



Uniwersytet im. A. Mickiewicza w Poznaniu

Wydział Matematyki i Informatyki

Praca inżynierska

Integracja systemów zarządzania w transporcie przy

użyciu aplikacji LogInt

**Integration of transport management systems using the LogInt
application**

Daria Dworzyńska

Numer albumu: 473560

Kierunek: Informatyka

Jakub Paszke

Numer albumu: 473556

Kierunek: Informatyka

Miłosz Rolewski

Numer albumu: 473634

Kierunek: Informatyka

Michał Wujec

Numer albumu: 473587

Kierunek: Informatyka

Promotor:

prof. UAM dr hab. Krzysztof Dyczkowski

Poznań, 2024

Spis treści

Wstęp	8
1. Opis projektu LogInt	11
1.1. Słownik	11
1.2. Cel i klient	13
1.2.1. Cel projektu	13
1.2.2. Klient - firma Boekestijn Transport Sp. z o.o	14
1.2.3. Grupa docelowa	15
1.3. Konkurencyjne produkty	16
1.3.1. Power Automate	16
1.3.2. Zapier	17
1.3.3. Integromat (Make)	17
1.3.4. Przewaga LogInt	18
1.4. Wymagania projektu	18
1.4.1. Wymagania funkcjonalne	18
1.4.2. Wymagania нефункционалне	19
1.4.3. Ryzyka projektowe	21
1.5. Komponenty projektu	23
1.5.1. Środowisko kreacji integracji	23
1.5.2. Środowisko wykonywalne	25
1.5.3. Architektura systemu	27

1.6. Metodyka pracy i role w projekcie	27
1.6.1. Metodyka pracy	27
1.6.2. Role w projekcie	28
2. Projektowanie, testowanie i wdrażanie bazy danych	30
2.1. Wprowadzenie	30
2.2. Wstęp teoretyczny	31
2.2.1. Języki baz danych	31
2.2.2. Modele baz danych	32
2.2.3. Architektura komunikacyjna baz danych	33
2.2.4. Podział systemów baz danych	34
2.3. Etapy projektowania baz danych	36
2.3.1. Analiza wymagań (firmy)	36
2.3.2. Wymagania funkcjonalne i нефункционалне	37
2.3.3. Modelowanie koncepcyjne (diagramy UML)	39
2.3.4. Implementacja baz danych na przykładzie aplikacji Lo- gInt	39
2.3.5. Środowisko pracy - PostgreSQL - implementacja bazy danych	40
2.3.6. Modelowanie koncepcyjne i logiczne	42
2.4. Podsumowanie	54
3. Zastosowanie OpenCV i uczenia maszynowego w automa- tyzacji interfejsów mobilnych	56
3.1. Wprowadzenie	56
3.2. Podstawy teoretyczne	58
3.2.1. Uczenie maszynowe w rozpoznawaniu obiektów	58
3.2.2. Tradycyjne techniki widzenia komputerowego	59

3.2.3. Uczenie głębokie	60
3.2.4. Optyczne rozpoznawanie znaków	61
3.3. Wybrane technologie na potrzeby systemu LogInt	62
3.3.1. Wymagania biznesowe systemu LogInt	62
3.3.2. Wyzwania w automatyzacji aplikacji mobilnych	64
3.3.3. Automatyzacja procesów w środowisku LogInt	64
3.3.4. Zasada działania systemu LogInt	66
3.4. Praktyczne zastosowanie technik CV i OCR w aplikacji LogInt	67
3.4.1. Dopasowywanie wzorców w OpenCV	67
3.4.2. Znajdowanie tekstu za pomocą PyTesseract	71
3.5. Podsumowanie	74
4. Automatyczna interakcja z emulatorem urządzenia mobilnego za pomocą Android Debug Bridge	76
4.1. Wprowadzenie	76
4.1.1. Automatyzacja procesów biznesowych dla procesów logistycznych	77
4.1.2. Zrobotyzowana automatyzacja procesów	77
4.1.3. Wpływ RPA na rynek pracy	78
4.2. Środowisko emulatora urządzenia mobilnego	78
4.2.1. Android Studio Emulator	78
4.2.2. Docker-Android	80
4.2.3. BlueStacks	80
4.3. Android Virtual Device jako komponent emulatora	81
4.4. Android Debug Bridge - możliwości i zastosowania	82
4.4.1. Powłoka Android Debug Bridge	82
4.4.2. Sposób działania	83
4.4.3. Połączenie urządzenia z ADB	84

4.4.4. Wydawanie poleceń powłoki	85
4.5. Projekt systemu automatycznych interakcji	88
4.5.1. Wstęp	88
4.5.2. Proces definiowania sekwencji interakcji w systemie Lo- gInt	89
4.5.3. Zastosowanie środowiska Python w celu integracji kom- ponentów	92
4.5.4. Format JSON jako ustandaryzowana struktura przeka- zywania danych między komponentami systemu	94
4.6. Komponenty wspierające	96
4.6.1. Open Broadcaster Software i symulacja wirtualnej ka- mery	96
4.6.2. OpenCV i PyTesseract	96
5. Budowa backendu aplikacji webowej z wykorzystaniem fra- mework Django	98
5.1. Django jako framework	98
5.2. Model View Controller (MVC)	99
5.2.1. Wprowadzenie	99
5.2.2. Koncepcja	100
5.2.3. Zastosowanie MVC w Django	102
5.3. Struktura projektu w Django	103
5.3.1. Główne pliki i katalogi	104
5.3.2. Aplikacje	112
Spis ilustracji	122
Bibliografia	127

Streszczenie

Celem pracy było opracowanie aplikacji integrującej systemy zarządzania w transporcie, eliminującej problemy wynikające z różnorodności narzędzi używanych przez klientów. Transportowe firmy rejestrują dane o działaniach kierowców, jednak ręczne wprowadzanie danych jest często pomijane z powodu liczby aplikacji. Stworzony system automatyzuje przesyłanie danych, odciążając kierowców. Rozwiązanie jest skalowalne, intuicyjne w obsłudze, z GUI umożliwiającym administratorom zarządzanie integracjami, w tym dodawanie nowych aplikacji. Modułowa architektura obejmuje backend i frontend, zapewniając kompatybilność z różnorodnym oprogramowaniem, nawet przy braku dostępu do API.

Abstract

The aim of this thesis was to design an application integrating transport management systems, addressing issues from diverse client tools. Transport companies collect data on driver activities, but manual input is often skipped due to numerous applications. The system automates data transmission, relieving drivers. It is scalable, user-friendly, and features a GUI for managing integrations. Its modular architecture, with backend and frontend components, ensures compatibility with existing software and integration with platforms, even without API access.

Wstęp

Współczesny świat biznesu coraz bardziej opiera się na optymalizacji procesów operacyjnych. Usprawnienia te nie tylko podnoszą efektywność pracy, ale również poprawiają rentowność i konkurencyjność przedsiębiorstw, umożliwiając szybsze dostosowywanie się do zmieniających się potrzeb rynku oraz klientów. Dynamiczny rozwój technologii stawia przed firmami nowe wyzwania, lecz jednocześnie oferuje zaawansowane narzędzia, które pozwalają na lepsze zarządzanie danymi, procesami i zasobami.

W odpowiedzi na te potrzeby powstał projekt LogInt – integrator logistyczny, mający na celu zoptymalizowanie kluczowego aspektu działalności logistycznej: zarządzania danymi związanymi z pracą kierowców ciężarówek. Obecnie firmy transportowe, takie jak Boekestijn Transport Sp. z o.o., borykają się z problemem czasochłonnego i podatnego na błędy ręcznego wypełniania danych w wielu aplikacjach mobilnych. Proces ten angażuje liczne osoby i zasoby, co wpływa na efektywność operacyjną oraz jakość danych.

Projekt LogInt dostarcza dedykowane rozwiązanie, które automatyzuje wprowadzanie danych, wykorzystując istniejące zbiory danych firmy. Produkt został zaprojektowany jako kompleksowe narzędzie, które integruje nowoczesne technologie z istniejącą infrastrukturą klienta. Składa się z dwóch głównych komponentów:

Praca została podzielona na pięć głównych rozdziałów, które w sposób kompleksowy opisują projekt LogInt. Każdy rozdział koncentruje się na innym aspekcie realizacji i wdrożenia projektu, począwszy od jego opisu i wymagań, przez projektowanie i implementację bazy danych, aż po bardziej zaawansowane zagadnienia związane z automatyzacją procesów oraz budową backendu aplikacji webowej.

Rozdział 1 - Opis projektu LogInt

Autorzy: Daria Dworzyńska, Jakub Paszke, Miłosz Rolewski, Michał Wujec

Zawiera podstawowe informacje na temat projektu, jego celów, grupy docelowej oraz analizy konkurencji. Rozdział ten wprowadza w tematykę projektu i określa wymagania funkcjonalne, нефункционалне oraz ryzyka związane z realizacją. Dodatkowo opisuje komponenty środowiska, w którym działa LogInt.

Rozdział 2 - Projektowanie, testowanie i wdrażanie bazy danych

Autor: Daria Dworzyńska

Ten rozdział skupia się na teorii i praktyce związanej z projektowaniem oraz wdrażaniem baz danych, ze szczególnym uwzględnieniem środowiska PostgreSQL. Omówione zostały różne etapy, takie jak analiza wymagań, modelowanie koncepcyjne, a także implementacja bazy danych dla aplikacji LogInt.

Rozdział 3 - Zastosowanie OpenCV i uczenia maszynowego

Autor: Jakub Paszke

Dotyczy wykorzystania technik widzenia komputerowego oraz uczenia maszynowego w automatyzacji interfejsów mobilnych. Przedstawione są konkretne technologie, takie jak OpenCV i PyTesseract, oraz sposób ich zastosowania w praktyce w ramach projektu LogInt.

Rozdział 4 - Automatyczna interakcja z emulatorem urządzenia

mobilnego**Autor: Miłosz Rolewski**

Rozdział ten omawia integrację Android Debug Bridge (ADB) z systemem LogInt oraz automatyzację procesów na emulatorach urządzeń mobilnych. Zawiera szczegóły dotyczące środowisk emulatorów, takich jak Android Studio Emulator i Bluestacks, a także sposobów wykorzystania formatu JSON do przesyłania danych między komponentami systemu.

Rozdział 5 - Budowa backendu aplikacji webowej**Autor: Michał Wujec**

Przedstawia wykorzystanie frameworka Django do budowy backendu aplikacji LogInt. Omawia koncepcję Model-View-Controller (MVC), strukturę projektu w Django oraz szczegóły implementacji aplikacji webowych.

Deklaracja o użyciu narzędzi AI

W procesie przygotowywania niniejszej pracy inżynierskiej wykorzystano narzędzia oparte na sztucznej inteligencji, w tym model językowy GPT-4o, w celu poprawy stylistyki tekstu, wyszukiwania błędów ortograficznych oraz korekty drobnych błędów językowych. Wszystkie wprowadzone poprawki zostały poddane weryfikacji i zatwierdzone przez autora pracy, zapewniając zgodność z intencją i charakterem treści.

Rozdział 1

Opis projektu LogInt

1.1. Słownik

Podczas prac nad systemem LogInt opracowano i stosowano następujący słownik pojęć:

- **Integracja** – przygotowana w systemie LogInt logika zawierająca:
 - źródło danych,
 - zakres danych (wyekstraktowane dane ze źródła)
 - aplikację,
 - definicję wypełniania aplikacji.

Celem integracji jest umożliwienie automatyzacji procesu wypełniania danych w aplikacjach mobilnych kontrahentów.

- **Aplikacja** – przez aplikację twórcy rozumieją aplikację mobilną kontrahenta. Dotychczas dane były do niej wprowadzane ręcznie, natomiast system LogInt umożliwia kreowanie wzorców automatycznych działań (Integracji).

- **Źródło danych (source)** – jest to link do API bazodanowego klienta, z którego pobierane będą dane używane w środowisku wykonywania.
- **Wykonanie integracji** - pojedyncze wykonanie czynności określonych w definicji wypełniania aplikacji.
- **Definicja wypełniania aplikacji** – Moduł pozwalający na definiowanie akcji niezbędnych do poruszania się po aplikacji (np. kliknij przycisk, wpisz login, znajdź tekst).
- **Raport** – plik generowany po wykonaniu integracji. Zawiera informacje o:
 - danych oraz ich wartościach podczas konkretnego wykonania integracji,
 - logach ze środowiska wykonywalnego opisujących czynności, które zaszły na urządzeniu.
- **Środowisko wykonywalne** – emulator urządzenia mobilnego lub podłączony do komputera smartfon.
- **Historia** – zapis wykonanych czynności w systemie LogInt, takich jak:
 - utworzenie nowej integracji,
 - dodanie nowego źródła danych,
 - dezaktywacja istniejącej integracji.

1.2. Cel i klient

1.2.1. Cel projektu

Celem projektu LogInt było stworzenie systemu integracyjnego, który umożliwi automatyczne wypełnianie danych w aplikacjach mobilnych używanych przez kontrahentów firmy Boekestijn Transport Sp. z o.o. Kluczowe obszary, które zostały usprawnione dzięki wdrożeniu systemu to:

- Integracja danych między bazą klienta a aplikacjami kontrahentów.
- Automatyzacja uzupełniania danych o czynnościach kierowców, takich jak załadunek, rozładunek, dane ciężarówek.
- Eliminacja błędów wynikających z manualnego wprowadzania informacji.

Największym wyzwaniem dla firmy był brak uzupełniania danych w aplikacjach klienckich przez kierowców. Dane które zostały uzupełnione w jednej aplikacji (odpowiedzialnej za kontakt na linii dyspozytor firmy Boekestijn - kierowca firmy Boekestijn), najczęściej nie zostały uzupełnione przez kierowcę w drugiej aplikacji (kontrahenta). LogInt rozwiązuje ten problem, wykorzystując dane przechowywane w bazie klienta i przekazując je bezpośrednio do aplikacji mobilnych, nawet w przypadku braku API w tych aplikacjach.

Rozwiązanie obejmuje tworzenie integracji zdefiniowanych przez administratorów, którzy mogą wskazać źródło danych, określić ich zakres oraz skonfigurować harmonogramy automatycznych operacji. Dodatkowo system generuje raporty dokumentujące przeprowadzone operacje, co pozwala na bieżące monitorowanie wyników.

1.2.2. Klient - firma Boekestijn Transport Sp. z o.o

Boekestijn Transport Sp. z o.o. to międzynarodowa firma logistyczna specjalizująca się w transporcie drogowym, oferująca kompleksowe usługi przewozowe oraz obsługę celno-logistyczną. Dzięki innowacyjnemu podejściu do zarządzania procesami logistycznymi, firma zapewnia elastyczne, niezawodne i bezpieczne rozwiązania transportowe dostosowane do indywidualnych potrzeb klientów.

Firma świadczy szeroki wachlarz usług transportowych, obejmujących zarówno przewozy krajowe, jak i międzynarodowe. Główne obszary działalności to:

- Transport międzynarodowy – specjalizacja w przewozach pomiędzy krajami Europy Zachodniej, Skandynawią oraz Bałkanami. Firma obsługuje trasy m.in. między Holandią, Niemcami, Wielką Brytanią i Macedonią.
- Transport specjalistyczny – przewóz towarów wysokowartościowych (TAPA TSR), towarów niebezpiecznych (ADR) oraz farmaceutycznych (GDP).
- Usługi celne – uproszczenie procedur celnych, eliminacja błędów dokumentacyjnych oraz zapewnienie zgodności z przepisami międzynarodowymi.

Dynamiczny rozwój firmy, rosnąca liczba kontrahentów oraz zróżnicowane wymagania stawiają przed Boekestijn Transport wyzwania związane z efektywnym zarządzaniem danymi i automatyzacją procesów. Między innymi potrzeby klienta obejmują:

- Usprawnienie pracy kierowców oraz administratorów danych.

- Redukcję błędów w bazach danych.
- Integrację różnych systemów używanych przez kontrahentów.

LogInt to dedykowane narzędzie, które pozwala sprostać tym wyzwaniom, oferując rozwiązanie specjalnie dostosowane do działalności firmy. Jego elastyczność umożliwia integrację nawet z systemami, które nie posiadają interfejsu API (w szczególności z aplikacjami mobilnymi).

1.2.3. Grupa docelowa

System LogInt został zaprojektowany z myślą o administratorach danych w firmie Boekestijn Transport, którzy zajmują się obsługą procesów integracyjnych oraz zarządzaniem informacjami o czynnościach kierowców. W pierwszej fazie wdrożenia użytkownikami systemu są osoby odpowiedzialne za:

- Konfigurację źródeł danych.
- Tworzenie integracji.
- Definiowanie akcji.
- Tworzenie zależności (dane umieszczone w źródle danych wpływają na interakcję środowiska wykonywalnego z aplikacją kontrahenta).
- Zarządzanie harmonogramami przesyłania informacji do aplikacji kontrahentów.
- Monitorowanie wyników operacji poprzez generowane raporty.

Główne korzyści dla grupy docelowej to:

- Automatyzacja monotonnego i czasochłonnego procesu wypełniania danych.
- Redukcja błędów ludzkich dzięki bezpośredniej integracji danych. Usprawnienie codziennej pracy poprzez łatwy dostęp do narzędzi w jednym prostym interfejsie.
- Usprawnienie codziennej pracy poprzez łatwy dostęp do narzędzi w jednym prostym interfejsie.

Dzięki wdrożeniu systemu LogInt firma Boekestijn Transport zyska narzędzie, które nie tylko poprawi efektywność działań, ale również umożliwi szybsze reagowanie na potrzeby kontrahentów, co znacząco wpłynie na poziom satysfakcji klientów oraz konkurencyjność na rynku.

1.3. Konkurencyjne produkty

Na rynku istnieje wiele rozwiązań integracyjnych, które umożliwiają przesyłanie i synchronizację danych pomiędzy różnymi usługami. Wśród najpopularniejszych narzędzi tego typu znajdują się Power Automate [4], Zapier[2] oraz Integromat (obecnie znany jako Make) [3]. Każde z tych rozwiązań ma swoje mocne strony, jednak żadne z nich nie jest w stanie w pełni sprostać specyficznym wymaganiom, które stawia klient.

1.3.1. Power Automate

Power Automate[4] to platforma automatyzacyjna rozwijana przez Microsoft, umożliwiająca integrację różnych aplikacji i usług poprzez przepływy pracy (tzw. "flows"). Produkt opiera swoje działanie na wyzwalaczach (triggerach),

które inicjują procesy, takie jak np. wprowadzenie nowego rekordu do bazy danych, czy otrzymanie wiadomości e-mail. Następnie system wykonuje odpowiednie akcje, np. przesyłanie danych do innej aplikacji przez API. Choć Power Automate doskonale sprawdza się w środowisku integracyjnym, jego ograniczeniem jest konieczność istnienia API w aplikacjach docelowych. W przypadku kontrahentów bez takich interfejsów platforma nie jest w stanie zapewnić wymaganej funkcjonalności.

1.3.2. Zapier

Zapier[2] to popularne narzędzie low-code umożliwiające łączenie aplikacji i automatyzację procesów w środowisku chmurowym. System działa na zasadzie łańcucha "trigger → action", co oznacza, że po wykryciu określonego zdarzenia w jednej aplikacji (np. dodanie pliku do chmury) automatycznie wywoływana jest akcja w innej (np. przesłanie danych do arkusza kalkulacyjnego).

Zapier obsługuje tysiące aplikacji, co czyni go wszechstronnym narzędziem, jednak jego słabością jest ograniczona elastyczność w przypadku systemów zamkniętych lub aplikacji, które nie oferują gotowych API. Ogranicza to możliwość wykorzystania w bardziej specyficznych przypadkach.

1.3.3. Integromat (Make)

Integromat [3] to zaawansowane narzędzie do automatyzacji procesów, które pozwala na tworzenie złożonych przepływów danych między różnymi systemami. Charakteryzuje się wysoką elastycznością i możliwością konfiguracji wielu scenariuszy automatyzacji. Podobnie jak Power Automate i Zapier, Integromat działa na zasadzie wyzwalaczy i akcji, co wymaga obecności API

w aplikacjach docelowych.

Choć Integromat jest bardziej wszechstronny w porównaniu do innych platform low-code, nadal nie spełnia wymagań w sytuacjach, gdy docelowe aplikacje nie oferują interfejsów programistycznych.

1.3.4. Przewaga LogInt

W przeciwieństwie do wymienionych produktów, system LogInt jest zaprojektowany tak, aby działać także w przypadku braku API w aplikacjach kontrahentów. Dzięki wykorzystaniu technik uczenia maszynowego do wykonywania zdefiniowanych akcji w aplikacji oraz emulatorów urządzeń mobilnych LogInt umożliwia automatyczne wypełnianie danych bez konieczności modyfikacji aplikacji docelowych. Ta unikalna funkcjonalność czyni LogInt rozwiązaniem lepiej dostosowanym do specyficznych potrzeb klientów, oferując im większą elastyczność i szerokie możliwości integracyjne.

1.4. Wymagania projektu

1.4.1. Wymagania funkcjonalne

1. System autoryzacji:

- Administrator loguje się do systemu za pomocą loginu i hasła.
- Administrator ma możliwość zarządzania (dodawania/usuwania/edytowania) kontami użytkowników.
- Użytkownik, który nie jest administratorem ma dostęp do wszystkich funkcjonalności systemu, oprócz zarządzania kontami użytkowników.

2. Zarządzanie integracjami:

- Dodawanie, edycja i usuwanie integracji z aplikacjami kontrahentów.
- Dodawanie, edycja i usuwanie kierowców dla danej integracji.
- Dodawanie "Workflow" do danej integracji.
- Dodawanie "Condition" i "Step" do modułu "Workflow".
- Modyfikowanie i usuwanie "Condition", "Step", "Workflow".

3. Zarządzanie źródłami:

- Dodawanie, edycja i usuwanie źródeł ("Sources")

4. Monitorowanie historii wszystkich operacji:

- Filtrowanie operacji po:
 - Typie: Źródle ("Source"), Kierowcy ("Driver"), Integracji ("Integration")
 - Rodzaju operacji
 - Użytkownika, który wykonał operację
 - Dacie (od i do)

5. Generowanie raportów:

- Generowanie raportu po wyżej wymienionych filtrach modułu historia.

1.4.2. Wymagania niefunkcjonalne**1. Dostępność systemu:**

- Gwarancja dostępności systemu dla użytkownika (administrato-ra).
- Minimalny poziom dostępności systemu powinien wynosić 90% w ciągu każdego miesiąca kalendarzowego. Poziom ten będzie mierzony ilością pozytywnie wykonanych integracji przez ilość wszystkich integracji.

2. Wydajność systemu:

- Wysoka wydajność systemu integracji, umożliwiająca obsługę dużej liczby aplikacji i integracji równocześnie.
- Trigger będzie uruchamiany co 10 sekund, aby pobierać nowe rekordy.

3. Elastyczność systemu:

- Dodanie nowej aplikacji, nie określonej wcześniej przez klienta bez ingerencji programistów.
- Każda aplikacja zawsze ma dostęp do wszystkich niezbędnych danych udostępnianych przez klienta.

4. Zgodność z przeglądarkami:

- Kompatybilność środowiska kreacji z przeglądarką Google Chrome.

5. Skalowalność:

- W przypadku dużej ilości przekazywanych danych lub/i zwiększonych ilości zintegrowanych aplikacji, istnieje możliwość wykonywania wielu integracji jednocześnie.

1.4.3. Ryzyka projektowe

Wszystkie projekty informatyczne wiążą się z pewnego rodzaju ryzykiem, które może wpłynąć na kompletność czy też skuteczność danego rozwiązania. Szczegółowe przeanalizowanie wymagań klienta, zbadanie rozwiązań dostępnych na rynku, konsultacje społeczne czy też dogłębne rozpoznanie w technologiach, które mogłyby być pomocne przy danym projekcie, jest kluczowe, aby możliwie to ryzyko zniwelować. Opisywany w tej pracy projekt obarczony był szczególnie wysokim zagrożeniem ze strony analizy wymagań klienta, które od samego początku nie były do końca ustalone, oraz doboru technologii, gdyż produkt musiał być jak najbardziej uniwersalny dla różnorodnych aplikacji kontrahentów.

Niezgodność z oczekiwaniami klienta

Podczas współpracy z klientem, który żąda niestandardowego rozwiązania, którego nie ma na rynku należy liczyć się z ryzykiem nie do końca klarownych oraz sprecyzowanych oczekiwań względem produktu. W cyklu tworzenia systemu LogInt zdecydowano, aby być w ciągłym kontakcie z klientem w celu redukcji tego zagrożenia. Regularne prezentacje, częste spotkania w celu weryfikacji postępów, wczesna identyfikacja ewentualnych niezgodności i szybkie dostosowywanie się do nowych wymagań pozwoliły zespołowi działać płynnie, oraz dynamicznie zmieniać produkt poprzez ewentualne nawroty.

Dobór nieodpowiednich technologii i problemy z wydajnością

Z racji na innowacyjność rozwiązania wybranie odpowiedniej technologii było kluczowym elementem. O ile środowisko kreacji mogło być stworzone w różnorodnych wariantach, to środowisko wykonywalne musiało być możliwie

uniwersalne oraz otwarte na dołączanie różnorodnych narzędzi. Co więcej potencjał na zwiększenie wydajności rozwiązania był dla klienta ważny z racji na ciągły rozwój firmy i zwiększanie się przepustowości floty transportowej.

Nieintuicyjny interface

Dużym ryzykiem obarczone było również tworzenie interface'u. System LogInt od początku był tworzony z myślą o użytkowniku, który nie posiada zaawansowanych umiejętności programistycznych, lecz jest pracownikiem biurowym więc jest w stanie sprawnie korzystać z narzędzi komputerowych takich jak Excel, Power Automate czy Google Forms. Skomplikowany i nieintuicyjny interface, który sprawiałby problemy w użytkowaniu skutkowałby niską używalnością czy nawet niechęcią do systemu LogInt. Prototyp stworzony w programie Figma pozwolił przedstawić wizję zespołu developerskiego klientowi co wyeliminowało początkowe problemy. Wraz z rozwojem oprogramowania i powstawaniem nowych funkcjonalności interface stawał się co raz bardziej skomplikowany, jednak liczne konsultacje z klientem sprawiły, że w kolejnych iteracjach nanoszono poprawki mające na celu przystosowanie warstwy wizualnej aby była przyjazna użytkownikowi i intuicyjna podczas korzystania.

Opóźnienia w integracji nowych aplikacji kontrahenckich

Dodawanie do systemu integracji nowych aplikacji testowych czy też otrzymanych od kontrahentów początkowo mogło wiązać się z ryzykiem opóźnień. Nowe, dotychczas nie używane akcje które mogły być wymagane podczas korzystania z aplikacji mogły być wymagane do prawidłowego przesłania danych. Twórcy systemu LogInt musieli dynamicznie dodawać nowe komponenty, takie jak elementy widzenia komputerowego, symulacja specyficznych ru-

chów np. przewinięcie listy, czy załączenie zdjęcia z galerii urządzenia, aby co raz to nowe aplikacje mogły być integrowane w systemie. Niezwykle trudnym zagadnieniem była analiza potencjalnych akcji jakie mogą wystąpić w przyszłych integracjach, dlatego postawiono na uniwersalność w podstawowych akcjach symulowanych przez system, oraz dostępność i otwartość bibliotek Python na nowe komponenty wspierające emulacje akcji użytkownika.

1.5. Komponenty projektu

1.5.1. Środowisko kreacji integracji

System został zaprojektowany na podstawie wymagań funkcjonalnych określonych we współpracy z firmą Boekestijn Transport. Jest to platforma internetowa do zarządzania integracjami, użytkownikami oraz źródłami dla aplikacji LogInt. Do budowy backendu oraz frontendu zastosowano framework "Django", a jako system zarządzania bazą danych wykorzystano "PostgreSQL".

Korzystanie z platformy

Aby poprawnie korzystać z platformy, należy przejść przez kolejne kroki:

1. Założenie konta

- Pierwszym etapem jest utworzenie konta przez administratora systemu. Konto to jest niezbędne do zalogowania się na platformie i rozpoczęcia pracy.

2. Dodanie źródła danych

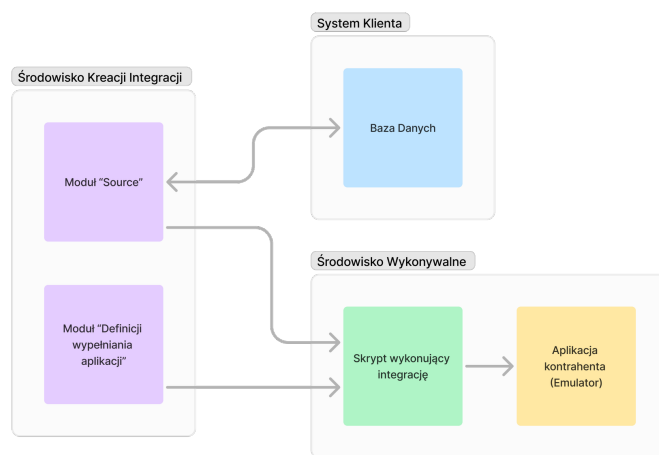
- Po zalogowaniu, w zakładce "Sources" należy utworzyć nowe źródło danych. Źródło definiuje miejsce, z którego system będzie pobierał dane. Przy jego dodawaniu podajemy link, który zawiera dane potrzebne do uzupełniania informacji w kolejnych krokach.
3. Tworzenie integracji - w zakładce "Integrations" przystępujemy do tworzenia nowej integracji, która obejmuje:
- Zadeklarowanie nazwy integracji.
 - Zdefiniowanie nazwy aplikacji, dla której dane będą uzupełniane.
 - Wybranie wcześniej utworzonego źródła, z którego będą pobierane dane.
 - Podanie nazwy klienta.
 - Załadować plik z rozszerzeniem .apk, który zawiera aplikację, w której dane będą przetwarzane.
4. Dodanie kierowców - po poprawnym utworzeniu integracji przechodzimy do jej szczegółów:
- Jeśli aplikacja tego wymaga, należy dodać kierowców, podając dla nich login i hasło.
5. Konfiguracja operacji w module "Workflow", która polega na:
- Dodawanie kroków (Step), w których określamy akcje, jakie mają zostać wykonane, oraz, opcjonalnie, operacje, które mają nastąpić po ich zakończeniu.
 - Dodawaniu bloków warunkowych (Condition), w których definiujemy, jaka operacja ma zostać wykonana, jeśli warunek zostanie spełniony, oraz co zrobić, jeśli warunek nie zostanie spełniony.

1.5.2. Środowisko wykonywalne

Skrypt wykonujący integrację

Skrypt wykonujący integrację odgrywa kluczową rolę w procesie automatyzacji i efektywnego zarządzania danymi między różnymi systemami. Jego główna funkcja polega na regularnym pobieraniu rekordów z bazy danych systemu klienta. Skrypt ten analizuje rekordy pod kątem określenia, czy dotyczą one akcji związanych z aplikacjami zintegrowanymi w systemie.

Jeżeli skrypt znajdzie odpowiednie akcje, inicjuje proces tworzenia pliku w formacie JSON. Plik ten jest kompilowany na podstawie danych z systemu klienta oraz modułu "Definicji wypełniania aplikacji", który jest częścią środowiska kreacji integracji. Moduł ten dostarcza metadane i parametry niezbędne do poprawnego sformatowania danych w pliku JSON, co umożliwia ich dalsze przetwarzanie (patrz Rysunek 3.2).



Rys. 1.1. Diagram przedstawiający proces integracji danych w systemie, z wyszczególnieniem funkcji skryptu wykonującego integrację.

Następnie, w ramach środowiska wykonawczego, skrypt korzysta z zaawansowanych technik widzenia komputerowego (CV) i optycznego rozpoznawania tekstu (OCR), aby analizować i interpretować dane zawarte w pliku JSON. Dzięki temu możliwe jest automatyczne wypełnianie aplikacji kontrahenta działającej na emulatorze Androida. Skrypt zapewnia, że aplikacja jest wypełniana w sposób zdefiniowany przez administratora systemu, co jest kluczowe dla zapewnienia spójności i dokładności przepływu danych.

Działanie skryptu stanowi zatem istotny element systemu, zapewniający ciągłość operacji i automatyzację procesów, które są krytyczne dla efektywności i skuteczności całego środowiska integracyjnego. Jest to przykład wykorzystania nowoczesnych technologii informatycznych w praktyce biznesowej, gdzie szybkość, dokładność i automatyzacja stanowią o konkurencyjności przedsiębiorstwa.

Emulator urządzenia Android i aplikacje kontrahenta

W celu symulowania automatycznych interakcji środowiska LogInt z aplikacjami kontrahenta konieczne było użycie emulatora urządzenia mobilnego lub fizycznym urządzeniu które byłoby połączone ze środowiskiem wykonywalnym. Zdecydowano się oprzeć cały system na emulatorze urządzenia z systemem Android (ADV).

AVD Manager dostarczany wraz z oprogramowaniem Android Studio stanowi wręcz idealne wręcz narzędzie do tworzenia, zarządzania oraz edycji emulatorów. Jest to narzędzie intuicyjne dla osób zaznajomionych z dokumentacją systemu LogInt, oraz pozwala w łatwy sposób skonfigurować opisywany system w nowym środowisku.

Niezwykle istotnym elementem środowiska wykonywalnego jest Android Debug Bridge (ADB). Jest to uniwersalne narzędzie powłoki, umożliwiające

ce komunikację z urządzeniem mobilnym czy jego emulatorem. Opisywany most komunikacyjny daje dostęp do wielu akcji, jednak jego pierwotnym założeniem nie jest symulacja akcji użytkowników na AVD przez co konieczne było wdrożenie bardziej złożonych funkcjonalności, które pozwoliły na proste i intuicyjne tworzenie integracji w środowisku kreacji integracji.

Poszczególne aplikacje kontrahentów są dostarczane jako pliki z rozszerzeniem .apk do systemu LogInt, a wprowadzane do środowiska wykonywalnego są za pomocą powłoki ADB, której jedną z funkcjonalności jest instalowanie pakietów z plików źródłowych, które niekoniecznie muszą występować w Sklepie Play.

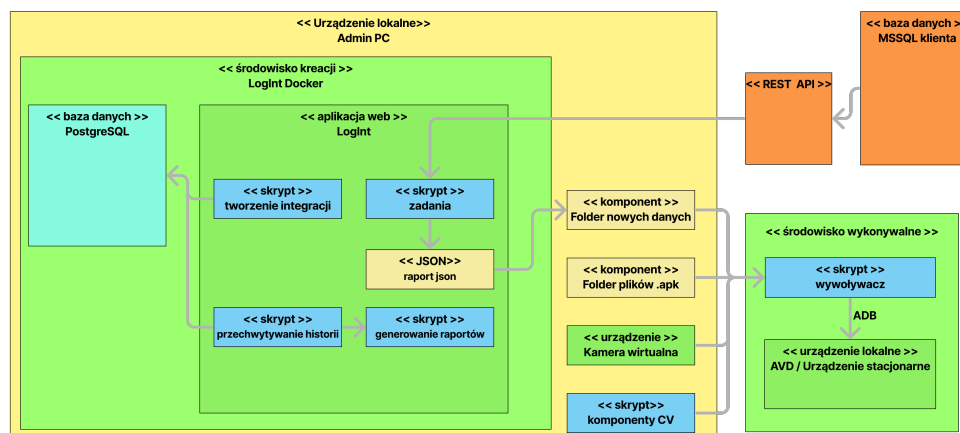
1.5.3. Architektura systemu

System LogInt opisać można jako komponent integrujący bazę danych klienta, do której dostęp uzyskano poprzez REST API z aplikacjami mobilnymi kontrahentów. Złożoność architektury 1.2 wymagała od zespołu deweloperskiego integracji wielu pomniejszych komponentów aby zapewnić funkcjonalności konieczne do stworzenia schematu integracji oraz jej późniejszego wykonania na emulatorze urządzenia mobilnego tak jakby zadeklarowane akcje wykonywał człowiek.

1.6. Metodyka pracy i role w projekcie

1.6.1. Metodyka pracy

Ze względu na złożoność oraz innowacyjność projektu zespół zdecydował się prowadzić go w metodyce Scrum. Metodyka ta zalicza się do metod zwinnych (Agile) i pozwala na efektywne zarządzanie dynamicznie zmieniającymi



Rys. 1.2. Diagram przedstawiający architekturę systemu LogInt.

się projektami. Dzięki każdej iteracji, zwanej sprintem, zespół mógł wdrażać coraz to nowe funkcjonalności. Zmieniające się podejście zespołu do projektu oraz głębsze zrozumienie problematyki prowadziły do nawrotów, które umożliwiały udoskonalenie lub lepsze wdrożenie istniejących już rozwiązań. Spotkania stand-up pozwalały również na zachowanie ciągłej komunikacji w zespole, dzięki czemu każdy członek wiedział, co dzieje się w obszarach nieobjętych jego obowiązkami. Umożliwiały także wspólne rozpoznawanie problemów w większym gronie, co z kolei prowadziło do ich efektywnego rozwiązania.

1.6.2. Role w projekcie

W zespole zdecydowano, że każdy z jego członków będzie zajmował się rozwojem aplikacji pod kątem programistycznym, co pozwala stwierdzić, że wszyscy należeli do zespołu deweloperskiego. Dodatkowo wprowadzono dwie role, które znacznie usprawniły prowadzenie projektu w metodyce Scrum.

- **Scrum Master** – zajmował się planowaniem spotkań stand-up, przy-

pisywaniem zadań poszczególnym członkom zespołu oraz dbaniem o równy podział pracy w zespole.

- **Product Owner** – do jego zadań należało utrzymanie ciągłości kontaktu z klientem, tworzenie user stories oraz określanie wymagań dotyczących funkcjonalności, jakie projekt miał realizować.
- **Zespół deweloperski** – przede wszystkim zajmował się wdrażaniem funkcjonalności opisywanych w user stories, dzielonych na poszczególne zadania. Dostarczał również gotowe produkty, takie jak nowe moduły czy funkcje, na przestrzeni poszczególnych przyrostów oraz dbał o integrację komponentów.

Rozdział 2

Projektowanie, testowanie i wdrażanie bazy danych

Autor: Daria Dworzyńska

2.1. Wprowadzenie

Współczesne aplikacje opierają swoją funkcjonalność na bazach danych, które stanowią fundament zarządzania informacjami w niemal każdym przedsiębiorstwie. Poprawne projektowanie, implementacja oraz testowanie baz danych jest niezbędne dla zapewnienia niezawodności i wydajności systemów informatycznych. Praca przedstawia proces projektowania i wdrażania bazy danych na przykładzie aplikacji LogInt, która została stworzona w odpowiedzi na potrzeby biznesowe firmy Boekestijn Transport. Omówione zostaną zarówno teoretyczne aspekty projektowania baz danych, jak i praktyczne kroki ich implementacji, uwzględniając realne wyzwania pojawiające się w trakcie pracy nad projektem.

2.2. Wstęp teoretyczny

Baza danych to uporządkowany zbiór danych opisujących pewien fragment rzeczywistości. Pozwala ona na przechowywanie, zarządzanie i przetwarzanie informacji w sposób zorganizowany i zoptymalizowany. Przykładowo, baza danych banku może zawierać dane o klientach, ich rachunkach, operacjach finansowych czy udzielonych kredytach. Dzięki zastosowaniu odpowiednich systemów zarządzania bazami danych (DBMS), informacje te mogą być łatwo przetwarzane i analizowane, co czyni bazy danych bardzo ważnym narzędziem w codziennym funkcjonowaniu wielu organizacji. [39]

2.2.1. Języki baz danych

Do opisu, tworzenia oraz zarządzania bazą danych wykorzystuje się specjalistyczny język - w tym celu opracowano Strukturalny Język Zapytań, znany jako SQL (Structured Query Language). SQL należy do grupy języków deklaratywnych, co oznacza, że użytkownik określa jedynie, co chce uzyskać, bez konieczności wskazywania, w jaki sposób system zarządzania bazą danych ma to zrobić. DBMS samodzielnie dobiera najbardziej efektywną metodę wykonania zapytania. [24]

Spójrzmy bardziej szczegółowo na język baz danych. Każdy język powinien mieć zdefiniowaną grupę poleceń [29]:

- DDL (Data Definition Language) - polecenia do definiowania struktury bazy danych, takie jak CREATE, DROP czy ALTER.
- DML (Data Manipulation Language) - instrukcje do operacji na danych (manipulacji danymi), m.in. SELECT, INSERT, UPDATE, DELETE.

- DCL (Data Control Language) - narzędzia do zarządzania dostępem do danych, np. GRANT, REVOKE.

2.2.2. Modele baz danych

Modele baz danych to sposoby organizowania i przechowywania informacji. Model opisuje, jak dane są strukturyzowane i przechowywane w bazie. Przed zapisaniem danych należy ustalić, w jakiej formie będą one przechowywane[27].

Do najczęściej stosowanych modeli baz danych należą:

Model hierarchiczny - Dane są zorganizowane w strukturę drzewa, w której każdy zbiór nadrzędny (rodzic) może mieć jeden lub więcej podzbiorów (potomków). Informacje są przechowywane zarówno w węzłach drzewa, jak i w ich strukturze. Przykładem może być organizacja systemu plików na serwerze, gdzie katalog główny zawiera podfoldery z różnymi dokumentami.

Model sieciowy - W tym modelu dane są powiązane w formie sieci, umożliwiając dowolne połączenia między zbiorami. Dane są przechowywane w węzłach, a relacje między nimi w połączeniach sieci. Przykładem może być system rezerwacji lotów, w którym lotniska (węzły) są połączone trasami (relacje), co pozwala na szybkie wyszukiwanie połączeń między dowolnymi lotniskami.

Model obiektowy - Łączy cechy programowania obiektowego z funkcjonalnością baz danych. Dane są reprezentowane jako obiekty, które mogą zawierać zarówno dane, jak i operacje na nich. Przykładem zastosowania tego modelu może być baza danych w aplikacji do zarządzania projektami, gdzie obiekt "Projekt" zawiera dane (np. nazwę, daty, budżet) oraz metody (np. obliczanie kosztów, zarządzanie harmonogramem).

Model relacyjny - Najpopularniejszy model baz danych, oparty na

przechowywaniu danych w tabelach. Tabele są połączone relacjami, które umożliwiają łatwe wyszukiwanie i łączenie danych. Przykładem jest baza danych sklepu internetowego. Jedna tabela może przechowywać dane o klientach (np. imię, nazwisko, adres), a inna informacje o zamówieniach (np. numer zamówienia, data, wartość). Tabele te są połączone kluczem wspólnym, np. identyfikatorem klienta. Dane w tabelach są podzielone na rekordy (wiersze) i pola (kolumny). Każde pole ma określony typ danych (np. tekstowy, liczbowy, logiczny). Przykładem typu danych może być pole "Cena produktu", które ma format liczbowy, albo "Data zakupu", które ma format daty. Proces organizacji danych w tabelach nazywa się normalizacją.

Model relacyjny jest najczęściej wykorzystywany, ponieważ oferuje dużą elastyczność, łatwość przetwarzania danych oraz szerokie wsparcie w systemach bazodanowych, takich jak MySQL czy PostgreSQL.

2.2.3. Architektura komunikacyjna baz danych

Architektura komunikacyjna baz danych to inaczej sposób w jaki baza danych komunikuje się z klientem.[26] Wyróżniamy:

- Architekturę jednowarstwową - W architekturze jednowarstwowej baza danych, interfejs użytkownika i logika aplikacji znajdują się na tym samym komputerze lub serwerze. Ponieważ w tej architekturze nie występują opóźnienia sieciowe, jest to zazwyczaj szybki sposób dostępu do danych.
- Architekturę dwuwarstwową - Architektura dwuwarstwowa składa się z wielu klientów łączących się bezpośrednio z bazą danych. Ten rodzaj architektury, znany również jako architektura klient-serwer, był bardziej powszechny w czasach, gdy aplikacje desktopowe łączyły się z poje-

dynczą bazą danych hostowaną na serwerze bazodanowym w siedzibie firmy. Przykładem może być wewnętrzny system zarządzania relacjami z klientami (CRM), który łączy się z bazą danych Access.

- Architekturę trójwarstwową - Większość nowoczesnych aplikacji internetowych korzysta z architektury trójwarstwowej. W tej architekturze klienci łączą się z warstwą zaplecza (back-end), która następnie łączy się z bazą danych.

2.2.4. Podział systemów baz danych

Bazy danych możemy sklasyfikować ze względu na:[32]

Liczbę użytkowników

- dla jednego użytkownika - dopuszczone jest tylko jedno połączenie do bazy danych w tym samym czasie
- dla wielu użytkowników - dozwolony jest dostęp wielu użytkowników do jednej bazy danych w tym samym czasie

Lokalizację

- scentralizowane - bazy danych znajdują się w jednym miejscu, dostęp do wszystkich danych jest zebrany w jednym miejscu
- rozproszone - "zespół baz danych znajdujących się na różnych komputerach połączonych ze sobą w taki sposób, że użytkownik nie wie, iż dane, z którymi pracuje, pochodzą z różnych baz i komputerów." [37]

Model danych

Jak opisano w 2.2.2

Sposób użycia

- Operacyjne bazy danych (OLTP ang. On Line Transaction Processing) - używane do zarządzania danymi w czasie rzeczywistym. Wykorzystuje dane dynamiczne czyli takie, które ulegają ciągłej zmianie i przedstawiają aktualny stan rzeczy, której dotyczą np. bazy inwentaryzacyjne
- Analityczne bazy danych (OLAP ang. On Line Analytical Processing) - dane są braze z OLTP, jednak po wprowadzeniu ich do bazy analitycznej nie są modyfikowane. Wykorzystuje się je głównie do wyszukiwania danych, przeprowadzania analiz, czy zestawień statystycznych

Typ danych

- Ogólnego przeznaczenia
- Temporalne - jest to baza danych przechowująca informacje o czasie wprowadzenia danych lub pamiętać kiedy dana akcja (np. dodanie lub usunięcie danych) miała miejsce [30]
- Strumieniowe - jest to program komputerowy przetwarzający dane napływające z dużą szybkością w postaci nieskończonych strumieni, umożliwiający analizę serii czasowych. Znajduje zastosowanie m.in. w medycznych systemach monitorujących,
- Dedukcyjne - jest to system bazodanowy, który potrafi wnioskować nowe informacje bazując na wcześniej zapisanych zasadach i informacjach w bazie. Wykorzystywany za pomocą użycia języka Datalog

- Multimedialne - są to systemy baz danych, które służą do przechowywania i wyszukiwania danych związanych z zawartościami multimedialnymi, takimi jak muzyka, filmy czy grafika.
- Przestrzenne - Baza danych przestrzennych, zwana także bazą danych geograficznych, to system zaprojektowany do efektywnego przechowywania i przetwarzania informacji o obiektach zlokalizowanych w przestrzeni geometrycznej.

Zużycie zasobów

- mikro bazy danych
- bazy danych dla urządzeń mobilnych
- bazy danych czasu rzeczywistego
- bazy danych w pamięci operacyjnej

2.3. Etapy projektowania baz danych

Projektowanie baz danych to złożony proces, który wymaga odpowiedniego zrozumienia potrzeb biznesowych oraz technicznych danego przedsiębiorstwa. Każdy etap projektowania ma na celu zapewnienie, że baza danych będzie efektywna, skalowalna i łatwa w utrzymaniu. Poniżej przedstawiono główne etapy projektowania baz danych.

2.3.1. Analiza wymagań (firmy)

Pierwszym krokiem w projektowaniu bazy danych jest analiza wymagań przedsiębiorstwa. Proces ten obejmuje zrozumienie charakteru działalności,

rozmiaru organizacji oraz jej specyficznych potrzeb w zakresie zarządzania danymi. Na podstawie tych informacji można określić główne cele bazy danych i zaprojektować jej strukturę w sposób najlepiej odpowiadający tym wymaganiom.

W przypadku firmy Boekestijn Transport Sp. z o.o. analiza wymagań koncentrowała się na następujących aspektach:

- Rozmiar firmy i jej struktura organizacyjna, obejmująca międzynarodowy zasięg działania.
- Rodzaj usług oferowanych przez firmę, w tym transport międzynarodowy, przewozy specjalistyczne oraz usługi celne.
- Specyficzne wyzwania związane z zarządzaniem danymi, takie jak integracja z systemami kontrahentów, automatyzacja procesów oraz eliminacja błędów wynikających z manualnego wprowadzania danych.

Analiza wykazała, że baza danych musi obsługiwać dużą ilość danych, umożliwiać ich szybkie przetwarzanie oraz integrację z aplikacjami mobilnymi kontrahentów, nawet w przypadku braku interfejsu API tych aplikacji.

2.3.2. Wymagania funkcjonalne i нефunkcjonalne

Zdefiniowanie wymagań funkcjonalnych i нефunkcjonalnych jest kluczowe dla prawidłowego określenia modeli danych oraz zależności między nimi.

Wymagania funkcjonalne

Wymagania funkcjonalne określają, jakie zadania bazy danych powinna realizować. W przypadku Boekestijn Transport zidentyfikowano następujące funkcjonalności:

- Automatyczne wypełnianie danych w aplikacjach mobilnych kontrahentów, wykorzystując dane przechowywane w bazie firmy.
- Obsługa integracji z systemami kontrahentów, w tym definiowanie harmonogramów przesyłania danych.
- Generowanie raportów dokumentujących przeprowadzone operacje.
- Monitorowanie historii operacji w celu śledzenia zmian i zapewnienia zgodności z wymaganiami biznesowymi.

Wymagania niefunkcjonalne

Wymagania niefunkcjonalne określają, jakie cechy techniczne powinna posiadać baza danych. Główne wymagania niefunkcjonalne dla systemu obejmują:

- **Wydaźność:** Baza danych musi obsługiwać dużą ilość operacji jednocześnie, zapewniając krótki czas odpowiedzi.
- **Dostępność:** System powinien być dostępny co najmniej przez 90% miesiąca
- **Skalowalność:** Struktura bazy danych musi umożliwiać rozbudowę w miarę wzrostu liczby danych i nowych wymagań biznesowych.
- **Bezpieczeństwo:** System musi zapewniać odpowiedni poziom ochrony danych, szczególnie w kontekście wrażliwych informacji.
- **Elastyczność:** Rozwiązanie powinno umożliwiać szybkie dostosowywanie do zmieniających się potrzeb biznesowych, takich jak dodawanie nowych integracji.

Dzięki jasno zdefiniowanym wymaganiom funkcjonalnym i нефункциональным, możliwe było opracowanie modeli danych oraz relacji między nimi, co zapewniło efektywne działanie systemu LogInt.

2.3.3. Modelowanie koncepcyjne (diagramy UML)

Diagram klas to najczęściej używany typ diagramu w procesie tworzenia oprogramowania. Służy do przedstawiania logicznego i fizycznego projektu systemu oraz ilustruje jego klasy. Wygląda podobnie do schematu blokowego, ponieważ klasy są reprezentowane za pomocą pól [31]. Każda klasa w diagramie posiada trzy sekcje:

- **Sekcja górna:** zawiera nazwę klasy.
- **Sekcja środkowa:** przedstawia atrybuty klasy.
- **Sekcja dolna:** definiuje metody lub operacje klasy.

2.3.4. Implementacja baz danych na przykładzie aplikacji LogInt

Implementacja baz danych w aplikacji LogInt obejmuje szczegółowe odwzorowanie tabel, encji oraz zależności między nimi, co umożliwia realizację wymagań narzuconych przez firmę. Główne tabele w systemie to

- History
- Integration
- Integration Account
- Source

- Workflow
- Steps
- Condition

Zależności między tabelami są realizowane głównie za pomocą kluczy obcych, co zapewnia integralność danych.

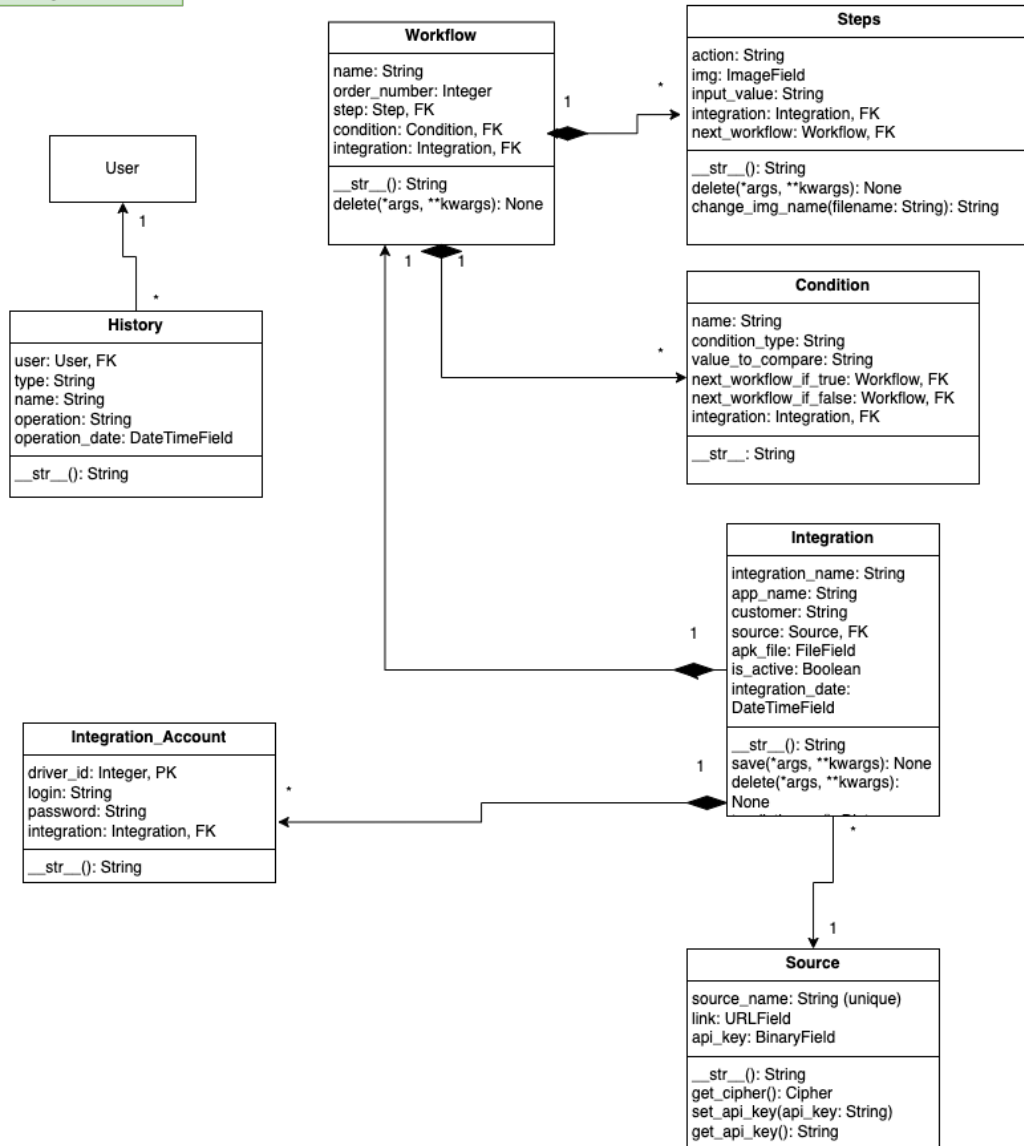
2.3.5. Środowisko pracy - PostgreSQL - implementacja bazy danych

Do implementacji bazy danych wybrano PostgreSQL, które jest najbardziej zaawansowanym open source system relacyjnych baz danych. Wybór tej technologii uzasadnia jej wydajność, elastyczność oraz możliwość pracy z dużymi zbiorami danych.

Zalety PostgreSQL:

- Obsługa wielu typów danych m.in - podstawowych (takich jak liczby całkowite, czy zmienne przecinkowe), strukturalnych (data i czas), czy dokumentowych (umożliwione przechowywanie danych w formatach JSON, czy XML)
- Integralność danych - Mechanizmy takie jak UNIQUE, NOT NULL, klucze główne i obce zapewniają spójność danych.
- Współbieżność i wydajność - wsparcie dla różnego typu indeksów, równoległe zapytania odczytowe i budowanie indeksów.
- Niezawodność i odzyskiwanie danych - Dziennikowanie Write-Ahead Logging (WAL), umożliwiające odbudowę danych po awarii.

Diagram UML



Rys. 2.1. Diagram klas systemu LogInt - rysunek własny

- Rozszerzalność - Obsługa konstruktorów SQL/JSON, zapytań funkcjonalnych oraz wyrażeń ścieżkowych

Podsumowując, PostgreSQL jest wszechstronnym narzędziem, które odpowiednio wpisało się w potrzeby i wymagania projektu LogInt. [21] [20]

2.3.6. Modelowanie koncepcyjne i logiczne

Modelowanie koncepcyjne i logiczne w aplikacji LogInt stanowiło podstawę do opracowania tabel i atrybutów bazy danych. Główne decyzje projektowe obejmowały:

- Klucze główne (Primary Key): W każdej tabeli zdefiniowano unikalne identyfikatory, takie jak id dla tabel Integration czy driver id w tabeli Integration Account.
- Klucze obce (Foreign Key): Relacje między tabelami są zrealizowane poprzez klucze obce, np. integration w tabeli Workflow odwołuje się do tabeli Integration.
- Atrybuty opcjonalne: W niektórych przypadkach atrybuty mogą przyjmować wartość NULL, np. source w tabeli Integration, co umożliwia większą elastyczność przy wprowadzaniu danych.
- Zasady integralności: Ustalono, że dane krytyczne, takie jak operation date, nie mogą być NULL, aby zapewnić spójność danych.
- Dane zewnętrzne: Zdefiniowano mechanizmy importowania danych z raportów lub systemów zewnętrznych, szczególnie w kontekście źródeł danych dla tabeli Source.

Poniżej znajdują się zadeklarowane modele:

Listing 2.1. Model Historii w systemie LogInt

```
1 class History(models.Model):
2     user = models.ForeignKey(User, on_delete=models.
3         CASCADE)
4     type = models.CharField(max_length=50)
5     name = models.CharField(max_length=100)
6     operation = models.CharField(max_length=50)
7     operation_date = models.DateTimeField(default=
8         timezone.now)
9
10    def __str__(self):
11        return f"{self.type} | {self.name} | \
12            {self.operation} | {self.user.username}"
```

Model History w Django służy do przechowywania informacji o działaniach użytkowników w aplikacji. Pola modelu obejmują:

- **user** - klucz obcy do modelu **User**, wskazujący, który użytkownik wykonał operację.
- **type** oraz **name** - pola tekstowe, określające typ i nazwę obiektu, którego dotyczy operacja.
- **operation** - pole tekstowe opisujące rodzaj wykonanej operacji.
- **operation_date** - znacznik czasu z aktualną datą i godziną (`timezone.now`) określający, kiedy dana operacja się wykonała

Listing 2.2. Modele "Integration" i "Integration Account" w systemie LogInt

```
1 class Integration(models.Model):
```

```
2     integration_name = models.CharField(max_length
      =100)
3     app_name = models.CharField(max_length=100)
4     customer = models.CharField(max_length=100)
5     source = models.ForeignKey(Source, on_delete=
      models.SET_NULL, null=True)
6     apk_file = models.FileField(upload_to='apk_files/'
      )
7     is_active = models.BooleanField(default=True)
8     integration_date = models.DateTimeField(default=
      timezone.now)
9
10    def __str__(self):
11        return self.integration_name
12
13    def save(self, *args, **kwargs):
14        if self.id:
15            existing = Integration.objects.get(id=self
              .id)
16            if (
17                existing.apk_file
18                and self.apk_file
19                and existing.apk_file != self.apk_file
20            ):
21                if os.path.isfile(existing.apk_file.
                  path):
22                    os.remove(existing.apk_file.path)
23
24        super().save(*args, **kwargs)
```

```
25
26     if self.apk_file:
27         new_name = f"{self.id}.apk"
28         new_path = os.path.join('apk_files',
29                                 new_name)
29
30         if self.apk_file.name != new_path:
31             old_path = self.apk_file.path
32             new_full_path = os.path.join(settings.
33                                         MEDIA_ROOT, new_path)
34             os.rename(old_path, new_full_path)
35             self.apk_file.name = new_path
36             super().save(update_fields=['apk_file',
37                                       ])
38
39 def delete(self, *args, **kwargs):
40     if self.apk_file:
41         if os.path.isfile(self.apk_file.path):
42             os.remove(self.apk_file.path)
43         super().delete(*args, **kwargs)
44
45 def to_dictionary(self):
46     fields = ['integration_name', 'app_name', '
47             customer']
48     integration_dict = model_to_dict(self, fields=
49     fields)
50
51     integration_dict['source'] = None
52     if self.source:
```

```
49         integration_dict['source'] = self.source.  
            source_name  
50  
51         integration_dict['apk_file'] = None  
52         if self.apk_file:  
53             integration_dict['apk_file'] = self.  
                apk_file.url  
54  
55         return integration_dict  
56  
57  
58 class Integration_Account(models.Model):  
59     driver_id = models.IntegerField(primary_key=True)  
60     login = models.CharField()  
61     password = models.CharField()  
62     integration = models.ForeignKey(  
63         Integration, on_delete=models.CASCADE, null=  
            True, blank=True  
64     )  
65  
66     def __str__(self):  
67         return self.login
```

Model `Integration` w Django reprezentuje integracje aplikacyjne oraz informacje o klientach i plikach APK.

1. Pole `integration.name` przechowuje nazwę integracji jako ciąg znaków (`CharField`) o maksymalnej długości 100 znaków.
2. Pole `app_name` przechowuje nazwę aplikacji (`CharField`) o maksymal-

nej długości 100 znaków.

3. Pole `customer` przechowuje nazwę klienta (`CharField`) o maksymalnej długości 100 znaków.
4. Pole `source` to klucz obcy do modelu `Source`, które umożliwia powiązanie z danym źródłem. Może być `null`.
5. Pole `apk_file` (`FileField`) przechowuje plik APK, który jest zapisywany w katalogu `apk_files/`.
6. Pole `is_active` to pole logiczne (`BooleanField`), które wskazuje, czy integracja jest aktywna.
7. Pole `integration_date` przechowuje datę i godzinę utworzenia integracji (`DateTimeField`) z domyślną wartością ustawioną na bieżący czas.
8. Metoda `__str__` zwraca nazwę integracji.
9. Metoda `save` usuwa istniejący plik APK, jeśli nowy plik zostaje przesłany, oraz zmienia nazwę pliku na format `{id}.apk`.
10. Metoda `delete` usuwa powiązany plik APK z systemu plików przy usuwaniu obiektu.
11. Metoda `to_dictionary` zwraca słownik zawierający wybrane pola modelu, takie jak `integration_name`, `app_name`, `customer`, a także `source` (nazwa źródła) i `apk_file` (adres URL do pliku).

Model `Integration_Account` w Django reprezentuje konta użytkowników powiązane z integracjami.

1. Pole `driver_id` to klucz główny (`IntegerField`) identyfikujący użytkownika.
2. Pole `login` przechowuje nazwę użytkownika (`CharField`).
3. Pole `password` przechowuje hasło użytkownika (`CharField`).
4. Pole `integration` to klucz obcy do modelu `Integration`, umożliwiający powiązanie konta z konkretną integracją. Jest opcjonalne (`null=True`, `blank=True`).
5. Metoda `__str__` zwraca login użytkownika.

Listing 2.3. Model "Source" (Źródło) w systemie LogInt

```
1
2 class Source(models.Model):
3     source_name = models.CharField(max_length=255,
4                                     unique=True)
5     link = models.URLField()
6     api_key = models.BinaryField(null=True, blank=True)
7
8     def __str__(self):
9         return self.source_name
10
11     @staticmethod
12     def get_cipher():
13         fernet_key = settings.FERNET_KEY
14         if not fernet_key:
15             raise ValueError("FERNET_KEY not found")
16         return Fernet(fernet_key.encode())
```



```
16
17     def set_api_key(self, api_key: str):
18         cipher = self.get_cipher()
19         self.api_key = cipher.encrypt(api_key.encode()
20                                     )
21
22     def get_api_key(self):
23         if not self.api_key:
24             raise ValueError("API key is not set or
25                             missing from the database.")
26
27         if isinstance(self.api_key, memoryview):
28             encrypted_key = self.api_key.tobytes()
29         else:
30             encrypted_key = self.api_key
31
32         cipher = self.get_cipher()
33         try:
34             return cipher.decrypt(encrypted_key).
35                               decode()
36         except (TypeError, ValueError) as e:
37             raise ValueError("API key decryption
38                             failed.") from e
```

Model Source w Django reprezentuje zewnętrzne źródła, takie jak API lub inne usługi sieciowe.

1. Pole `source_name` przechowuje unikalną nazwę źródła jako ciąg znaków o maksymalnej długości 255 znaków.

2. Pole `link` jest typu `URLField` i przechowuje adres URL danego źródła.
3. Pole `api_key` to `BinaryField`, które przechowuje zaszyfrowany klucz API; jest opcjonalne (`null=True`, `blank=True`).
4. Metoda `__str__` zwraca nazwę źródła, co upraszcza jego reprezentację w interfejsie administracyjnym lub debugowaniu.
5. Statyczna metoda `get_cipher` tworzy obiekt szyfrujący przy użyciu klucza `FERNET_KEY` z ustawień aplikacji.
6. Jeśli klucz `FERNET_KEY` jest nieobecny, metoda `get_cipher` zgłasza wyjątek `ValueError`.
7. Metoda `set_api_key` umożliwia zapisanie klucza API w zaszyfrowanej postaci za pomocą algorytmu Fernet.
8. Metoda `get_api_key` odszyfrowuje i zwraca klucz API, jeśli jest on dostępny w bazie danych.
9. Jeśli klucz API jest nieobecny, metoda `get_api_key` zgłasza wyjątek `ValueError`.
10. W przypadku błędu odszyfrowania klucza API zgłaszany jest wyjątek `ValueError`, z informacją o niepowodzeniu operacji.

Listing 2.4. Modele "Workflow", "Step" i "Condition" w systemie LogInt

```
1 class Workflow(models.Model):
2     name = models.CharField(max_length=128)
3     order_number = models.IntegerField()
4     step = models.ForeignKey(
5         'Step',
```

```
6         null=True,
7         blank=True,
8         on_delete=models.CASCADE,
9         related_name='step_workflow',
10    )
11    condition = models.ForeignKey(
12        'Condition',
13        null=True,
14        blank=True,
15        on_delete=models.CASCADE,
16        related_name='condition_workflow',
17    )
18    integration = models.ForeignKey(Integration,
19                                   on_delete=models.CASCADE)
19
20 class Step(models.Model):
21     action = models.CharField(max_length=10, choices=
22                               ACTION_CHOICES.items())
23     img = models.ImageField()
24     input_value = models.CharField(max_length=128,
25                                    null=True, blank=True)
26     integration = models.ForeignKey(Integration,
27                                    on_delete=models.CASCADE)
28     next_workflow = models.ForeignKey(
29         Workflow,
30         null=True,
31         blank=True,
32         on_delete=models.SET_NULL,
33         related_name='next_workflow',
```

```
31     )
32
33
34 class Condition(models.Model):
35     CONDITION_CHOICES = [
36         ('equals', 'Equals'),
37         ('greater_than', 'Greater Than'),
38         ('less_than', 'Less Than'),
39         ('contains', 'Contains'),
40         ('starts_with', 'Starts With'),
41         ('ends_with', 'Ends With'),
42     ]
43
44     name = models.CharField(max_length=128)
45     condition_type = models.CharField(
46         max_length=50,
47         choices=CONDITION_CHOICES,
48     )
49     value_to_compare = models.CharField(
50         max_length=128,
51         help_text="The value against which to compare.",
52     )
53     next_workflow_if_true = models.ForeignKey(
54         Workflow,
55         null=True,
56         blank=True,
57         on_delete=models.SET_NULL,
58         related_name="true_conditions",
```

```
59     )
60     next_workflow_if_false = models.ForeignKey(
61         Workflow,
62         null=True,
63         blank=True,
64         on_delete=models.SET_NULL,
65         related_name="false_conditions",
66     )
```

Model `Workflow` w Django reprezentuje procesy składające się z kroków (`Step`) i warunków (`Condition`), które są wykonywane w ustalonej kolejności. Poniżej znajduje się szczegółowy opis struktur danych:

Model Workflow

1. Pole `name` przechowuje nazwę procesu jako ciąg znaków o maksymalnej długości 128 znaków.
2. Pole `order_number` przechowuje liczbę całkowitą, reprezentującą kolejność procesu.
3. Relacja `step` wskazuje opcjonalnie na model `Step` za pomocą klucza obcego (`ForeignKey`) z ustawieniami `null=True` i `blank=True`.
4. Relacja `condition` wskazuje opcjonalnie na model `Condition`, również jako klucz obcy.
5. Pole `integration` łączy workflow z integracją (`Integration`).
6. Metoda `delete` usuwa powiązane kroki i warunki przed usunięciem samego obiektu.

Model Step

1. Pole `action` przechowuje rodzaj akcji, wybieranej spośród predefiniowanych opcji (`ACTION_CHOICES`).
2. Pole `img` przechowuje obraz związany z krokiem.
3. Pole `input_value` przechowuje opcjonalny tekst wejściowy.
4. Relacja `next_workflow` wskazuje na następny proces (`Workflow`).
5. Metoda `delete` usuwa obraz z systemu plików przy usuwaniu kroku.
6. Metoda `change_img_name` umożliwia zmianę nazwy obrazu w oparciu o informacje o kroku i integracji.

Model Condition

1. Pole `name` przechowuje nazwę warunku.
2. Pole `condition_type` wskazuje rodzaj porównania (np. `equals`, `greater_than`).
3. Pole `value_to_compare` przechowuje wartość do porównania.
4. Relacje `next_workflow_if_true` i `next_workflow_if_false` wskazują na następny proces w zależności od wyniku warunku.
5. Pole `integration` łączy warunek z integracją.

2.4. Podsumowanie

Podsumowując, projekt LogInt pokazuje, jak teoretyczne zasady projektowania baz danych mogą być zastosowane w praktyce, odpowiadając na konkretne potrzeby przedsiębiorstwa. Dzięki odpowiedniej analizie i narzędziom

udało się stworzyć rozwiązanie wydajne, bezpieczne i gotowe na przyszłe wyzwania związane z rozwojem firmy.

Rozdział 3

Zastosowanie OpenCV i uczenia maszynowego w automatyzacji interfejsów mobilnych

Autor: Jakub Paszke

3.1. Wprowadzenie

Tradycyjne podejścia do automatyzacji aplikacji mobilnych opierają się na bezpośredniej interakcji z elementami interfejsu użytkownika (UI) poprzez identyfikację ich w strukturze drzewa dokumentu aplikacji. Jedną z powszechnie stosowanych metod jest wykorzystanie XPath do lokalizowania elementów na podstawie ich atrybutów, hierarchii czy pozycji[8]. XPath umożliwia precyzyjne wskazanie elementu w strukturze aplikacji, co czyni go przydatnym w tworzeniu skryptów testowych. Niemniej jednak, w przypadkach kiedy pożądana jest automatyzacja programów, których struktura może się zmieniać, XPath nie spełnia swojej roli.[38]

Innym przykładem istotnych narzędzi do automatyzacji aplikacji mobilnych jest Appium, które umożliwia testowanie aplikacji mobilnych na różnych platformach (np. Android, iOS). Appium wspiera zarówno aplikacje natywne, jak i hybrydowe, oferując elastyczność dzięki wykorzystaniu standardowego protokołu JSON Wire Protocol. W literaturze podkreśla się zalety tego narzędzia, takie jak obsługa wielu języków programowania, ale również wskazuje na wyzwania związane z konfiguracją i wydajnością, zwłaszcza przy testach na emulatorach [8].

Innym podejściem jest wykorzystanie natywnych SDK do testowania, takich jak UI Automator dla Androida czy XCUITest dla iOS. Narzędzia te pozwalają na głębszą integrację z platformą, co poprawia stabilność i wydajność testów. Jednak ich użycie wymaga zaawansowanej wiedzy o ekosystemach poszczególnych systemów operacyjnych, co również wiąże się z dodatkowymi kosztami dla stron, które chciałyby z takiego rozwiązania skorzystać.

Warto zauważyć, że te tradycyjne rozwiązania techniczne służą przede wszystkim automatyzacji testowania i utrzymania aplikacji, a niekoniecznie imitowaniu jej rzeczywistego użycia. Dlatego nie są optymalnym wyborem dla tworzenia rozwiązań typu RPA (Robotic Process Automation) dedykowanego urządzeniom mobilnym.

Mimo to tradycyjne podejścia, takie jak XPath, Appium czy natywne SDK, wciąż znajdują zastosowanie. Ich ograniczenia – takie jak brak odporności na zmiany w interfejsie oraz skomplikowana konfiguracja – motywują jednak do eksploracji bardziej zaawansowanych technologii. W szczególności techniki oparte na uczeniu maszynowym i wizji komputerowej oferują nowe możliwości: analiza tekstu wyświetlanego na urządzeniu i rozpoznawanie elementów UI umożliwia tworzenie bardziej elastycznych i skalowalnych rozwiązań automatyzacyjnych.

3.2. Podstawy teoretyczne

3.2.1. Uczenie maszynowe w rozpoznawaniu obiektów

Uczenie maszynowe to dziedzina sztucznej inteligencji, której celem jest rozwijanie algorytmów zdolnych do nauki na podstawie danych, zamiast opierania się na statycznych regułach programistycznych. Proces ten obejmuje automatyczne uczenie się z przykładów poprzez analizowanie wzorców w danych i wykorzystanie ich do podejmowania decyzji lub prognozowania. Kluczową rolę odgrywają tutaj techniki nadzorowanego, nienadzorowanego i półnadzorowanego uczenia. [23]

W tradycyjnym podejściu algorytmy uczenia maszynowego opierały się na ręcznie definiowanych cechach, takich jak kontury, krawędzie, tekstury czy kolory, które stanowiły podstawę do klasyfikacji obrazów. Metody takie jak maszyny wektorów nośnych (SVM), k-najbliższych sąsiadów (k-NN) oraz drzewa decyzyjne były szeroko stosowane do analizy danych obrazowych. Zastosowanie tych technik umożliwiało identyfikację elementów graficznych, takich jak przyciski czy pola tekstowe, w automatyzacji interfejsów użytkownika. [23]

Jednak tradycyjne podejścia mają swoje ograniczenia. Kluczowe wyzwania obejmują:

- **Skalowalność:** Metody te często nie radzą sobie z dużymi zbiorami danych i wymagają dużej ilości etykietowanych przykładów, co czyni proces kosztownym i czasochłonnym. [22]
- **Dynamiczność środowiska:** Zmiany w interfejsie, takie jak modyfikacje wizualne, mogą wymagać ponownej inżynierii cech, co ogranicza elastyczność tych metod.

Najnowsze badania wskazują, że zastosowanie uczenia maszynowego w widzeniu komputerowym obejmuje szeroką gamę aplikacji, takich jak detekcja obiektów, klasyfikacja oraz analiza wzorców. Algorytmy takie jak SVM, klasteryzacja k-średnich i sieci neuronowe znajdują zastosowanie w problemach związanych z przetwarzaniem obrazów oraz detekcją obiektów. [23, 22]

3.2.2. Tradycyjne techniki widzenia komputerowego

Tradycyjne techniki widzenia komputerowego opierają się na algorytmach i metodach analizy obrazów, które są zaprojektowane w sposób deterministyczny i nie wymagają uczenia się na danych. W przypadku automatyzacji interfejsów mobilnych takie podejścia, jak dopasowanie wzorców, są szeroko stosowane ze względu na ich prostotę i niskie wymagania obliczeniowe.

Dopasowanie wzorców to technika, która polega na porównywaniu fragmentu obrazu (wzorca) z większym obrazem w celu znalezienia najlepiej dopasowanego regionu. Wykorzystuje korelację między pikselami wzorca a obrazem wejściowym. W bibliotece OpenCV, wykorzystywaną przez system LogInt, technikę tę realizuje funkcja `cv2.matchTemplate`, która zwraca współczynniki dopasowania dla każdej możliwej pozycji wzorca w obrazie. Ta metoda sprawdza się w scenariuszach, gdzie elementy interfejsu użytkownika mają stały wygląd, na przykład ikony lub przyciski. [28]

Jednak tradycyjne metody, w tym dopasowanie wzorców, mają swoje ograniczenia:

- **Wrażliwość na zmiany:** Niewielkie modyfikacje w wyglądzie obiektu (np. zmiana koloru lub skalowanie) mogą znacząco obniżyć skuteczność dopasowania.
- **Brak generalizacji:** Algorytmy te są zoptymalizowane pod kątem spe-

cyficznych przypadków użycia i nie radzą sobie w środowiskach o dużej zmienności wizualnej.

W przypadku bardziej dynamicznych aplikacji mobilnych, gdzie interfejsy często zmieniają się w wyniku aktualizacji, takie podejście może wymagać częstej ponownej kalibracji wzorców. To ograniczenie staje się szczególnie istotne w kontekście zadań RPA, gdzie wymagana jest wysoka elastyczność i odporność na zmiany.

3.2.3. Uczenie głębokie

Uczenie głębokie stanowi nowoczesne podejście do analizy obrazów, które eliminuje wiele ograniczeń tradycyjnych technik. Dzięki zastosowaniu sieci neuronowych o wielu warstwach (ang. *deep neural networks*), uczenie głębokie umożliwia automatyczne wykrywanie i hierarchiczne przetwarzanie cech bez konieczności ręcznego projektowania ich przez człowieka.

W kontekście analizy obrazów, konwolucyjne sieci neuronowe (CNN) odgrywają kluczową rolę. Struktura CNN umożliwia efektywne wyodrębnianie wzorców z danych wizualnych, począwszy od prostych elementów, takich jak krawędzie czy kąty, po bardziej złożone struktury, takie jak twarze czy obiekty w tle. Proces ten polega na przechodzeniu obrazu przez kolejne warstwy konwolucyjne i operacje, takie jak filtrowanie, subsampling i normalizacja, co prowadzi do coraz bardziej abstrakcyjnych reprezentacji danych. [14]

W automatyzacji interfejsów mobilnych, uczenie głębokie umożliwia dynamiczne dostosowywanie się do zmian w wyglądzie elementów UI. Na przykład, sieci CNN mogą być wykorzystywane do wykrywania przycisków lub ikon niezależnie od ich koloru, rozmiaru czy orientacji. Dzięki temu systemy oparte na CNN są w stanie działać niezawodnie w złożonych środowiskach

aplikacji mobilnych. [28]

Jednak uczenie głębokie ma swoje wady:

- **Wysokie wymagania obliczeniowe:** Trenowanie sieci głębokich wymaga znacznych zasobów sprzętowych, takich jak GPU.
- **Zależność od danych treningowych:** Skuteczność modelu głębokiego uczenia w dużej mierze zależy od jakości i różnorodności danych treningowych.
- **Niekontrolowane zmiany:** Po dokonaniu zmian w zautomatyzowanej aplikacji, model może wybierać niepoprawny obiekt, który statystycznie najbardziej pasuje do swojej roli. Nie sprawdza się, gdy aplikacje nie są intuicyjne, lub tworzone zgodnie z poprawnymi technikami UX/UI.

Mimo tych wyzwań, uczenie głębokie rewolucjonizuje automatyzację procesów RPA, umożliwiając identyfikację i obsługę elementów interfejsów mobilnych w sposób elastyczny i niezawodny. W szczególności techniki te pozwalają na tworzenie systemów, które adaptują się do zmieniających się interfejsów bez konieczności ręcznej modyfikacji algorytmów.

3.2.4. Optyczne rozpoznawanie znaków

Optyczne rozpoznawanie znaków (ang. *Optical Character Recognition (OCR)*) to technologia, która umożliwia automatyczną konwersję tekstu z obrazów na formaty edytowalne i przeszukiwalne. Proces OCR składa się z kilku etapów, takich jak wstępne przetwarzanie obrazu, segmentacja tekstu, rozpoznawanie znaków oraz analiza kontekstowa. OCR znajduje szerokie zastosowanie w digitalizacji dokumentów, przetwarzaniu tekstu w obrazach w czasie rzeczywistym oraz automatyzacji procesów wymagających ekstrakcji danych.

[19, 36]

Współczesne metody OCR, takie jak Tesseract OCR, wykorzystują zaawansowane algorytmy analizy obrazu, w tym adaptacyjne progowanie i klasyfikację znaków. Tesseract, rozwijany jako projekt open-source, opiera się na modelach sieci neuronowych LSTM (Long Short-Term Memory), co pozwala na dokładne rozpoznawanie tekstu nawet w trudnych warunkach, takich jak różnorodność czcionek czy obecność szumów w obrazie. Sieci LSTM, uwzględniając kontekst znaków sąsiadujących, znacznie poprawiają dokładność rozpoznawania sekwencji tekstowych. [19]

3.3. Wybrane technologie na potrzeby systemu LogInt

3.3.1. Wymagania biznesowe systemu LogInt

W poprzednim podrozdziale omówiono kluczowe techniki automatyzacji aplikacji mobilnych oraz ich zalety i ograniczenia. W kontekście wymagań biznesowych klienta, firmy Boekestijn Transport sp. z o.o., system LogInt został zaprojektowany z uwzględnieniem specyficznych potrzeb i ograniczeń. Najważniejsze wymagania obejmują:

- **Elastyczność rozwiązania:** System powinien umożliwiać automatyzację wielu aplikacji, w których dane wejściowe są przekazywane w formacie JSON.
- **Niezależność od zespołu programistycznego:** Dodawanie nowych integracji nie powinno wymagać zaangażowania programistów. System

musi pozwalać na definiowanie logiki wypełniania aplikacji przez użytkowników nietechnicznych.

- **Specyfika automatyzowanych aplikacji:** Automatyzowane aplikacje często cechują się prostym interfejsem użytkownika, skierowanym do wąskiej grupy odbiorców. Brak wsparcia dla API oraz niska częstotliwość zmian graficznych to dodatkowe założenia. Jeśli zmiany są wprowadzane, często wpływają również na strukturę drzewa dokumentu.
- **Ograniczone zasoby obliczeniowe:** System musi działać na lokalnym serwerze klienta, co może oznaczać konieczność wykorzystania komputerów o ograniczonej mocy obliczeniowej.
- **Mała liczba aplikacji:** Brak dużych zbiorów danych wyklucza możliwość trenowania zaawansowanych modeli głębokiego uczenia na potrzeby specyficznego UI.

Powyższe wymagania stanowią podstawę doboru rozwiązań technologicznych w systemie LogInt. Kluczowym aspektem jest umożliwienie użytkownikowi końcowemu tworzenia i edytowania automatyzacji bez konieczności posiadania umiejętności programistycznych. Współczesne systemy RPA (Robotic Process Automation) skupiają się głównie na automatyzacji procesów w aplikacjach desktopowych lub z dostępem do API. Żadne z obecnie dostępnych rozwiązań nie spełnia jednak wymagań stawianych przez aplikacje mobilne w kontekście prostoty obsługi i elastyczności integracji.

3.3.2. Wyzwania w automatyzacji aplikacji mobilnych

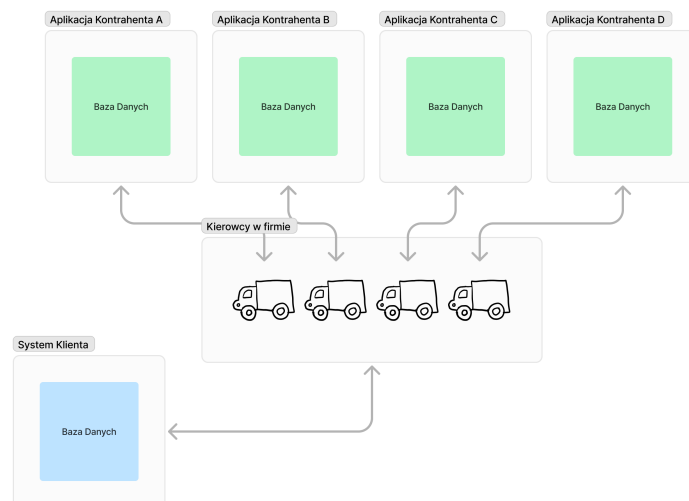
Pomimo popularności systemów RPA, ich zastosowanie w środowisku aplikacji mobilnych niesie ze sobą liczne ograniczenia. Nawet w systemach typu low-code czy no-code wymagane jest definiowanie szczegółowej logiki działania automatyzacji, obejmującej:

- Określenie sekwencji działań, takich jak wpisywanie danych czy wybieranie elementów interfejsu.
- Tworzenie warunków logicznych, które pozwalają dostosować zachowanie aplikacji do zmiennych scenariuszy użytkowych.

LogInt opiera swoje działanie na założeniu, że elementy interfejsu graficznego rzadko zmieniają się wizualnie, nawet jeśli struktura dokumentu może ulegać modyfikacjom. Użytkownik systemu może wskazać interesującego go elementy interfejsu (np. pola tekstowe, przyciski), a system na podstawie widzenia komputerowego i OCR podejmie odpowiednie akcje.

3.3.3. Automatyzacja procesów w środowisku LogInt

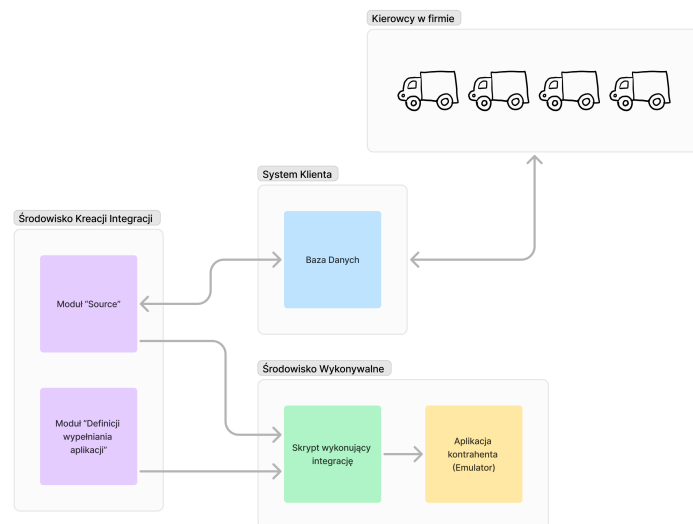
Przed wprowadzeniem systemu LogInt kierowcy byli zobligowani do ręcznego wprowadzania danych do aplikacji mobilnych kontrahentów. Każda aplikacja posiadała inny interfejs użytkownika oraz różne wymagania dotyczące danych, co znacząco utrudniało standaryzację procesów. Przykładowo, niektóre aplikacje rozdzielały typy przystanków (np. załadunek, rozładunek, serwis), podczas gdy inne traktowały każdy przystanek jako jednorodną aktywność.



Rys. 3.1. Diagram przedstawiający proces uzupełniania dwóch aplikacji tymi samymi danymi.

Po wdrożeniu systemu LogInt możliwe stało się częściowe przejęcie obowiązków kierowców przez system automatyzacji. Aplikacja LogInt pozwala użytkownikowi (administratorowi) na definiowanie logiki automatyzacji przy użyciu intuicyjnego interfejsu. Przykładowo:

- Użytkownik wskazuje elementy interfejsu, takie jak pola tekstowe, które mają zostać wypełnione danymi.
- Wybierane są zmienne (np. numer rejestracyjny pojazdu), które zostaną automatycznie wprowadzone do aplikacji.
- System interpretuje te instrukcje i generuje odpowiednie polecenia ADB, co pozwala na interakcję z aplikacją mobilną.



Rys. 3.2. Diagram przedstawiający proces uzupełniania wielu aplikacji kontrahenta z jednego źródła.

Warto podkreślić, że dynamiczność elementów interfejsu, takich jak listy rozwijane, wymaga zastosowania technologii OCR. Przykładowo, w aplikacji kontrahenta może być konieczne wybranie pozycji odpowiadającej numerowi rejestracyjnemu pojazdu kierowcy. LogInt jest w stanie wykrywać i rozpoznawać te elementy interfejsu dzięki integracji technologii widzenia komputerowego oraz OCR, co umożliwia elastyczne dostosowanie się do wymagań aplikacji.

3.3.4. Zasada działania systemu LogInt

System LogInt łączy funkcjonalności znane z popularnych rozwiązań RPA, takich jak Power Automate czy Zapier, z możliwościami analizy obrazu. Użytkownik definiuje automatyzację poprzez wskazanie interesujących elementów interfejsu na zrzutach ekranu oraz określenie logicznych instrukcji, takich jak

”kliknij ten element” lub ”wypełnij ten formularz danymi z kolumny C w bazie danych”.

Zastosowane podejście umożliwia intuicyjne tworzenie procesów automatyzacji, które nie wymagają od użytkownika znajomości programowania. W następnych rozdziałach szczegółowo omówiono sposób generowania poleceń ADB oraz komunikacji między aplikacjami mobilnymi a systemem LogInt.

3.4. Praktyczne zastosowanie technik CV i OCR w aplikacji LogInt

3.4.1. Dopasowywanie wzorców w OpenCV

W podrozdziale 3.2.2 omówiono techniki identyfikacji elementów interfejsu użytkownika w aplikacjach mobilnych, w tym tradycyjne podejścia wizji komputerowej. Jedną z kluczowych metod stosowanych w systemie LogInt jest dopasowywanie wzorców (*template matching*), które umożliwia precyzyjne lokalizowanie elementów interfejsu graficznego.

Template matching to technika widzenia komputerowego, której celem jest identyfikacja regionów obrazu odpowiadających określonemu wzorcowi. Proces polega na przesuwaniu wzorca po obrazie źródłowym i obliczaniu miary podobieństwa, takiej jak korelacja krzyżowa lub znormalizowana korelacja krzyżowa (*normalized cross-correlation*). Regiony osiągające najwyższe wartości podobieństwa są uznawane za dopasowania.

W systemie LogInt zastosowano funkcję odpowiedzialną za obliczanie współrzędnych środka dopasowanego wzorca. Kod funkcji przedstawiono poniżej:

Listing 3.1. Funkcja obliczająca współrzędne środka danego wzorca

```
1 def find_coordinates_of_screenshot(target_img_path,
   source_img):
2     target_img = cv2.imread(target_img_path, cv2.
       IMREAD_GRAYSCALE)
3     source_img_gr = cv2.imread(source_img, cv2.
       IMREAD_GRAYSCALE)
4     result = cv2.matchTemplate(source_img_gr,
       target_img, cv2.TM_CCORR_NORMED)
5
6     min_val, max_val, min_loc, max_loc = cv2.minMaxLoc
       (result)
7     if max_val > 0.9:
8         h, w = target_img.shape[:2]
9         center_x = max_loc[0] + w // 2
10        center_y = max_loc[1] + h // 2
11        return (center_x, center_y)
12    else:
13        return None
```

Omówienie działania funkcji

Pierwsze dwie linie funkcji:

Listing 3.2. Wczytanie obrazu

```
1 target_img = cv2.imread(target_img_path, cv2.
   IMREAD_GRAYSCALE)
2 source_img_gr = cv2.imread(source_img, cv2.
   IMREAD_GRAYSCALE)
```

odpowiadają za wczytanie obrazu i jego konwersję do skali szarości. Taki

proces redukuje redundancję danych, upraszcza analizę obrazu i zmniejsza złożoność obliczeniową. W kontekście interfejsów użytkownika mogłoby się wydawać, że usunięcie informacji o kolorach może negatywnie wpłynąć na wyniki analizy, jednak testy przeprowadzone w systemie LogInt wykazały, że skuteczność dopasowania pozostaje niezmienną, a dodatkowo uzyskuje się wyżej wymienione korzyści.

Kluczowa część funkcji to:

Listing 3.3. Szukanie wzorca

```
1 result = cv2.matchTemplate(source_img_gr, target_img,
    cv2.TM_CCORR_NORMED)
```

Funkcja `cv2.matchTemplate` przesuwając wzorzec po obrazie źródłowym i oblicza znormalizowaną korelację krzyżową (`TM_CCORR_NORMED`) jako miarę podobieństwa. Dzięki zastosowaniu tej metody wynik jest częściowo odporny na zmienności oświetlenia, co czyni ją odpowiednią w środowiskach o stabilnych warunkach wizualnych.

Następna linia:

Listing 3.4. Obliczanie współrzędnych

```
1 min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(
    result)
```

znajduje współrzędne najlepszego dopasowania wzorca w obrazie. Wartość `max_val` wskazuje poziom dopasowania, a współrzędne `max_loc` określają pozycję wzorca. Aby uniknąć fałszywych dopasowań, ustawiono próg podobieństwa na poziomie 0.9. Dzięki temu drobne zmiany w wyglądzie interfejsu (np. wynikające z aktualizacji aplikacji) nie wpływają znacząco na działanie systemu.

Końcowa część funkcji:

Listing 3.5. Próg podobieństwa

```
1 if max_val > 0.9:
2     h, w = target_img.shape[:2]
3     center_x = max_loc[0] + w // 2
4     center_y = max_loc[1] + h // 2
5     return (center_x, center_y)
6 else:
7     return None
```

sprawdza, czy dopasowanie spełnia próg podobieństwa (`max_val > 0.9`). Jeśli tak, funkcja zwraca współrzędne środka wzorca w obrazie. W przeciwnym razie zwracana jest wartość `None`, co sygnalizuje brak dopasowania.

Dopasowywanie wzorców jest kluczowym elementem w automatyzacji działań na aplikacjach mobilnych w systemie LogInt. Przykładowo, funkcja może zostać użyta do zlokalizowania przycisku „Zaloguj” w aplikacji kontrahenta. Jeśli wzorzec nie zostanie dopasowany (np. z powodu aktualizacji aplikacji), system pomija daną instrukcję, co minimalizuje ryzyko błędów w działaniu.

Takie podejście pozwala na intuicyjne definiowanie działań przez użytkowników nietechnicznych, jednocześnie zapewniając elastyczność w dostosowywaniu systemu do zmian w interfejsie aplikacji, bez potrzeby dostępu do API, kodu źródłowego czy stosowania inżynierii wstecznej.

Co więcej, w testach przeprowadzonych na bardzo prostej aplikacji imitującej rozwiązania kontrahentów naszego klienta udało się osiągnąć 100% skuteczności. Jednak w warunkach produkcyjnych, przy użyciu tych samych technik widzenia komputerowego, efektywność całego procesu wyniosła 86%.

Różnica ta wynika prawdopodobnie z błędów danych, spowolnień w działaniu aplikacji lub innych nieprzewidzianych usterek. Warto jednak zauważyć, że sama skuteczność rozpoznawania wzorców była nieco wyższa, niż wskazuje finalny wynik całego procesu.

3.4.2. Znajdowanie tekstu za pomocą PyTesseract

Podobnie jak technika dopasowywania wzorców, optyczne rozpoznawanie znaków (OCR, *Optical Character Recognition*) jest kluczowym elementem realizacji wymagań biznesowych systemu LogInt. OCR jest szczególnie przydatne w przypadkach dynamicznej zawartości aplikacji, gdzie wartości tekstowe zmieniają się w zależności od kontekstu. Przykładem jest wybór przystanku na trasie pojazdu, gdzie decyzja kierowcy zależy od odczytania adresu widocznego na ekranie.

W systemie LogInt zastosowano technologię OCR przy wykorzystaniu biblioteki PyTesseract, bazującej na otwartoźródłowym silniku Tesseract OCR. Kod odpowiedzialny za identyfikację tekstu i jego lokalizację w obrazie przedstawiono poniżej:

Listing 3.6. Funkcja identyfikująca tekst i jego lokalizację na obrazie

```
1 def find_coordinates_of_text(text, source_img,
2   similarity_threshold=70):
3     gray_img = cv2.imread(source_img, cv2.
4       IMREAD_GRAYSCALE)
5     blurred_img = cv2.GaussianBlur(gray_img, (5, 5),
6       0)
7     processed_img = cv2.adaptiveThreshold(
8       blurred_img, 255, cv2.
9       ADAPTIVE_THRESH_GAUSSIAN_C, cv2.
```

```

        THRESH_BINARY, 11, 2
6    )
7    custom_config = r'--oem 3 --psm 11 --lang eng'
8    data = pytesseract.image_to_data(processed_img,
        config=custom_config, output_type=pytesseract.
        Output.DICT)
9
10   best_match = None
11   best_similarity = 0
12   for i, t in enumerate(data['text']):
13       if t.strip():
14           similarity = fuzz.ratio(t.lower(), text.
               lower())
15           if similarity > best_similarity:
16               best_similarity = similarity
17               best_match = (data['left'][i], data['
                   top'][i], data['width'][i], data['
                   height'][i], t)
18
19   if best_match:
20       x, y, w, h, matched_text = best_match
21       center_x = x + w // 2
22       center_y = y + h // 2
23       tap((center_x, center_y))
24       return True
25   return False

```

Działanie funkcji można podzielić na następujące etapy:

1. **Przetwarzanie i przygotowanie obrazu:**

- Za pomocą funkcji `cv2.imread` obraz jest konwertowany do skali szarości (`GRAYSCALE`). Redukuje to złożoność danych wejściowych, upraszczając analizę tekstu.
- Funkcja `cv2.GaussianBlur` wygładza obraz, usuwając szумы i gradienty, co zwiększa skuteczność rozpoznawania.
- Funkcja `cv2.adaptiveThreshold` wzmacnia kontrast między tekstem a tłem, co jest kluczowe w sytuacjach, gdzie tło jest nierówne.

2. Konfiguracja Tesseract OCR:

- Parametr `--oem 3` umożliwia wykorzystanie hybrydowego trybu działania, który łączy klasyczne algorytmy OCR z modelami LSTM (*Long Short-Term Memory*), co poprawia skuteczność w rozpoznawaniu tekstu.
- Parametr `--psm 11` pozwala na analizę pojedynczych elementów tekstu. Jest to szczególnie przydatne w aplikacjach mobilnych, gdzie tekst jest rozproszony po całym interfejsie.

3. Poszukiwanie wzorca tekstowego:

- Funkcja `pytesseract.image_to_data` zwraca rozpoznany tekst wraz z jego metadanymi (`left`, `top`, `width`, `height`).
- Aby uwzględnić potencjalne literówki lub błędy wynikające z ręcznego wprowadzania danych, zastosowano bibliotekę `rapidfuzz`, która porównuje podobieństwo tekstów za pomocą metody `fuzz.ratio`. Zidentyfikowany tekst o najwyższym stopniu dopasowania, przekraczającym zadany próg (`similarity_threshold`), jest wybierany jako wynik dopasowania.

4. Symulacja akcji:

- Po obliczeniu współrzędnych środka tekstu funkcja symuluje kliknięcie za pomocą `tap`. Polecenie ADB (Android Debug Bridge) jest wysyłane do emulatora aplikacji, co pozwala na interakcję z interfejsem użytkownika.

Zastosowanie w systemie LogInt Technika OCR w systemie LogInt umożliwia automatyzację zadań opartych na dynamicznych danych tekstowych, takich jak identyfikacja adresów na przystankach trasy. Dzięki PyTesseract użytkownik może definiować logikę działania systemu w sposób intuicyjny, wskazując teksty do rozpoznania oraz odpowiednie akcje. Funkcja działa niezależnie od API aplikacji, umożliwiając interakcję z interfejsem użytkownika w sposób zbliżony do ludzkiego.

Rozwiązanie to zwiększa elastyczność i niezawodność automatyzacji, jednocześnie upraszczając proces definiowania kolejnych kroków automatyzacji przez administratora systemu.

3.5. Podsumowanie

W przedstawionym rozdziale omówiono kluczowe metody wykorzystywane w systemie LogInt do automatyzacji procesów w aplikacjach mobilnych. Opisano zarówno wymagania biznesowe klienta, które determinowały wybór technik, jak i konkretne rozwiązania technologiczne, takie jak dopasowywanie wzorców (*template matching*) oraz optyczne rozpoznawanie znaków (OCR).

Znormalizowana korelacja krzyżowa jest skuteczną metodą dopasowania wzorców w warunkach stabilnych wizualnie interfejsów aplikacji mobilnych, gdzie zmiany kontrastu i jasności są jednorodne. Dzięki przewidywalności

układu graficznego oraz ograniczeniu deformacji i lokalnych zakłóceń, metoda ta spełnia wymagania systemu LogInt. Jednak w przypadkach bardziej dynamicznych interfejsów, w aplikacjach które nie są typowymi aplikacjami kontrahentów naszego klienta, wymagających adaptacji do większych zmian wizualnych, jej skuteczność może być ograniczona.

Technologia OCR, zaimplementowana z użyciem biblioteki PyTesseract, umożliwia rozpoznawanie dynamicznych treści tekstowych, takich jak adresy czy dane wprowadzane przez użytkowników. Dzięki integracji z funkcją fuzzy matching, system LogInt jest w stanie skutecznie identyfikować elementy tekstowe mimo drobnych różnic wynikających z literówek lub błędów OCR.

Przedstawione metody podkreślają elastyczność systemu LogInt w dostosowywaniu się do zmiennych warunków i dynamicznych środowisk aplikacji mobilnych. System, choć prosty w obsłudze, oferuje zaawansowane możliwości automatyzacji, które nie wymagają dostępu do API aplikacji ani zaawansowanej wiedzy technicznej. Tym samym LogInt stanowi uniwersalne narzędzie spełniające specyficzne wymagania klienta, jednocześnie upraszczając proces definiowania automatyzacji dla nietechnicznych użytkowników.

Podsumowując, opisane w rozdziale techniki dopasowują się do założeń współczesnych systemów RPA, jednak w unikalny sposób przenoszą ich funkcjonalności na środowiska aplikacji mobilnych, co stanowi wyróżnik rozwiązania LogInt.

Rozdział 4

Automatyczna interakcja z emulatorem urządzenia mobilnego za pomocą Android Debug Bridge

Autor: Miłosz Rolewski

4.1. Wprowadzenie

System LogInt został stworzony głównie z myślą o automatyzacji procesów biznesowych związanych z zarządzaniem danymi logistycznymi firmy Boekstijn Transport. Wymagało to od zespołu deweloperskiego opracowania kreatora integracji oraz uniwersalnego narzędzia do realizacji wcześniej zdefiniowanych automatyzacji. Celem tego rozdziału jest wyjaśnienie roli środowiska wykonywalnego w projekcie, omówienie specyfiki jego działania oraz przedstawienie, jak złożonym procesem jest symulacja akcji użytkownika poprzez

automatyczne interakcje z emulatorem urządzenia mobilnego.

4.1.1. Automatyzacja procesów biznesowych dla procesów logistycznych

Procesem biznesowym określamy strukturę działań, które prowadzą do realizacji określonego zadania [9]. Dobrze zdefiniowane procesy biznesowe, przy wsparciu systemów informatycznych, mogą znacznie usprawnić komunikację w procesach logistycznych. Jest ona kluczowa dla zachowania wiarygodności wobec klienta. Automatyzacja tych procesów odgrywa istotną rolę w dobie cyfryzacji przemysłu. Umożliwia usprawnienie przepływu danych, eliminację błędów spowodowanych czynnikiem ludzkim oraz oszczędność czasu, który człowiek musiałby poświęcić na wykonywanie drobnych zadań. Właśnie w duchu tych trzech idei firma Boekestijn Transportzleciła zespołowi deweloperskiemu stworzenie systemu LogInt.

4.1.2. Zrobotyzowana automatyzacja procesów

Postępującym zjawiskiem, które można zaobserwować w przestrzeni biznesowej, jest zrobotyzowana automatyzacja procesów (ang. Robotic Process Automation, RPA). Rozwiązania te naśladują pracę i działania człowieka za pomocą specjalistycznego oprogramowania [33]. System LogInt jest przestrzenią do tworzenia oraz zarządzania robotami softwarowymi. Są to wirtualne narzędzia mające na celu wyręczenie człowieka w obsłudze oprogramowania przy czynnościach powtarzalnych. Środowisko wykonywalne, będące drugą składową systemu LogInt, jest przestrzenią, w której roboty te mogą pracować, wykonując wcześniej zdefiniowane akcje.

4.1.3. Wpływ RPA na rynek pracy

RPA jest stosunkowo nowym i innowacyjnym rozwiązaniem na rynku, co sprawia, że jego koszty wdrożenia są wysokie.

Do tej pory RPA było rzadko stosowane w Polsce. Jednakże można zaobserwować jego popularyzację, wynikającą z braku specjalistów na rynku. Automatyzacja procesów umożliwia przedsiębiorcom znaczne oszczędności oraz lepsze zarządzanie zasobami ludzkimi, ponieważ część zadań wykonywana jest automatycznie. Roboty softwarowe eliminują również błędy popełniane przez ludzi i opóźnienia w realizacji zadań, które mogą skutkować karami finansowymi nakładanymi na przedsiębiorstwo.

4.2. Środowisko emulatora urządzenia mobilnego

Jak opisano powyżej, RPA opiera się na naśladowaniu działań człowieka. W systemie LogInt środowiskiem, w którym wykonywane są te akcje, jest emulator Androida. Wybrano go ze względu na otwartość tego systemu oraz szeroką dostępność aplikacji używanych w logistyce transportowej. Emulator Androida działa w oparciu o wirtualne urządzenie Android (Android Virtual Device, AVD).

4.2.1. Android Studio Emulator

Najpopularniejszym wyborem wśród deweloperów aplikacji mobilnych na system Android jest emulator dostarczany wraz z oprogramowaniem Android Studio. Pozwala on uruchamiać aplikacje na urządzeniach o różnych profilach

sprzętowych i poziomach interfejsu API Android [18]. Rozwiązanie to niesie za sobą następujące zalety:

- **Elastyczność** - Poza dostarczaniem interfejsów o różnych poziomach API, można zasymulować, jak aplikacja będzie funkcjonowała na urządzeniach innego typu, takich jak tablet, smartwatch czy telewizor.
- **Wysoka wiarygodność** - Emulator ten dostarcza szereg funkcjonalności mających na celu jak najbardziej wiarygodne odwzorowanie urządzenia mobilnego. Mowa tu o symulacji przychodzących połączeń i wiadomości tekstowych, dostosowywaniu różnych przepustowości sieci oraz wpływaniu na fizyczne sensory urządzenia, takie jak czujnik temperatury, wykrywacz pola magnetycznego, barometr czy higrometr.
- **Szybkość** - Uruchamianie i testowanie aplikacji na emulatorze jest szybsze i wygodniejsze niż na fizycznym urządzeniu. Łatwość migracji danych i plików wejściowych oraz możliwość symulacji różnych czynników zewnętrznych za pomocą sensorów odgrywają tu znaczącą rolę.

Rozwiązanie to nie jest jednak perfekcyjne. Emulator oferujący tak wiele funkcjonalności znacznie obciąża system. Aplikacje przesyłające dane logistyczne najczęściej potrzebują jedynie podstawowych komponentów, dlatego system LogInt korzystał z tego rozwiązania jedynie w początkowych fazach rozwoju projektu. Dodatkowo emulator został wyeliminowany z użytku przez fakt, że Google skutecznie blokuje możliwość symulacji wirtualnej kamery. Powodem tej decyzji były liczne incydenty związane z podszywaniem się pod inne osoby i oszukiwaniem systemów weryfikacji tożsamości.

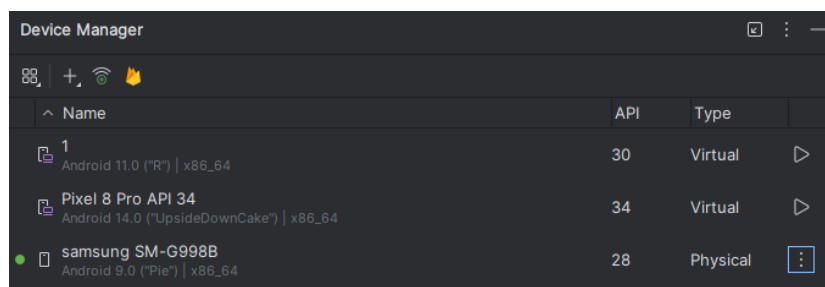
4.2.2. Docker-Android

Emulator będący wirtualnym urządzeniem nie musi koniecznie działać na fizycznym komputerze gospodarza (ang. host). Na rynku dostępne są rozwiązania umożliwiające uruchomienie środowiska Android w formie kontenerów platformy Docker. Rozwiązanie to nie zostało wdrożone w ramach projektu ze względu na wysokie zużycie zasobów przez skonteneryzowane emulatory oraz ograniczone możliwości konfiguracji urządzenia wirtualnego [25]. Projekt Docker-Android rozwijany na platformie GitHub oferuje siedem wersji Androida oraz kilka predefiniowanych profili sprzętowych urządzeń wirtualnych. Niestety, nie zapewnia elastycznego dostosowania warstwy wizualnej emulatora [10], co jest kluczowe dla prawidłowego działania widzenia komputerowego – krytycznego elementu integracji systemu LogInt.

4.2.3. BlueStacks

Rozwiązaniem, które okazało się optymalne dla automatycznego wykonywania zadeklarowanych procedur, był BlueStacks. Podobnie jak Android Studio Emulator, BlueStacks korzysta z wirtualizacji. System wirtualny (gość) ma wrażenie, że funkcjonuje na fizycznym urządzeniu [1]. Istotnym czynnikiem w realizacji celów projektowych było to, że BlueStacks posiada własną, zmodyfikowaną nakładkę systemu Android zoptymalizowaną pod kątem gier mobilnych. Dzięki temu zwiększono wydajność systemu poprzez sprawniejszą komunikację z emulatorem.

Kolejnym czynnikiem, który wyróżnia BlueStacks spośród pozostałych emulatorów, jest fakt, że na komputerze gospodarza widoczny jest jako zewnętrznie podłączone urządzenie fizyczne, a nie jako urządzenie wirtualne (patrz: urządzenie Samsung na rys. 4.1).



Rys. 4.1. Samsung SM-G998B – emulator BlueStacks rozpoznawany jako urządzenie fizyczne w systemie.

Pozwala to na korzystanie z takich funkcjonalności, jak symulacja pola widzenia kamery za pomocą tworzonych na bieżąco wirtualnych kamer (więcej w 4.6.1).

BlueStacks nie oferuje takiej swobody personalizacji AVD jak Android Studio, jednak jego funkcjonalności są wystarczające do stworzenia w pełni wydajnego środowiska wykonywalnego. Zapewnia możliwość dostosowania liczby rdzeni procesora, wielkości pamięci RAM, częstotliwości odświeżania ekranu, rozdzielczości i zagęszczenia pikseli, a co najważniejsze – pozwala ustawić skalę wyświetlacza na 1:1 oraz wybrać kamerę wirtualną jako aparat sprzętowy.

4.3. Android Virtual Device jako komponent emulatora

Terminem Android Virtual Device (AVD) nazywamy konfigurację opisującą urządzenie, które będzie symulowane przez emulator. Składa się na nią specyfikacja urządzenia (wielkość wyświetlacza, rozdzielczość, model wirtualnego urządzenia), parametry sprzętowe (ilość pamięci RAM, wielkość pamięci wewnętrznej), obraz systemu operacyjnego (API), wykaz obsługiwanych przez

urządzenie funkcji (obsługa i symulacja kamer, odczytywane czujniki, obsługa mikrofonu i głośników)[17]. Wpływając na AVD, można symulować warunki zewnętrzne, takie jak poziom naładowania baterii, temperatura urządzenia czy położenie geograficzne. W projekcie ustawieniem, które odgrywa najważniejszą rolę, jest odblokowanie komunikacji z urządzeniem za pomocą Android Debug Bridge (ADB) (patrz 4.4) w celu udostępnienia łączności pomiędzy skryptem Python interpretującym przychodzące do wykonania integracje, a emulatorem, na którym mają się one wykonać. Użytkownik może zadeklarować więcej niż jedno urządzenie wirtualne i uruchamiać je w zależności od ich przeznaczenia. W przypadku tworzenia i zarządzania takimi instancjami należy korzystać z Device Managera, będącego kolejną składową emulatora.

4.4. Android Debug Bridge - możliwości i zastosowania

4.4.1. Powłoka Android Debug Bridge

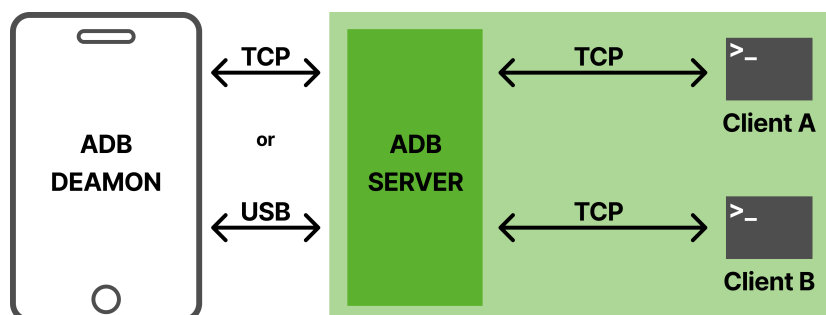
Android Debug Bridge jest narzędziem wiersza poleceń umożliwiającym komunikację pomiędzy systemem hosta i systemem gościa, którym może być wirtualne urządzenie z systemem Android lub połączone urządzenie fizyczne. Głównymi funkcjonalnościami, jakie dostarcza ta powłoka, jest wykonywanie akcji wymagających interakcji z interfejsem za pomocą komend wiersza poleceń [15]. Powłoka ADB została stworzona z myślą o debugowaniu i testowaniu aplikacji mobilnych, przez co jest używana w rozwiązaniach przeznaczonych do automatycznego testowania oprogramowania, takich jak Appium czy Selenium. W systemie LogInt pełni ona rolę mostu komunikacyjnego pomiędzy skryptem Python, będącym interpreterem przychodzących danych i instruk-

cji, a emulatorem będącym środowiskiem wykonania tych integracji.

4.4.2. Sposób działania

Sposób działania ADB jest oparty o komunikację protokołem sterowania transmisją (ang. Transmission Control Protocol, TCP), który działa w trybie klient-serwer [34]. Program ten składa się z trzech komponentów (spójrz Rys. 4.2):

- Klient - wysyła polecenia. Może być wywoływany z poziomu wiersza poleceń lub jako podproces skryptu Python.
- Serwer - łączy komunikację klienta z demonem. Działa w tle i jest uruchamiany wraz z klientem. Po uruchomieniu łączy się z lokalnym portem TCP 5037 i nasłuchuje poleceń ze strony klienta.
- Demon (adbd) - jest odpowiedzialny za uruchamianie poleceń na połączonym urządzeniu. Podobnie jak serwer, działa w tle.



Rys. 4.2. Architektura serwera ADB - rysunek własny na podstawie <https://emteria.com/learn/android-debug-bridge>

Po uruchomieniu serwera tworzy on połączenia z emulatorami, skanując porty o nieparzystych numerach z zakresu 5555 do 5585. Każda para portów

służy do utworzenia połączenia z jednym emulatorem. Parzysty numer portu odpowiada za konsolę, a nieparzysty jest odpowiedzialny za ADB [15].

4.4.3. Połączenie urządzenia z ADB

Włączenie mostu debugowania na urządzeniu

W celu włączenia mostu debugowania na urządzeniu należy odblokować ustawienia programisty [16], a następnie włączyć debugowanie USB, niezależnie od tego, czy korzystamy z emulatora, czy urządzenia fizycznego podłączonego do portu USB.

W instrukcji korzystania z systemu LogInt dla wygody osoby zarządzającej systemem opisano, jak dokonać tej czynności na emulatorze.

Łączenie urządzeń za pomocą Wi-Fi

Do ADB można połączyć wiele fizycznych urządzeń bez konieczności łączenia ich fizycznym kablem USB ze stacją roboczą. Od Androida 11 możliwe jest debugowanie przez Wi-Fi. Należy wtedy na urządzeniu włączyć debugowanie przez Wi-Fi, a następnie dodać takie urządzenie do Device Managera stacji roboczej za pomocą interfejsu graficznego lub wiersza poleceń.

Rozwiązanie to daje znaczące możliwości w kwestii rozwoju systemu LogInt. Gdy utworzona zostanie znacząca liczba integracji, zadania zaczną się kolejkować. Podłączenie wielu emulatorów i rozdzielenie zadań pomiędzy nie sprawi, że system znacznie zwiększy swoją przepustowość w kwestii przepływu danych i wydajności.

4.4.4. Wydawanie poleceń powłoki

Cały system automatycznego wykonywania integracji w środowisku wykonywalnym systemu LogInt oparty jest na wydawaniu odpowiednich, predefiniowanych poleceń powłoki ADB. Niektóre z tych komend zostały połączone ze sobą w celu realizacji funkcjonalności o wyższej złożoności (patrz 4.5.3). Bazowe polecenia odgrywające najważniejszą rolę w systemie to:

Instalowanie aplikacji

```
1 adb -s [target_ip] install [apk_path]
```

Polecenie to pozwala zainstalować aplikację z pliku o rozszerzeniu APK. Flagą `-s` określa numer seryjny, czyli urządzenie, na jakim ma zostać wykonane polecenie. W tym przypadku parametr ten opisuje `target_IP`. Jako `apk_path` do funkcji należy podać ścieżkę do rzekomego pliku źródłowego.

Możliwe jest również instalowanie wielu plików APK przy użyciu polecenia `install_multiple` [12]. W systemie LogInt nie skorzystano jednak z tej funkcjonalności z powodu dynamicznej zmiany ilości integracji oraz potencjału na rozproszenie systemu na wiele emulatorów.

Uruchamianie aplikacji

```
1 adb -s [target_ip] shell monkey -p [package_name] 1
```

W tym przypadku argument `target_IP` pełni analogiczną funkcję. `shell` odpowiada za wykonanie polecenia `monkey` w wewnętrznej powłoce emulatora. Jako argument `package_name` należy podać wewnętrzną nazwę pakietu, który jest już zainstalowany na urządzeniu. Wartość tej zmiennej można uzyskać bezpośrednio z pliku instalacyjnego APK.

Odinstalowywanie aplikacji

```
1 adb -s [target_ip] -e uninstall [package_name]
```

Polecenie to używane jest w przypadku, gdy w systemie zostaną usunięte wszystkie integracje pracujące na danej aplikacji. Odinstalowanie aplikacji z urządzenia wiąże się z usunięciem jej pozwoleń systemowych, takich jak dostęp do systemu plików, aparatu czy danych geolokalizacji. Wiąże się to z koniecznością ponownej wstępnej konfiguracji aplikacji po jej zainstalowaniu w celu prawidłowego działania systemu automatycznych integracji.

Zamknięcie aplikacji

```
1 adb -s [target_ip] shell am force-stop package_name
```

Zamknięcie aplikacji pełni bardzo ważną rolę w środowisku emulatora. Działanie to pozwala przede wszystkim na powrót do ekranu startowego z możliwością uruchomienia innych aplikacji. Co więcej, służy do efektywnego zarządzania pamięcią RAM, gdyż aplikacje działające w tle wciąż mogą zużywać zasoby i obciążać urządzenie wirtualne.

Wykonanie kliknięcia

```
1 adb -s [target_ip] shell input tap [x] [y]
```

Jest to podstawowa funkcjonalność użytkowa w przypadku interakcji z urządzeniem mobilnym lub jego emulatorem. Przez kliknięcia wybierane są poszczególne przyciski czy opcje z list w aplikacjach kontrahenta. W celu określenia, w które miejsce ekranu należy kliknąć, do komendy podawane są dwa współrzędne [x] i [y] określające pozycję na ekranie.

Wprowadzenie tekstu

```
1 adb -s [target_ip] shell input text [text]
```

Niektóre aplikacje, poza wybraniem przycisku z interfejsu graficznego, wymagają również wprowadzenia tekstu w pole tekstowe. W tym celu należy połączyć komendę `tap` z `input text`, aby wprowadzić tekst w odpowiednie miejsce. Argument `[text]` pozwala podać ciąg znaków, który ma zostać wprowadzony, co umożliwia wygodniejsze wprowadzanie tekstu niż programowanie sekwencji poleceń `tap` jako ciągu kliknięć na wirtualnej klawiaturze emulatora.

Wykonanie zrzutu ekranu

```
1 adb -s [target_ip] shell screencap -p [
    path_on_emulator]
```

System LogInt, korzystając z widzenia komputerowego, potrafi określić położenie elementów, w które należy kliknąć. W celu analizy komponentów na wyświetlaczu emulatora system pracuje na zrzutach ekranu. Metoda ta okazała się być bardzo wydajna, ponieważ uzyskanie zrzutów w oryginalnej rozdzielczości urządzenia oraz ustalenie skali na 1:1 pozwala na niemal bezbłędne odnalezienie współrzędnych szukanego komponentu. Argument `[path_on_emulator]` pozwala zdefiniować katalog, w którym ma zostać zapisany plik oraz jego nazwę.

Skopiowanie pliku na urządzenie gospodarza

```
1 adb -s [target_ip] pull [guest_path] [host_path]
```

Kolejną ważną funkcjonalnością komunikacyjną pomiędzy gospodarzem a gościem jest transfer plików. Skopiowanie pliku zawierającego zrzut ekranu z urządzenia gościa na urządzenie gospodarza jest używane do poddania tego zrzutu analizie w celu odnalezienia współrzędnych szukanego elementu na ekranie emulatora.

Skopiowanie pliku na urządzenie gościa

```
1 adb -s [target_ip] push [host_path] [guest_path]
```

Analogicznie można przekazać plik z systemu gospodarza na system gościa. Funkcjonalność ta jest używana w przypadku, gdy podczas wykonywania integracji konieczne będzie załączenie zdjęcia z galerii urządzenia.

4.5. Projekt systemu automatycznych interakcji

4.5.1. Wstęp

Jednym z największych wyzwań projektowych dla zespołu deweloperskiego było utworzenie intuicyjnego narzędzia pozwalającego tworzyć integracje dla osób nieposiadających zaawansowanych umiejętności programistycznych. Narzędzie to miało być możliwie proste w obsłudze, ale dostarczać takie funkcjonalności, by możliwe było, definiując kroki integracji, sporządzenie listy kroków opisującej, co należy wykonać kolejno w aplikacji kontrahenta.

4.5.2. Proces definiowania sekwencji interakcji w systemie LogInt

Rozwiązanie, jakie zdecydowano się wdrożyć, to kreator sekwencji operacji. Pozwala on użytkownikowi zadeklarować, jakie czynności kolejno wykonywałby człowiek w celu przejścia przez zewnętrzną aplikację kontrahenta. Schemat tworzony jest z podstawowych operacji, które można wykonać jako użytkownik aplikacji, oraz instrukcji warunkowych, lecz odpowiednio je łącząc, można uzyskać zaawansowane funkcjonalności. W kreatorze użytkownik ma dostęp do trzech podstawowych akcji:

- TAP - Akcja ta pozwala odnaleźć na ekranie emulatora element interfejsu, który został zadeklarowany przy tworzeniu tego kroku, a następnie wykonać kliknięcie w jego środek (patrz Rys.4.3).
- TYPE IN - Operacja ta pełni funkcję uzupełniania pól tekstowych. Podobnie jak akcja TAP, odnajduje przypisany jej element na ekranie, a następnie wprowadza do niego tekst znajdujący się w wybranej kolumnie w raporcie wygenerowanym przez system (patrz Rys.4.4).
- FIND TEXT - Niekiedy użytkownik zamiast znaleźć element taki jak guzik czy checkbox musi wybrać z listy element opisany jakimś ciągiem znaków. W tym celu stworzono ten typ operacji, by przy użyciu widzenia komputerowego i rozpoznawania tekstu możliwe było jego odnalezienie oraz kliknięcie (patrz Rys.4.5).

Podczas tworzenia całego workflow (patrz Rys.4.6) użytkownik może zastosować bloki warunkowe w celu rozgraniczenia różnych wariantów użytkowania aplikacji. Dodanie operacji warunkowej zobowiązuje użytkownika do wybrania jej następników, tj. która operacja ma się wykonać, gdy warunek

Add Operation

×

Operation Order Number

6

Operation Name

Add Step or Condition

Add Step

Action

TAP

Screenshot

Choose File

confirm_button_pic.png

Next Operation (Optional)

None

Cancel

Save

Rys. 4.3. Deklaracja operacji TAP w kreatorze integracji.

jest spełniony, a która w przeciwnym wypadku. Rozgraniczenia te pozwalają budować schematy przejścia przez aplikację w zależności od danych wejściowych dostarczanych przez system klienta.

Action

TYPE IN

Screenshot

Choose File TYPE_IN_field.png

Column

assignment_id

Next Operation (Optional)

None

Cancel Save

Rys. 4.4. Deklaracja operacji TYPE IN w kreatorze integracji.

Action

FIND TEXT

Column

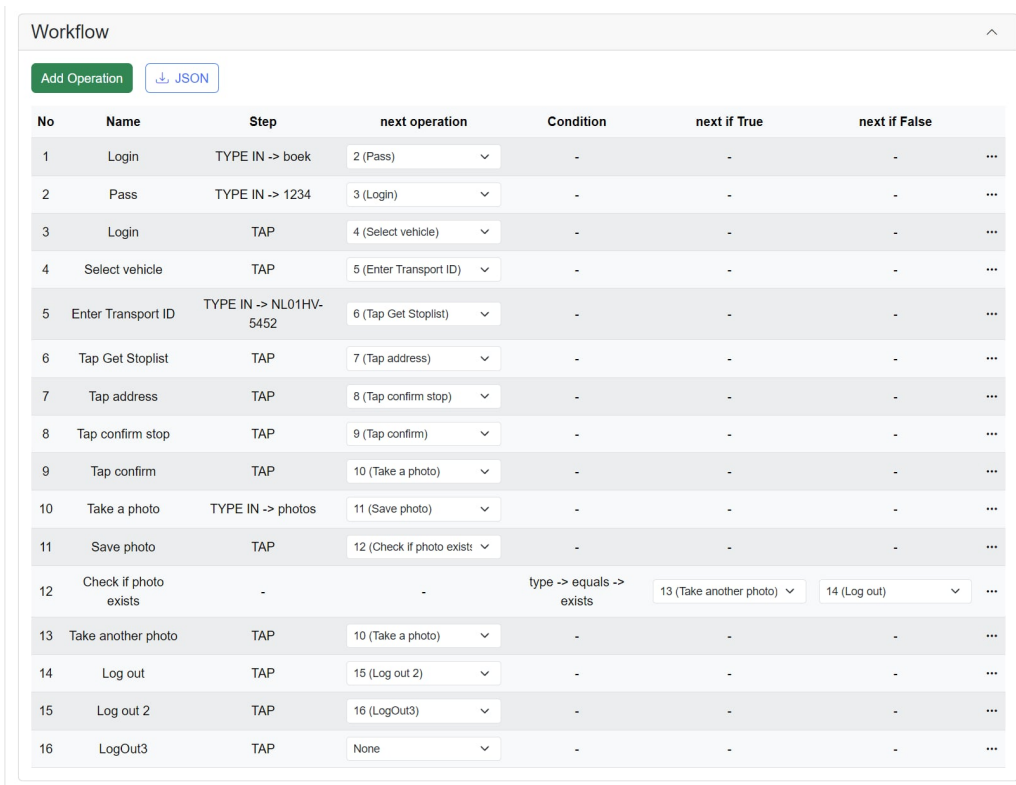
entered_by_id

Next Operation (Optional)

None

Cancel Save

Rys. 4.5. Deklaracja operacji FIND TEXT w kreatorze integracji.



No	Name	Step	next operation	Condition	next if True	next if False
1	Login	TYPE IN -> book	2 (Pass)	-	-	-
2	Pass	TYPE IN -> 1234	3 (Login)	-	-	-
3	Login	TAP	4 (Select vehicle)	-	-	-
4	Select vehicle	TAP	5 (Enter Transport ID)	-	-	-
5	Enter Transport ID	TYPE IN -> NL01HV-5452	6 (Tap Get Stoplist)	-	-	-
6	Tap Get Stoplist	TAP	7 (Tap address)	-	-	-
7	Tap address	TAP	8 (Tap confirm stop)	-	-	-
8	Tap confirm stop	TAP	9 (Tap confirm)	-	-	-
9	Tap confirm	TAP	10 (Take a photo)	-	-	-
10	Take a photo	TYPE IN -> photos	11 (Save photo)	-	-	-
11	Save photo	TAP	12 (Check if photo exists)	-	-	-
12	Check if photo exists	-	-	type -> equals -> exists	13 (Take another photo)	14 (Log out)
13	Take another photo	TAP	10 (Take a photo)	-	-	-
14	Log out	TAP	15 (Log out 2)	-	-	-
15	Log out 2	TAP	16 (LogOut3)	-	-	-
16	LogOut3	TAP	None	-	-	-

Rys. 4.6. Przykładowe workflow dla aplikacji kontrahenta.

4.5.3. Zastosowanie środowiska Python w celu integracji komponentów

Ważną rolę w systemie wykonywania automatycznych interakcji pełni środowisko Python. To właśnie skrypt napisany w tym języku pełni rolę interpretera dla przychodzących plików JSON, mówiących, jak dane integracje mają się wykonać, oraz spaja w całość środowisko kreacji integracji z środowiskiem wykonania. Skrypt ten dostarcza również dostęp do komponentów wspierających, czyli narzędzi służących do widzenia komputerowego, rozpoznawania tekstu na ekranie czy symulacji wirtualnej kamery dla emulatora.

Automatyczne wywoływanie metod powłoki ADB

Jak opisano powyżej (patrz 4.4.4), cały proces komunikacji odbywa się przez polecenia powłoki ADB. Wszystkie te polecenia są wywoływane z poziomu Pythona. Na podstawie tych podstawowych poleceń zespół deweloperski stworzył funkcje usprawniające wykonywanie integracji, które są używane przez interpreter podczas rozczytywania pliku wejściowego JSON.

Jako przykład można przedstawić funkcję `type_in(coordinate, text)` (patrz 4.1). Zastosowano w niej połączenie trzech poleceń powłoki ADB: `input tap` (wywołane przez zewnętrzną funkcję `tap(coordinates)`), `input text` i `input keyevent`. Funkcja ta ma następujące działanie:

1. Kliknij w punkt o danych współrzędnych.
2. Wprowadź dany tekst.
3. Usuń klawiaturę z wyświetlacza poprzez `keyevent 4`.

Listing 4.1. Deklaracja operacji TYPE IN w kreatorze integracji.

```
1 def type_in(coordinates, text):
2     if coordinates:
3         x, y = coordinates
4         tap(x, y)
5         if text != 'photo':
6             call(["adb", "-s", target_device, "shell",
7                  "input", "text", text])
8             call(["adb", "-s", target_device, "shell",
9                  "input", "keyevent", "4"])
10        time.sleep(0.1)
```

Tak zbudowane funkcje łączące w sobie kilka poleceń powłoki oraz wywołania komponentów wspierających pozwalają na płynne przejście przez cały cykl wykonania integracji, symulując akcje, jakie mógłby wykonać w niej użytkownik.

4.5.4. Format JSON jako ustandaryzowana struktura przekazywania danych między komponentami systemu

Format JSON jest używany jako nośnik instrukcji wraz z danymi, zaciągniętymi z bazy danych klienta, potrzebnych do wykonania danej integracji (patrz listing 5.16). W sposób ustandaryzowany opisuje kolejne kroki, które są rozpoznawane przez środowisko wykonywalne Python w celu wykonania konkretnych akcji.

W przykładowym kroku przedstawionym poniżej (patrz 5.16) opisano krok pierwszy z workflow widocznego na grafice 4.6. Instrukcja ta reprezentu-

je proces wprowadzania loginu zawartego pod kluczem `input_value` w pole tekstowe opisane atrybutem `img_url` do aplikacji kontrahenta. Jako że nie wprowadzono w tym przypadku instrukcji warunkowej, pole `condition` pozostawiono puste lub wpisano w nie wartość `null`. Atrybut pola `next_workflow` pozwala odnaleźć interpreterowi następny krok w pliku JSON.

Listing 4.2. Przykładowy krok pozbawiony danych opisany w pliku JSON.

```
1 {  
2   "name": "Login",  
3   "order_number": 1,  
4   "step": {  
5     "action": "TYP",  
6     "input_value": "",  
7     "img_url": "",  
8     "next_workflow": "Pass"  
9   },  
10  "condition": {  
11    "name": "",  
12    "condition_type": "",  
13    "value_to_compare": "",  
14    "next_workflow_if_true": null,  
15    "next_workflow_if_false": null  
16  }  
17 }
```

4.6. Komponenty wspierające

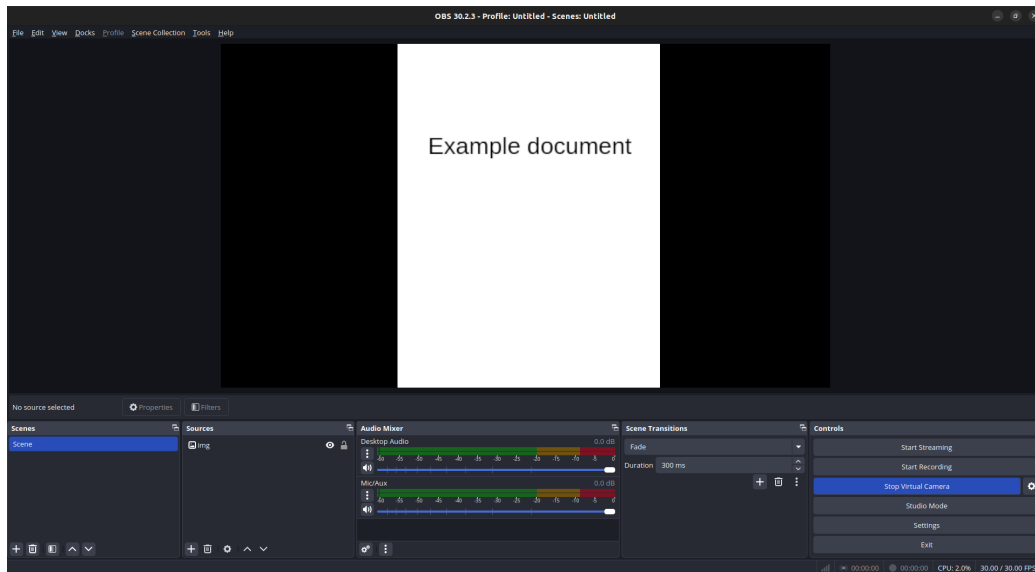
4.6.1. Open Broadcaster Software i symulacja wirtualnej kamery

W celu symulacji wirtualnej kamery na potrzeby emulatora użyto zewnętrznego programu Open Broadcaster Software (OBS)[5] wraz z wtyczką Virtual Camera (VC) [7]. Oprogramowanie to pozwala tworzyć sceny [6], wirtualne kamery oraz decydować, co ma być wyświetlane jako pole widzenia takiej kamery. W celu przesłania pliku graficznego do programu OBS użyto połączenia poprzez websocket [35]. Skrypt odpowiedzialny za wykonanie integracji otrzymuje w pliku JSON link do API umożliwiającego pobranie żądanej grafiki i przesyła otrzymany obraz do programu OBS za pomocą websocketu, jednocześnie wyświetlając go na działającej scenie (przykład patrz Rys.4.7). Rozwiązanie to okazało się być niezwykle istotne w tworzeniu całego systemu, gdyż w wielu aplikacjach kontrahenta wymagane jest zrobienie zdjęcia, a nie załączenie go z pamięci urządzenia.

4.6.2. OpenCV i PyTesseract

Komponentem wspierającym odpowiedzialnym za rozpoznawanie elementów i wyszukiwanie tekstu na ekranie emulatora jest segment widzenia komputerowego. Schemat działania w tych przypadkach jest do siebie zbliżony i zaprojektowano go w następujący sposób:

1. Wykonaj zrzut ekranu urządzenia wirtualnego.
2. Pobierz z bazy danych obraz elementu, który należy znaleźć, lub pobierz do zmiennej szukany tekst.



Rys. 4.7. Przykładowy dokument wyświetlany w polu widzenia wirtualnej kamery w programie OBS.

3. Przy pomocy widzenia komputerowego znajdź obraz lub tekst i zwróć współrzędne środka tego elementu.

Rozwiązanie to pozwoliło zespołowi deweloperskiemu na szybsze wdrożenie systemu w działającej formie i nie wiązało się z elementami inżynierii wstecznej aplikacji kontrahentów.

Rozdział 5

Budowa backendu aplikacji webowej z wykorzystaniem framework Django

Autor: Michał Wujec

5.1. Django jako framework

Wprowadzenie

Django to jeden z najbardziej popularnych frameworków, który został napisany w języku Python. Jest to wysokopoziomowy framework, który służy do tworzenia aplikacji internetowych (webowych).

Cechy szczególne Django to:

- **Szybkość:** Django umożliwia szybkie tworzenie aplikacji dzięki wbudowanym funkcjom i narzędziom (np. system zarządzania użytkownikami, obsługa baz danych).

- **Wyższa abstrakcja:** Framework dostarcza wiele uproszczeń i abstrakcji. Dobrym przykładem jest ORM (Object-Relational Mapping), które pozwala na pracę z bazami danych bez konieczności pisania zapytań SQL.
- **Bezpieczeństwo:** Dostępna jest automatyczna obsługa typowych zagrożeń związanych z aplikacjami webowymi, przede wszystkim SQL Injection, Cross-Site Scripting (XSS) czy Cross-Site Request Forgery (CSRF).
- **Elastyczność:** Django pozwala na tworzenie szerokiego zakresu aplikacji, tzn. od prostych stron internetowych po złożone systemy (np. Instagram, Spotify czy YouTube).
- **Łatwość nauki:** Dzięki bogatej dokumentacji [13] oraz wielu forum internetowym, Django jest frameworkiem łatwym do nauki i użytkowania.

5.2. Model View Controller (MVC)

5.2.1. Wprowadzenie

Wzorce projektowe

Wzorce projektowe to uniwersalne i sprawdzone rozwiązania często występujących problemów programistycznych. Ułatwiają one tworzenie, organizację i rozwój oprogramowania, dostarczając zrozumiałych schematów dla typowych wyzwań w projektowaniu systemów.

Definicja wzorca projektowego na podstawie [11]:

Wzorzec projektowy to opis sprawdzonego sposobu rozwiązywania powtarzających się problemów w systemach obiektowych. Wyjaśnia, na czym polega problem, jak go rozwiązać, kiedy warto zastosować to rozwiązanie oraz jakie mogą być jego konsekwencje. Dodatkowo zawiera wskazówki, jak je wdrożyć, oraz przykłady. Rozwiązanie przedstawia ogólny układ obiektów i klas, które wspólnie rozwiązują problem. Może być ono dostosowane i wdrożone w zależności od konkretnego przypadku.

Model View Controller (MVC)

Model View Controller (MVC) to rodzaj architektonicznego wzorca projektowego. Jest często stosowany w projektowaniu aplikacji, pomagając w uporządkowaniu kodu i zwiększając modularność oraz przejrzystość projektu. Wzorzec ten powstał pod koniec lat 70. XX wieku. Za jego autora uważa się Trygve'a Reenskauga, który opracował główne idee podczas prac w laboratoriach "Palo Alto Research Centre" firmy Xerox nad językiem SmallTalk.

5.2.2. Koncepcja

Głównym założeniem MVC jest podział aplikacji na trzy warstwy: Model, Widok, i Kontroler. Taka struktura pozwala na niezależny rozwój, testowanie oraz modyfikowanie poszczególnych komponentów, co ułatwia utrzymanie i rozwój aplikacji.

- **Model (Model)**

Model zawiera logikę biznesową oraz dane aplikacji. To tutaj przechowywane są informacje, które aplikacja musi przetwarzać, na przykład

dane użytkowników, produkty w sklepie internetowym czy wyniki obliczeń. Model jest odpowiedzialny za komunikację z bazą danych i zapewnia aktualizowanie danych na żądanie innych komponentów.

- **Widok (View)**

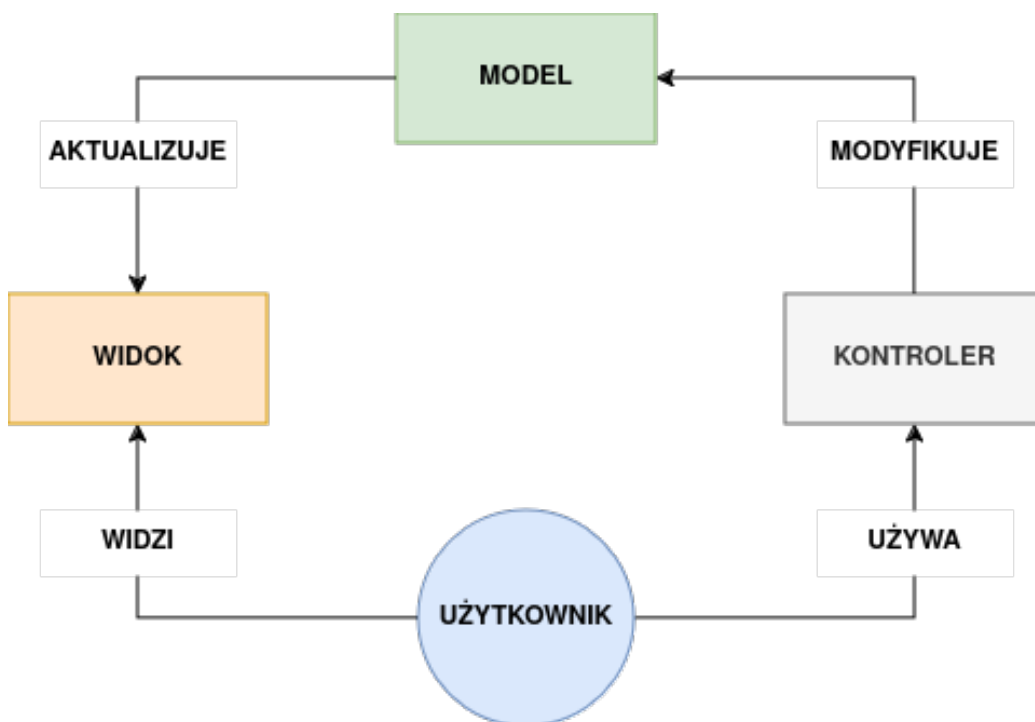
Widok odpowiada za wyświetlanie danych użytkownikowi. Jego zadaniem jest prezentacja informacji w czytelnej i przejrzystej formie. Co ważne, widok nie przetwarza danych — odbiera je z modelu i generuje w interfejsie użytkownika. Przykładem może być generowanie wykresów czy formularzy.

- **Kontroler (Controller)**

Kontroler obsługuje dane wejściowe wprowadzane przez użytkownika. Jest odpowiedzialny za odbieranie żądań (np. kliknięcia przycisków, przesyłanie formularzy) oraz ich interpretowanie. Następnie przekazuje odpowiednie polecenia modelowi lub widokowi. Kontroler łączy interakcję użytkownika z logiką aplikacji, umożliwiając dynamiczne reagowanie na działania użytkownika.

Przepływ informacji w MVC

Struktura przepływu informacji pomiędzy warstwami Model, Widok oraz Kontroler została przedstawiona na rysunku 5.1. Schemat ilustruje, w jaki sposób poszczególne komponenty współpracują ze sobą, aby zapewnić prawidłowe działanie aplikacji.



Rys. 5.1. Schemat przepływu informacji w MVC.

5.2.3. Zastosowanie MVC w Django

Django stosuje wzorzec MVC, ale w swojej implementacji przyjęło zmodyfikowane nazewnictwo komponentów, nazywane Model-Template-View (MTV). Główna różnica polega na przypisaniu ról poszczególnym komponentom:

- **Model (Model)**

W obu wzorcach model odpowiada za zarządzanie danymi i logikę biznesową. W Django model opisuje strukturę tabel w bazie danych. Jest to klasa dziedzicząca po `django.db.models.Model`, której atrybuty odpowiadają polom w tabeli bazy danych. Operacje na danych są realizowane przy użyciu Object-Relational Mapping (ORM), co pozwala na uniezależnienie kodu od konkretnego typu bazy danych (np. SQLite, PostgreSQL czy MySQL).

- **Widok (View)**

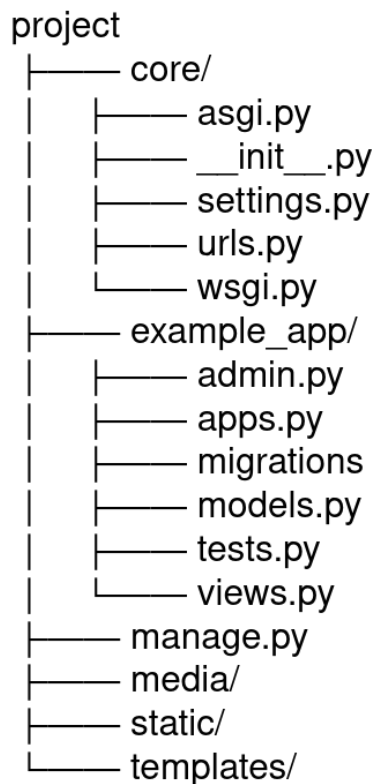
W tradycyjnym MVC widok odpowiada za prezentację danych, natomiast w Django pełni rolę kontrolera opisanego w MVC. Widok w Django to funkcja lub klasa, która przetwarza żądanie użytkownika, następnie pobiera dane z modelu i przekazuje je do szablonu. Przyjmuje parametr `request` i zwraca odpowiedź `Http` (np. stronę HTML lub dane JSON). Widoki mogą także obsługiwać autoryzację, przekierowania oraz inne zadania logiczne.

- **Szablon (Template)**

W MTV szablon odpowiada za warstwę prezentacji. Jest to plik tekstowy zawierający kod HTML oraz specjalne tagi Django. Tagi te umożliwiają dynamiczne generowanie zawartości, takie jak wyświetlanie danych z bazy czy logika kontrolna. Django wspiera także inne formaty, takie jak CSS czy JavaScript.

5.3. Struktura projektu w Django

W trakcie pisania projektu bardzo istotną kwestią jest jego czytelność oraz odpowiednia organizacja. Czytelność oznacza taki sposób pisania kodu oraz strukturyzowania plików, który umożliwia łatwe zrozumienie i pracę nad projektem zarówno obecnym, jak i przyszłym członkom zespołu. Odpowiednia struktura plików wspiera efektywność pracy, ułatwia skalowanie i również utrzymanie projektu w dłuższej perspektywie. Standardowa struktura projektu Django została przedstawiona na rysunku 5.2, stanowiąc punkt wyjścia do omówienia kluczowych elementów każdego projektu.



Rys. 5.2. Struktura plików w projekcie Django.

5.3.1. Główne pliki i katalogi

W typowym projekcie Django można wyróżnić kilka najważniejszych plików i katalogów, które pełnią określone funkcje:

- **manage.py**

To plik do zarządzania projektem. Jest używany do wykonywania poleceń administracyjnych w Django. Przy jego użyciu można uruchamiać serwer deweloperski, tworzyć migracje, zarządzać bazą danych oraz generować nowe aplikacje w projekcie. Możliwe jest również tworzenie własnych funkcji, które można wywoływać za pomocą tego pliku.

Ogólny wzór użycia:

```
$ python manage.py <command> [options]
```

Przykład użycia **manage.py** do uruchomienia serwera:

```
$ python manage.py runserver 0.0.0.0:8000
```

Opis: Argument 0.0.0.0 wskazuje adres IP, na którym serwer nasłuchuje, a 8000 określa port.

- **asgi.py** i **wsgi.py**

Te pliki są punktami wejścia dla serwera aplikacji.

- **asgi.py** — Odpowiada za obsługę asynchronicznego serwera ASGI (Asynchronous Server Gateway Interface), który pozwala na implementację nowoczesnych protokołów, takich jak WebSocket.
- **wsgi.py** — Służy do obsługi synchronicznego serwera WSGI (Web Server Gateway Interface) i jest powszechnie używany przy wdrażaniu aplikacji na serwerach takich jak Gunicorn albo WSGI.

- **settings.py**

Plik `settings.py` jest jedną z najważniejszych części projektu Django. Zawiera całą konfigurację instalacji Django, kontroluje takie aspekty jak:

- Klucz tajny aplikacji (`SECRET_KEY`)
Jest używany do kryptograficznego podpisywania w Django. Powinien być tajny i unikalny dla każdej instalacji Django. Aby zadbać

o bezpieczeństwo w środowisku produkcyjnym nigdy nie należy go przechowywać bezpośrednio w pliku ustawień. Jednym z możliwych rozwiązań jest użycie zmiennych środowiskowych:

Listing 5.1. Settings.py definiowanie 'SECRET_KEY'

```
1 SECRET_KEY = os.environ.get('DJANGO_SECRET_KEY')
```

– Lista hostów (ALLOWED_HOSTS)

Jest to lista nazw hostów lub domen, które mogą obsługiwać daną stronę Django. Taka metoda ma za zadanie zwiększyć bezpieczeństwo poprzez zapobieganie atakom na nagłówki HTTP Host. W środowisku deweloperskim często używa się:

Listing 5.2. Settings.py definiowanie 'ALLOWED_HOSTS' lokalnie

```
1 ALLOWED_HOSTS = ['localhost', '127.0.0.1']
```

W środowisku produkcyjnym należy natomiast podać nazwę swojej domeny:

Listing 5.3. Settings.py definiowanie 'ALLOWED_HOSTS' produkcyjnie

```
1 ALLOWED_HOSTS = ['www.domena.com']
```

– Zainstalowane aplikacje (INSTALLED_APPS)

Jest to lista, która dostarcza informacje Django, które aplikacje są aktywne w tym projekcie. Domyślna lista zawiera wbudowane aplikacje Django:

Listing 5.4. Settings.py definiowanie aktywnych aplikacji w 'INSTALLED_APPS'

```
1 INSTALLED_APPS = [
```

```
2     'django.contrib.admin',
3     'django.contrib.auth',
4     'django.contrib.contenttypes',
5     'django.contrib.sessions',
6     'django.contrib.messages',
7     'django.contrib.staticfiles',
8 ]
```

Można również tworzyć swoje własne aplikacje. Wówczas należy dopisać je do listy:

Listing 5.5. Settings.py dodawanie własnych aplikacji w 'INSTALLED_APPS'

```
1 INSTALLED_APPS = [
2     ...
3     'moja_aplikacja',
4 ]
```

– MIDDLEWARE

Jest to mechanizm w Django, który umożliwia ingerencję w proces obsługi żądań i odpowiedzi. Innymi słowy to niskopoziomowy system "wtyczek", pozwalający globalnie modyfikować dane wejściowe lub wyjściowe Django.

Domyślna konfiguracja MIDDLEWARE:

Listing 5.6. Settings.py definiowanie 'MIDDLEWARE'

```
1 MIDDLEWARE = [
2     'django.middleware.security.
      SecurityMiddleware',
3     'django.contrib.sessions.middleware.
      SessionMiddleware',
```

```
4      'django.middleware.common.CommonMiddleware',
5      'django.middleware.csrf.CsrfViewMiddleware',
6      'django.contrib.auth.middleware.AuthenticationMiddleware',
7      'django.contrib.messages.middleware.MessageMiddleware',
8      'django.middleware.clickjacking.XFrameOptionsMiddleware',
9  ]
```

Podobnie jak w przypadku aplikacji istnieje możliwość dodania własnego `middleware` do listy, tak aby dostosować działanie projektu do swoich potrzeb.

– Szablony (TEMPLATES)

To ustawienie konfiguruje renderowanie szablonów w Django. Klucz `DIRS` pozwala określić katalogi, w których Django ma szukać plików szablonów. Na przykład:

Listing 5.7. Settings.py definiowanie 'TEMPLATES'

```
1  TEMPLATES = [
2      {
3          'BACKEND': 'django.template.backends.
4              django.DjangoTemplates',
5          'DIRS': [BASE_DIR / 'templates'],
6          'APP_DIRS': True,
7          'OPTIONS': {
8              'context_processors': [
```

```
8         'django.template.  
          context_processors.debug',  
9         'django.template.  
          context_processors.request',  
10        ,  
11        'django.contrib.auth.  
          context_processors.auth',  
12        'django.contrib.messages.  
          context_processors.messages',  
13    ],  
14 },  
15 ]
```

- Konfiguracja bazy danych (DATABASES).

To ustawienie konfiguruje połączenie z bazą danych dla projektu Django. Domyślnie używana jest baza SQLite, która jest prostym rozwiązaniem odpowiednim do środowisk deweloperskich.

Listing 5.8. Settings.py definiowanie 'DATABASES' metoda standardowa

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.sqlite3',  
4         ,  
5         'NAME': BASE_DIR / 'db.sqlite3',  
6     }  
}
```

W środowisku produkcyjnym można skonfigurować bazę danych

PostgreSQL, która jest bardziej wydajna i odpowiednia do dużych aplikacji.

Listing 5.9. Settings.py definiowanie 'DATABASES' dla PostgreSQL

```
1 DATABASES = {  
2     'default': {  
3         'ENGINE': 'django.db.backends.  
4             postgresql',  
5         'NAME': 'db_name',  
6         'USER': 'db_user',  
7         'PASSWORD': 'db_password',  
8         'HOST': 'localhost',  
9         'PORT': '5432',  
10    }
```

- **urls.py**

Plik odpowiedzialny za mapowanie adresów URL na odpowiednie widoki. Dzięki niemu można określić, która funkcja lub klasa widoku odpowiada na określony adres URL w aplikacji. Przykładowy zapis:

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('blog/', include('blog.urls')),  
]
```

- **urls.py**

W Django plik `urls.py` zawiera listę wzorców URL (URL patterns), które mapują adresy URL na odpowiednie widoki (views). Wzorzec

URL jest definiowany za pomocą funkcji `path()` lub `re_path()`. Gdy użytkownik wysyła żądanie do określonego adresu URL, Django przeszukuje listę wzorców w pliku `urls.py` i wybiera pierwszy pasujący wzorzec. Jeśli żaden wzorzec nie pasuje, Django zwraca błąd 404.

Listing 5.10. Przykład zdefiniowania pliku `urls.py`

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path('', views.index, name='index'),
6     path('info/', views.info, name='info'),
7 ]
```

- **media/**

Katalog `media/` służy do przechowywania plików przesyłanych przez użytkowników, takich jak obrazy, dokumenty czy inne zasoby. W Django, lokalizacja katalogu `media/` jest definiowana w ustawieniach za pomocą zmiennych `MEDIA_URL` i `MEDIA_ROOT`.

Listing 5.11. Przykład zdefiniowania `MEDIA_URL` oraz `MEDIA_ROOT`

```
1 MEDIA_URL = '/media/'
2 MEDIA_ROOT = BASE_DIR / 'media'
```

W środowisku produkcyjnym pliki z katalogu `media/` powinny być obsługiwane przez serwer WWW, taki jak Nginx lub Apache.

- **static/**

Katalog `static/` przechowuje pliki statyczne projektu, takie jak style CSS, skrypty JavaScript oraz obrazy. Django oferuje mechanizm zbierania

rania plików statycznych z różnych aplikacji do jednego miejsca za pomocą polecenia:

```
python manage.py collectstatic
```

Po uruchomieniu tego polecenia pliki statyczne są gotowe do serwowania przez serwer WWW, co zwiększa szybkość ładowania strony. W ustawieniach Django konfigurujemy ścieżki dla plików statycznych za pomocą `STATIC_URL` i `STATIC_FILES_DIRS`.

- **templates/**

Katalog `templates/` przechowuje szablony HTML wykorzystywane w warstwie prezentacji aplikacji. Szablony te umożliwiają dynamiczne generowanie treści, iteracje oraz używanie warunków dzięki specjalnym tagom Django. Informacje na temat konfiguracji szablonów w pliku `settings.py` oraz szczegółowy przykład został opisany już wcześniej. (zobacz: Konfiguracja szablonów)

5.3.2. Aplikacje

```
example_app
├── admin.py
├── apps.py
├── migrations\
├── models.py
├── tests.py
└── views.py
```

Rys. 5.3. Struktura aplikacji w projekcie Django.

Aplikacje w Django to niezależne moduły, które realizują konkretne funkcjonalności w ramach projektu. Pozwalają na logiczne organizowanie kodu i zwiększają modularność oraz skalowalność aplikacji. Dzięki zastosowaniu aplikacji łatwiej zarządzać kodem, rozwijać poszczególne części projektu oraz wielokrotnie wykorzystywać gotowe moduły w różnych projektach. Dobrą praktyką jest aby każda aplikacja odpowiadała za jedną, jasno określoną funkcjonalność. Należy unikać mieszania odpowiedzialności między aplikacjami. W dużych projektach warto podzielić aplikacje na mniejsze moduły, co ułatwi zarządzanie kodem.

Ważne jest również zrozumienie różnicy między projektem a aplikacją:

- **Projekt** - to zbiór konfiguracji i aplikacji, które wspólnie tworzą system webowy. Może zawierać wiele aplikacji, z których każda odpowiada za określoną funkcjonalność.
- **Aplikacja** - to niezależny komponent, który można wykorzystywać w wielu projektach, bez potrzeby wprowadzania większych modyfikacji.

Na przykładzie systemu **LogInt**, projekt może być podzielony na wiele aplikacji, z których każda realizuje określoną funkcjonalność:

- **accounts** - obsługa rejestracji i logowania użytkowników,
- **history** - gromadzenie danych dotyczących operacji wykonywanych przez użytkowników,
- **home** - moduł integracji, umożliwiający zarządzanie głównymi ustawieniami,
- **reports** - generowanie raportów na podstawie dokonanych operacji,

- `sources` - zarządzanie źródłami zewnętrznymi, służącymi do pobierania danych,
- `steps` - tworzenie i obsługa instrukcji wykonywania kroków.

Tworzenie nowej aplikacji

Aby utworzyć aplikację w Django, należy przejść do katalogu projektu, gdzie znajduje się plik `manage.py`. Następnie wprowadzić poniższą komendę w wierszu poleceń:

```
python manage.py startapp <nazwa_aplikacji>
```

Po utworzeniu aplikacji należy dopisać jej nazwę do sekcji `INSTALLED_APPS` w pliku `settings.py`, aby została zarejestrowana w projekcie.
(zobacz Zainstalowane aplikacje))

Struktura aplikacji

Po zainstalowaniu aplikacji w folderze zostaje utworzony zestaw plików:

- `__init__.py`
Plik ten jest automatycznie generowany w każdej aplikacji i wskazuje, że dany katalog jest pakietem. Zwykle pozostaje pusty, ale można w nim umieścić logikę inicjalizacyjną dla aplikacji.
- `admin.py`
W tym pliku definiowane są konfiguracje interfejsu administracyjnego Django dla modeli aplikacji. Można rejestrować modele, aby były

widoczne w panelu administracyjnym, oraz zdefiniować, jakie kolumny powinny być wyświetlane w panelu. Dzięki temu możesz dostosować wygląd i funkcjonalność panelu administracyjnego do potrzeb projektu.

Listing 5.12. Przykład rejestracji modelu i konfiguracji widocznych kolumn

```
1 from django.contrib import admin
2 from .models import Source
3
4 @admin.register(Source)
5 class SourceAdmin(admin.ModelAdmin):
6     list_display = ('source_name', 'link')
```

- apps.py

Plik `apps.py` odpowiada za konfigurację ustawień i metadanych aplikacji. Służy do definiowania klasy konfiguracji aplikacji (`AppConfig`), która określa nazwę aplikacji oraz inne opcjonalne parametry, takie jak domyślne pola modeli, nazwa w panelu administracyjnym czy rejestracja sygnałów.

Listing 5.13. Konfiguracja w `apps.py` na przykładzie aplikacji "sources" w systemie LogInt

```
1 from django.apps import AppConfig
2
3 class SourcesConfig(AppConfig):
4     default_auto_field = 'django.db.models.
5         BigAutoField'
6     name = 'sources'
```

- migrations/

Katalog `migrations/` zawiera pliki migracji, które są wykorzystywane

przez Django do zarządzania zmianami w strukturze bazy danych. Migracje odzwierciedlają zmiany w modelach aplikacji, takie jak dodanie nowych pól, usunięcie kolumn czy zmiana typów danych. Pliki w tym katalogu są generowane automatycznie za pomocą polecenia:

```
python manage.py makemigrations
```

Aby zastosować migracje w bazie danych, należy użyć polecenia:

```
python manage.py migrate
```

Katalog `migrations/` umożliwia śledzenie i kontrolowanie historii zmian w bazie danych, co jest szczególnie przydatne w dużych projektach z wieloma modelami.

- `models.py`

Plik `models.py` służy do definiowania modeli danych w aplikacji Django. Modele to podstawowe elementy aplikacji, które reprezentują strukturę danych i umożliwiają interakcję z bazą danych. Każdy model jest klasą, która dziedziczy z `django.db.models.Model`, a jego pola odpowiadają kolumnom w tabeli bazy danych.

Listing 5.14. Przykład modelu dla History

```
1 from django.contrib.auth.models import User
2 from django.db import models
3 from django.utils import timezone
4
5
```

```
6 class History(models.Model):
7     user = models.ForeignKey(User, on_delete=
8         models.CASCADE)
9     type = models.CharField(max_length=50)
10    name = models.CharField(max_length=100)
11    operation = models.CharField(max_length=50)
12    operation_date = models.DateTimeField(default=
13        timezone.now)
14
15    def __str__(self):
16        return f"{self.type} | {self.name} | {self
17            .operation} | {self.user.username}"
```

Modele Django obsługują różne typy pól, m. in.:

- CharField - tekst o określonej długości,
- IntegerField - liczby całkowite,
- DateTimeField - daty i czasy,
- BooleanField - wartości logiczne (True/False),
- ForeignKey, ManyToManyField, OneToOneField - relacje między modelami.

- tests.py

Plik `tests.py` służy do definiowania testów jednostkowych w aplikacji Django. Testy pozwalają na weryfikację poprawności działania kodu, zapewniając, że aplikacja działa zgodnie z założeniami. Testy są szczególnie przydatne podczas rozwoju i modyfikacji kodu, ponieważ pomagają wykrywać błędy na wczesnym etapie.

Listing 5.15. Przykład testu jednostkowego dla modelu History

```
1 from django.test import TestCase
2 from django.contrib.auth.models import User
3 from .models import History
4
5 class HistoryModelUnitTests(TestCase):
6     def setUp(self):
7         self.user = User.objects.create_user(
8             username="testuser", password="password"
9         )
10        self.history = History.objects.create(
11            user=self.user,
12            type="File",
13            name="test_file.txt",
14            operation="CREATE",
15            operation_date="2024-11-14T10:00:00Z"
16        )
17
18    def test_str_representation(self):
19        expected_str = "File | test_file.txt |"
20                       "CREATE | testuser"
21        self.assertEqual(str(self.history),
22                          expected_str)
```

Metoda `setUp()` przygotowuje środowisko testowe, tworząc użytkownika i instancję modelu `History`.

`test_str_representation()` weryfikuje, czy metoda `__str__()` modelu `History` zwraca oczekiwaną wartość.

Testy w Django uruchamiane są za pomocą polecenia:

```
python manage.py test
```

Istnieją również narzędzia, które pozwalają sprawdzić, jaki procent kodu aplikacji został pokryty przez testy. Jednym z popularnych narzędzi jest paczka `coverage`. Aby z niej skorzystać, należy ją zainstalować w środowisku projektu. Następnie testy można uruchomić wraz z mierzeniem pokrycia kodu za pomocą polecenia:

```
coverage run manage.py test
```

Po zakończeniu testów można wygenerować raport w terminalu lub formie html:

```
coverage report
```

- `views.py`

Plik `views.py` jest miejscem, w którym definiowana jest logika obsługi żądań HTTP w aplikacji Django. Widoki są centralnym elementem aplikacji, ponieważ zarządzają przepływem danych pomiędzy użytkownikiem a systemem oraz decydują o tym, jak dane są przetwarzane i prezentowane. Widok to funkcja lub klasa, która przetwarza żądanie użytkownika (`HttpRequest`) i zwraca odpowiedź (`HttpResponse`). Widoki pełnią rolę pośrednika pomiędzy danymi a warstwą prezentacji.

W Django dostępne są dwa rodzaje widoków:

- Widoki oparte na funkcjach (Function-Based Views, FBV)
- Widoki oparte na klasach (Class-Based Views, CBV)

Listing 5.16. Przykład widoku opartego na funkcji (FBV) i obsługującego żądania GET i zwracającego dane w formacie JSON

```
1 from django.contrib.auth.decorators import
    login_required
2 from django.http import JsonResponse
3 from django.shortcuts import get_object_or_404
4 from .models import Integration
5
6 @login_required(login_url='/auth/login/')
7 def get_integration_details(request):
8     if request.method == "GET":
9         integration_id = request.GET.get('id')
10        integration = get_object_or_404(
11            Integration, pk=integration_id)
12        data = integration.to_dictionary()
13
14        return JsonResponse(data)
15    else:
16        return JsonResponse({'valid': 'false'})
```


Spis rysunków

1.1. Diagram przedstawiający proces integracji danych w systemie, z wyszczególnieniem funkcji skryptu wykonującego integrację.	25
1.2. Diagram przedstawiający architekturę systemu LogInt.	28
2.1. Diagram klas systemu LogInt - rysunek własny	41
3.1. Diagram przedstawiający proces uzupełniania dwóch aplikacji tymi samymi danymi.	65
3.2. Diagram przedstawiający proces uzupełniania wielu aplikacji kontrahenta z jednego źródła.	66
4.1. Samsung SM-G998B – emulator BlueStacks rozpoznawany ja- ko urządzenie fizyczne w systemie.	81
4.2. Architektura serwera ADB - rysunek własny na podstawie https://emteria.com/learn/android-debug-bridge	83
4.3. Deklaracja operacji TAP w kreatorze integracji.	90
4.4. Deklaracja operacji TYPE IN w kreatorze integracji.	91
4.5. Deklaracja operacji FIND TEXT w kreatorze integracji.	91
4.6. Przykładowe workflow dla aplikacji kontrahenta.	92
4.7. Przykładowy dokument wyświetlany w polu widzenia wirtu- alnej kamery w programie OBS.	97
5.1. Schemat przepływu informacji w MVC.	102

5.2. Struktura plików w projekcie Django.	104
5.3. Struktura aplikacji w projekcie Django.	112

Bibliografia

- [1] Eli5: how does bluestacks work? https://www.reddit.com/r/explainlikeimfive/comments/n3p66a/eli5_how_does_bluestacks_work/, Accessed on: 2024-12-05.
- [2] How zapier works. <https://docs.zapier.com/platform/quickstart/how-zapier-works>, Accessed on: 2025-06-19.
- [3] Make use cases. <https://www.make.com/en/use-cases>, Accessed on: 2025-06-19.
- [4] Microsoft power automate documentation. <https://learn.microsoft.com/en-us/power-automate/>, Accessed on: 2025-06-19.
- [5] Obs quick start. <https://obsproject.com/kb/quick-start-guide>, Accessed on: 2024-12-27.
- [6] Obs studio guide. <https://obsproject.com/kb/obs-studio-overview#scenes-and-sources>, Accessed on: 2024-12-27.
- [7] Obs virtual camera guide. <https://obsproject.com/kb/virtual-camera-guide>, Accessed on: 2024-12-27.

-
- [8] APPIUM PROJECT. Appium Documentation, 2023. Latest version accessed.
- [9] BITKOWSKA, A. *Zarządzanie procesami biznesowymi w przedsiębiorstwie*. VIZJA PRESS IT, 2009.
- [10] BUDI UTOMO. Docker-android - documentation. <https://github.com/budtmo/docker-android?tab=readme-ov-file>, Accessed on: 2024-12-05.
- [11] ERICH GAMMA, RICHARD HELM, R. J. J. V. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [12] FIG. obs-websocket - remote-control obs studio using websockets 5.0.1. <https://fig.io/manual/adb/install-multiple>, Accessed on: 2024-12-10.
- [13] FOUNDATION, D. S., AND INDIVIDUAL CONTRIBUTORS. Django documentation. <https://docs.djangoproject.com/en/5.1/>, 2024.
- [14] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, Cambridge, MA, 2016. Online version available.
- [15] GOOGLE. Android developer documentation - android debug bridge (adb). <https://developer.android.com/tools/adb?hl=pl>, Accessed on: 2024-12-05.
- [16] GOOGLE. Android developer documentation - configure on-device developer options. <https://developer.android.com/studio/debug/dev-options#enable>, Accessed on: 2024-12-05.

- [17] GOOGLE. Android developer documentation - create and manage virtual devices. <https://developer.android.com/studio/run/managing-avds>, Accessed on: 2024-12-05.
- [18] GOOGLE. Android developer documentation - run apps on the android emulator. <https://developer.android.com/studio/run/emulator>, Accessed on: 2024-12-04.
- [19] GOOGLE DEVELOPERS. Tesseract OCR Documentation. Accessed: 2023-12-13.
- [20] GROUP, P. G. D. About. <https://www.postgresql.org/about/>.
- [21] GROUP, P. G. D. Postgresql. <https://www.postgresql.org.pl/>.
- [22] KHAN, A. A., LAGHARI, A. A., AND AWAN, S. A. Machine learning in computer vision: A review. *EAI Endorsed Transactions on Scalable Information Systems* 8 (2021), e4.
- [23] KHAN, A. I., AND AL-HABSIB, S. Machine learning in computer vision. *Procedia Computer Science* 167 (2020), 1444–1451.
- [24] MACHAJ, M. Wprowadzenie. http://www.informatyka.orawskie.pl/?pl_wprowadzenie, 165.
- [25] MINDBOX. Docker – czym jest konteneryzacja i dlaczego może się ona przydać w twojej firmie? <https://mindboxgroup.com/pl/docker-czym-jest-konteneryzacja-i-dlaczego-moze-sie-ona-przydac-w-twojej-fi>. Accessed on: 2024-12-05.
- [26] MONGODB, I. Database architecture introduction. <https://www.mongodb.com/resources/basics/databases/database-architecture>.

- [27] NEWDATALABS.COM. Halo, baza? czyli słów kilka o bazach danych. <https://newdatalabs.com/bazy-danych/>.
- [28] NIALL O' MAHONY, SEAN CAMPBELL, A. C. S. H. Deep learning vs. traditional computer vision. *arXiv 1910.13796* (2019).
- [29] ORZEŁEK, R. 13. strukturalny język zapytań sql - wprowadzenie. <https://rafalorzelek.pl/nauka-menu/12-systemy-baz-danych/60-13-strukturalny-j%C4%99zyk-zapyta%C5%84-sql-wprowadzenie.html>.
- [30] PAWEŁ GŁUCHOWSKI, P. W. Logika temporalna i automaty czasowe. [urlhttp://staff.iiar.pwr.wroc.pl/pawel.gluchowski/wp-content/uploads/miasi/logika_w10.pdf](http://staff.iiar.pwr.wroc.pl/pawel.gluchowski/wp-content/uploads/miasi/logika_w10.pdf).
- [31] PEREZ, M. Przewodnik po tworzeniu diagramów uml i modelowaniu baz danych. <https://www.microsoft.com/pl-pl/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling>.
- [32] PERLIŃSKI, R. Klasyfikacja systemów baz danych ze względu na:. <https://icis.pcz.pl/~rperlinski/strona/files/sbd/SBDw07.p.pdf>.
- [33] PYPLACZ, P., AND SASAK, J. Rpa jako narzędzie automatyzacji i optymalizacji procesów. *Organizacja i Kierowanie*, 2 (191) (cze. 2022), 173–188.
- [34] TERESA REIDT. Using android debug bridge. <https://emteria.com/learn/android-debug-bridge>, Accessed on: 2024-12-05.

-
- [35] TT2468. obs-websocket - remote-control obs studio using websockets 5.0.1. <https://obsproject.com/forum/resources/obs-websocket-remote-control-obs-studio-using-websockets.466/updates#resource-update-4796>, Accessed on: 2024-12-10.
- [36] WANG, H., PAN, C., GUO, X., AND JI, C. From object detection to text detection and recognition: A brief evolution history of optical character recognition. *WIREs Computational Statistics 2021* (2021), e1547.
- [37] WIKIPEDIA. Rozproszona baza danych. https://pl.wikipedia.org/wiki/Rozproszona_baza_danych.
- [38] WORLD WIDE WEB CONSORTIUM (W3C). XML Path Language (XPath) 3.1, 2017. Published as a W3C Recommendation.
- [39] WREMBEL, R. Bazy danych - wykład. <https://wazniak.mimuw.edu.pl/index.php?title=BD-2st-1.2-w01.tresc-1.1-Slajd1>.