connection_string = result["connection"]

    with open('sas_keys.json', 'w') as file:
        file.write(decrypted_data)
        file.close()

    encryptKeys()

    zipFiles(path, zip_name)
    file_name = zip_name+'.zip'

    blob_service_client = BlobServiceClient.from_connection_string(connection_string)
    blob_client = blob_service_client.get_blob_client(container="store", blob=file_name)

    logging.basicConfig(level=logging.DEBUG, filename='logs.log', format='%(asctime)s %(levelname)s:%(message)s')
    logger = logging.getLogger(__name__)

    logger.info("Attempting archiving")

    try:
        with open(os.path.join(path, file_name), "rb") as data:
            blob_client.upload_blob(data, overwrite=True)
    except HttpResponseError:
        logger.warning("Archiving failed")
    finally:
        logger.debug("Archiving done")

    os.remove(path + "/" + file_name)
    messagebox.showinfo("Done", "Archiving done!")
```

Фиг. 3.10. Качване на blob в Data Lake хранилище

Библиотеката azure позволява достъп до контейнер вътре в Data Lake хранилището. В него се качва компресиран файл, съдържащ всички снимки от избраната при отварянето на приложението директория. Този файл се изпраща под формата на blob.

Тъй като се изпраща само един файл няма проблем да се добавят и снимките, чиито размер е повече от 4MB. Името, избрано през съответния прозорец, се подава на функцията за компресиране и след установяване на връзка с хранилището новосъзданеният файл се изпраща. След

приключване на действието компресираният файл се премахва от директорията.

3.4.1. Сваляне на архивирани файлове

Във функцията downloadBlob(path, zip_name) се извършва свалянето на вече архивираните файлове. За да се изпълни правилно потребителят трябва да помни с какво име е качил снимките си в облачното хранилище и да го напише във втория прозорец от точка 3.2.2.

```
def downloadBlob(path, zip_name):
    decrypted_data = str(decryptKeys())
    decrypted_data.replace("'", '"')

    result = json.loads(decrypted_data)
    connection_string = result["connection"]

    with open('sas_keys.json', 'w') as file:
        file.write(decrypted_data)
        file.close()

    encryptKeys()

    blob_name = zip_name+'.zip'
    blob_service_client = BlobServiceClient.from_connection_string(connection_string)
    container = blob_service_client.get_container_client("store")

    logging.basicConfig(level=logging.DEBUG, filename='logs.log', format='%(asctime)s %(levelname)s:%(message)s')
    logger = logging.getLogger(__name__)

    logger.info("Attempting download")

     try:
         with open(os.path.join(path, blob_name), "wb") as file:
             file.write(container.get_blob_client(blob_name).download_blob().readall())
     except HttpResponseError:
         logger.warning("Download failed")
     finally:
         logger.debug("Download done")

     messagebox.showinfo("Done", "Download done!")
```

Фиг. 3.11. Извличане на архивирани файлове от Data Lake хранилище

След осъществяване на връзка с облачното хранилище се взима container client, чрез който може да се достъпят съхраняваните blob обекти.

Създава се нов файл и в него се записва всичката информация, която се сваля от Data Lake.

3.4.2. Компресиране на файлове

С цел избягването на безразборното архивиране на множество снимки, както и драстично намаляване на възможната цена за ползване на услугата, компресирането на изображенията в дадената директория е разумен подход. Освен това по този начин възстановяването на файловете е по-бързо.

Функцията за компресиране е zipFiles(path, name) - минава през подадения път и добавя изображенията от него в ZIP файл, който може да се намери в същата директория.

```
def zipFiles(path, name):
    extensions = ('.JPG', '.jpg', '.JPEG', '.jpeg', '.PNG', '.png')

    with ZipFile(path+'/'+name+'.zip', 'w') as zip_object:
        for file in os.listdir(path):
            if file.endswith(extensions):
                file_path = os.path.join(path, file)
                zip_object.write(file_path, basename(file_path))
    zip_object.close()
```

Фиг. 3.12. Компресиране на файлове

3.5. Качване към FTP сървър

Повечето фотобанки имат критерии, по които определят кой може да качва снимки. Най-често срещаният подход е преди да се качват файлове потребителят да предостави портфолио. Ако отговаря на изискванията, профилът получава одобрение и публикуваните снимки не подлежат на проверка, освен ако съдържанието не нарушава правилата. За

реализирането на проекта беше нужно да се намерят платформи, които не изискват предварително одобрение, а разглеждат снимките една по една. Освен това тези фотобанки трябва да поддържат качване през FTP. Двете платформи, използвани в проекта, са Zoonar и Alamy.

Изпращането към FTP сървърите се реализира в upload(username_input, password_input, path, server_address), което се намира във файла upload.py.

```
def upload(username_input, password_input, path, server_address):
    logging.basicConfig(level=logging.DEBUG, filename='logs.log', format='%(asctime)s %(levelname)s:%(message)s')
    logger = logging.getLogger(__name__)

    # Connect to FTP server
    session = ftplib.FTP(server_address)
    session.login(username_input.get(), password_input.get())

    if server_address == 'upload.alamy.com':
        session.cwd('/Stock')
```

Фиг. 3.13. Свързване с FTP сървър

В началото на функцията се инициализира връзка със сървъра. В зависимост от това кой от двата сайта ще бъде използван се сменя директорията, в която се изпращат снимките, защото при Alamy има няколко възможни категории.

```
logger.info("Image upload started")
try:
    # Gets images from directory
    img_extensions = ('.JPG', '.jpg', '.PNG', '.png')
    for file in os.listdir(path):
        if file.endswith(img_extensions):
            filename, ext = os.path.splitext(os.path.join(path, file))
            os.rename(os.path.join(path, file), filename+'.JPEG')

    for file in os.listdir(path):
        if file.endswith(('.JPEG', '.jpeg')):
            img_path = os.path.join(path, file)

            # Azure API only works with images 4MB and under
            if os.path.getsize(img_path) / (1024 * 1024) < 4:
                addIPTCInfo(img_path)
                session.storbinary('STOR ' + file, open(img_path, 'rb'))
                print("upload "+img_path)
except EOFError:
    logger.warning("Image upload failed")
finally:
    logger.debug("Image upload done")
```

Фиг. 3.14. Качване на изображения в FTP сървър

При качването на снимките се променят файловите разширения, причината за което е обяснена в точка 2.1., както и добавянето на метаданни към изображенията, алгоритмът за което се намира в точка 3.5.1. Всички снимки с размер над 4MB се изключват от качването, защото няма как да получат ключови думи или описание. Тъй като библиотеката ftplib предлага изпращане само под бинарна форма или тази на текст [7], методът storbinary() е използван за предаването към сървъра.

```
# Upload IPTC data to FTP server
logger.info("Metadata upload started")
try:
    iptc_extensions = ('.JPEG~', '.jpeg~')
    for file in os.listdir(path):
        if file.endswith(iptc_extensions):
            iptc_path = os.path.join(path, file)
            session.storbinary('STOR ' + file, open(iptc_path, 'rb'))
            print("upload meta "+iptc_path)
except EOFError:
    logger.warning("Metadata upload failed")
finally:
    logger.debug("Metadata upload done")
    messagebox.showinfo("Done", "Upload done!")

session.quit()
```

Фиг. 3.15. Качване на метаданни в FTP сървър

В последната част на функцията се добавят и изпращат метаданните, които се записват под формата на JPEG~ файл.

3.5.1. Добавяне на метаданни към изображения

Всички фотобанки се нуждаят от ключови думи и описания на снимките, по този начин се улеснява намирането и продаването на изображения. При качването на файлове през сайтовете или приложенията на фотобанките включването на тагове е лесно, но нещата се усложняват

ако вместо това се избере изпращане по FTP сървър. Едно решение на този проблем е добавянето на метаданни към снимките, предвидени за изпращане.

```
def addIPTCInfo(img_path):
    temp_tag_arr = []

    tag_arr = getTags(img_path)
    for t in tag_arr:
        temp_tag_arr.append(t.name)

    img_data = IPTCInfo(img_path, force=True)
    img_data['keywords'] = []
    img_data['keywords'] = temp_tag_arr

    img_data['caption/abstract'] = []
    img_data['caption/abstract'] = [getCaption(img_path), "stock photo"]

    img_data.save()
```

Фиг. 3.16. Добавяне на метаданни към изображения

В getImageData.py се намира и функцията addIPTCInfo(img_path), която добавя нова информация към файла, съдържащ метаданни за снимката. Ако няма такъв файл ще се създаде нов, в който да се налеят данните. За да се избегне възможното обърквaне на метаданните, преди да се добавят съответната категория се изчиства от друга информация.

4. ГЛАВА ЧЕТВЪРТА – Ръководство на потребителя

4.1. Инсталиране на библиотеки

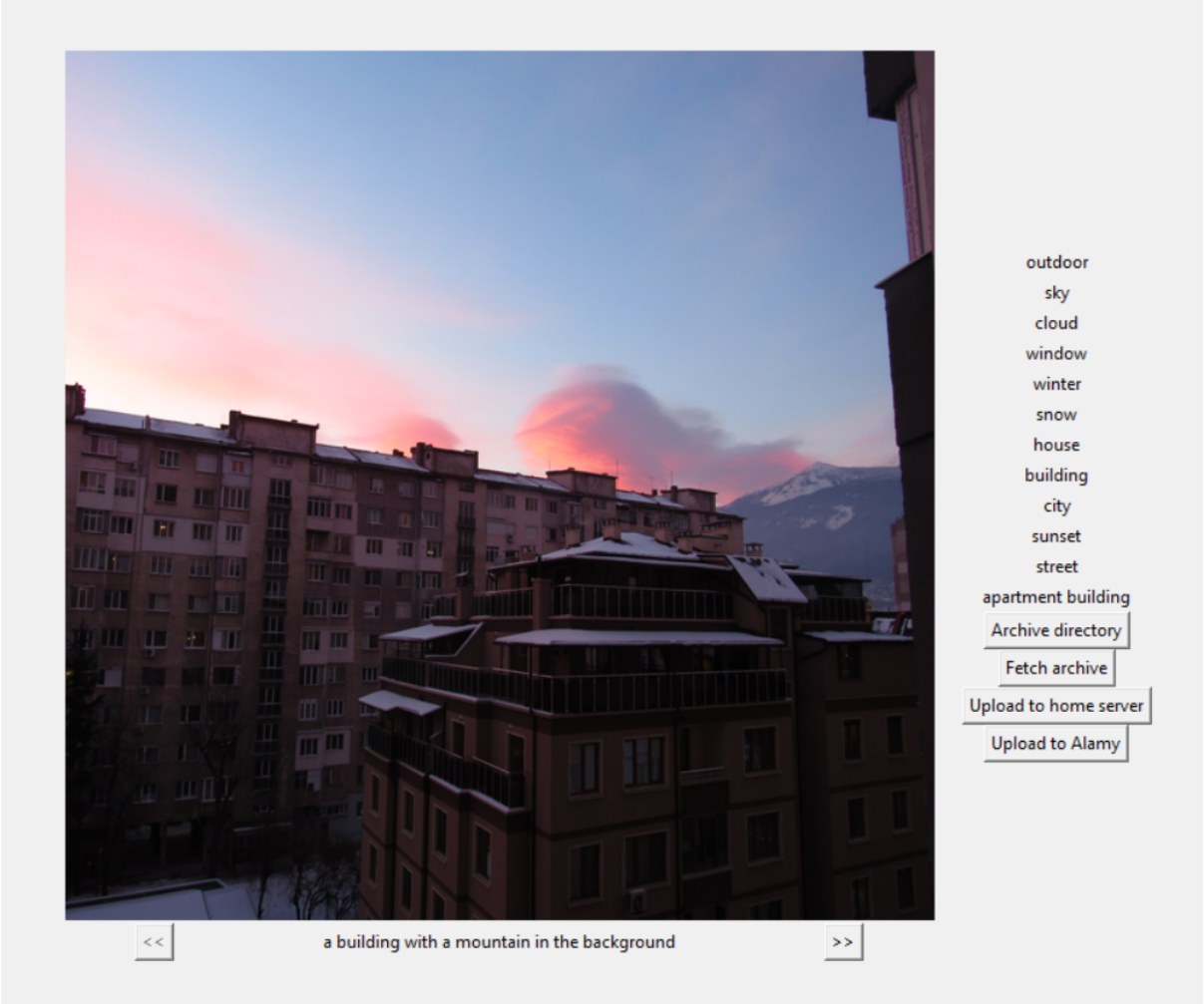
За да функционира проектът се нуждае от Python 3.9, както и няколко външни библиотеки, които могат да се инсталират по следния начин:

- pip install IPTCInfo3, което е нужно за промяната и добавянето на метаданни
- pip install azure-storage-blob, което е нужно за достъп до blob контейнера
- pip install requests, което е нужно за комуникация с Azure Computer Vision
- pip install Pillow, което е нужно за визуализиране и манипулиране на изображения
- pip install cryptography, което е нужно за разчитане и криптиране на SAS ключа и адреса на хранилището

Стартирането на приложението става с изпълнение на файла mainWindow.py.

4.2. Използване на приложението

След стартиране на приложението се отваря Tk прозорец, където може да се избере директорията, с която ще се работи. Това се случва с натискането на бутонът с текст “Choose directory”. Когато действието приключи се появява първата снимка, заедно с описание, ключови думи и бутони за операция.



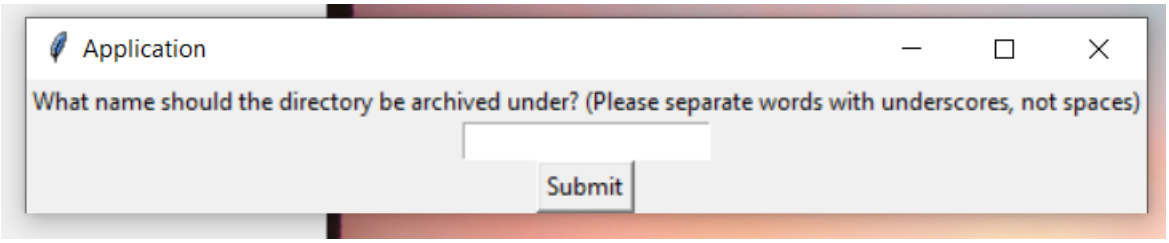
Фиг. 4.1. Начален екран след зареждане на директория

4.2.1. Навигиране между снимки

За да се минава през изображенията се използват бутоните от двете страни на описанието, маркирани с “<<” и “>>”. Ако сегашната снимка е първата в заредената директория, бутонът за минаване назад (“<<”) е блокиран. Същото важи и в случая, че се разглежда последното възможно изображение, но тогава се блокира минаването напред.

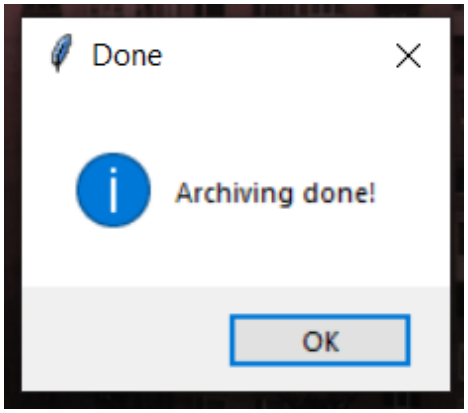
4.2.2. Архивиране

Точно под таговете за снимката седи бутонът “Archive directory”. При неговото натискане на екранът се появява прозорецът за избиране на името на архива.



Фиг. 4.2. Прозорец за именуване на архив

Натискането на бутонът “Submit” инициализира архивирането. Необходимо е малко да се изчака, за да се извърши функцията. При успешно приключване се появява уведомление, след което потребителят може спокойно да затвори всички прозорци, с които няма да работи повече.

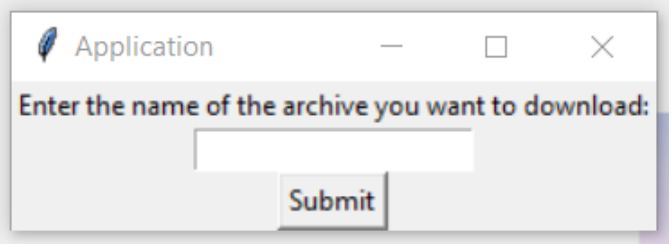


Фиг. 4.3. Уведомление за успешно архивиране

4.2.3. Сваляне на архивирани файлове

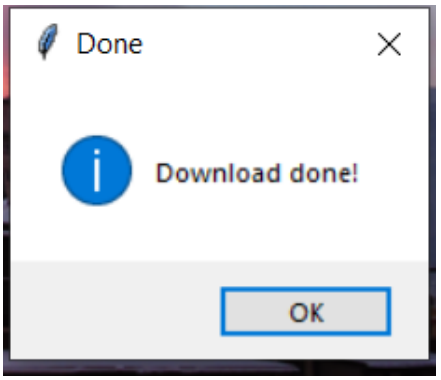
След като поне веднъж файлове са архивирани потребителят може да свали снимките си от облачното хранилище. За да се направи това трябва да се натисне бутона с текст “Fetch archive”. Това ще изкара

прозорец на екрана, в който може да се въведе името на желания за сваляне архив.



Фиг. 4.4. Прозорец за сваляне на архивирани файлове

Когато функцията приключи действие се показва известие. Свалените файл може да бъде намерен в директорията, която е отворена от приложението.



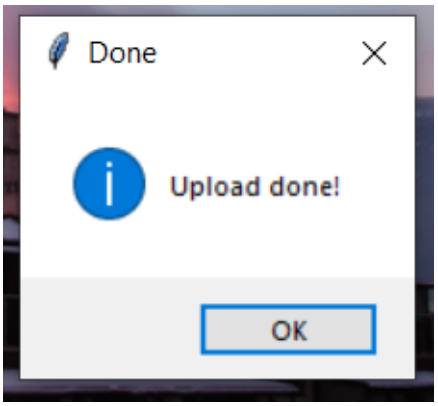
Фиг. 4.5. Уведомление за успешно сваляне

4.2.4. Качване към FTP сървър

Освен архивиране, потребителят има възможност да качи снимките и метаданните от директорията към FTP сървъра на Zoonar или Alamy. С зависимост от избраната платформа ще се покаже различен прозорец за влизане в системата.

Фиг. 4.6. Прозорец за влизане в Alamy (ляво) и Zoonar (дясно)

Процесът по пращане на файловете и добавянето на метаданните отнема повече време от архивирането, тъй като е по-сложен. Възможно е приложението да отиде в режим на Not Responding, но това не е притеснително. Ако потребителят изчака функцията да се изпълни докрай на екрана ще се появи известие.



Фиг. 4.7. Известие за успешно качване на файловете

Заключение

Дипломната работа реализира всички изисквания. Успешно са имплементирани функционалности за добавяне на ключови думи и описания към изображения, архивиране и качване към фотобанки, запазване на потребителска активност, манипулиране на файлови разширения, както и графичен потребителски интерфейс.

Едно от нещата, които се нуждаят от подобрение, е съхраняването на ключовете. Използването само на криптиране, и то такова от библиотека, не пази данните достатъчно добре. Този проблем може да бъде опрваен с помощта на Azure Key Vault, което е облачно хранилище, специално направено за пазенето на данни.

Използвана литература

[1] Tamás Gulácsi, IPTCInfo Documentation, https://github.com/20minutes/iptcinfo/blob/master/iptcinfo.py, June 2012

[2] John E. Grayson, Python and Tkinter Programming, Manning Publications, January 2000, p. 14

[3] What is Computer Vision?, https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/overview

[4] Introduction to Azure Blob storage, https://docs.microsoft.com/en-us/azure/storage/blobs/storage-blobs-introduction

[5] Symmetric Cryptography, https://www.sciencedirect.com/topics/computer-science/symmetric-cryptography

[6] What is IPTC metadata? Everything you need to know, https://smartframe.io/blog/what-is-iptc-metadata-everything-you-need-to-know/

[7] ftplib – FTP protocol client, https://docs.python.org/3/library/ftplib.html

[8] What is Image Analysis?, https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/overview-image-analysis

[9] Object detection - Computer Vision, https://docs.microsoft.com/en-us/azure/cognitive-services/computer-vision/concept-object-detection

[10] Microsoft Azure Data Lake, https://www.techtarget.com/searchcloudcomputing/definition/Microsoft-Azure-Data-Lake

[11] What Is Hadoop Yarn Architecture & It's Components, https://www.upgrad.com/blog/what-is-hadoop-yarn-architecture-its-components/

[12] A tour of the C# language, https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[13] What is Python? Executive Summary, https://www.python.org/doc/essays/blurb/

[14] What Is Cloud Storage? Definition, Types, Benefits, and Best Practices, https://www.toolbox.com/tech/cloud/articles/what-is-cloud-storage/

[15] SaaS vs PaaS vs IaaS: What's The Difference & How To Choose, https://www.bmc.com/blogs/saas-vs-paas-vs-iaas-whats-the-difference-and-how-to-choose/

[16] Image analysis, https://en.wikipedia.org/wiki/Image_analysis

[17] Azure Storage Pricing, https://azure.microsoft.com/en-us/pricing/details/storage/blobs/

[18] AWS Storage Pricing, https://aws.amazon.com/s3/pricing/

[19] GCP Storage Pricing, https://cloud.google.com/storage/pricing#europe

[20] Azure Computer Vision Pricing, https://azure.microsoft.com/en-us/pricing/details/cognitive-services/computer-vision/

[21] GCP Google Vision Pricing, https://cloud.google.com/vision/pricing

Съдържание