Politechnika Warszawska

WYDZIAŁ ELEKTRYCZNY

Instytut Elektrotechniki Teoretycznej
i Systemów Informacyjno-Pomiarowych
Zakład Elektrotechniki Teoretycznej
i Informatyki Stosowanej

Praca dyplomowa inżynierska

na kierunku Informatyka stosowana
w specjalności Inżynieria oprogramowania

Aplikacja do zarządzania publikacjami naukowymi z automatyczną analizą PDF

Piotr Jeleniewicz
nr albumu 291072

promotor
dr inż. Bartosz Chaber

WARSZAWA 2021

APLIKACJA DO ZARZĄDZANIA PUBLIKACJAMI NAUKOWYMI Z AUTOMATYCZNĄ ANALIZĄ PDF

Streszczenie

Praca składa się z

Słowa kluczowe: praca dyplomowa, LaTeX, jakość

REFERENCE MANAGER WITH AUTOMATIC PDF ANALYSIS

Abstract

This thesis presents.

Keywords: thesis, LaTeX, quality

WARSZAWA, 1 lutego 2021

POLITECHNIKA WARSZAWSKA
WYDZIAŁ ELEKTRYCZNY

OŚWIADCZENIE

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa inżynierska pt. Aplikacja do zarządzania publikacjami naukowymi z automatyczną analizą PDF:

- została napisana przeze mnie samodzielnie,
- nie narusza niczyich praw autorskich,
- nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam, że przedłożona do obrony praca dyplomowa nie była wcześniej podstawą postępowania związanego z uzyskaniem dyplomu lub tytułu zawodowego w uczelni wyższej. Jestem świadom, że praca zawiera również rezultaty stanowiące własności intelektualne Politechniki Warszawskiej, które nie mogą być udostępniane innym osobom i instytucjom bez zgody Władz Wydziału Elektrycznego.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Piotr Jeleniewicz.....

Spis treści

Podziękowania

Piotr Jeleniewicz

Rozdział 1

Wstęp

Przy pisaniu artykułów naukowych często sięga się po efekty prac przedstawione w innych publikacjach naukowych. Wraz z pisaniem coraz większej liczby tego typu dokumentów, ilość wykorzystywanych w nich pozycji bibliograficznych może zacząć znacznie wzrastać co może utrudnić odnajdowanie potrzebnych publikacji w stale rozszerzającym się ich zbiorze.

Dlatego też ta praca będzie przedstawiała efekty prac nad aplikacją do zarządzania publikacjami naukowymi z automatyczną analizą plików PDF, której głównym celem jest ułatwienie procesu zarządzania publikacjami naukowymi, które przechowywane są w formie plików PDF. System będzie składał się aplikacji klienckiej przygotowanej dla systemu Android oraz aplikacji serwerowej działającej w kontenerze Dockera.

1.1 Założenia projektowe

System powstający w ramach pracy inżynierskiej będzie opierał się o następujące założenia:

1. Aplikacja serwerowa będzie napisana przy użyciu Node.js oraz języka TypeScript.
2. W celu ułatwienia konfiguracji środowiska deweloperskiego jaki produkcyjnego, baza danych wraz z aplikacją serwerową będą uruchamiane w sposób skonteneryzowany przy wykorzystaniu technologii Docker.
3. Aplikacja kliencka będzie przeznaczona na system Android oraz do jej napisania wykorzystany zostanie język Kotlin.
4. W celu korzystania z aplikacji użytkownik będzie musiał utworzyć konto w systemie.

5. Do uzyskania informacji z pliku PDF dotyczących publikacji wykorzystane zostanie API dostępne pod adresem:
   `https://api.crossref.org/`
6. Aplikacje będą tworzone w sposób modułowy, umożliwiając stosunkowo łatwą możliwość rozbudowywania funkcjonalności.W
7. Podczas implementacji obu aplikacji w miarę możliwości stosowane będą najlepsze praktyki programistyczne.

W systemie będą dostępne następujące funkcjonalności:

1. Wyświetlanie listy publikacji.
2. Wyświetlanie opisu publikacji naukowych.
3. Tworzenie nowej publikacji w oparciu o metadane oraz numer DOI, jeśli są dostępne w dodawanym pliku PDF.
4. Edycja publikacji.
5. Pobranie pliku PDF powiązanego z daną publikacją.

Rozdział 2

Przedstawienie wykorzystywanych technologii

Przed rozpoczęciem implementacji projektu przeprowadzona została szczegółowa analiza dostępnych technologii zarówno w kontekście aplikacji serwerowej jak również klienckiej. W trakcie jej trwania pod uwagę brane były przede wszystkim aspekty dotyczące specyfik danych technologii takich jak wydajność czy też sugerowane przeznaczenie poszczególnych rozwiązań, ale także kwestie dotyczące osobistych preferencji odnoszących się do danych technologii.

2.1 Aplikacja serwerowa

W obecnych czasach ilość technologii pozwalających na pisanie aplikacji serwerowych jest ogromna. Podjęcie jednoznacznego wyboru dotyczącego, której z nich należy użyć do danego zadania jest niemalże niemożliwe, jednakże każda z nich cechuje się swoimi indywidualnymi cechami, które pozwalają w pewnym stopniu ocenić, które z dostępnych narzędzi będzie odpowiednie do rozwiązania wybranego problemu. W przypadku aplikacji serwerowej, tworzonej w ramach tej pracy, głównymi cechami branymi pod uwagę była wydajność danej technologii oraz ilość kodu wymagana do poprawnego i bezpiecznego działania aplikacji. Po przeanalizowaniu dostępnych rozwiązań, aplikacja zostanie napisana w środowisku Node.js, przy użyciu języka TypeScript. Do przechowywania danych wykorzystana zostanie baza danych MongoDB, natomiast dane dotyczące sesji użytkowników będą przechowywane w bazie danych Redis. Cała aplikacja serwerowa będzie uruchomiona jako kontenery Dockera.

2.1.1 Node.js

Node.js jest środowiskiem uruchomieniowym bazującym na koncepcie *„JavaScript everywhere"*, który unifikuje język używany podczas procesu tworzenia aplikacji webowej, wykorzystując JavaScript zarówno po stronie klienta jaki i serwera. W związku z tym, że w ramach projektu aplikacja kliencka będzie stworzona w formie aplikacji mobilnej na system Android, koncept ten nie zostanie spełniony. jednakże przy tworzeniu hipotetycznej aplikacji klienckiej działającej w przeglądarkach internetowych wykorzystywanie jednego języka, jest sytuacją ułatwiającej rozwój i utrzymania takiej aplikacji.

Do niewątpliwych zalet środowiska Node.js należy ogromna ilość bibliotek dostępnych do wykorzystania w aplikacji. Dzięki temu wiele problemów może zostać rozwiązanych przy użyciu gotowych komponentów, co znacznie wpływa na zmniejszenie ilości kodu, który jest wymagany do rozwiązania konkretnego problemu. Z punktu widzenia aplikacji serwerowej tworzonej w ramach tego projektu, niezwykle istotna okazała się dostępność biblioteki umożliwiającej przetwarzanie i odczyt danych z plików *pdf*.

Inną ważną cechą środowiska Node.js, w odróżnieniu od środowisk wykorzystujących język Python, jest bardzo dobra obsługa standardu JSON, która pozwala na używanie obiektów zgodnych ze standardem JSON niemal jako standardowych obiektów języka JavaScript, co znacznie wpływa na ułatwienie komunikacji pomiędzy aplikacją serwerową a kliencką.

2.1.2 TypeScript

Językiem który zostanie wykorzystany do napisania aplikacji serwerowej będzie TypeScipt. Jest on nadzbiorem języka JavaScript, utworzonym i utrzymywanym przez firmę Microsoft. Zasadniczą różnicą pomiędzy tymi języki jest typowanie.

Typowanie można podzielić między innymi na następujące dwa rodzaje:

- Typowanie statyczne - zmienne mają ustalany typ w momencie deklaracji. Wykorzystywane w języku TypeScript
- Typowanie dynamiczne - typ zmiennej wynika wynika z wartości jaka jest w niej przechowywana. Wykorzystywane w języku JavaScript.

Pomimo że typowanie dynamiczne pozwala na większą elastyczność przy wykorzystaniu zmiennych, typowanie statyczne pozwala na wykrycie części błędów w momencie kompilacji, bądź też transpilacji kodu.

W celu uruchomienia kodu napisanego w języku TypeScript, musi najpierw zostać dokonana jego transpilacja do języka JavaScript. Dopiero ten kod jest realnie wykonywany w środowisku uruchomieniowym.

2.1.3 MongoDB

MongoDB jest nierelacyjną bazą danych, wykorzystywanym w ramach aplikacji serwerowej. W systemach nierelacyjnych w przeciwieństwie od relacyjnych, informacje nie są przechowywane w tabelach, a w formie dokumentów w stylu JSON, które oprócz samych danych przechowują także ich schemat. Dzięki temu zmiana schematu danych, nie wymaga zazwyczaj skomplikowanych migracji i pozwala na bardzo dynamiczne zarządzanie strukturą danych. Wadą takiego rozwiązania jest utrudnione wyszukiwanie żądanych informacji, jeśli wymagają one bardziej skomplikowanych kwerend, a także znacznie mniejsza szybkość wyszukiwania, zwłaszcza w przypadku większej liczby danych.

W celu łączenia się z bazą danych i mapowania obiektów z bazy danych na obiekty języka JavaScript wykorzystywany jest ODM Mongoose. Dzięki temu że baza danych przechowuje dokumenty w stylu JSON, w bardzo prosty sposób można zapisywać obiekty otrzymane od klienta w ramach żądania HTTP do bazy danych, a także wysyłać klientowi obiekty z bazy, bez konieczności znaczącej ingerencji w ich format i strukturę.

2.1.4 Redis

Redis jest bazą danych typu klucz-wartość, wykorzystywaną głównie w formie pamięci podręcznej dla aplikacji. W odróżnieniu od typowych baz danych, dane przechowywane są w pamięci, a nie na dysku twardym. Dzięki temu szybkość odczytu danych jest znacznie większa niż w przypadku klasycznych rozwiązań. Jest ona w stanie pod wybranym kluczem, przechowywać zwykle ciągi znaków, ale także słowniki, co wraz z szybkim odczytem danych sprawia, że baza ta dobrze nadaje się do przechowywania danych dotyczących sesji użytkownika.

2.1.5 Docker

W celu ułatwienia procesu wdrożenia aplikacji serwerowej w środowisku produkcyjnym, aplikacja wraz z bazą danych zostały zamknięte w kontenery Dockera. Jest to oprogramowanie umożliwiające uruchamianie skonteneryzowanych aplikacji w środowisku odizolowanym od głównego systemy operacyjnego. Kontener zawiera w sobie całe środowisko potrzebne do uruchomienia aplikacji, w odróżnieniu jednak od typowej maszyny wirtualnej kontener nie musi zazwyczaj zawierać w sobie pełnego systemu operacyjnego co znacznie poprawia wydajność takiego rozwiązania, a jednocześnie sprawia że jesteśmy w stanie uruchomić aplikację zamkniętą w kontenerze, bez ko-

nieczności wstępnej konfiguracji środowiska uruchomieniowego, co znacznie poprawia przenośność takiej aplikacji. Izolacja kontenerów pomiędzy sobą, a także od pozostałych procesów głównego systemu operacyjnego może dodatkowo mieć wpływ na podniesienie bezpieczeństwa. Komunikacja pomiędzy różnymi kontenerami może mieć miejsce tylko w specjalnie przygotowanych kanałach komunikacyjnych, np. poprzez wydzieloną sieć, w skład której wchodzą tylko te kontenery, które muszą pomiędzy sobą wymieniać dane. Taka sytuacja ma miejsce przy wymianie danych pomiędzy aplikacją serwerową przygotowywaną w ramach tej pracy, a bazą danych MongoDB.

2.2 Aplikacja kliencka

W obecnych czasach telefony komórkowe nie są wykorzystywane wyłącznie do wykonywania połączeń, ale także do korzystania z zasobów internetu czy też do obsługi różnych aplikacji o najróżniejszym zastosowaniu. Dzięki temu zyskują one coraz większą popularność - obecnie znaczna część społeczeństwa posiada swojego osobistego smartfona. Dlatego też aplikacja kliencka przygotowana została z myślą o urządzeniach mobilnych, a konkretnie tych, które działają pod kontrolą systemu Android.

2.2.1 Android

Android jest systemem operacyjnym opartym o jądro Linuxa, przygotowany głównie z myślą o smartfonach, choć można go także znaleźć innych urządzeniach tj. telewizory czy smartwatche. Obecnie rozwijany jest i utrzymywany przez firmę Google. W badaniu przeprowadzonym przez firmę Gemius, polegającym na sprawdzeniu z jakiego systemu operacyjnego pochodzą odsłony wybranych stron internetowych, Android zajął pierwsze miejsce z około 54% udziałem. Można zatem wnioskować, że jest to jeden z najpopularniejszych systemów na świecie, dlatego też aplikacja kliencka będzie przygotowywana do działania na urządzeniach z system Android.

2.2.2 Kotlin

Na początku istnienia systemu Android, natywne aplikacje na ten system były pisane w języku Java. Jednak w roku 2019, firma Google umożliwiła rozwijanie programów także w języku Kotlin. Jest to język wieloparadygmatowy, zaprezentowany w 2011 roku przez firmę JetBrains - autorów środowiska InteliJ Idea, na którym oparte jest środowisko Android Studio. Kotlin

uruchamiany jest w wirtualnej maszynie Javy, dzięki czemu możliwe jest wykorzystywanie bibliotek napisanych w języku Java w aplikacjach bazujących na Kotlinie. Dodatkowo środowisko Android Studio umożliwia translację plików napisanych w Javie na język Kotlin, a także odwrotnie.

Do głównych przewag języka Kotlin na Javą należy zaliczyć przede wszystkim fakt, że często wymaga znacznie mniej kodu, w celu rozwiązania tego samego problemu, dzięki czemu kod jest czytelniejszy co zmniejsza prawdopodobieństwo powstawania błędów. Dodatkowo Kotlin posiada mechanizmy chroniące programistę przed wyjątkiem *NullPointerExecption*, co także minimalizuje prawdopodobieństwo powstania wadliwego kodu.

Dla osób mających doświadczenie w programowaniu w innych językach niż Kotlin, jego wadą może być składnia, która znacząco różni się względem innych popularnych języków takich jak Javy, czy C++ co może powodować problemy dla programistów, którzy uczą się pisania aplikacji przy użyciu Kotlina, pisząc wcześniej w języku, którego składnia wywodzi się z C. Wartym wspomnienia jest również fakt, że Kotlin jest obecnie językiem mniej popularnym niż Java, co powoduje, że ilość problemów opisywanych na forach internetowych dotyczących rozwijania aplikacji na system Android, częściej dotyczy języka Java. Jednakże problem ten nie jest bardzo dotkliwy, ze względu na to, że rozwiązania z języka Java może stosunkowo łatwo wykorzystać w Kotlinie. Dodatkowo oficjalna dokumentacja systemu Android, w której przykładowy kod napisany zarówno w języku Kotlin jak i w Javie, również minimalizuje tą niedogodność.

Rozdział 3

Architektura

Cały system, który zostanie utworzony w ramach opisywanego projektu będzie składał się z trzech głównych części:

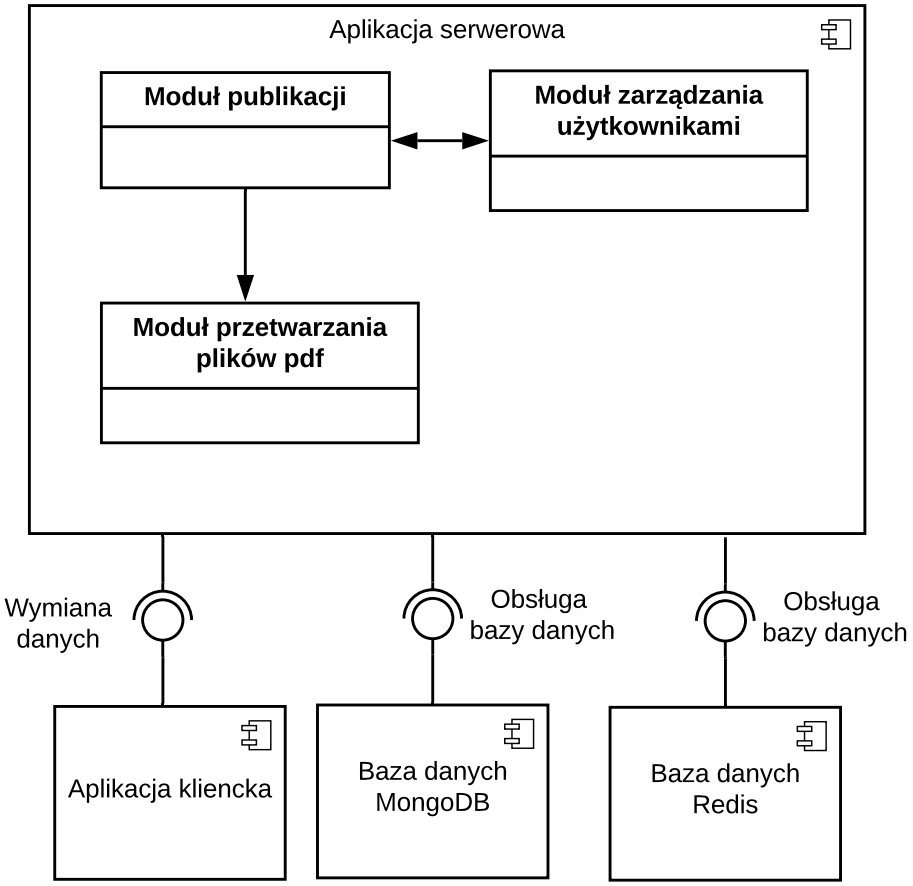
- Aplikacja serwerowa - udostępnia ona interfejs programistyczny służący do szeroko rozumianego zarządzania publikacjami naukowymi.
- Baza danych MongoDB - będzie służyła do przechowywanie informacji na temat publikacji naukowych. Bezpośredni dostęp będzie miała do niej tylko aplikacja serwerowa
- Mobilna aplikacja kliencka - przeznaczona na urządzenia z systemem Android, wykorzystująca interfejs udostępniany przez aplikację serwerową

Ze względu na pełną niezależność aplikacji serwerowej od mobilnej, możliwe jest przygotowanie w przyszłości klienta działającego w przeglądarce internetowej lub innej platformie.

Każda z aplikacji ze wykorzystuje inne, specyficzne dla swoich platform wzorce projektowe, dlatego też po omówieniu ogólnej architektury całego systemu każda z aplikacji zostanie przedstawiona osobno, w sposób bardziej szczegółowy.

3.1 Architektura ogólna

Na poniższym diagramie znajduje się ogólny schemat przedstawiający architekturę całego systemu:



Rysunek 3.1: Ogólna architektura systemu

Przedstawiona na powyższym diagramie aplikacja serwerowa oraz baza danych MongoDB będę uruchamiana jako kontenery Dockera, natomiast aplikacja kliencka będzie pracowała na urządzeniach z systemem Android. Wymiana danych pomiędzy tymi aplikacjami będzie odbywała się za pomocą interfejsu programistycznego(API), w sposób szyfrowany przy wykorzystaniu protokółu HTTPS. Baza danych będzie działać w wewnętrznej sieci Dockerowej, do której bezpośredni dostęp będzie posiadała jedynie aplikacja serwerowa.

3.2 Architektura aplikacji serwerowej

3.2.1 Struktura

Aplikacja serwerowa zbudowana będzie w oparciu o środowisko Node.js. Składać się będzie z 3 podstawowe modułów:

1. Moduł zarządzania użytkownikami - będzie odpowiadał za obsługę żądań HTTP dotyczących rejestracji i logowania użytkowników oraz zarządzaniu sesją
2. Moduł publikacji - będzie posiadał następujące zadania:
   - obsługa żądań HTTP dotyczących publikacji
   - zapis, odczyt i modyfikacja publikacji w bazie danych
   - zlecanie operacji przetwarzania pliku PDF
   - zapisywanie i odczyt plików PDF przypisanych do publikacji
3. Moduł przetwarzania plików pdf - jako jedyny nie będzie odpowiadał za obsługę żądań, a za przetwarzania plików pdf. Do przetwarzania plików pdf zostanie wykorzystana biblioteka *pdf-parse*. Będzie dostarczał następujące funkcjonalności:
   - odczyt informacji takich jak tytuł lub autor publikacji z metadanych plików PDF
   - wyszukiwanie DOI w metadanych pliku PDF, bądź w jego treści
   - pobieranie szczegółów dotyczących przetwarzanej publikacji na podstawie znalezionego wcześniej numery DOI

Dodatkowo żądania HTTP będą przetwarzane wstępnie, także przez następujące moduły pośredniczące:

1. Moduł autoryzacyjny - jego rolą będzie sprawdzanie czy dane żądanie wysłane została przez użytkownika, który posiada uprawnienia do jego wykonania
2. Moduł walidacji danych - będzie odpowiedzialny za sprawdzania czy dane przychodzące w żądaniach, typu POST oraz PUT, które tworzą lub modyfikują informacje w bazie danych, są w odpowiednim formacie

3.2.2 Środowisko

Aplikacja serwerowa będzie przygotowana do pracy jako kontener Dockera. Dzięki temu przy wdrażaniu aplikacji konfiguracja środowiska produkcyjnego, nie będzie wymagała pełnego przygotowania środowiska Node.js,

potrzebując do pracy jednie aplikacji Docker oraz docker-compose, która pozwala na zarządzanie i konfigurację wielu kontenerów, mogących współpracować między sobą. Dzięki temu, że baza danych MongoDB może również działać w formie kontenera Dockerowego, przy wykorzystaniu aplikacji docker-compose, uruchomienie całej aplikacji serwerowej wraz bazą danych będzie ograniczać się do pobrania kontenerów oraz ich uruchomienia za pomocą jednego polecenia, a dodatkowo pozwala na wdrożenie aplikacji zarówno przy użyciu systemu Linux i Windows jaki i MacOs.

3.3 Architektura aplikacji klienckiej

Aplikacja kliencka będzie przeznaczona na urządzenia mobilne z systemem Android. Zostanie wykonana w sposób natywny dla tej platformy przy wykorzystaniu środowiska Android Studio oraz języka Kotlin. Zgodnie z zaleceniami występującymi w dokumentacji Android, wykorzystany zostanie uproszczony wzorzec Model–view–viewmodel(MVVM).

3.3.1 Wzorzec Model–view–viewmodel(MVVM)

Wzorzec MVVM bazuje na popularnym wzorcu Model-view-controller. Jest często wykorzystywany podczas tworzenia aplikacji z interfejsem graficznym, do których możemy zaliczyć programy przeznaczone na system Android. Zgodnie z tą architekturą aplikacja składa się z głównych elementów:

1. model - przechowuje dane, pobrane przez viewmodel, później prezentowane w warstwie view, nie powinien zawierać logiki biznesowej
2. view - jest to reprezentacja graficznego widoku, widzianego przez użytkownika aplikacji
3. viewmodel - element spajający model i view, pod pewnym względem odpowiadający dla wartswy controller z wzorca MVC. Jest odpowiedzialny za wypełnianie widoku danymi, pobranymi z modelu oraz za obsługę interakcji w użytkownikiem.

Jedną z podstawowych cech tego wzorca jest tak zwany databinding. Polega on automatycznym odświeżaniu danych pochodzących z modelu w widoku, w momencie gdy dojdzie do ich modyfikacji w viewmodelu. Dzięki temu, programista jest zwolniony z ciągłego dbania o uaktualnianie danych prezentowanych w widoku.

3.3.2 Struktura

Natywne aplikacje działające pod kontrolą systemu Android zbudowaną są z komponentów zwanych aktywnościami(Activity). Zgodnie z definicją dostępną w dokumentacji, jest to pojedyncza konkretna czynność, która może zostać wykonana przez użytkownika. Dodatkowo aktywności mogą składać się z mniejszych części zwanych fragmentami. Aplikacja kliencka przygotowywana w ramach tej pracy, będzie składała się z trzech aktywności:

1. LoginActivity - aktywność odpowiedzialna za proces logowania użytkownika.
2. MainActivity - główna aktywność służąca do prezentacji listy publikacji danego użytkownika.
3. PublicationDetailsActivity - aktywność w której dodaje się nowe publikacje oraz odczytuje i edytuje istniejące.

Do komunikacji z aplikacja serwerową za pomocą protokołu HTTP wykorzystana zostanie biblioteka Retrofit. Wykorzystywana będzie w klasach należących do paczki *network*, która odpowiadać za obsługę żądań HTTP wykorzystywanych do logowania użytkowników oraz do odczytywania, dodawania i edycji publikacji naukowych.

Do obsługi asynchroniczności zapytań, wszystkie klasy korzystające z klas pakietu *network*, będą wymagały zaimplementowania interfejsu *RequestObserver*, który będzie zawierał dwie metody:

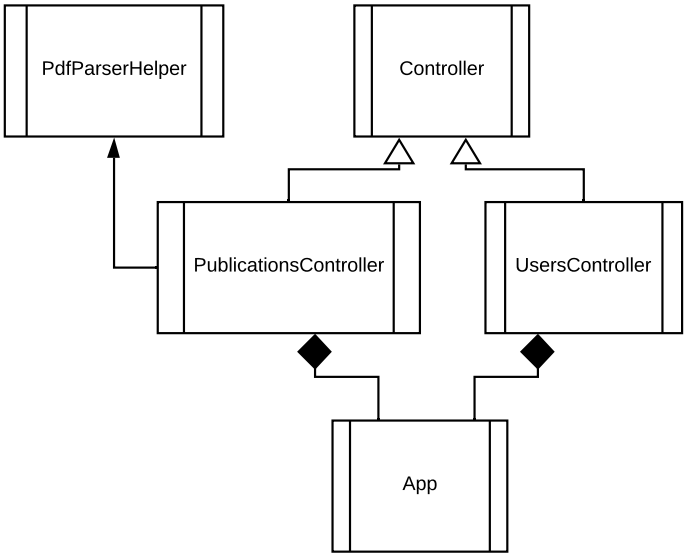
1. onSuccess - metoda wywoływana, w przypadku gdy żądanie zakończyło się pomyślnie.
2. onFail - metoda wywoływana, w przypadku gdy żądanie zakończyło się błędem.

Rozdział 4

Szczegóły implementacyjne aplikacji serwerowej

4.1 Ogólna struktura

Zgodnie z przedstawioną w poprzednim rozdziale architekturą aplikacji serwerowej została ona podzielona na trzy moduły. W tym rozdziale zostaną przedstawione ich szczegóły implementacyjne oraz sposób komunikacji pomiędzy sobą. Organizacja kodu aplikacji serwerwoej została przedstawiona, na poniższym diagramie klas:



Rysunek 4.1: Diagram klas

4.1.1 Opis podstawowych klas

Przedstawione na powyższym diagramie klasy pełnią następujące funkcje:

1. App - główna klasa aplikacji, która inicjalizuje wszystkie klasy rozszerzające klasę Controller, a także odpowiedzialna za skonfigurowanie oprogramowania pośredniczącego(middleware) .
2. Controller - klasa rozszerzana przez inne klasy, odpowiedzialne za obsługę żądań HTTP.
3. UsersController - klasa obsługująca żądania dotyczące rejestracji, logowania, oraz wylogowywania użytkowników.
4. PublicationsController - klasa obsługująca żądania dotyczące dodawania, odczytywania, usuwania oraz edycji publikacji.

4.2 Obsługa użytkowników

Cała obsługa użytkowników w systemie została napisana od podstaw. Odpowiedzialna jest za to klasa *UsersController*, odpowiadająca za proces rejestracji i logowania użytkowników oraz oprogramowanie pośredniczące *authMiddleware*, którego zadaniem jest sprawdzanie czy dane żądanie jest wykonywane przez zalogowanego użytkownika, który posiada wystarczające uprawnienia. Do zapewnienia bezpiecznego przechowywania hasła wykorzystana została biblioteka *bcrypt*, za pomocą której uzyskiwana jest sól, wykorzystywana podczas obliczania funkcji skrótu dla hasła.

Dla celów autoryzacji, dokumenty reprezentujące użytkowników, które przechowywane są w bazie danych, posiadają w swojej strukturze dwa pola:

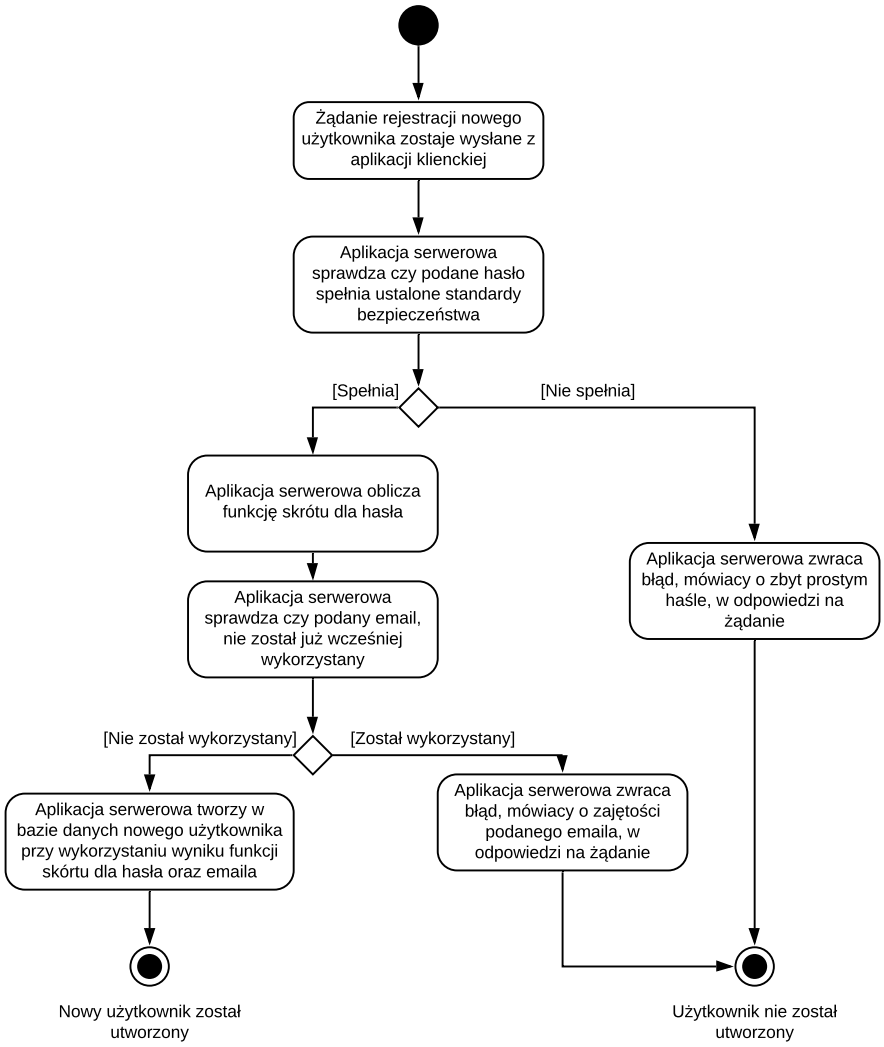
- email - służący do identyfikacji, użytkownika, który próbuje zalogować się do systemu
- hash - pole zawierające sól oraz wynik funkcji skrótu dla hasła danego użytkownika
- type - pole służące do określania typu użytkownika. Obecnie nie jest one wykorzystywane i zawsze posiada wartość *USER*, lecz umożliwia wprowadzenie w przyszłości podziału użytkowników na różne grupy, o specyficznych uprawnieniach.

Dzięki przechowywaniu wyniku funkcji skrótu dla hasła, zamiast jawnego jego przechowywania, podnoszony jest poziom bezpieczeństwa. Nawet w przypadku, gdy dane z bazy danych zostaną wykradzione, hasła użytkowników systemu nie będą mogły zostać odczytane. Wynika to z własności funkcji

skrótu, która nie pozwala w łatwy sposób odczytać wartości wejściowej, nawet przy posiadania soli, która została wykorzystana podczas obliczania wyniku funkcji.

4.2.1 Proces rejestracji użytkownika

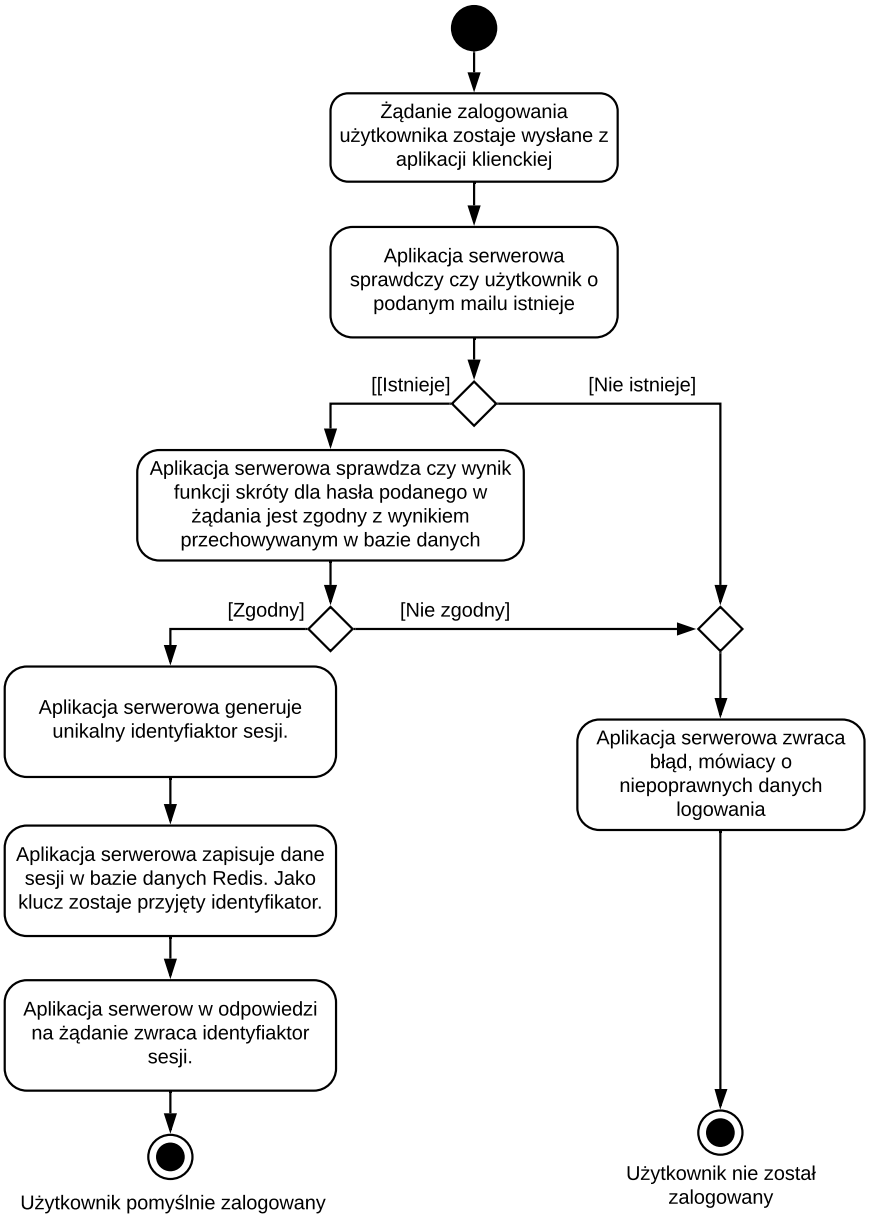
Proces rejestracji użytkownika posiada następujący przebieg:



Rysunek 4.2: Diagram aktywności dla procesu rejestracji

4.2.2 Proces logowania użytkownika

Po pomyślnym przejściu procesu rejestracji, użytkownik nabywa możliwość zalogowania się do systemu za pomocą wcześniej podanego emaila i hasła. Proces ten wygląda następująco:



Rysunek 4.3: Diagram aktywności dla procesu logowania

W celu przeprowadzania logowania do systemu, wysyłane jest żądanie do aplikacji serwerowej z danymi, na podstawie, których użytkownik jest uwierzytelniany, a w odpowiedzi zwracany jest identyfikator sesji, który jest później wykorzystywany podczas wysyłania innych żądań. Posiada on ograniczoną czasową ważność, wynoszącą 7 dni.

4.3 Uwierzytelnienie i autoryzacja żądań

Większość żądań obsługiwanych przez aplikacją serwerową, może być obsłużona tylko w przypadku, gdy użytkownik, które je wysłał został poprawnie uwierzytelniony i zautoryzowany. Uwierzytelnienie żądania odbywa się na podstawie weryfikacji nagłówka żądania HTTP o nazwie *Authorization*, który powinien zawierać token pełniący rolę identyfikatora sesji. Za to zadanie odpowiedzialne są dwie funkcje pośredniczące w obsłudze żądania:

1. *authMiddlewareUser* - funkcja zezwala na pełne wykonanie żądania, tylko gdy użytkownik jest zalogowany.
2. *authMiddlewareOwner* - funkcja zezwala na pełne wykonanie żądania, tylko gdy użytkownik jest właścicielem zasobu, którego dotyczy żądanie.

Obie te funkcje wykorzystują w trakcie działania jedną z najważniejszych funkcji biorących udział w procesie obsługi użytkowników o nazwie *checkAuth*. Jej rolą jest sprawdzenie ważności sesji i zgodności roli użytkownika zidentyfikowanego na podstawie tokena przesłanego w nagłówkach żądania, z wymaganiami bezpieczeństwa danego żądanie. Jeśli warunki te są spełnione, wtedy obsługa żądania przekazywana jest już do docelowej funkcji.

Nagłówek funkcji *checkAuth* prezentuje się następująco:

```
async function checkAuth: boolean (
    req: Request,
    next: NextFunction,
    profileTypes: string[]
)
```

Do wywołania tej funkcji należy jej przekazać następujące parametry:

1. req - obiekt zawierający wszystkie parametry żądania, takie jak nagłówki czy ciało.
2. next - funkcja, która powinna zostać wywołana jako kolejna, podczas procesu obsługi żądania.
3. profileTypes - tablica zawierająca role użytkowników, którzy mają prawo do wywołania danego żądania.

4.3.1 Sposób działania funkcji checkAuth

Na początku działania, sprawdza czy w żądaniu, które zostało odebrane, znajduje się nagłówek *Authorization* zawierający token identyfikujący sesję użytkownika. Jeśli token został odnaleziony, na jego podstawie wyszukiwany jest obiekt sesji w bazie danych Redis. Jeśli wyszukiwanie powiedzie się, funkcja *checkAuth* przeprowadza następujące czynności:

1. Sprawdzenie czy czas ważności sesji nie został przekroczony.
2. Sprawdzenie czy rola użytkownika, którego sesja została zidentyfikowana, jest zgodna z parametrem *profileTypes* przykazanym do funkcji jako parametr.

W przypadku nie odnalezienia nagłówka *Authorization* lub obiektu sesji w bazie danych Redis, bądź też gdy choć jedna z czynności wymienionych powyżej zakończy się niepowodzeniem, funkcja zwraca wartość false, która oznacza, że proces autoryzacji również zostanie zakończony niepomyślnie. W przypadku gdy wszystkie wyżej wymienione warunki zostaną spełnione, do obiektu żądania zostaną dopisane parametry użytkownika powiązanego z sesją a następnie funkcja zwróci wartość true, która oznacza, że proces autoryzacji użytkownika zakończył się pomyślnie.

4.3.2 Funkcja authMiddlewareUser

Funkcja authMiddlewareUser wykorzystuje wzorzec projektowy Adapter, w celu dostosowania funkcji *checkAuth* do interfejsu, który jest wymagany dla funkcji pośredniczących w środowisku Node.js. Jej nagłówek prezentuje się następująco:

```
async function authMiddlewareUser: void (
    req: Request,
    res: Response,
    next: NextFunction
)
```

W odróżnieniu od funkcji *checkAuth* wśród argumentów wejściowych brakuje parametru *profileTypes*, będącego listą ról użytkowników, którzy mają prawo do wywołania danego żądania. Dodatkowo w celu zapewnienia zgodności z interfejsem funkcji pośredniczącej w Node.js, pojawił się parametr *res*, który jest obiektem odpowiedzi na żądanie.

Jej działanie polega na wywołaniu funkcji *checkAuth* oraz sprawdzeniu czy zwróci ona wartość `true`. Jeśli tak się stanie, wywoływana jest funkcja

next, która jest następną funkcją odpowiedzialną za przetworzenie żądania. Przy wywoływaniu funkcji *checkAuth*, jako parametr *profileTypes* zawsze zostaje przekazana tablica `[ "USER" ]`, co oznacza, że obługa żądania będzie kontynuowana, tylko gdy użytkownik, z którym będzie powiązany identyfikator sesji zawarty w nagłówku żądania jest typu `USER`.

Ze względu na to, że obecnie wszyscy użytkownicy posiadają typ `USER`, funkcja *authMiddlewareUser* pozwala w rzeczywistości na badanie czy użytkownik wywołujący żądanie jest zalogowany.

4.3.3 Funkcja authMiddlewareOwner

Funkcja authMiddlewareOwner w odróżnieniu authMiddlewareUser, nie posiada interfejsu zgodnego z funkcjami pośredniczącymi, gdyż wykorzystuje ona domknięcie, zwracając funkcję, posiadającą interfejs funkcji pośredniczącej.

Jedynym argumentem, który przyjmuje funkcja *authMiddlewareOwner* jest obiekt *DBModel* typu *Model<IOwnership Document>* pochodzącego z biblioteki Mongoose, który służy do zarządzania wybranym rodzajem dokumentów w bazie danych MongoDB. Do funkcji może zostać przekazany jedynie taki model, który obsługuje dokumenty które są jednocześnie implementują interfejs *Document*, który również pochodzi z biblioteki Mongoose i reprezentuje pojedynczy dokument z bazy danych oraz *IOwnership*, który wykorzystywany jest we wszystkich dokumentach, które mogą posiadać właścicieli, jak ma to miejsce przykładowo dla dokumentów przechowujących dane pojedynczej publikacji.

Poniżej przedstawiony został interfejs *IOwnership*:

```
interface IOwnership {
    owners: UserType[ ]
}
```

Interfejs ten jest bardzo prosty i posiada tylko jedno pole *owners*, które jest tablicą użytkowników, będących właścicielami obiektu, który implementuje ten interfejs. Dzięki zastosowaniu tego interfejsu, nie ma potrzeba pisania osobnej obsługi rozpoznawania czy użytkownik jest właścicielem wybranego obiektu, dla każdego rodzaju dokumentów osobno. Z tej właśnie własności korzysta funkcja *authMiddlewareOwner*.

Uproszczony kod funkcji *authMiddlewareOwner* prezentuje się następująco:

```
const authMiddlewareOwner =
  (DbModel: Model<IOwnership & Document>) => (
    async (req: Request, res: Response, next: NextFunction) => {
    const authStatus = await checkAuth(req, next, [USER, ADMIN]);
    const { documentId } = req.params;
    const { _id as userId } = (<any>req).user;
    let user: UserType = await User.findById(userId);
    dbObject = await DbModel.findOne({ _id: documentId, owners: user });
  })
}
```

Jej działanie opiera się o domknięcie - jako argument otrzymuje ona obiekt modelu, który później zostaje zawarty w środowisku zwracanej funkcji. Dzięki temu, że model ten musi implementować interfejs *IOwnership*, dokumenty, które zostaną pobrane z bazy danych za jego pomocą muszą posiadać pole *owners*, określające ich właścicieli. Zatem funkcja *authMiddlewareOwner* po wywołaniu zwraca inną funkcję, która jest już właściwą funkcją pośredniczącą, zawierającą dodatkowo w swoim środowisko odwołanie do obiektu modelu dokumentu z bazy danych. Sposób działania zwracanej funkcji wygląda następująco:

Wywołanie funkcji `authMiddlewareOwner(Publication)`, zwróci funkcję pośredniczą, która pozwoli na wywołanie kolejnej metody obsługującej żądanie, tylko gdy użytkownik wywołujący żądanie, jest właścicielem obiektu typu `Publication`, którego id została przekazane jako parametr w ścieżce żądania.

1. W sposób analogiczny do funkcji *authMiddlewareUser*, wykorzystując funkcję *checkAuth*, sprawdzana jest tożsamość i typ użytkownika, który wysłał żądanie.
2. Jeśli funkcja *checkAuth* potwierdzi, to że użytkownik jest zalogowany oraz to czy jego typ jest odpowiedni, sprawdzane jest czy dokument, którego dotyczy żądanie jest własnością tego użytkownika.
3. Jeśli użytkownik okaże się właścicielem dokumentu, wywoływana jest funkcja *next*, która jest następną funkcją odpowiedzialną za przetworzenie żądania.

Dzięki tej funkcji, nie trzeba każdorazowo sprawdzać czy użytkownik wysyłający żądanie usunięcia lub edycji danego dokumentu, ma do tego prawo, co ułatwia bezpośrednio kod tych funkcji które wykonują już właściwe operacje na dokumentach przechowywanych w bazie danych.

4.3.4 Proces wylogowania użytkownika

Proces wylogowywania użytkownika jest stosunkowo prosty i polega najpierw na jego zidentyfikowaniu przy pomocy funkcji pośredniczącej *authMiddlewareUser*, a następnie na usunięciu danych dotyczących jego sesji z bazy danych Redis. Dzięki temu, pomimo że identyfikator sesji może pozostać zapisany po stronie klienta, wysłane żądanie zawierające ten identyfikator nie zostanie wykonane.

4.4 Zarządzanie publikacjami oraz przetwarzanie plików pdf

W bazie danych MongoDB dokument reprezentujący publikację posiada następujące pola:

1. `name: string` - tytuł publikacji. Może zostać odczytany podczas z pliku PDF.
2. `authors: string[ ]` - tablica zawierająca autorów publikacji. Pole to nie powinno być mylone z `owners: UserType[ ]`, który w odróżnieniu od `authors: string[ ]`, przechowuje właścicieli publikacji
3. `file: string` - ścieżka do pliku PDF
4. `description: string` - opis publikacji
5. `doi: string` - nr doi publikacji

Dodatkowo zawiera pole *owners* pochodzące z interfejsu *IOwnership*, co pozwala korzystać z funkcji pośredniczącej *authMiddlewareOwner*, w odniesieniu do publikacji.

Zarządzanie publikacjami odbywa się poprzez obsługę 7 żądań. Aby wywołać każde z nich użytkownik wcześniej zalogować się do systemu. Posiadają one następujące funkcjonalnościach:

1. Żądanie typu GET - pobranie informacji dotyczących pojedynczej publikacji. Aby pobrać publikację
2. Żądanie typu GET - pobranie listy z informacjami o wszystkich publikacjach użytkownika.
3. Żądanie typu PUT - edycja informacji dotyczących pojedynczej publikacji.
4. Żądanie typu DELETE - Usunięcie informacji i pliku pdf należących do pojedynczej publikacji.

5. Żądanie typu GET - pobranie pliku pdf przypisanego do wybranej publikacji.
6. Żądanie typu POST - przetworzenie pliku pdf. Jako odpowiedź zwracane było informacje, które udało się odczytać z wysłanego pliku pdf.
7. Żądanie typu POST - dodanie nowej publikacji.

Pierwsze pięć wymienionych wyżej opiera się o zwykłe operacje CRUD przeprowadzane na bazie danych, natomiast dwa pozostałe zależne od siebie żądania również korzystają z bazy danych, lecz wykonują także inne czynności, takie jak operacje na plikach, czy też komunikacja z API służącym do pobierania informacji na temat danej publikacji na podstawie wyszukanego wcześniej numeru DOI.

4.4.1 Definiowanie żądań

Przykładowa deklaracja obsługi żądania wygląda następująco:

```
router.get(
   '/:id',
   authMiddlewareOwner(Publication),
   getPublication
)
```

Przedstawiony w powyższym fragmencie obiekt `router`, służy do definiowania sposobu obsługi żądań. W celu dodania nowego żądania, należy wywołać metodę on nazwie zgodnej z jego typem. Przykładowo w celu dodania obsługi żądania GET należy wywołać metodę `get`. Każda z tych metod jako pierwszy argument przyjmuje ścieżkę żądania, a następnie funkcje obsługujące żądnie, gdzie pozycja wśród argumentów odpowiada kolejności, w której poszczególne funkcje będą wywoływane. Zatem w przypadku dla powyższej próbki kodu po odebraniu żądania, najpierw wywołana zostanie metoda zwrócona przez funkcję `authMiddlewareOwner(Publication)`, a następnie w zależności wyniku działania tej funkcji może zostać wywołana funkcja `getPublication` lub też obsługa żądania może zakończyć się błędem przed wywołaniem tej funkcji.

4.4.2 Pobranie listy publikacji

Proces ten polega na wyszukaniu w bazie danych tych dokumentów opisujących publikacje, których właścicielem jest użytkownik rozpoznany podczas procesu autoryzacji. Uproszczony kod funkcji prezentuje się następująco:

```
const { _id as userId } = (<any>req).user;
user = await User.findById(userId);
const publications = await Publication.find({owners: user});
```

4.4.3 Pobranie dokumentu pojedynczej publikacji

Proces rozpoczyna się od sprawdzenia czy użytkownik wywołujący to żądanie, jest właścicielem publikacji, której szczegóły próbuje odczytać. Jeśli tak jest, publikacja zostaje wyszukana w bazie danych i zwrócona użytkownikowi.

4.4.4 Edycja dokumentu pojedynczej publikacji

Początek obsługi tego żądania jest taki sam jak w przypadku pobrania pojedynczej publikacji - najpierw sprawdzane jest użytkownik wywołujący to żądanie, jest właścicielem publikacji, którą próbuje edytować. Jeśli tak publikacja zostaje zedytowana zgodnie z danymi otrzymanymi w żądaniu.

4.4.5 Usunięcie pojedynczej publikacji

Analogicznie to dwóch wyżej wymienionych żądań, w przypadku usuwania aplikacji, najpierw sprawdzane jest użytkownik wywołujący to żądanie, jest właścicielem publikacji, a następnie dochodzi do usunięcia dokumentu zawierającego jej szczegółu oraz PDF do niej należącego.

4.4.6 Pobranie pliku PDF należącego do pojedynczej publikacji

Obsługa tego żądania jest niemal identyczna do obsługi pobierania dokumentu pojedynczej publikacji, z tą różnicą, że zamiast zwracać obiekt, zwracany jest plik, którego ścieżka zapisana jest w dokumencie wybranej publikacji.

4.4.7 Dodawanie publikacji na podstawie pliku PDF

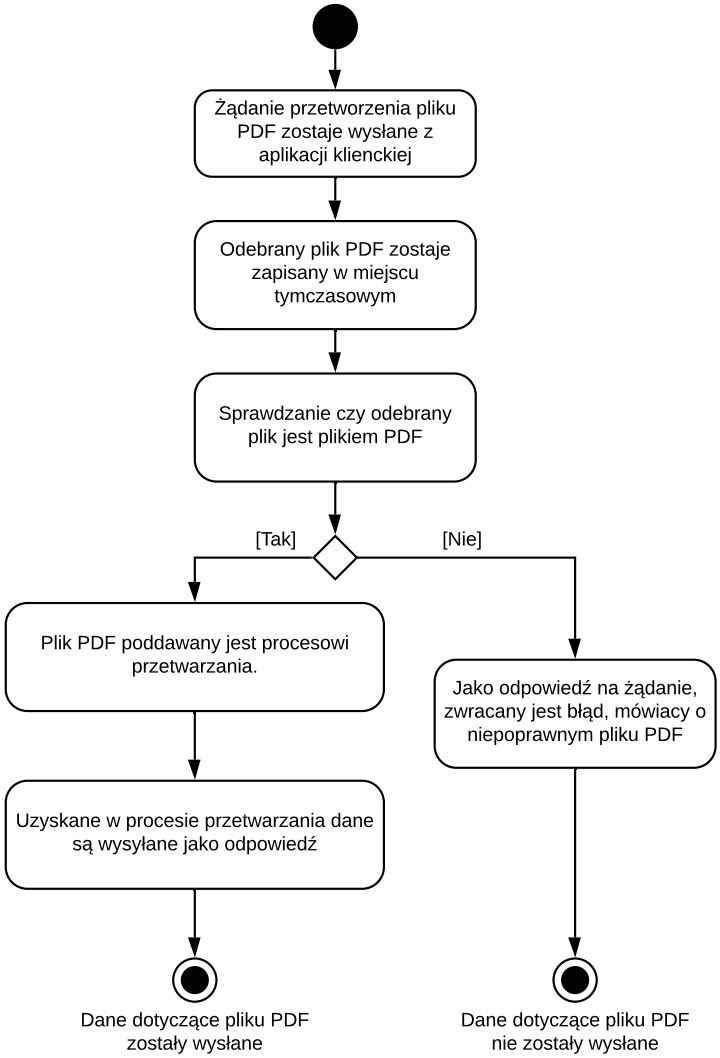
Proces dodawania publikacji, składa się dwóch podstawowych kroków:

1. Przetworzenie pliku pdf, w wyniku którego mogą zostać odczytane informacje dotyczące publikacji.

2. Zapisanie publikacji, po zreadgowaniu i uzupełnieniu przez użytkownika informacji na temat publikacji, które pochodzą z procesu przetwarzania

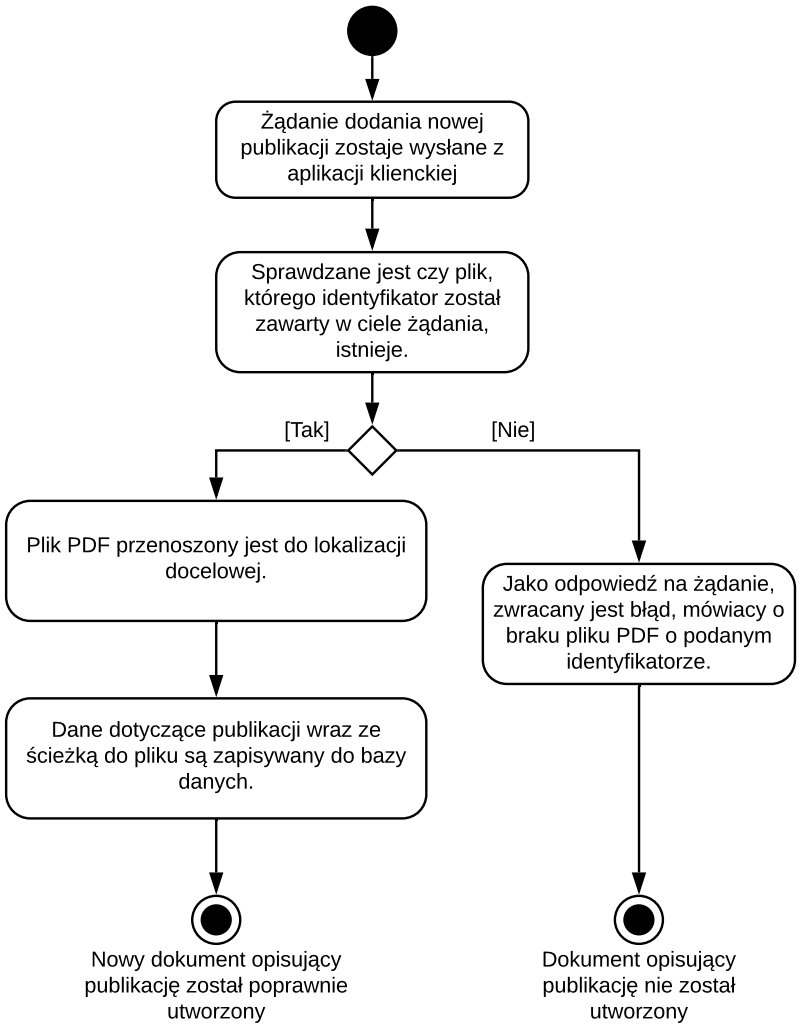
Cały proces składa się także z dwóch osobnych żądań:

1. Żądanie przetwarzające plik PDF - w ramach ciała żądania wysyłany jest plik PDF, który zostaje zapisany w lokalizacji tymczasowej, a w odpowiedzi zwracane są informacje, które udało się odczytać z pliku.



Rysunek 4.4: Diagram aktywności dla obsługi żądania przetwarzania pliku PDF

2. Żądanie dodające dokument publikacji - w ciele żądania powinny znajdować się informacje pochodzące z wcześniejszego przetwarzania pliku PDF, które zostały zredagowane przez użytkownika. Wśród odebranych parametrów publikacji, musi znaleźć się identyfikator przetworzonego wcześniej pliku PDF.



Rysunek 4.5: Diagram aktywności dla obsługi żądania przetwarzania pliku PDF

4.4.8 Przetwarzanie plików PDF

Funkcja przetwarzająca plik PDF jest wywoływana podczas obsługi żądania przetwarzania pliku PDF. Proces ten polega na przeszukaniu treści pliku oraz jego metadanych w celu odnalezienia identyfikatora DOI, który zostanie wykorzystany później w celu pobrania szczegółów dotyczących publikacji. Jeśli identyfikator DOI nie zostanie odnaleziony, następuje wtedy próba odczytania informacji o publikacji wprost z metadanych pliku PDF.

W przypadku, gdy identyfikator DOI nie zostanie odnaleziony wśród metadanych pliku, jego poszukiwanie w treści pliku wykonywane jest przy użyciu wyrażenia regularnego:

```
const regex = /\S+(doi.org)\S+/g;
const found = pdfData.text.match(regex);
```

Stała `regex` zawiera w sobie definicję poszukiwanego ciągu znaków. Obiekt pdfData powstaje w wyniku wykorzystania funkcji dostarczonej w bibliotece *pdf-parse*. Polę text należące do tego obiektu, zawiera cały tekst znajdujący się wewnątrz przetwarzanego pliku. Przy wykorzystaniu funkcji `match`, która wyszukuje ciągi tekstu zgodne z wyrażeniem regularnym, który został przekazany do funkcji jako argument, wyszukiwane są wszystkie możliwie identyfikatory DOI zawarte w pliku. Jako właściwy identyfikator DOI zostaje uznany ten, który jako pierwszy zostanie odnaleziony w tekście, gdyż z duża dozą prawdopodobieństwa pozostałe identyfikatory należą do innych publikacji, do których występują odwołania, wewnątrz badanego pliku.

Jeśli identyfikator DOI został pomyślnie odnaleziony, następuje próba pobrania na jego podstawie szczegółów dotyczących publikacji. Następnie pobrane dane są uzupełniane o te odczytane bezpośrednio z metadanych pliku PDF.

Gdyby identyfikator DOI nie został odnaleziony, nastąpiła by próba odczytania wszystkich wartości wprost z metadanych pliku. Jeśli i ta operacja zakończyła by się niepowodzeniem, użytkownik będzie musiał ręcznie wprowadzić wszelkie szczegóły dotyczące publikacji.

4.5 Walidacja danych

W celu zwiększenia bezpieczeństwa oraz łatwiejszego odnajdowania błędów użytkownika, ciała prawie wszystkich żądań typu POST i PUT, są walidowane, po procesie autoryzacji użytkownika ale przed wywołaniem właściwej funkcji obsługującej żądanie. Do tego celu wykorzystana została funkcja pośrednicząca *validationMiddleware*.

W swoim działania funkcja ta korzysta z biblioteki *class-validator*, która umożliwia walidowanie obiektów wykorzystując zdefiniowane wcześniej do tego celu wzorce. Wykorzystuje ona również domknięcie, zwracając funkcję, która jest zgodna z interfejsem funkcji pośredniczącej.

Nagłówek funkcji *validationMiddleware* prezentuje się następująco:

```
function validationMiddleware<T>(type: any): express.RequestHandler
```

Jej jedynym argumentem jest obiekt, będący definicją typu, z którym musi być zgodne ciało żądania, do którego obsługi wykorzystana została ta funkcja.

Przykładowa definicja typu, który musi być zgodny z ciałem żądania dodającego publikację ma postać:

```
class AddPublicationDto {

  @IsString()
  public readonly name: string;

  @IsString()
  public readonly description: string;

  @IsString()
  public readonly file: string;

  @IsArray()
  @ValidateNested({ each: true })
  @ArrayMinSize(0)
  @Type(() => String)
  public readonly authors: string[];

}
```

Definiowanie warunków, które muszą zostać spełnione przez poszczególne pola ciała żądania definiowane są za pomocą dekoratorów. Znaczenie części z tych, które zostały wykorzystane w powyższym przykładzie jest następujące:

- `@IsString()` - sprawdza czy wartość danego pola jest ciągiem znaków
- `@IsArray()` - sprawdza czy wartość danego pola jest tablicą
- `@ValidateNested()` - określa czy w przypadku tablic oraz obiektów należy walidować, każdą zagnieżdżoną wartość
- `@ArrayMinSize(0)` - służy do określenia minimalnej wielkości tablicy

Rozdział 5

Szczegóły implementacyjne aplikacji klienckiej

Aplikacja kliencka przygotowana w ramach tego projektu, została napisana w celu działania na urządzeniach mobilnych wyposażonych w system Android. Zgodnie z informacjami zawartymi w poprzednich rozdziałach, będzie ona wykorzystywać wzorzec Model-View-Viewmodel(MVVM) oraz do jej stworzenia wykorzystany został język Kotlin.

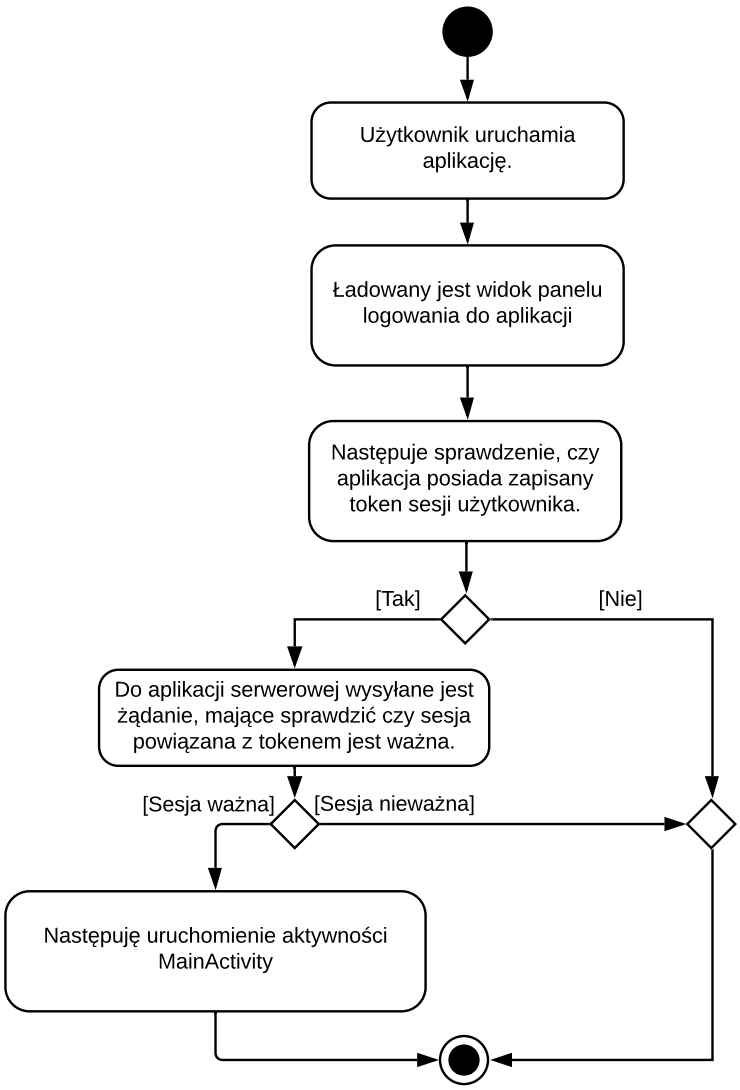
Najważniejsze klasy definiujące trzy aktywności, z których składa się aplikacja kliencka należą do podstawowego pakietu publicationManager Dodatkowo została podzielona następujące podpakiety:

1. network - zawiera wszystkie klasy odpowiedzialne za wymianę danych z aplikacją serwerową, przy użyciu żądań wysyłanych za pomocą protokołu HTTP
2. ui - składa się z klas odpowiedzialnych za obsługę interfejsu użytkownika.
3. interfaces - w tym pakiecie znajdują się interfejsy wykorzystywane wewnątrz aplikacji, służące między innymi do obsługi odpowiedzi na żądania
4. utils - w skład tego pakietu wchodzą różnego rodzaju klasy i funkcje pomocnicze

Oprócz kodu napisanego w języku Kotlin, aplikacja składa się także z plików XML. Są one wykorzystywane do konfiguracji ustawień wymaganych przez system Android, a także do definiowania styli oraz układu poszczególnych elementów interfejsu użytkownika.

5.1 LoginActivity

LoginActivity jest pierwszą aktywnością, która jest uruchamiana po włączeniu aplikacji. Zgodnie z nazwą, odpowiada ona za obsłużenie procesu logowania użytkownika, a także za sprawdzenie czy użytkownik nie zalogował się wcześniej do aplikacji. Jej sposób działania prezentuje się następująco:



Rysunek 5.1: Diagram aktywności dla LoginActivity

Ze względu na stosunkową prostotę i jedynie jeden ekran, który należy do tej aktywności, nie została ona podzielona na fragmenty. Inna cechą tej ak-

tywności jest fakt, że nie jest ona zapisywana na stosie cofania, dzięki czemu użytkownik po zalogowaniu, nie jest w stanie cofnąć się do tej aktywności przy użyciu klawisza cofnij.

Ze względu na fakt wywoływania asynchroniczną naturę metod obsługujących żądania wysyłany do aplikacji serwerowej, klasa LoginActivity implementuje interfejs *LoginObserver* służący do obsługi odpowiedzi na żądania logowania. Prezentuje się on następująco:

```
interface LoginObserver {
  fun onUserLoginSuccessful(userDTO: UserDTO)
  fun onUserLoginFailure(errorResponse: ErrorResponse?)
}
```

Funkcja *onUserLoginSuccessful* zaimplementowana w LoginActivity zostaje wywołana gdy żądanie logowania zostanie pomyślnie zakończone. Jako argument funkcja otrzymuje obiekt transferu danych typu *UserDTO*, który zawiera informacje dotyczące zalogowanego użytkownika oraz token sesji, który zostaje zapisany w pamięci urządzenia przy użyciu interfejsu *SharedPreferences*, służącego do przechowywania informacji w postaci klucz-wartość. Funkcja ta odpowiada także za uruchomienie głównej aktywności *MainActivity*

Funkcja *onUserLoginFailure* jest wywoływana w przypadku gdy żądanie logowania zakończy się błędem. Jeśli taka sytuacja wystąpi, funkcja ta wyświetli komunikat użytkownikowi w formie chmurki zwanej tostem(Toast).

5.2 MainActivity

MainActivity jest główną aktywnością w aplikacji składającą się z dwóch fragmentów:

1. PublicationsListFragment - fragment odpowiedzialny za ekran z listą publikacji użytkownika
2. UserDetailsFragment - ekran powiązany z tym fragmentem przedstawia szczegóły zalogowanego użytkownika oraz zezwala na ich edycję.

5.2.1 PublicationsListFragment

Ekran reprezentujący *PublicationsListFragment* składa się z przewijanej listy zawierającej odnośniki do poszczególnych publikacji należących do obecnie zalogowanego użytkownika. Za jej wyświetlenie odpowiedzialny jest komponent *RecyclerView*. Jego największą zaletą jest fakt, iż w odróżnieniu od

komponentu *ScrollView* jest on w stanie wyświetlać bez większych spowolnień bardzo długie listy.

Przewaga komponentu *RecyclerView* wynika z faktu, że zamiast trzymać w pamięci wszystkie elementy listy jak ma to miejsce w *ScrollView*, przechowuje jedynie te, które są widoczne na ekranie. W momencie gdy dany element listy znika z pola widzenia, jest przesuwany na początek ze zmienioną zawartością, odpowiadającą kolejnemu elementowi listy.

Wadą komponentu *RecyclerView* jest niewątpliwe stosunkowo skomplikowana obsługa w kodzie, gdyż aby móc z niego korzystać należy przygotować specjalną klasę będącą adapterem listy wejściowej na jej graficzną reprezentację.

Za pobranie i aktualizację danych odpowiada klasa *PublicationsListViewModel*, stanowiąca warstwę viewmodelu. Jej deklaracja prezentuje się następująco:

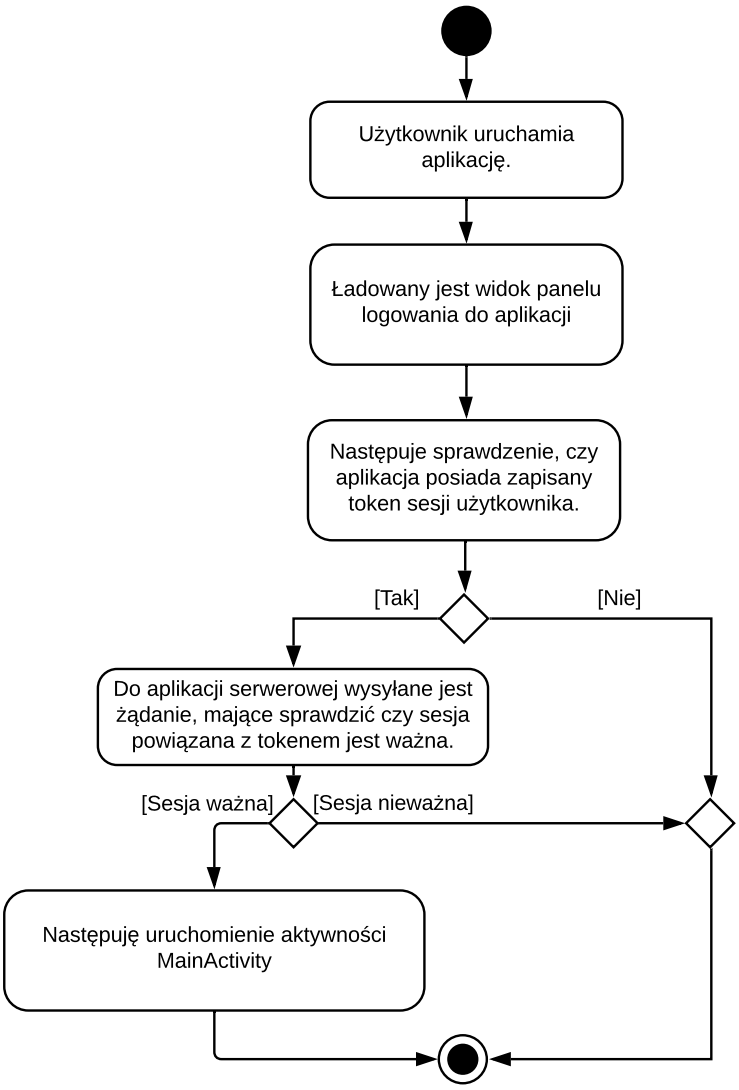
```
class PublicationsListViewModel :
  ViewModel(),
  RequestObserver<ArrayList<PublicationListItem>>
```

Klasa *PublicationsListViewModel* dziedziczy po klasie ViewModel, co pozwala w łatwy sposób połączyć ją klasą z warstwy View - w tym przypadku z klasą *PublicationsListFragment*. Dodatkowo implementuje ona interfejs *RequestObserver*, który umożliwia obsługę odpowiedzi na żądanie. Jest on zbliżony do interfejsu *LoginObserver*, gdyż również posiada on dwie metody: jedną wywoływaną gdy żądanie zakończy się pomyślnie oraz drugą która uruchamiana jest gdy odpowiedź żądania będzie zawierała błąd. Jednak dzięki wprowadzeniu do niego typów generycznych, można go wykorzystywać do obsługi wielu typów odpowiedzi na żądania. Jego kod wygląda następująco:

```
interface RequestObserver<T> {
  fun onSuccess(response: T, type: String)
  fun onFail(errorResponse: ErrorResponse?, type: String)
}
```

Klasa *PublicationsListViewModel* posiada jedno publiczne pole *publications*, które jest typu *LiveData<List<PublicationListItem»*. Typ *LiveData* ten daje możliwość obserwacji zmian wartości, dzięki czemu w warstwie widoku wystarczy zdefiniować obserwatora pola typu LiveData, który zostanie poinformowany o każdej zmianie wartości obserwowanego pola i pozwoli na dynamiczne odświeżenie prezentowanych informacji.

Poniższy diagram przedstawia w jaki współpracują między sobą klasy *PublicationsListFragment*, *PublicationsListViewModel*, a także klasa *PublicationRepository*, odpowiedzialna za obsługę żądań dotyczących publikacji:



Rysunek 5.2: Diagram aktywności dla LoginActivity

5.2.2 UserDetailsFragment

Klasa *UserDetailsFragment* jest odpowiedzialna za prezentowanie i edycję danych zalogowanego użytkownika. Prezentuje ona na ekranie pola tekstowe zawierające email użytkownika oraz jego pseudonim, a także dwa przyciski odpowiadające za zapisanie zmian oraz wylogowanie użytkownika.

5.3 PublicationDetailsActivity

Bibliografia

[1] daemon9, „LOKI2”, Phrack Magazine, Issue 51. `http://phrack.org`

[2] van Hauser, Reverse WWW Shell, THC, The Hacker's Choice. `www.thc.org`

Opinia

o pracy dyplomowej magisterskiej wykonanej przez dyplomanta
Zdolnego Studenta i Pracowitego Kolegę
Wydział Elektryczny, kierunek Informatyka, Politechnika Warszawska
Temat pracy
TYTUŁ PRACY DYPLOMOWEJ

Promotor: **dr inż. Miły Opiekun**
Ocena pracy dyplomowej: **bardzo dobry**

Treść opinii

Celem pracy dyplomowej panów dolnego Studenta i Pracowitego Kolegi było opracowanie systemu pozwalającego symulować i opartego o oprogramowanie o otwartych źródłach (ang. Open Source). Jak piszą Dyplomanci, starali się opracować system, który łatwo będzie dostosować do zmieniających się dynamicznie wymagań, będzie miał niewielkie wymagania sprzętowe i umożliwiał dalszą łatwą rozbudowę oraz dostosowanie go do potrzeb. Przedstawiona do recenzji praca składa się z krótkiego wstępu jasno i wyczerpująco opisującego oraz uzasadniającego cel pracy, trzech rozdziałów (2-4) zawierających opis istniejących podobnych rozwiązań, komponentów rozpatrywanychjako kandydaci do tworzonego systemu i wreszcie zagadnień wydajności wirtualnych rozwiązań. Piąty rozdział to opis przygotowanego przez Dyplomantów środowiska obejmujący opis konfiguracji środowiska oraz przykładowe ćwiczenia laboratoryjne. Ostatni rozdział pracy to opis możliwości dalszego rozwoju projektu. W ramach przygotowania pracy Dyplomanci zebrali i przedstawili w bardzo przejrzysty sposób duży zasób informacji, co świadczy o dobrej orientacji w nowoczesnej i ciągle intensywnie rozwijanej tematyce stanowiącej zakres pracy i o umiejętności przejrzystego przedstawienia tych wyników. Praca zawiera dwa dodatki, z których pierwszy obejmuje wyniki eksperymentów i badań nad wydajnością, a drugi to źródła skryptów budujących środowisko.

Dyplomanci dość dobrze zrealizowali postawione przed nimi zadanie, wykazali się więc umiejętnością zastosowania w praktyce wiedzy przedstawionej w rozdziałach 2-4. Uważam, że cele postawione w założeniach pracy zostały pomyślnie zrealizowane. Proponuję ocenę bardzo dobrą (5).

(data, podpis)

Recenzja

pracy dyplomowej magisterskiej wykonanej przez dyplomanta
Zdolnego Studenta i Pracowitego Kolegę
Wydział Elektryczny, kierunek Informatyka, Politechnika Warszawska
Temat pracy
TYTUŁ PRACY DYPLOMOWEJ

Recenzent: **prof. nzw. dr hab. inż. Jan Surowy**
Ocena pracy dyplomowej: **bardzo dobry**

Treść recenzji

Celem pracy dyplomowej panów dolnego Studenta i Pracowitego Kolegi było opracowanie systemu pozwalającego symulować i opartego o oprogramowanie o otwartych źródłach (ang. Open Source). Jak piszą Dyplomanci, starali się opracować system, który łatwo będzie dostosować do zmieniających się dynamicznie wymagań, będzie miał niewielkie wymagania sprzętowe i umożliwiał dalszą łatwą rozbudowę oraz dostosowanie go do potrzeb. Przedstawiona do recenzji praca składa się z krótkiego wstępu jasno i wyczerpująco opisującego oraz uzasadniającego cel pracy, trzech rozdziałów (2-4) zawierających bardzo solidny i przejrzysty opis: istniejących podobnych rozwiązań (rozdz. 2), komponentów rozpatrywanychjako kandydaci do tworzonego systemu (rozdz. 3) i wreszcie zagadnień wydajności wirtualnych rozwiązań, zwłaszcza w kontekście współpracy kilku elementów sieci (rozdział 4). Piąty rozdział to opis przygotowanego przez Dyplomantów środowiska obejmujący opis konfiguracji środowiska oraz przykładowe ćwiczenia laboratoryjne (5 ćwiczeń). Ostatni, szósty rozdział pracy to krótkie zakończenie, które wylicza także możliwości dalszego rozwoju projektu. W ramach przygotowania pracy Dyplomanci zebrali i przedstawili w bardzo przejrzysty sposób duży zasób informacji o narzędziach, Rozdziały 2, 3 i 4 świadczą o dobrej orientacji w nowoczesnej i ciągle intensywnie rozwijanej tematyce stanowiącej zakres pracy i o umiejętności syntetycznego, przejrzystego przedstawienia tych wyników. Drobne mankamenty tej części pracy to zbyt skrótowe omawianie niektórych zagadnień technicznych, zakładające dużą początkową wiedzę czytelnika i dość niestaranne podejście do powołań na źródła. Utrudnia to w pewnym stopniu czytanie pracy i zmniejsza jej wartość dydaktyczną (a ta zdaje się być jednym z celów Autorów), ale jest zrekompensowane zawartością merytoryczną. Praca zawiera dwa dodatki, z których pierwszy obejmuje wyniki eksperymentów i badań nad wydajnością, a drugi to źródła skryptów budujących środowisko. Praca zawiera niestety dość dużą liczbę drobnych błędów redakcyjnych, ale nie wpływają one w sposób istotny na na jej czytelność i wartość. W całej pracy przewijają się samodzielne, zdecydowane wnioski

Autorów, które są wynikiem własnych i oryginalnych badań. Rozdział 5 i dodatki pracy przekonują mnie, że Dyplomanci dość dobrze zrealizowali postawione przed nimi zadanie. Pozwala to stwierdzić, że wykazali się więc także umiejętnością zastosowania w praktyce wiedzy przedstawionej w rozdziałach 2-4. Kończący pracę rozdział szósty świadczy o dużym (ale moim zdaniem uzasadnionym) poczuciu własnej wartości i jest świadectwem własnego, oryginalnego spojrzenia na tematykę przedstawioną w pracy dyplomowej. Uważam, że cele postawione w założeniach pracy zostały pomyślnie zrealizowane. Proponuję ocenę bardzo dobrą (5).

(data, podpis)