

AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki, Elektroniki i Telekomunikacji
Katedra Informatyki



Projekt z przedmiotu Inżynieria Oprogramowania
Aplikacja do synchronizowania repozytoriów
Nazwa kodowa: Git-sync

Krzysztof Bieniasz, Rafał Juraszek,
Adrian Maciej, Benedykt Roszko

KIERUNEK: Informatyka

Opis problemu	4
Charakterystyka problemu	4
Motywacja projektu	4
Wizja rozwiązania	5
Wizja produktu	5
Analiza zagrożeń	6
Koncepcja systemu	7
Studium wykonalności:	8
Diagram sekwencji dla obsługi synchronizacji repozytoriów	8
Diagram sekwencji dla obsługi notifikatora mailowego	10
Dokumentacja developerska	11
Backend	11
Synchronizer	11
Pulling	13
Pushing	13
REST API	14
POST /api/addRepo	14
PUT /api/modifyRepo	15
GET /api/repos	15
DELETE /api/repos/<id>	16
POST /api/notify	16
Notificator	17
Scrapper	17
Klasa Scrapper	17
Email client	18
Plik notifikator.py	18
Baza danych	18
Diagram	19
Struktura bazy danych	19
Plik database_handler.py	20
Management Panel	21
Electron	21
Angular	21
Przewodnik użytkownika	22
Istota działania aplikacji	22
Instrukcja uruchomienia aplikacji	22
Serwer	22
Instrukcja korzystania z poszczególnych funkcjonalności i ekranów	24

Ekran startowy	24
Dodanie synchronizacji dla nowego repozytorium	25
Modyfikacja ustawień synchronizacyjnych dla repozytorium	28
Wysyłanie powiadomień	29

Opis problemu

Charakterystyka problemu

W dzisiejszych czasach repozytoria przeznaczone do przechowywania cyfrowych zasobów są dla wielu z nas (szczególnie w branży IT) nierozłącznym elementem życia codziennego. Bez nich synchronizacja pracy wielu osób w zespole byłaby o wiele cięższa. Co się jednak stanie jeżeli z jakiegoś powodu dostęp do naszego repozytorium na danym serwerze zostanie zablokowany? Wtedy ani my nie możemy kontynuować pracy nad produktem, ani potencjalni klienci (jeżeli repozytorium jest publiczne) nie mogą z niego korzystać. Rozwiązaniem, które nasuwa się jako pierwsze jest przechowywanie naszych zasobów na większej ilości repozytoriów na innych serwerach. Gdy dostęp do jednego z nich zostanie zablokowany, mamy wtedy wciąż dostęp do innych repozytoriów na innych serwerach. Wtedy jednak powstaje nowy problem - jak zsynchronizować to co jest na naszym głównym repozytorium z pozostałymi "zapasowymi"? Odpowiedź na to pytanie z pewnością nie jest już trywialna. Możliwości jest wiele, jednak z pewnością widać, iż takie rozwiązanie jest potrzebne i z pewnością przydałoby się do codziennej pracy, w której przechowywanie zasobów w repozytoriach jest czynnością rutynową.

Motywacja projektu

Z racji popularności publicznych repozytoriów istnieją już pewne rozwiązania zabezpieczające przed utratą dostępu do repozytorium. Jednak często jedyną oferowaną przez nich funkcjonalnością jest cykliczne tworzenie kopii zapasowej lokalnie na komputerze albo ewentualnie w przestrzeni chmurowej. Nie znaleźliśmy jednak popularnych rozwiązań, które by synchronizowały publiczne repozytoria umieszczone na różnych publicznych serwerach, tak aby w razie utraty dostępu do jednego z nich użytkownicy mogli automatycznie korzystać z innych repozytoriów. Po analizie popularnych serwisów oferujących przechowywanie kodu stwierdziliśmy, iż w sporej części dotychczasowych rozwiązań problemu jest bardzo niewielkie wykorzystanie interfejsów programistycznych aplikacji, które te serwisy udostępniają. W tworzeniu produktu warto też zwrócić uwagę na korzyści jakie mogą odnieść beneficjentami aplikacji nie będzie tylko sam właściciel repozytorium, ale osoby, które z niego korzystają w codziennej pracy. Wykorzystując nasz produkt właściciel repozytorium będzie mógł zapewnić innym użytkownikom ciągłą dostępność do kodu

źródłowego. Motywacją do projektu może być zatem, stworzenie kompleksowego narzędzia, którego jeszcze nie oferuje rynek i które mogłoby zyskać istotną popularność.

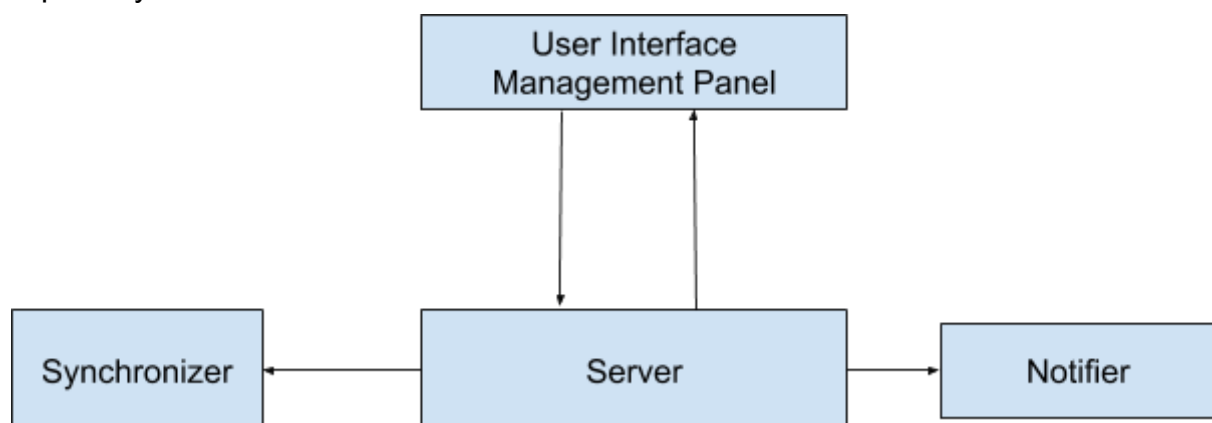
Wizja rozwiązania

Wizja produktu

Celem naszego projektu jest stworzenie narzędzia pozwalającego na automatyczną synchronizację danych na różnych publicznych repozytoriach z danymi z repozytorium na platformie GitHub. Podjęliśmy decyzję, że najlepszym rozwiązaniem jest aby nasz projekt miał postać procesu działającego w tle na komputerze użytkownika oraz prostej graficznej aplikacji do zarządzania parametrami synchronizacji.

Użytkownik posiada prostą aplikację z interfejsem graficznym, gdzie jest w stanie ustalić gdzie chce przechowywać kopie zapasowe określonego przez siebie repozytorium na platformie GitHub. Będzie on również w stanie podać parametry synchronizacji takie jak częstotliwość. Pozwoli też na podejrzenie logów wygenerowanych przez Synchronizer. W przypadku zablokowania publicznego dostępu do naszego repozytorium, będzie możliwość poinformowania wszystkich użytkowników, którzy korzystali z naszego repozytorium, o tym gdzie znajduje się najnowsza publiczna wersja, poprzez notyfikację wiadomością e-mail.

Synchronizer jest to proces działający w tle, który odpowiada za automatyczną synchronizację danych co określony okres czasu. Korzystając z mechanizmu kontroli wersji GIT pobiera on najnowsze zmiany z głównego repozytorium do repozytorium lokalnego i wysyła je na repozytoria będące kopiami zapasowymi.



Analiza zagrożeń

Proces tworzenia systemu może wiązać się z następującymi problemami:

- **Obciążenie zasobów klienta** - im częściej dane będą synchronizowane między repozytoriami i im więcej danych będzie synchronizowanych, tym większe będzie obciążenie łącza. Również źle zaprojektowany proces działający w tle będzie stałym obciążeniem zasobów procesora.
- **Łączność między repozytoriami** - każde repozytorium ma własne zasady przetrzymywania danych, co może powodować niekompletną kompatybilność w przesyłaniu (w szczególności metadane). Wszystkie dane nie wspierane przez system kontroli wersji GIT takie jak gwiazdki na platformie GitHub mogą nie zostać zapisane w kopiach zapasowych
- **Zagrożenie organizacji pracy związane z pandemią Coronavirusa** - aktualna sytuacja na świecie związana z pandemią Coronavirusa negatywnie wpływa na organizację pracy i możliwość wspólnego spotkania, które umożliwiłoby sprawniejsze wytwarzanie systemu

Koncepcja systemu

Nasz system będzie się składał z trzech głównych modułów.

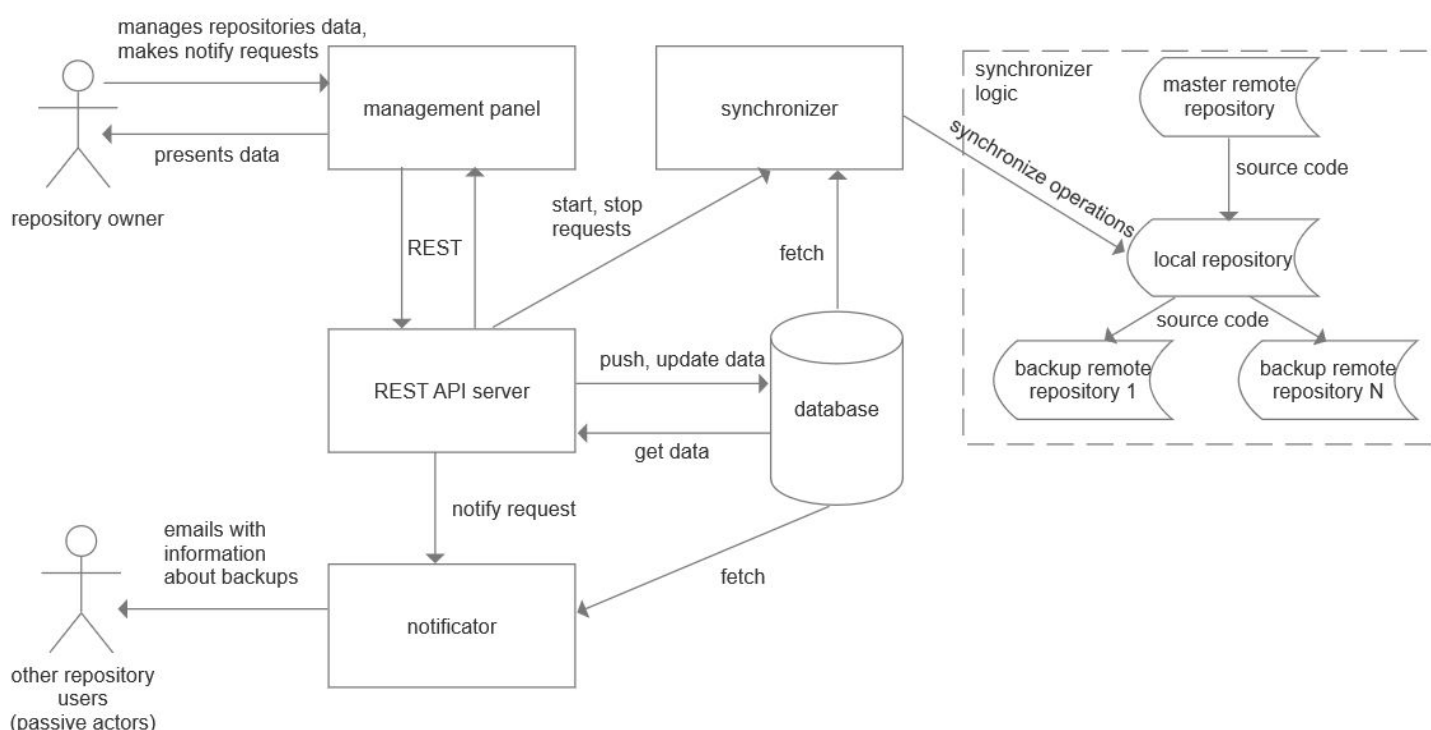
Moduł panelu sterowania pozwoli na zarządzanie całą aplikacją. Użytkownik będzie mógł wprowadzać dane dotyczące repozytoriów (adresy, nazwy, hasła) oraz wszelkie parametry konfigurujące synchronizację (częstotliwość, pora itp). Następnie informacje te zostaną zapisane w odpowiedniej hurtowni danych (pliki JSON lub baza danych). Po skończonej konfiguracji użytkownik będzie miał możliwość uruchomienia modułu synchronizującego. Dodatkowo, w przypadku utraty dostępu do repozytorium głównego, panel sterowania pozwoli na komunikację z notyfikatorem w celu powiadomienia innych użytkowników.

Moduł synchronizujący zapewni stałą aktualizację repozytoriów zapasowych. Wszelkie potrzebne informacje będą pobierane z hurtowni danych. Cykl działania tego modułu prezentuje się następująco:

- pobranie zmian z repozytorium głównego
- odczytanie informacji na temat przypisanych repozytoriów zapasowych
- uaktualnienie tych repozytoriów

Moduł notyfikatora umożliwi powiadamianie wiadomościami **e-mail** zainteresowanych użytkowników (współpracownicy lub osoby dające gwiazdkę) o utracie dostępu do repozytorium głównego oraz prześle informacje na temat lokalizacji repozytorium zapasowego. Komponent ten uruchamiany będzie przez panel sterowania.

Za warstwę pośredniczącą w komunikacji między modułami, będzie odpowiadał serwer REST API, który będzie bezpośrednio połączony z bazą danych.



Studium wykonalności:

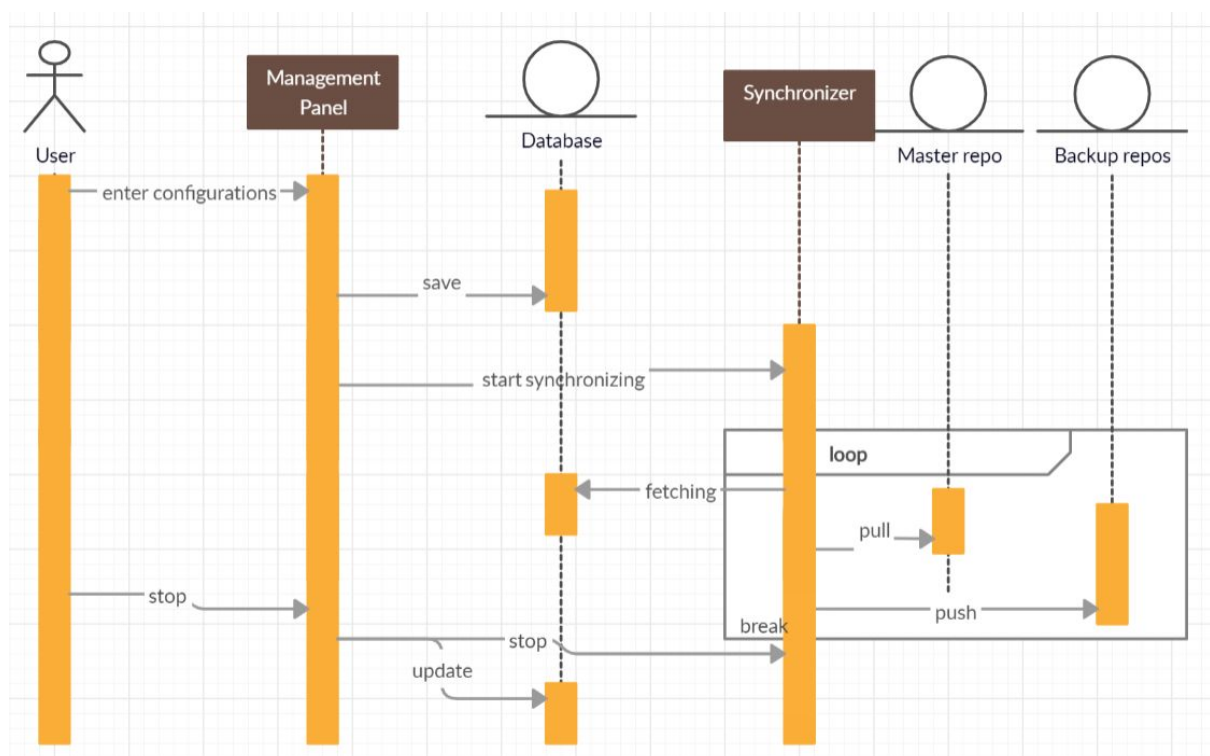
Technologie, które wybraliśmy na potrzeby wykonania projektu to Python3 (w tym framework Flask jako serwer REST API) oraz Angular.

Angular jako frontendowy framework idealnie sprawdzi się w tworzeniu User Interface - w naszej aplikacji będzie to panel zarządzania skąd użytkownik będzie sterował działaniem aplikacji.

Python3 zostanie wykorzystany do modułu zarządzającego synchronizacją zdalnych repozytoriów. Wybraliśmy w tym celu Pythona, ponieważ znaleźliśmy przydatną bibliotekę - GitPython, przy pomocy której z poziomu skryptu możemy operować na podstawowych komendach gita. Będzie ona bardzo przydatna w naszym rozwiązaniu.

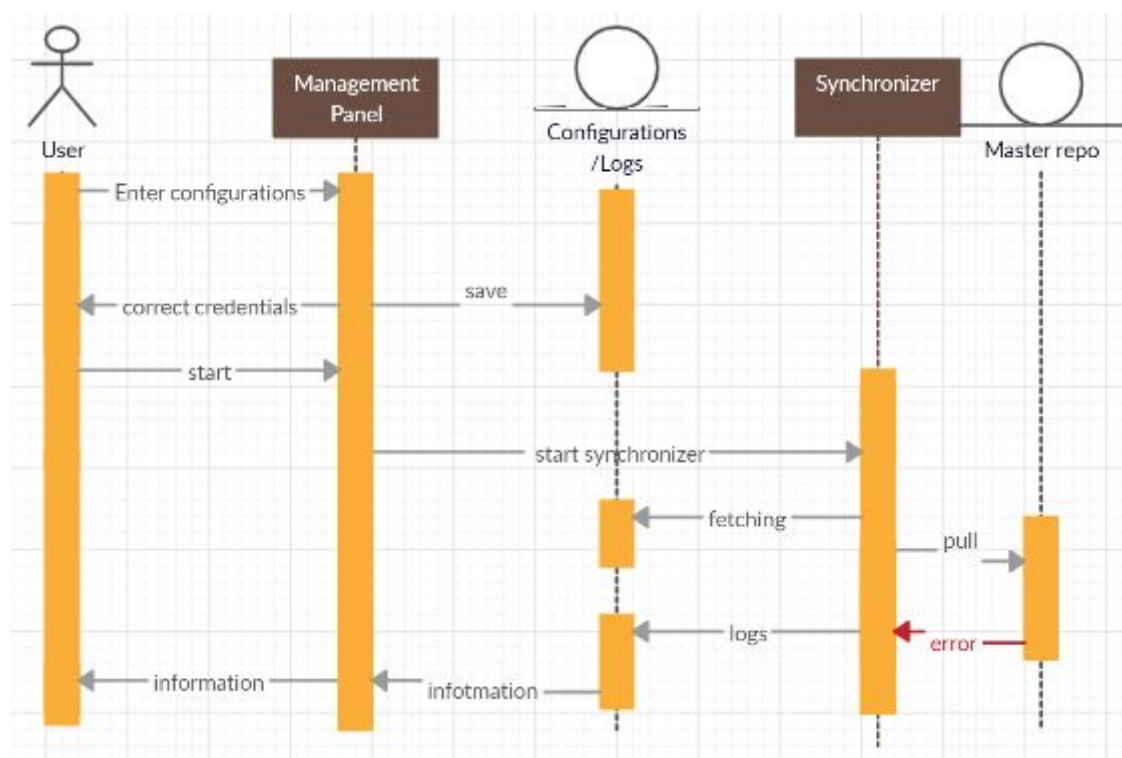
Do realizacji modułu wysyłającego notyfikacje mailowe również wykorzystamy Pythona oraz wbudowany w niego pakiet smtplib pozwalający wysyłać maile z poziomu programu Pythona.

Diagram sekwencji dla obsługi synchronizacji repozytoriów



Prawidłowy przebieg:

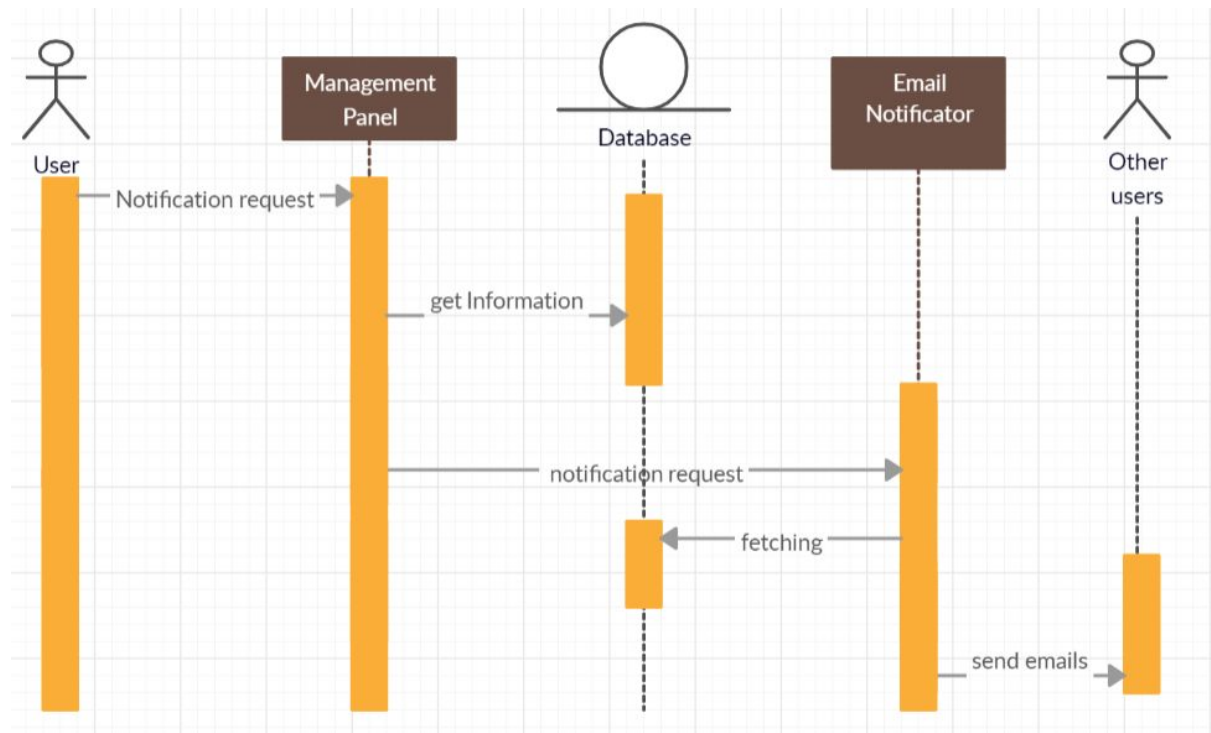
- User wpisuje potrzebne konfiguracje do modułu Management Panel
- Management Panel zapisuje wprowadzone informacje do hurtowni danych Configurations i zwraca komunikat o powodzeniu do Usera
- User poprzez Management Panel startuje moduł Synchronizer
- Synchronizer pobiera dane inicjalizujące z hurtowni Configurations
- Synchronizer wykonuje cyklicznie następujące kroki (w zależności od wybranej częstotliwości i pory)
 - Pobranie najnowszych zmian z Master Repo
 - Uaktualnienie repozytoriów zapasowych (Backup Repos) nowo pobranymi danymi
- User poprzez moduł Management Panel stopuje działanie modułu Synchronizer



Przebieg z wystąpieniem błędu:

- Do momentu wystąpienia błędu przebieg operacji jest analogiczny jak w opisanym wyżej przykładzie.
- Błąd może wystąpić podczas pobierania repozytorium z serwera internetowego, ponieważ może ono zostać usunięte i wtedy nie będzie można wykonać prawidłowej operacji
- Moduł synchronizera otrzymuje informację o błędzie i odkłada ją w logach
- Moduł panelu zarządzania pobiera informacje z logów i przekazuje ją klient

Diagram sekwencji dla obsługi notifikatora mailowego



- User wysyła "Notification Request" do modułu Management Panel wraz z informacją o lokalizacji nowego repozytorium głównego
- Management Panel przekierowuje żądanie do modułu Email Notificator
- Email Notificator pobiera potrzebne informacje z hurtowni danych Configurations, a następnie wysyła powiadomienia e-mail do zainteresowanych użytkowników (Other users)

Dokumentacja developerska

Backend

Część backendowa naszej aplikacji została stworzona w języku Python i używa najnowszej obecnie wersji Pythona 3.8. Pod względem struktury kod składa się z trzech głównych pakietów: synchronizer, notificador oraz database_maintenance. Backend jest uruchamiany poprzez uruchomienie pliku app.py, w którym jest tworzona i uruchamiana instancja klasy Flask.

Synchronizer

Synchronizer działa cyklicznie tak aby niezależnie od swojej długości czasu wykonania, sam proces synchronizacji uruchamiał się w równych odstępach czasu. Jeżeli będzie trwał dłużej niż ustalony okres, to po ukończeniu jednego procesu synchronizacji rozpocznie się kolejny. Dla każdego synchronizowanego repozytorium tworzony jest nowy wątek odpowiedzialny za uruchomienie procesu synchronizacji.

W każdym wątku, co wartość pola `check_if_alive_period` w sekundach (10s w wersji wstępnej) sprawdzany jest stan eventu zamykającego, który jest sposobem na prawidłowe zakończenie pracy wątku.

Każdy wątek oraz referencja do jego eventu zamykającego przechowywane są jako krotki w słowniku `threads` będącym polem klasy `Synchronizer` posiadającego za klucz ID repozytorium (string).

W każdym wątku, przy kolejnym uruchomieniu procesu synchronizacji, pobierane są na nowo dane z bazy danych opisujące repozytorium obsługiwane przez ten wątek. Używane są do stworzenia obiektu klasy `SyncRepository` obsługującego logikę procesu synchronizacji przy użyciu biblioteki `Gitpython`. Następnie na tak utworzonym obiekcie wywołana jest metoda `synchronize_all`.

`SyncRepository` poza konstruktorem posiada między innymi 3 metody:

Nazwa	Wartość zwracana	Parametry	Opis
<code>initialize</code>	<code>self</code>	<code>(self, local_path, login, password)</code>	inicjalizuje obiekt, z którego została wywołana, jeżeli załadowane repozytorium było bare repository, zostanie rzucony wyjątek <code>ImportError</code>
<code>create</code>	<code>self</code>	<code>(self, origin, local_path, login, password)</code>	klonuje synchronizowane repozytorium do podanej ścieżki i wywołuje metodę <code>initialize</code>
<code>end_synchronization</code>	<code>None</code>	<code>(self, repo_id)</code>	ustawia event zamykający dla pętli

_loop			synchronizującej repozytorium o danym repo_id
-------	--	--	---

SyncRepository posiada 3 pola:

Nazwa	Typ	Opis
localRepo	git.Repo	obiekt przechowujący załadowane repozytorium
origin	lista z 3 polami: git.remote.Remote, str, str	Obiekt reprezentujący zdalne repozytorium będące źródłem synchronizacji, login i hasło
remotes	lista list z 3 polami: git.remote.Remote, str, str	Lista obiektów reprezentujących zdalne repozytoria będące celem synchronizacji, loginy i hasła

Proces synchronizacji sprowadza się do fazy pulling-u i pushing-u (analogicznie do komend git).

Pulling

1. Zapamiętywana jest aktualna lista gałęzi
git branch
2. Dla każdej gałęzi na repozytorium źródłowym wywołana jest komenda
git checkout <branch> --force
co powoduje przełączenie się na tą gałąź, bądź pobranie jej, jeżeli nie była jeszcze utworzona lokalnie
3. Spośród zapamiętanej listy gałęzi usuwamy tą, którą właśnie obsługujemy
4. Dokonujemy pobrania aktualnej wersji gałęzi poprzez
git fetch --prune
git rebase
git submodule update --recursive
Ponieważ założeniem jest, że na lokalnym repozytorium nie będą wprowadzane zmiany, powyższy ciąg komend jest w stanie bez błędu zaktualizować stan repozytorium.
5. Po zaktualizowaniu wszystkich gałęzi, lista, którą na początku stworzyliśmy zawiera niepotrzebne (usunięte z repozytorium źródłowego) gałęzie, usuwamy je komendą
git branch -D <branch>

Jeżeli operacja Pulling się nie powiedzie, nie będzie przeprowadzona operacja Pushing.

Pushing

1. Zapamiętywana jest aktualna lista gałęzi
`git branch`
2. Iterujemy po liście remotes
3. Dla każdej pozycji budujemy string URL postaci
`https://login:hasło@url_zdalnego_repozytorium.git`
4. Usuwamy, na podstawie aktualnej listy gałęzi, stare gałęzie ze zdalnego repozytorium
`git ls-remote show --heads <remote.name>`
Ta komenda zwraca nam listę gałęzi na zdalnym repozytorium. Te, których nie ma lokalnie usuwamy
`git push <URL> --delete <branch>`
5. Aktualizujemy wszystkie gałęzie na zdalnym repozytorium poprzez
`git push <URL> --all --force`

Wszelkie informacje logowane są do pliku `synchronizer_log.log` znajdującego się w `server/synchronizer`.

REST API

Do obsługi zapytań REST wykorzystujemy framework Flask. Obsługa poszczególnych endpointów jest realizowana w pliku `app.py`.

POST /api/addRepo

Tworzy w bazie danych wpisy dotyczące repozytorium głównego oraz repozytoriów zapasowych.

Przykład poprawnego formatu body z obsługiwanej wiadomości:

```
{
  "id": "masterRepo",
  "url": "https://github.com/gitsync-tester/masterRepo",
  "login": "gitsync-tester",
  "password": "Gitsync1",
  "path": "C:\\Users\\kbien\\Downloads\\exampleRepo",
  "backups": [
    {
      "url": "https://github.com/gitsync-tester/backupForEx2",
      "login": "gitsync-tester",
      "password": "Gitsync1"
    }
  ],
  "frequency": 60
}
```

Wartość zwracana w przypadku sukcesu: []

Wartości błędów:

- brak możliwości dodania rekordu z powodu warunków integralności bazy danych - status 400, format wiadomości:

```
{
  "message": "error on database description"
}
```

PUT /api/modifyRepo

W zależności od otrzymanej wiadomości zmienia w bazie danych częstotliwość synchronizacji głównego repozytorium lub/oraz dodaje lokalizacje zapasowych repozytoriów.

Przykład poprawnego formatu body z obsługiwanej wiadomości:

```
{
  "id": "exampleRepo",
  "backups": [
    {
      "url": "https://github.com/gitsync-tester/backupForEx2",
      "login": "gitsync-tester",
      "password": "Gitsync1"
    }
  ],
  "frequency": 70
}
```

Wartość zwracana w przypadku sukcesu: []

GET /api/repos

Zwraca listę wszystkich repozytoriów głównych wraz z repozytoriami zapasowymi znajdującymi się w bazie danych.

Przykładowa zwracana wartość:

```
[
  {
    "backups": [
      {
        "login": "gitsync-tester",
        "password": "Gitsync1",
        "url": "https://github.com/gitsync-tester/backupForEx2"
      }
    ],
    {
```

```

    "login": "gitsync-tester",
    "password": "Gitsync1",
    "url": "https://github.com/gitsync-tester/backupRepo"
  },
  "frequency": 70,
  "id": "masterRepo",
  "login": "gitsync-tester",
  "password": "Gitsync1",
  "path": "C:\\Users\\kbien\\Downloads\\exampleRepo",
  "url": "https://github.com/gitsync-tester/masterRepo"
}
]

```

DELETE /api/repos/<id>

Parametr id jest nazwą repozytorium głównego, którego dotyczyć będzie zmiana na bazie danych. W przypadku, gdy żądanie zostało wysłane z pustym body, wtedy usuwamy z bazy danych główne repozytorium oraz wszystkie powiązane z nim repozytoria zapasowe. W przypadku, kiedy żądanie zawiera nie zawiera pustego body, to body powinno zawierać url z nazwą repozytorium, które należy usunąć.

Przykład żądania mającego usunąć repozytorium zapasowe dla repozytorium głównego "masterRepo":

DELETE <http://127.0.0.1:5000/api/repos/masterRepo>

Body:

```

{
  "url": "https://github.com/gitsync-tester/backupRepo"
}

```

Przykład żądania mającego usunąć repozytorium główne "masterRepo" oraz powiąane z nim repozytoria zapasowe:

DELETE <http://127.0.0.1:5000/api/repos/masterRepo>

Body: puste

POST /api/notify

Wywołuje funkcje notify z notify.py z nazwą repozytorium głównego, które przyszło w zapytaniu.

Przykład poprawnego body requestu:

```

{
  "id": "masterRepo"
}

```

Notificator

Moduł obsługujący funkcjonalność wysyłania powiadomień mailowych potencjalnie zainteresowanym osobom o lokalizacjach zapasowych repozytoriów.

Scraper

W przypadku, kiedy główne repozytorium pochodzi z serwisu github.com korzystamy z możliwości API githuba i pozyskujemy adresy mailowe użytkowników serwisu, którzy zaznaczyli gwiazdkę danemu repozytorium lub byli bezpośrednimi współpracownikami w tworzeniu repozytorium.

Klasa Scraper

Pola klasy:

Adres repozytorium	repo_url
Nazwa repozytorium w systemie	master_repo_id

Metody klasy:

Nazwa metody	Parametry	Wartość zwracana
__init__	(self, repo_url, master_repo_id)	Obiekt Scraper
scrap_associated	(self)	słownik {nazwa użytkownika na githubie : adres mailowy}

Działanie metody scrap_associated:

1. z urla repozytorium wydobywamy nazwę użytkownika i nazwę repozytorium
2. Odpytując github żadaniami:
`https://api.github.com/repos/<user_name>/<repo_name>/stargazers,`
`https://api.github.com/repos/<user_name>/<repo_name>/contributors`
otrzymujemy JSON-y zawierające loginy użytkowników, którzy dali gwiazdkę lub są kontrybutorami.
3. Mail użytkownika można wyłuskać za pomocą GET `/user/public_emails` w sytuacji, gdy użytkownik ma publicznie podany email na profilu. Często jednak zdarza się tak, iż jedyną metodą jest wydobycie emaila poprzez `https://api.github.com/users/<login>/events/public` - historię publicznych działań użytkownika. Dokładniej ten sposób jest opisany danym poniżej artykule:

<https://www.sourcecon.com/how-to-find-almost-any-github-users-email-address/>

4. W przypadku braku uzyskania maila nie dodajemy użytkownika do słownika wynikowego

Dokumentacja API githuba

<https://developer.github.com/v3/>

Email client

Moduł umożliwiający wysłanie wiadomości email do wskazanych odbiorców.

Plik notificator.py

Funkcje:

Nazwa funkcji	Parametry	Wartość zwracana
notify	(master_repo_id)	brak

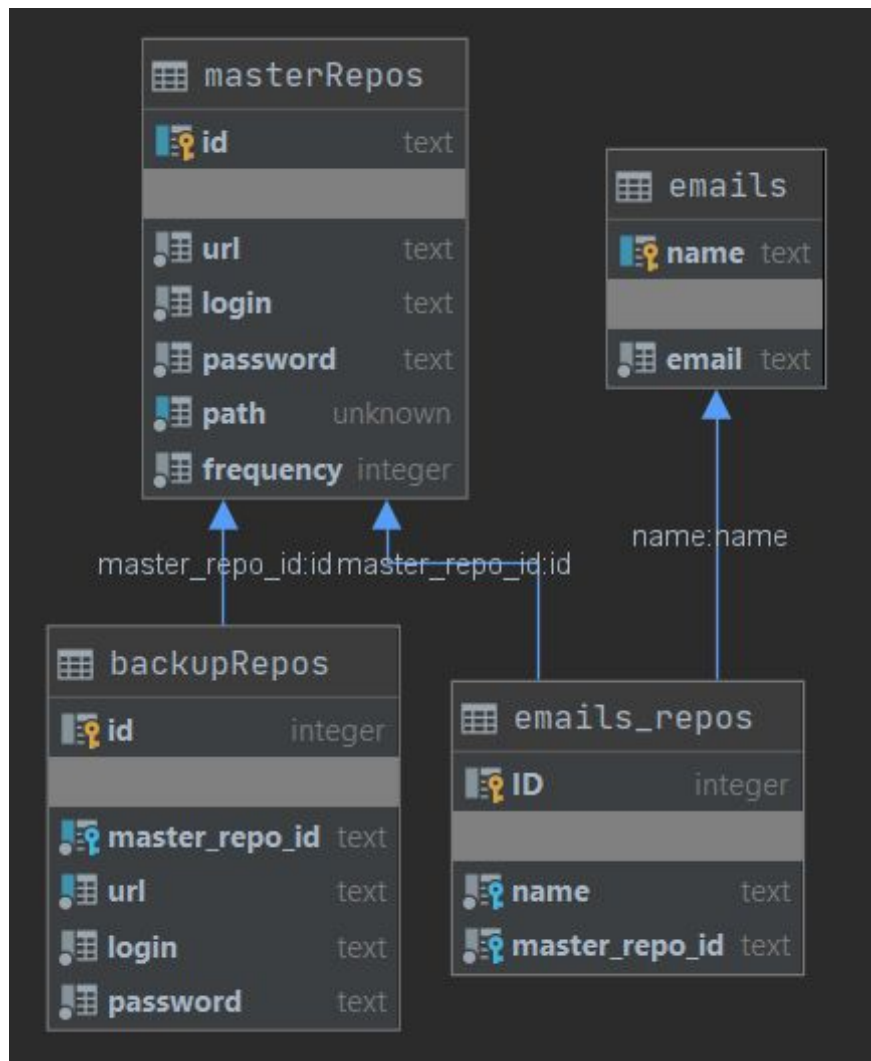
Działanie funkcji notify:

1. Obecnie wiadomości wysyłane są z adresu projekt.io.git@gmail.com założonego specjalnie w tym celu.
2. Maile są wysyłane z wykorzystaniem pakietu smtplib. Artykuł tłumaczący działania pakietu: <https://realpython.com/python-send-email/>
3. Klasa Scrapper jest wykorzystana do pobrania słownika maili zainteresowanych osób.
4. Z bazy danych pobierane są lokalizacje zapasowych repozytoriów.
5. Z pomocą obiektu MIMEMultipart tworzona jest wiadomość zawierająca informacje o zapasowych repozytoriach.
6. Wiadomość zostaje wysłana na maile ze słownika zainteresowanych osób

Baza danych

Wykorzystujemy technologię SQLite. Z racji tego, iż nasza baza nie jest bardzo skomplikowana doskonale się ona sprawdza. Domyślnie plik zawierający bazę danych jest tworzony w katalogu domowym użytkownika.

Diagram



Struktura bazy danych

Poniżej prezentujemy właściwości tabel przedstawione za pomocą instrukcji DDL:

1) masterRepos

```
create table masterRepos (
    id text primary key, url text not null,
    login text not null, password text not null,
    path not null unique, frequency integer not null
);
```

2) Tabela backupRepos

```
create table backupRepos (
    id INTEGER primary key autoincrement,
    master_repo_id text not null references masterRepos on
delete cascade,
```

```
url text not null,  
login text not null,  
password text not null,  
unique (master_repo_id, url)  
);
```

3) Tabela emails

```
create table emails (  
    name TEXT primary key, email TEXT not null  
);
```

4) Tabela emails_repos

```
create table emails_repos (  
    ID INTEGER primary key autoincrement,  
    name TEXT not null references emails on delete cascade,  
    master_repo_id TEXT not null references masterRepos on  
delete cascade  
);
```

Plik database_handler.py

W pliku database_handler.py znajdują się dwie klasy: ReposDatabaseHandler oraz EmailsDatabaseHandler, które zawierają metody umożliwiające zmiany w bazie danych.

Management Panel

W części klienckiej skorzystaliśmy z oprogramowania Electron. Pozwala on na tworzenie aplikacji desktopowych z użyciem najnowocześniejszych rozwiązań webowych do których należy m.in. framework Angular wykorzystywany w naszym projekcie.

Electron

Punktem startowym części frontendowej jest plik main.js. Korzystając z funkcjonalności platformy Electron uruchamia główne okno aplikacji oraz pozwala na połączenie z Angularem. Odbywa się to dzięki IPC (ang. inter-process communication) - mechanizmu, dostarczanego przez systemy operacyjne, pozwalającemu na asynchroniczną komunikację między procesem głównym (main.js), a procesem renderującym (kod angularowy). Dzięki takiemu rozwiązaniu można skorzystać zarówno z nowoczesnych rozwiązań frontendowych jak i ominąć ograniczenia kodu wykonywanego po stronie przeglądarki np. dostęp do systemu plików, uruchamianie innych procesów itp.

Angular

Odpowiada za logikę związaną z renderowaniem widoków użytkownikowi końcowemu jak i komunikację z częścią serwerową (z użyciem architektury REST). Struktura kodu podzielona jest na tzw. komponenty (pozwalające na uniknięcie redundancji kodu poprzez wielokrotne użycie tego samego elementu oraz zwiększające przejrzystość).

Komponenty:

- HomeComponent - komponent główny, pozwala na przeglądanie repozytoriów
- AddRepoComponent - komponent pozwalający na dodawanie nowych repozytoriów
- AddBackupComponent - komponent pozwalający na modyfikację repozytorium (backupy, częstotliwość)
- NotifyComponent- komponent umożliwiający notyfikację użytkowników o zmianach zaistniałych w głównym repozytorium za pomocą wiadomości email

Przewodnik użytkownika

Istota działania aplikacji

Główną funkcjonalnością aplikacji jest automatyczne propagowanie zmian z jednego głównego repozytorium do pozostałych podanych przez użytkownika repozytoriów zapasowych. Proces odbywa się tak, tworzymy lokalnie kopię repozytorium głównego i co ustalony okres czasu pobieramy zmiany jakich dokonano w repozytorium zdalnym. Następnie zmiany te są propagowane na repozytoria zdalne. Operacje te opierają się na wykorzystaniu gita popularnego systemu kontroli wersji. Opierają się na wykorzystaniu komend pull i push.

Drugą funkcjonalnością jest, w sytuacji gdy nasze główne repozytorium znajduje się na serwisie github, możliwość wysyłania powiadomień mailowych innym użytkownikom githuba o zapasowych lokalizacjach repozytorium. Użytkownicy, którym aplikacja stara się wysłać maila to osoby bezpośrednio powiązane z danym repozytorium. Są to bezpośredni współpracownicy, którzy współtworzyli repozytorium lub osoby, które przyznały gwiazdkę temu repozytorium. Ze względu na sposób przechowywania danych przez serwis github możliwe są specyficzne sytuacje, w których aplikacji poprzez API githuba nie uda się uzyskać maili niektórych wymienionych powyżej osób, zatem w obecnej wersji oprogramowania, nie ma gwarancji, iż wszyscy potencjalnie zainteresowani dostaną wiadomość mailową.

Instrukcja uruchomienia aplikacji

Serwer

Wymagania:

- Python 3.8 (<https://www.python.org/downloads/>)
- pip

Instalacja pipenv:

- `pip install --user pipenv`

Jeśli pipenv jest niedostępny z poziomu terminala, to należy dodać go do ścieżki (PATH)

- Windows
 - należy uruchomić komendę `py -m site --user-site` i w wyniku zastąpić *site-packages* na *Scripts*. Taką ścieżkę dodać do zmiennej PATH.
Przykładowy wynik po uruchomieniu komendy:
`C:\Users\username\AppData\Roaming\Python\Python38\site-packages`
Końcowa ścieżka (po zamianie):
`C:\Users\username\AppData\Roaming\Python\Python38\Scripts`
- Linux i macOS
 - należy uruchomić komendę `python -m site --user-base` i do wyniku dodać *bin*. Taką ścieżkę dodać do zmiennej PATH.

Przykładowy wynik po uruchomieniu komendy:

`~/local` (gdzie `~` oznacza ścieżkę do katalogu użytkownika)

Po dodaniu:

`~/local/bin`

W celu zainstalowania potrzebnych zależności i stworzenia środowiska należy z poziomu katalogu projektu wykonać komendy:

- `pipenv install` (instaluje potrzebne zależności na podstawie pliku *Pipfile*)
- `pipenv shell` (tworzy środowisko i automatycznie przechodzi do niego)

Uruchomienie (z poziomu katalogu projektu):

- `cd server`
- `python app.py`

Część kliencka

Foldery zawierające pliki wykonywalne:

- Windows 10 (x64)
 - https://drive.google.com/file/d/1TAnPcomYw_2IBVPXJbAwzFK1YJpbFaIQ/view?usp=sharing
- macOS
 - <https://drive.google.com/file/d/1btyLe4TVDgkOUJqjsyNE5VG3EAw2n3cm/view?usp=sharing>

Taki folder można wygenerować dla własnego systemu operacyjnego za pomocą następujących komend (wymagany *Node* wraz z *npm*):

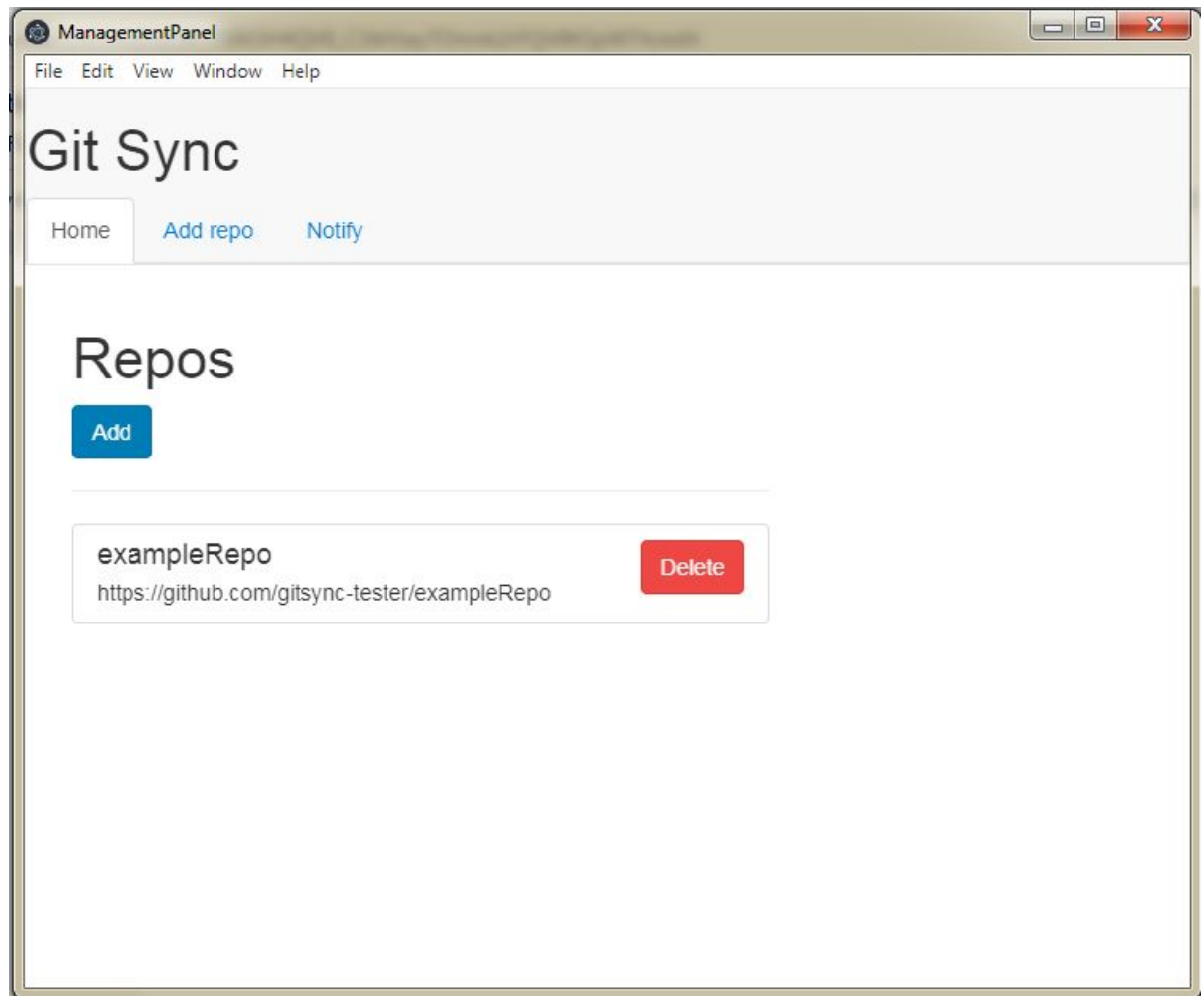
- `npm install -g @angular/cli`
- `npm install`
- `npm run pack`

Po wykonaniu powyższych komend powinien pojawić się w projekcie nowy katalog zawierający pliki umożliwiające bezpośrednie uruchomienie części klienckiej.

Instrukcja korzystania z poszczególnych funkcjonalności i ekranów

Ekran startowy

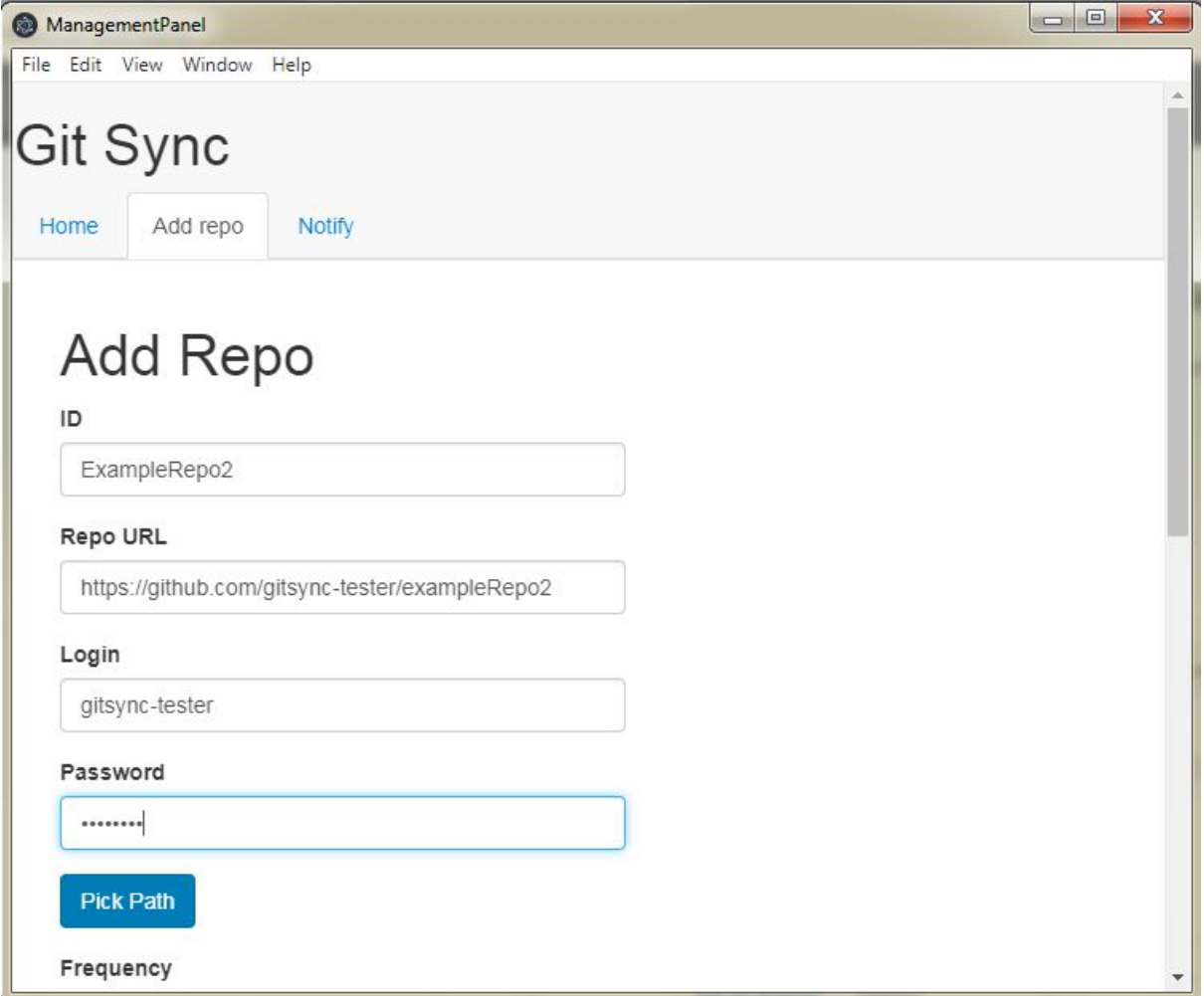
Po uruchomieniu użytkownikowi wyświetla się okno z listą repozytoriów, które są synchronizowane przez program.



Użytkownik może usunąć synchronizację dla danego repozytorium (przycisk delete) - wtedy są usuwane informacje o danym repozytorium oraz zmiany z głównego repozytorium nie są propagowane na repozytoria zapasowe. W sytuacji kiedy jakieś repozytorium jest wylistowane na tym ekranie oznacza to, że jego obecny stan jest propagowany na powiązane z nim repozytoria zapasowe w odstępach czasowych określonych w trakcie tworzenia synchronizacji repozytorium lub później po modyfikacji tego parametru. Parametr ten możemy zmodyfikować klikając przycisk z nazwą repozytorium.

Dodanie synchronizacji dla nowego repozytorium

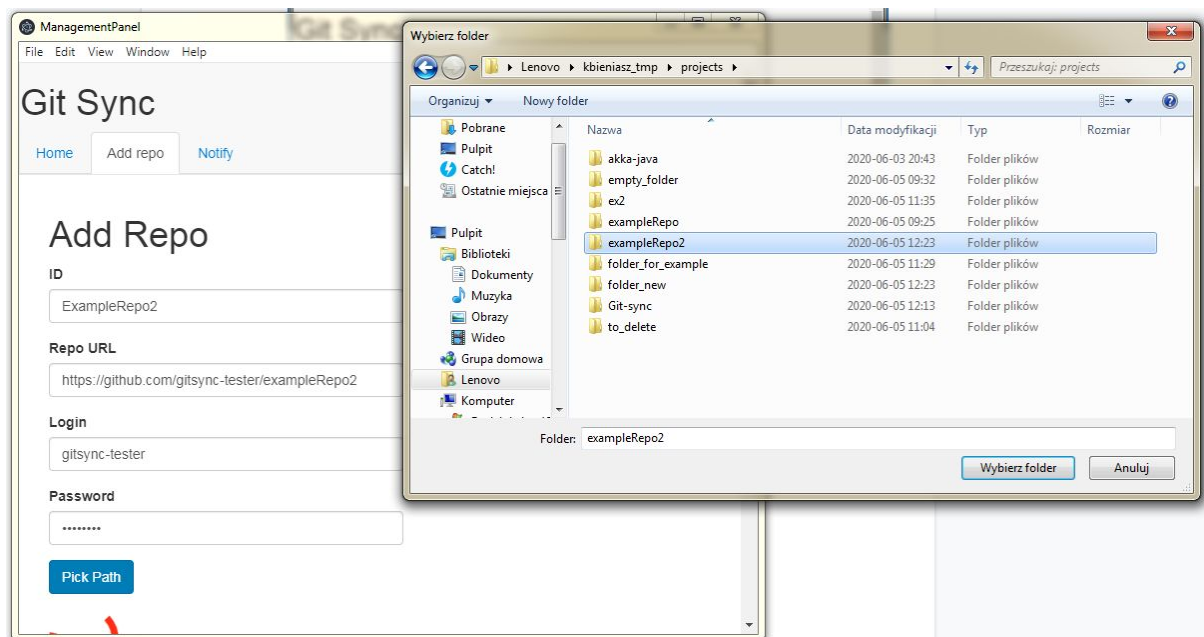
Po wciśnięciu przycisku add z ekranu głównego lub wybraniu zakładki add pokazuje się okno służące do dodania synchronizacji dla nowego repozytorium.



The screenshot shows a window titled "ManagementPanel" with a menu bar (File, Edit, View, Window, Help). The main content area is titled "Git Sync" and has three tabs: "Home", "Add repo" (selected), and "Notify". The "Add Repo" section contains the following fields and controls:

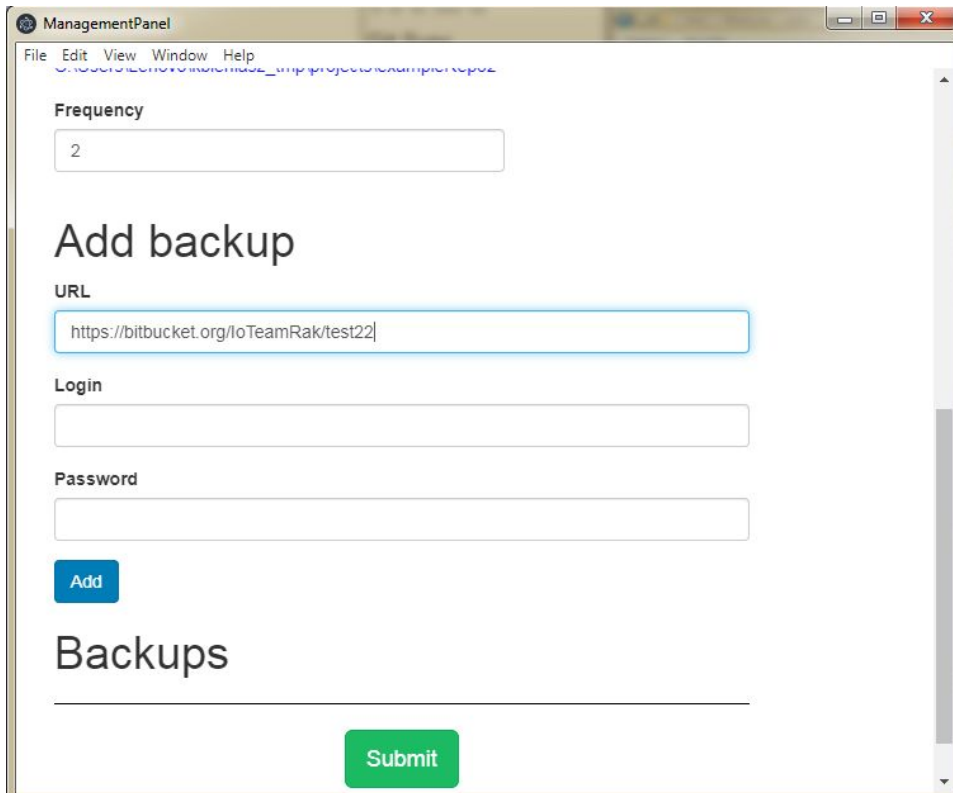
- ID**: A text input field containing "ExampleRepo2".
- Repo URL**: A text input field containing "https://github.com/gitsync-tester/exampleRepo2".
- Login**: A text input field containing "gitsync-tester".
- Password**: A password input field with masked characters ".....".
- Pick Path**: A blue button located below the password field.
- Frequency**: A label at the bottom of the form.

Po naciśnięciu przycisku "Pick Path" wyskakuje okienko systemowe, w którym wybieramy folder, gdzie będzie się znajdować kopia repozytorium na naszym lokalnym komputerze.



Ważne jest, aby wybrany folder był pusty. W przypadku wybrania niepustego folderu, system informuje, iż wybrany folder musi być pusty.

W dalszej części konfiguracji wybieramy jak często będzie przeprowadzana konfiguracja oraz wprowadzamy dane o repozytoriach zapasowych. Wartość w polu frequency oznacza co ile godzin będzie przeprowadzana synchronizacja.



Istotne jest, aby przy dodawaniu zapasowej lokalizacji repozytorium po wprowadzeniu danych nacisnąć przycisk “Add”

Add backup

URL	Login	Password
<input type="text" value="https://github.com/"/>	<input type="text" value="gitsync_tester"/>	<input type="password" value="....."/>
<input type="button" value="Add"/>		

Backups

Po jego wciśnięciu powinna się zaktualizować lista “Backups”

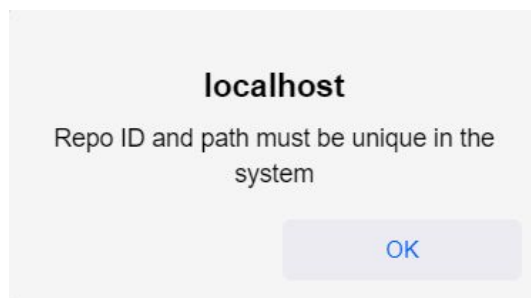
Add backup

URL	Login	Password
<input type="text"/>	<input type="text"/>	<input type="password"/>
<input type="button" value="Add"/>		

Backups

<input type="text" value="https://github.com/gitsync-tester/exampleRepo2"/>	<input type="button" value="Delete"/>
---	---------------------------------------

Po wprowadzeniu wszystkich danych użytkownik klika przycisk submit. W razie niepowodzenia operacji wyświetla się komunikat (np. repozytorium o danej nazwie już istnieje w systemie i nie można go wprowadzić drugi raz).



Pierwsza operacja synchronizacji repozytoriów może trwać stosunkowo długo, z tego powodu system nie wyświetla bezpośrednio informacji dla użytkownika o znajdowaniu się w trakcie tej czynności oraz jej bezpośrednim zakończeniu. W zależności od wielkości repozytorium i ilości lokalizacji zapasowych nawet kilkadziesiąt sekund. Zalecamy, aby w zależności od tych parametrów użytkownik odczekał odpowiednią ilość czasu, a następnie sprawdził na serwisach z repozytoriami, które wskazał czy synchronizacja przebiegła pomyślnie. Jeśli tak się stało kolejne synchronizacje po określonym czasie powinny się odbywać bez zarzutu. W przypadku niepowodzenia sugerujemy sprawdzenie czy wprowadzono poprawne dane do systemu (w szczególności URL, login, hasło).

Modyfikacja ustawień synchronizacyjnych dla repozytorium

Po kliknięciu w ikonkę zawierającą nazwę i urla głównego repozytorium otwiera nam się okienko służące do zmiany częstotliwości synchronizacji zmian oraz dodaniu nowych zapasowych repozytoriów lub też usunięciu starych.

ManagementPanel

File Edit View Window Help

Git Sync

Home Add repo Notify

masterRepo

Frequency

58

Add backup

URL

Login

Password

Add

Backups

https://github.com/gitsync-tester/backupForEx2

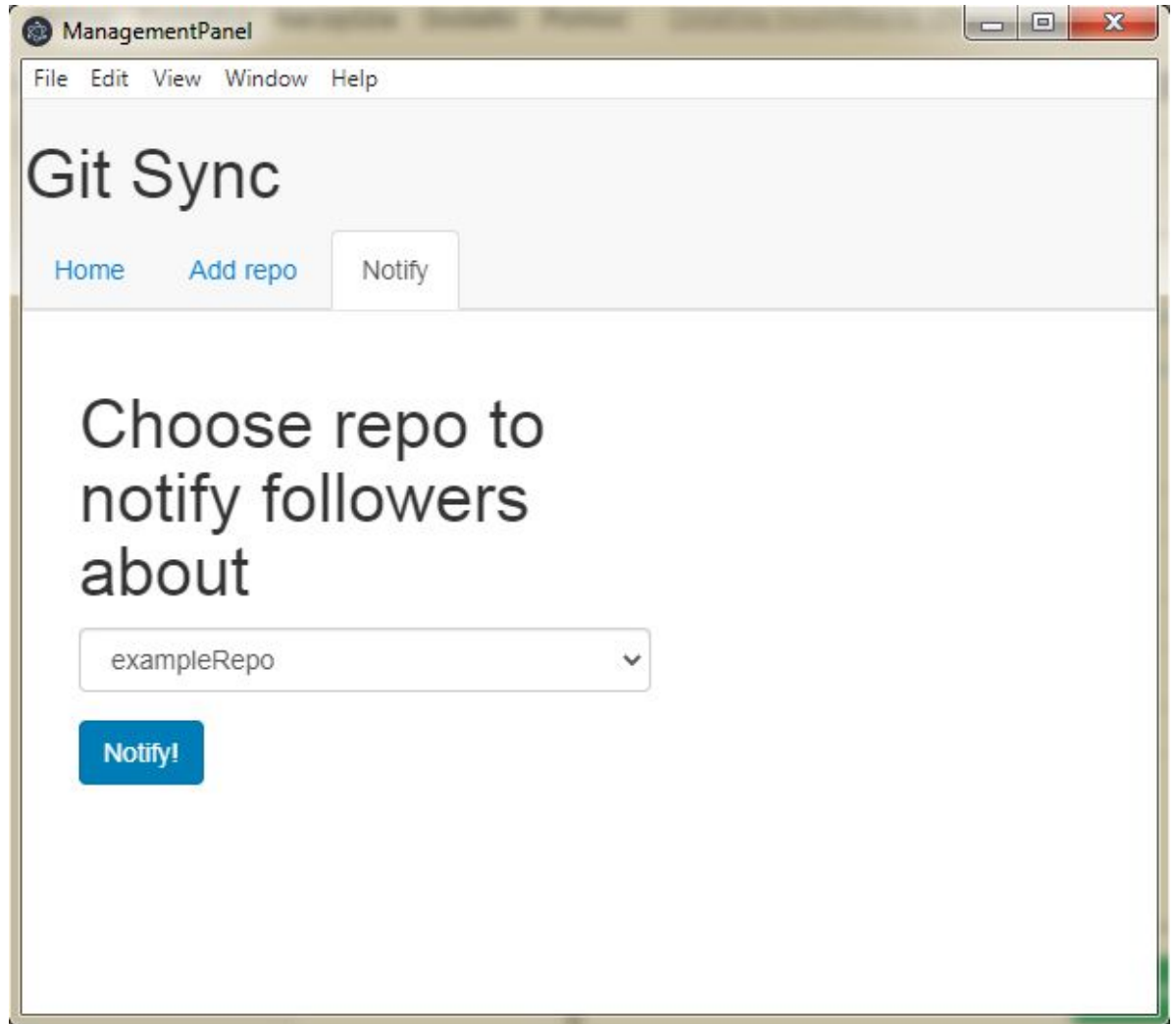
Delete

Submit

Operacja dodawanie i usuwania zapasowych lokalizacji przebiega w analogiczny sposób jak przy zarządzaniu zapasowymi repozytoriami dodawania synchronizacji dla nowego repozytorium

Wysyłanie powiadomień

Po kliknięciu w zakładkę notify przechodzimy do obszaru, z którego możemy zlecić wysłanie wiadomości o lokalizacji zapasowych repozytoriów wszystkim potencjalnym zainteresowanym.



Po kliknięciu notify nasz system wyśle wiadomość do wszystkich osób, które były współpracownikami w tworzeniu danego repozytorium lub polubiły dane repozytorium. Po wykonaniu tej operacji pojawi się krótka informacja o sukcesie czynności.