



POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra metod programowania

Programowanie aplikacji biznesowych

Rafał Reszczyński

Nr albumu: 14983

**System obsługi zleceń produkcyjnych
dla zakładu świadczącego usługę ekstrakcji
materiałów roślinnych**

Praca inżynierska

dr Krzysztof Barteczko

Warszawa, wrzesień, 2020

Streszczenie pracy

Aplikacje internetowe są często wykorzystywanym sposobem realizacji systemów typu CRM. W pracy opisano proces tworzenia tego typu systemu dla rzeczywiście funkcjonującego Zakładu, zaczynając od analizy istniejących procesów biznesowych w firmie. Następnie przedstawiono charakterystykę aplikacji webowej, omówiono wady i zalety aplikacji internetowej jako sposobu realizacji projektowanego systemu. Kolejnym etapem był wybór technologii, przy użyciu których aplikacja została wykonana, ze szczególnym naciskiem na bezpieczeństwo przechowywanych danych i zabezpieczenie przed nieautoryzowanym dostępem do nich.

W dalszej części pracy przedstawiono koncepcję nowego modelu biznesowego procesu obsługi zleceń produkcyjnych w Zakładzie. Na bazie przygotowanego modelu zostały określone wymagania, jakie powinien spełniać projektowany system. Następnym krokiem była implementacja systemu i omówienie użytych rozwiązań.

Końcowym etapem było dokonanie podsumowania wykonanych działań oraz przedstawienie koncepcji rozbudowy systemu o kolejne funkcjonalności.

Słowa kluczowe:

Aplikacja internetowa, WebApp, MySQL, Node.js, React.js, JWT, bcrypt

Spis treści

Wstęp.....	4
1 Cel i zakres pracy	5
2 Działalność zakładu.....	6
3 Obecny model obsługi zleceń produkcyjnych	8
4 Przegląd istniejących rozwiązań	11
5 Nowy model obsługi zleceń produkcyjnych	15
6 Zastosowane technologie	19
6.1 Aplikacja internetowa (Web App).....	19
6.2 Technologie używane w projektowaniu aplikacji internetowych	24
7 Koncepcja i budowa systemu	34
8 Implementacja	38
8.1 Frontend.....	38
8.2 Backend	68
Podsumowanie	80
Bibliografia.....	81
Spis ilustracji	83
Spis tabel	83
Spis listingów	84

Wstęp

Głównym zadaniem Zakładu jest wytwarzanie metodą ekstrakcji nadkrytycznym CO₂ produktów w postaci ekstraktów zawierających związki bioaktywne z dostarczonych przez zleceniodawców surowców roślinnych. Zakład nie prowadzi produkcji własnej, tzn. świadczy jedynie usługę ekstrakcji dla klientów.

Wstępna analiza procesów biznesowych zakładu dotyczących obsługi zleceń wykazała, że zlecenia obecnie przyjmuje się i obsługuje telefonicznie bądź e-mailowo, a klienci nie mają na bieżąco wglądu do tego na jakim etapie realizacji jest dane zlecenie. Kierownik wydziału finansowo-handlowego nie ma dostępu do informacji dotyczących przyjętych zleceń, danych dotyczących dostaw surowca i terminów odbioru produktów, a szef wydziału produkcyjnego - wiedzy na temat dostępnego zapasu surowców.

Na podstawie wyników przeprowadzonej analizy zdecydowano, że konieczne jest zaprojektowanie i wprowadzenie do użycia systemu informatycznego wspomagającego obsługę zleceń produkcyjnych.

1 Cel i zakres pracy

1.1 Cel pracy

Celem pracy jest zwiększenie efektywności działania i konkurencyjności Zakładu oraz usprawnienie przepływu informacji biznesowych pomiędzy Zakładem i innymi komórkami firmy, a klientami firmy, poprzez zastosowanie nowoczesnych i innowacyjnych rozwiązań z dziedziny IT.

1.2 Zakres pracy

W zakresie pracy leży zaprojektowanie i stworzenie systemu, który będzie składał się z bazy danych i dedykowanej aplikacji internetowej.

System będzie wspomagał użytkowników wewnętrznych firmy (dział magazynowy, dział produkcji) w obsłudze zleceń produkcyjnych.

Klienci korzystając z aplikacji będą mogli składać nowe zlecenia, a także mieć wgląd do zleceń w trakcie realizacji i informacji dotyczących zleceń anulowanych, bądź zakończonych. System powinien być prosty w obsłudze, zapewniać bezpieczeństwo przechowywanych danych oraz być zbudowany w sposób umożliwiający w przyszłości łatwe dodawanie nowych funkcjonalności.

2 Działalność zakładu

2.1 Profil działalności zakładu

Zakład, w którym będzie wdrażany nowy system zajmuje się działalnością produkcyjną polegającą na świadczeniu usługi ekstrakcji nadkrytycznym ditlenkiem węgla surowców roślinnych i materiałów naturalnych dostarczonych przez klientów. Wysoką jakość otrzymywanych produktów zapewnia prowadzenie procesu produkcyjnego w certyfikowanym przez niezależną jednostkę zewnętrzną systemie HACCP (Turlejska, 2014).

Głównymi produktami wytwarzanymi przez Zakład są:

- ekstrakt z szyszek chmielu,
- ekstrakty z nasion owoców jagodowych,
- ekstrakt z nasion soi,
- ekstrakt z kiełków pszenicy,
- ekstrakt z nasion palmy sabałowej.

Produkty te znajdują zastosowanie w przemyśle kosmetycznym spożywczym i farmaceutycznym, są również stosowane jako dodatki w suplementach diety oraz dodatki do pasz dla zwierząt.

2.2 Proces ekstrakcji

Ekstrakcja nadkrytycznym ditlenkiem węgla, to nowoczesny i innowacyjny proces, który w efektywny sposób pozwala wydobyć składniki bioaktywne z materiałów roślinnych (biomasy).

Największe zalety tej metody to:

- niska temperatura procesu, dzięki czemu substancje aktywne nie ulegają degradacji,
- nietoksyczny rozpuszczalnik, który można w prosty sposób oddzielić od produktu.

Zakład dysponuje przemysłową instalacją ekstrakcji nadkrytycznym CO₂. Proces ekstrakcji jest realizowany w obiegu zamkniętym, dzięki czemu zdecydowana większość rozpuszczalnika - ditlenku węgla, jest odzyskiwana. Metoda ta jest zaliczana do procesów tzw. „zielonej chemii”.

Ciekły ditlenek węgla, magazynowany w zbiorniku jest wstępnie chłodzony po czym kierowany na pompę, gdzie następuje sprężanie do wysokiego ciśnienia – od 150 do nawet 300 bar. Po stronie tłocznej pompy jest zlokalizowany podgrzewacz w którym rozpuszczalnik, podgrzewany do temperatury 40-60 °C, uzyskuje już parametry nadkrytyczne (ciśnienie powyżej 72,9 bar i temperatura powyżej 31,1 °C). Nadkrytyczny CO₂ przepływa przez materiał roślinny umieszczony w ekstraktorze – tam zachodzi właściwy proces ekstrakcji. Płyn nadkrytyczny idealnie łączy ze sobą zalety cieczy i gazu, to znaczy: wysoką

gęstość, a więc możliwość efektywnej solwatacji oraz niską lepkość - czyli dobre właściwości penetracyjne materiału.

Na instalacji ekstrakcji zlokalizowane są 4 ekstraktory mogące pracować niezależnie lub łącznie w połączeniu kaskadowym lub równoległym. Instalacja została zaprojektowana tak, że istnieje możliwość niezależnego wyłączenia z ruchu pojedynczego ekstraktora i wymiana materiału znajdującego się w nim bez konieczności przerywania pracy pozostałych ekstraktorów – w odniesieniu do przepływu ekstrahenta jest to więc praca w trybie ciągłym.

Po przepłynięciu przez ekstraktor, CO₂ nasycony ekstraktem jest rozprężany, ekstrakt oddziela się od ditlenku węgla w separatorze. CO₂ po schłodzeniu zawracany jest do zbiornika magazynowego, a ekstrakt po separacji kieruje się do zbiornika z mieszalnikiem, tzw. homogenizatora w celu ujednoludnienia składu w całej objętości ekstraktu.

Ostatnim etapem jest konfekcjonowanie ekstraktu w opakowania jednostkowe i zbiorcze.

Produkcja odbywa się zgodnie z zaleceniami systemu HACCP i normą ISO 9001:2015 (Kobylińska, 2014).

3 Obecny model obsługi zleceń produkcyjnych

Funkcjonujący obecnie w Zakładzie model obsługi zlecenia produkcyjnego jest daleki od optymalnego. Większość informacji dotyczących danego zlecenia przekazywanych wewnętrznie między komórkami firmy oraz między przedstawicielami firmy a klientem wykonywana jest drogą wymiany e-maili bądź telefonicznie. Taki sposób postępowania może powodować, że informacje trafiające do zainteresowanych stron mogą być niepunktualne, niepełne lub nawet błędne.

W rozdziale 3 przedstawiono funkcjonujący dotychczas w Zakładzie model obsługi zlecenia produkcyjnego w postaci opisowej oraz określono jednostki biorące udział przy realizacji zlecenia. Wskazano również obszary wymagające udoskonalenia.

W dalszej części pracy na podstawie informacji uzyskanych z analizy obecnego modelu obsługi zlecenia produkcyjnego zostanie stworzony nowy model zakładający wykorzystanie elektronicznego systemu obsługi zleceń produkcyjnych.

3.1 Jednostki biorące udział w procesie obsługi zlecenia

Jednym z celów analizy obecnego procesu obsługi zlecenia produkcyjnego była identyfikacja jednostek biorących udział w jego realizacji. Wykaz jednostek przedstawiono poniżej. Przyjęto uproszczone nazwy jednostek w celu poprawy czytelności zapisów:

- **Klient** – zlecający usługę,
- **Technolog** – kierownik Zakładu, odpowiedzialny za prawidłowość przebiegu procesu produkcyjnego,
- **Magazynier** – kierownik działu finansowo-handlowego, odpowiedzialny za dostawy materiałów opakowaniowych, fakturowanie i sekcję magazynowania.

3.2 Dotychczasowo funkcjonujący model obsługi zlecenia

Drugim celem przeprowadzonej analizy było sporządzenie dotychczas funkcjonującego w Zakładzie modelu obsługi zlecenia produkcyjnego. Opis sposobu postępowania przy obsłudze zlecenia przedstawiono poniżej:

1. Klient kontaktuje się z Technologiem w celu ustalenia wstępnego terminu wykonania usługi i ceny oraz podaje dane dotyczące planowanej ilości (masa surowca i liczba palet). Podaje również rodzaj surowca do ekstrakcji.
2. Technolog akceptuje lub odrzuca zlecenie (w przypadku odrzucenia proces obsługi jest przerywany).
3. Po zaakceptowaniu zlecenia Technolog przesyła druk zlecenia do klienta drogą e-mailową.
4. Klient wypełnia druk zlecenia i przesyła skan do Technologa.

5. Technolog przekazuje informacje dotyczące zlecenia Magazynierowi.
6. Klient powiadamia Technologa o wysłaniu transportu surowca do przerobu.
7. Technolog powiadamia Magazyniera o spodziewanej dostawie surowca.
8. Po odbiorze surowca Magazynier zawiadamia Technologa i Klienta o przyjętej dostawie.
9. Technolog pobiera surowiec z magazynu zgodnie z zaplanowanym harmonogramem przerobu.
10. Surowiec jest poddawany procesowi technologicznemu w celu wytworzenia produktu końcowego – ekstraktu. Ekstrakt jest konfekcjonowany w opakowania jednostkowe i zbiorcze.
11. Technolog przekazuje produkt końcowy na magazyn. Przekazuje Magazynierowi i Klientowi dane dotyczące masy wykonanego produktu i ilości opakowań zbiorczych (palet).
12. Magazynier wystawia fakturę za wykonanie usługi ekstrakcji. Produkt jest gotowy do odbioru przez klienta.
13. Klient zgłasza się po odbiór produktu.
14. Magazynier wydaje produkt, powiadamia o wydaniu Technologa.
15. Zlecenie jest uznane za zakończone.

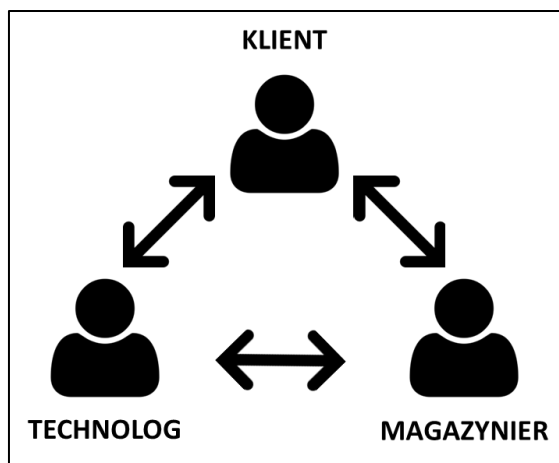
Można łatwo zauważyć, że przedstawiony algorytm postępowania nie jest optymalny. Klient po dostarczeniu surowca do firmy nie wie dokładnie na jakim etapie realizacji jest jego zlecenie (punkty 8 - 11).

Magazynier może nie mieć pełnej wiedzy na temat planowanych do realizacji zleceń, ponieważ Technolog może np. zapomnieć o powiadomieniu go. W związku z tym może niedoszacować niezbędnej w danym czasie ilości materiałów opakowaniowych.

Technolog może nie wiedzieć, którzy klienci wysłali surowce do przerobu i kiedy został odebrany produkt końcowy.

Przykładów obszarów, w których mogą wystąpić różnego rodzaju niezgodności jest oczywiście więcej, ale nawet na podstawie tych kilku przytoczonych przypadków można wyciągnąć wniosek, że model wymaga udoskonalenia.

Funkcjonujący obecnie model procesu obsługi zlecenia działa na zasadzie wymiany informacji „każdy z każdym”. Schemat wymiany informacji między jednostkami biorącymi udział w realizacji zlecenia przedstawiono na rys. 1



rys. 1. Wymiana informacji w dotychczasowym modelu

Wprowadzenie usprawnionego systemu będącego platformą wymiany informacji pomiędzy jego użytkownikami z pewnością wpłynie na poprawę jakości obsługi zleceń przez wyeliminowanie błędów wynikających z chaosu informacyjnego i zwiększy satysfakcję klienta ze współpracy z Zakładem.

4 Przegląd istniejących rozwiązań

Wiedząc, że projektowany system powinien mieć postać aplikacji internetowej wyszukano istniejące na rynku aplikacje oferujące zbliżone funkcjonalności. Na podstawie analizy przedstawionych rozwiązań zostanie sporządzony nowy model obsługi zlecenia oraz projekt aplikacji.

4.1 Atinea – obsługa zleceń

System obsługi zleceń firmy Atinea jest programem przeznaczonym dla firm zajmujących się świadczeniem usług, gdzie występuje duża liczba zleceń, które są realizowane przez wielu pracowników.

Aplikacja działa w oknie przeglądarki WWW, jest wykonana w autorskiej technologii firmy – AtineaGrids, dzięki której możliwe jest modyfikowanie aplikacji i dostosowywanie jej do potrzeb użytkownika. Poniżej zamieszczono przykładowe zrzuty ekranów systemu firmy Atinea (źródło grafik: <https://atinea.pl/index.html?p=73.html>).

The screenshot displays the 'Dodaj' (Add) form in the Atinea application. The interface is in Polish and shows a sidebar with navigation options like 'Zlecenia' (Orders), 'Nowe zlecenie' (New order), 'Moje zlecenia' (My orders), 'Wszystkie zlecenia' (All orders), 'Konfiguracja' (Configuration), and 'Pracownicy' (Employees). The main area is titled 'Zlecenia: 2012-08-17 BTZ s.c.' and contains a form for adding a new order. The form fields are as follows:

id	Data	Klient	Dane klienta	Zlecenie	Wartość	Opis zlecenia	Etap realizacji	Przypisane do	Data zakończenia
5	2012-08-17	BTZ s.c.	ul. Jedności Narodowej 1	fotokalendarze	180,00	wydruk 4 fotokalendarzy z projektu PDF/CMYK odb. osobisty	Fakturowane	Michał Sokołowski	

Below the form is a section for attachments titled 'Załączniki' (Attachments). It contains a table with columns 'Opis' (Description) and 'Plik' (File):

Opis	Plik
oryg. od klienta	Kalendarz projekt 12.08 poprawiony.pdf

The 'Plik' column includes a 'Przeglądaj...' (Browse...) button and a red 'X' icon. At the bottom of the form are buttons for 'Dodaj' (Add), 'Zapisz' (Save), 'Usuń' (Delete), and 'Powiel' (Duplicate).

rys. 2. Atinea - dodawanie nowego zlecenia

Grids Manager - Zlecenia Demo

Użytkownik: ts@atinea.pl Wyloguj

Zlecenia

Pole: Zlecenia Warunek: Wartość: Kolejność: Rosnąco

Zastosuj Filtry Resetuj widok

Zlecenia	id	Data	Klient	Zlecenie	Wartość	Etap realizacji	Przypisane do
2012-08-15 Marimex Sp. z o.o.	2	2012-08-15	Marimex Sp. z o.o.	ulotki A5	950,00	Zakończono	Michał Sokołowski
2012-08-15 Savia Creation Agency	1	2012-08-15	Savia Creation Agency	roll-up 120	779,00	Fakturowane	Michał Sokołowski
2012-08-16 AT Construction	3	2012-08-16	AT Construction	katalogi	335,00	Realizowane	Konrad Wójcik
2012-08-17 BTZ s.c.	5	2012-08-17	BTZ s.c.	fotokalendarze	180,00	Fakturowane	Michał Sokołowski
2012-08-17 NeuPharma Polska SA	4	2012-08-17	NeuPharma Polska SA	projekt graf. strony	3700,00	Realizowane	Maria Bielska
2012-08-20 Agencja Leśna	6	2012-08-20	Agencja Leśna	wizytówki	60,00	Realizowane	Maria Bielska
2012-08-21 Marimex Sp. z o.o.	7	2012-08-21	Marimex Sp. z o.o.	ulotki B6	380,00	Przyjęte	Konrad Wójcik
2012-08-21 TaloBank S.A.	8	2012-08-21	TaloBank S.A.	foldery	2120,00	Przyjęte	Elżbieta Maciejewska
2012-08-22 Joanna Konopka s.c.	9	2012-08-22	Joanna Konopka s.c.	plakaty	240,00	Realizowane	Maria Bielska
2012-08-24 Wójcik i Milewski	10	2012-08-24	Wójcik i Milewski	ulotki B6	543,00	Przyjęte	Elżbieta Maciejewska

Rekordy do wyświetlenia: 200 Rekordy: 1 - 10 (wszystko)

rys. 3. Atinea - lista zleceń

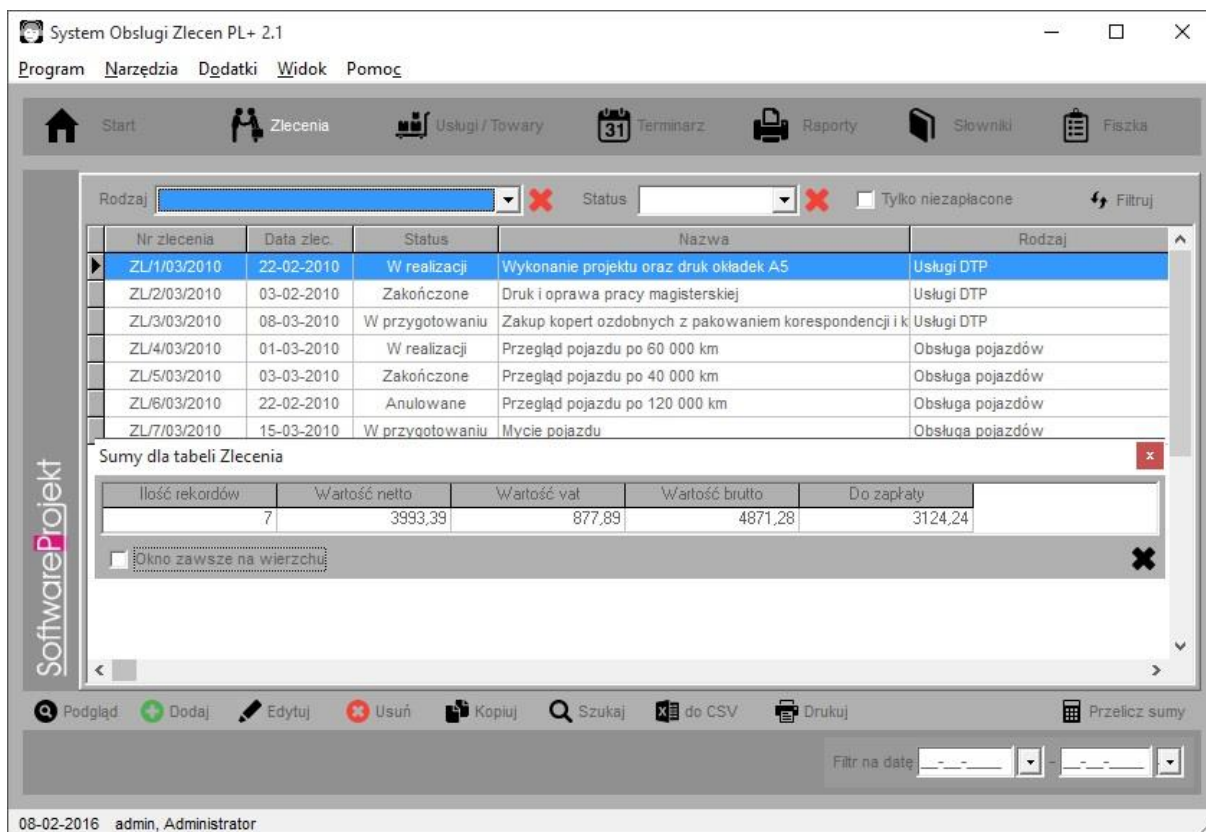
System obsługi zleceń firmy Atinea umożliwia:

- wprowadzanie nowych zleceń,
- załączanie plików do zlecenia na każdym etapie realizacji,
- przekazywanie zlecenia między pracownikami,
- łatwy dostęp do swoich zleceń,
- śledzenie historii zmian w realizacji zlecenia,
- wyszukiwanie zleceń z możliwością filtrowania parametrów,
- zarządzanie uprawnieniami dostępu do zleceń dla poszczególnych grup użytkowników,
- eksport danych do arkuszy kalkulacyjnych.

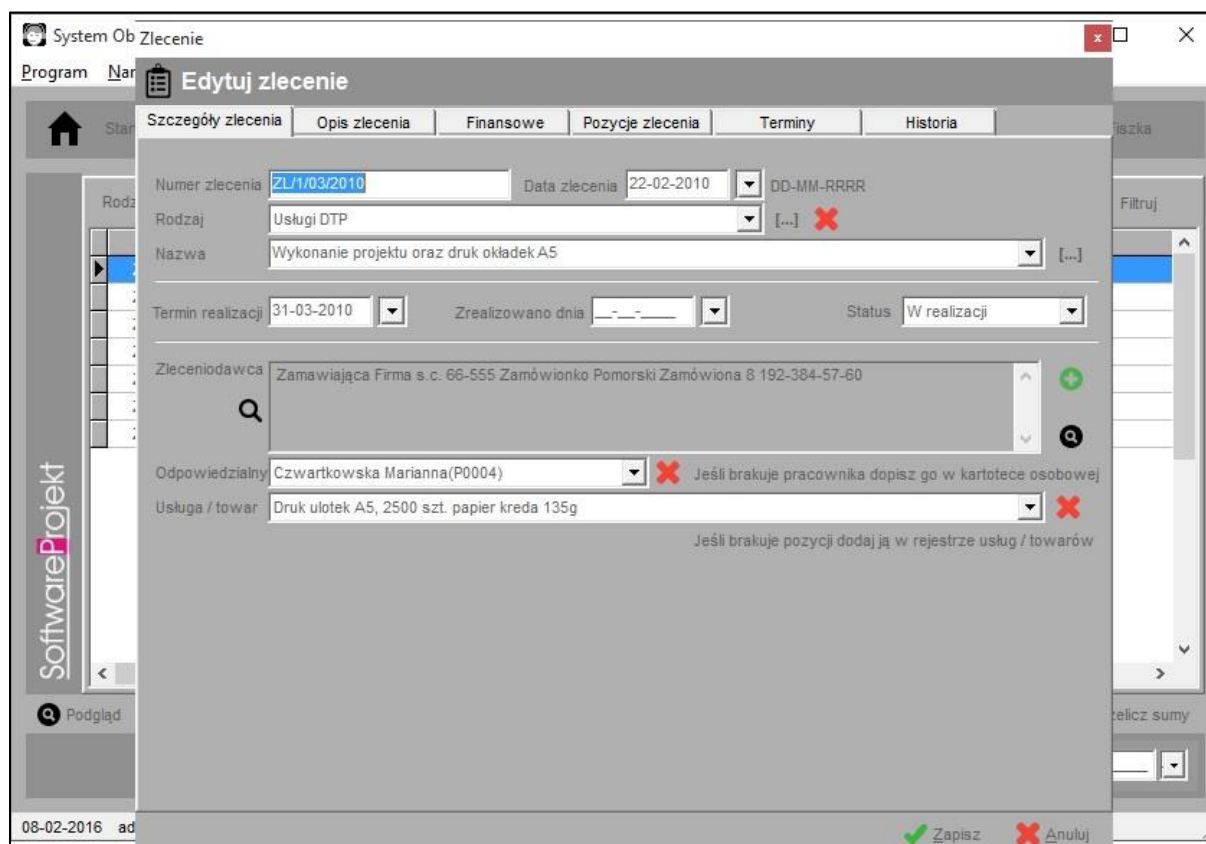
4.2 System obsługi zleceń PL+

System obsługi zleceń PL+, wykonany przez firmę SoftwareProjekt, jest rozwiązaniem przeznaczonym dla firm usługowo-produkcyjnych. Jego głównymi funkcjonalnościami są: kontrola postępu zleceń, prowadzenie historii zleceń i kontrola płatności. System jest aplikacją typu desktop dedykowaną dla systemów z rodziny Microsoft Windows.

Poniżej zamieszczono przykładowe zrzuty ekranów z programu (źródło grafik : http://www.softwareprojekt.com.pl/oprogramowanie/obsługa_zleceń_ekrany):



rys. 4. System obsługi zleceń PL+ - ewidencja zleceń



rys. 5. System obsługi zleceń PL+ - edycja zlecenia

Producent przedstawia swój program jako łatwy w użytkowaniu i dostosowany do obsługi różnego rodzaju działalności gospodarczych – od firm budowlanych po zakłady krawieckie. Na liście referencyjnej klientów można znaleźć takie firmy jak InPost, Centrum Serwisowe Orlen, czy Kancelaria Prezydenta Rzeczypospolitej Polskiej.

4.3 System śledzenia przesyłek InPost

Firma InPost świadcząca usługi doręczania przesyłek została założona w 2006 roku i od początku jej istnienia duży nacisk stawiany był na świadczenie najwyższej jakości usług kurierskich i e-commerce.

Już w 2007 InPost, jako jedna z pierwszych firm świadczących usługi pocztowe w Polsce, uruchomił system śledzenia listów przez Internet. Część systemu dostępna dla zwykłego użytkownika zawiera jedynie pole do wpisania numeru śledzonej przesyłki, przycisk aktywujący śledzenie i wyświetlaną po jego naciśnięciu listę zmian statusu przesyłki.

Zrzut ekranu systemu, obrazujący drogę przykładowej przesyłki przedstawiono na poniższym rysunku (*źródło grafiki: <https://inpost.pl/sledzenie-przesylek?number=664915208024100116692780>*):

The screenshot displays the InPost tracking page for package number 664915208024100116692780. The interface includes a search bar at the top with the package number and a 'Znajdź' (Find) button. Below the search bar, the package number is repeated. The main section shows a vertical timeline of events for the package, starting from 'Przygotowana przez Nadawcę' (Prepared by the sender) on July 16, 2020, at 13:18, and ending with 'Dostarczona.' (Delivered) on July 17, 2020, at 12:17. The events include: 'Odebrana od klienta' (Received from customer), 'Przyjęta w oddziale InPost' (Accepted at InPost branch), 'W trasie' (In transit), 'Przyjęta w oddziale InPost' (Accepted at InPost branch), 'Przekazano do doręczenia' (Passed to delivery), 'Umieszczona w Paczkomacie (odbiorczym)' (Placed in the mailbox (recipient)), and 'Dostarczona.' (Delivered). A yellow smiley face icon is next to the 'Dostarczona.' status. At the bottom, there is a section titled 'Odbiór paczek nigdy nie był tak prosty' (Package pickup has never been so easy) with a QR code and links to download the InPost app from Google Play, the App Store, and the AppGallery. To the right, there is a 'Pomoc' (Help) section with links to 'Jak nadać paczkę w Paczkomacie za pomocą Managera Paczek' (How to send a package in the mailbox using the Package Manager), 'Jak odebrać przesyłkę w Paczkomacie?' (How to receive a package in the mailbox?), and 'Jak przedłużyć termin odbioru paczki w Paczkomacie?' (How to extend the pickup deadline in the mailbox?).

Strona główna • Odbieram • Śledzenie paczki
Śledź paczkę

664915208024100116692780 **Znajdź**

Np. 873234987612340872938732 (24 znak)

Przesyłka nr: 664915208024100116692780

17 Lip 2020, 12:17 **Dostarczona.**
Podróż przesyłki od Nadawcy do Odbiorcy zakończyła się, ale nie musi to oznaczać końca naszej znajomości! Jeśli lubisz InPost, odwiedź nasz fanpage na Facebooku. Dziękujemy!

17 Lip 2020, 9:28 Umieszczona w Paczkomacie (odbiorczym).

17 Lip 2020, 6:54 Przekazano do doręczenia.

17 Lip 2020, 5:38 Przyjęta w oddziale InPost.

16 Lip 2020, 17:59 W trasie.

16 Lip 2020, 17:36 Przyjęta w oddziale InPost.

16 Lip 2020, 16:30 Odebrana od klienta.

16 Lip 2020, 13:18 Przygotowana przez Nadawcę.

Odbiór paczek nigdy nie był tak prosty
Odbieraj paczki szybciej niż kiedykolwiek, dzięki nowej funkcji zdalnego otwarcia skrytki! Pobierz aplikację i oszczędzaj czas!

Pomoc
Jak nadać paczkę w Paczkomacie za pomocą Managera Paczek -
Jak odebrać przesyłkę w Paczkomacie? -
Jak przedłużyć termin odbioru paczki w Paczkomacie? -

rys. 6. InPost - śledzenie przesyłki

Liczba wszystkich możliwych stanów przesyłki wynosi 18. Wykaz stanów przesyłki wraz z opisem i ścieżkami przejścia między stanami jest dostępny w postaci grafiki pod adresem: https://inpost.pl/sites/default/files/inline-images/droga%20paczki%20do%20paczkomatu_0.png

Specyfika systemu śledzenia przesyłek InPost znacząco odbiega od tej którą posiada typowy system obsługi zleceń, ważnym punktem wspólnym i powodem, dla którego autor zamieścił w pracy opis systemu InPost jest rozwiązanie dotyczące statusu paczki – jednocześnie proste i dające użytkownikowi odpowiednie informacje. W podobny sposób zostanie zaprojektowany proces realizacji zlecenia w budowanym na potrzeby pracy systemie.

5 Nowy model obsługi zleceń produkcyjnych

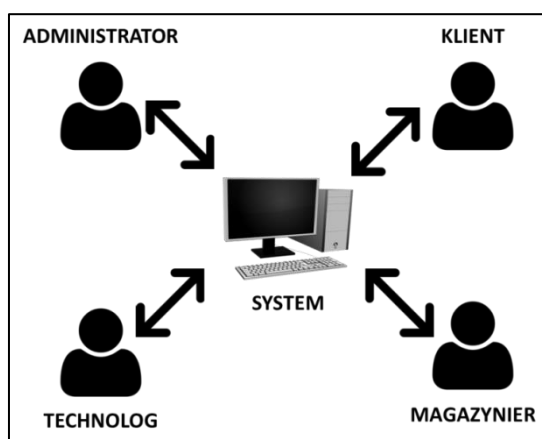
W rozdziale 5, na podstawie informacji uzyskanych z analizy dotychczas funkcjonującego modelu obsługi zlecenia produkcyjnego, przygotowano nowy model z przeznaczeniem wykorzystania go w elektronicznym systemie obsługi zleceń produkcyjnych.

Przy opracowywaniu nowego modelu przyjęto dwa główne założenia:

1. Centralizacja. Wymiana informacji powinna się odbywać między użytkownikami a systemem, a nie tak jak do tej pory na zasadzie „każdy z każdym”.
2. Prostota. Nowy model obsługi zlecenia produkcyjnego powinien być maksymalnie prosty, z zapewnieniem pełnej funkcjonalności.

Spełnienie pierwszego warunku zapewni łatwość implementacji modelu w systemie elektronicznym – centralizacja przenosi się na możliwość przechowywania informacji dotyczących zleceń w bazie danych. Przestrzeganie drugiego założenia powinno przełożyć się na późniejszą łatwość w programowaniu aplikacji, co jest szczególnie ważne biorąc pod uwagę fakt, że system powinien oferować możliwość dalszej rozbudowy.

Schemat wymiany informacji w nowym modelu obsługi zlecenia przedstawiono na rys. 7:



rys. 7. Wymiana informacji w nowym modelu

Wprowadzenie komunikacji w przedstawionej formie powinno po pierwsze zapobiec powstawaniu chaosu informacyjnego, a po drugie zminimalizować ryzyko zafałszowania lub utraty informacji.

5.1 Jednostki biorące udział w procesie obsługi zlecenia w nowym modelu

Zgodnie z nowym modelem w procesie obsługi zlecenia produkcyjnego będą brały udział te jednostki - aktorzy, które zostały wskazane podczas analizy modelu dotychczasowego. Dodatkowo zostali wprowadzeni nowi aktorzy – Administrator i Użytkownik niezalogowany. Wykaz jednostek – aktorów systemu przedstawiono poniżej:

- **Klient** – zlecający usługę
- **Technolog** – kierownik Zakładu, odpowiedzialny za prawidłowość przebiegu procesu produkcyjnego
- **Magazynier** – kierownik działu finansowo-handlowego, odpowiedzialny za dostawy materiałów opakowaniowych, fakturowanie i sekcję magazynowania
- **Administrator** – do jego zadań będzie należeć zarządzanie użytkownikami i zleceniami
- **Użytkownik niezalogowany** – może zalogować się w systemie, lub zarejestrować jako nowy Klient.

5.2 Nowy przebieg obsługi zlecenia

Nowy przebieg obsługi zlecenia produkcyjnego został opracowany w celu zidentyfikowania zadań poszczególnych aktorów systemu oraz wyspecyfikowania stanów, w jakich może znajdować się zlecenie. Opracowany model ułatwi dalsze prace związane z projektowaniem i wdrożeniem systemu.

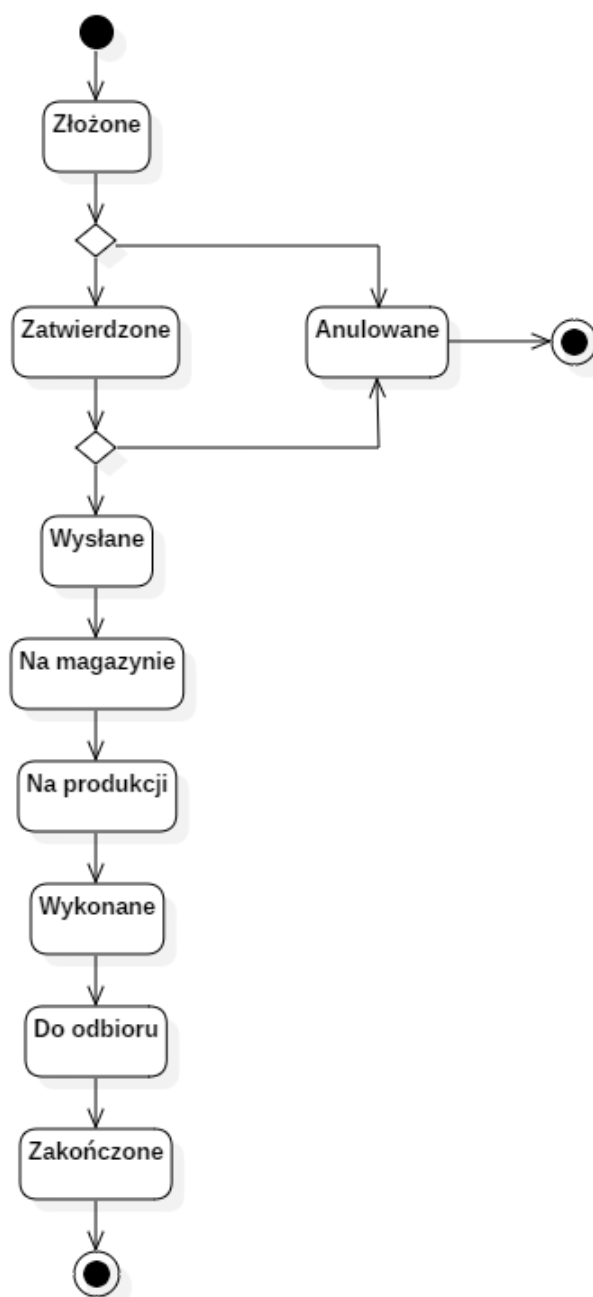
Zaplanowano przebieg procesu obsługi zlecenia produkcyjnego w następujący sposób:

1. Klient wprowadza dane nowego zlecenia w systemie (nazwa i masa surowca, ilość palet z surowcem).
2. Technolog zatwierdza zlecenie do realizacji bądź odrzuca je.
3. Klient wysyła surowiec do przerobu lub rezygnuje z wykonania zlecenia.
4. Magazynier przyjmuje surowiec na stan magazynu.
5. Technolog pobiera surowiec na produkcję zgodnie z zaplanowanym harmonogramem przerobu.
6. Po zakończeniu przerobu i uzyskaniu końcowego produktu, Technolog wprowadza dane dotyczące masy wykonanego produktu i ilości opakowań zbiorczych (palet) dla danego zlecenia.
7. Magazynier odbiera wykonany produkt z produkcji, wprowadza na stan magazynu i wystawia fakturę za wykonaną usługę.
8. Klient odbiera produkt.
9. Zlecenie jest uznane za zakończone.

Porównując nowy model procesu obsługi zlecenia produkcyjnego z modelem funkcjonującym do tej pory da się zauważyć, że nowy model jest znacznie prostszy dzięki modyfikacji sposobu wymiany informacji. Realizacja zlecenia od jego złożenia do zakończenia składa się z dziewięciu kroków, a nie jak do tej pory – piętnastu, przy czym żadna informacja dotycząca zlecenia, zawarta w starym modelu nie jest tracona.

5.3 Diagram stanów zlecenia

Bazując na przedstawionym w punkcie 5.2 nowym modelu przebiegu procesu obsługi zlecenia produkcyjnego sporządzono diagram stanów dla zlecenia. Diagram przedstawiono na rys. 8:



rys. 8. Diagram stanów zlecenia

Jak można zauważyć – droga zlecenia od jego złożenia do zakończenia jest prosta i składa się na nią niewielka liczba etapów pośrednich. Zostało więc spełnione główne założenie projektowe dla nowego modelu.

5.4 Przejścia między stanami

Po sporządzeniu diagramu stanów zlecenia kolejnym krokiem jest określenie, na bazie przedstawionego w punkcie 5.2 modelu, jakie przejścia między stanami będą realizowane przez poszczególnych aktorów systemu. Wykaz przedstawiono w postaci poniższej tabeli:

Stan przed	Stan po	Zmienia Aktor	Uwagi
brak	Złożone	Klient	Wprowadza wymagane dane surowca do zlecenia
Złożone	Zatwierdzone	Technolog	brak
Złożone	Anulowane	Technolog, Klient	brak
Zatwierdzone	Anulowane	Klient	brak
Zatwierdzone	Wysłane	Klient	brak
Wysłane	Na magazynie	Magazynier	brak
Na magazynie	Na produkcji	Technolog	brak
Na produkcji	Wykonane	Technolog	Wprowadza wymagane dane produktu do zlecenia
Wykonane	Do odbioru	Magazynier	brak
Do odbioru	Zakończone	Magazynier	brak

Tabela 1. Przejścia między stanami zlecenia

Przejścia między stanami zlecenia zostały jasno określone. Jedyny przypadek, gdzie dany stan może zmienić dwóch aktorów, to anulowanie zlecenia złożonego (może to zrobić Klient lub Technolog).

Dodatkowo aktor: Administrator powinien mieć możliwość zmiany stanu zlecenia na dowolnie wybrany.

5.5 Podsumowanie rozdziału

W rozdziale 5 przedstawiono nowy model obsługi zlecenia produkcyjnego od strony koncepcyjnej. Został określony diagram stanów dla zlecenia oraz podano przyporządkowanie aktorów do poszczególnych zmian stanu. Informacje te ułatwią specyfikację elektronicznego systemu obsługi zlecenia.

6 Zastosowane technologie

W rozdziale 6 opisano technologie użyte do wykonania elektronicznego systemu obsługi zleceń produkcyjnych w postaci aplikacji internetowej.

6.1 Aplikacja internetowa (Web App)

Aplikacja internetowa, zwana również aplikacją webową, to program komputerowy uruchomiony na serwerze i komunikujący się z użytkownikiem – klientem przez sieć Internet przy użyciu przeglądarki internetowej (Wawak, 2020). Aplikacje internetowe są programowane z wykorzystaniem architektury klient-serwer, gdzie serwer zapewnia usługi dla klientów zgłaszających do niego żądania obsługi. Jako przykłady aplikacji internetowych można podać między innymi różnego rodzaju portale społecznościowe, sklepy internetowe, systemy płatności elektronicznych, itd.

6.1.1 Jak powstały aplikacje internetowe?

Serwisy internetowe ewoluowały wraz z rozwojem sieci Internet. Ciągły przyrost ilości użytkowników Sieci, zwiększanie jej przepustowości oraz konkurencja między podmiotami świadczącymi usługi w Internecie, wpłynęły na opracowywanie coraz to doskonalszych i nowocześniejszych rozwiązań w celu zwiększenia atrakcyjności usług dla potencjalnych użytkowników.

Można wyróżnić następujące etapy rozwoju serwisów internetowych prowadzące do powstania aplikacji internetowych (Miłosz, 2008):

1. Statyczne strony WWW – są to proste witryny służące jedynie do prezentacji treści. Umożliwiają użytkownikom przeglądanie prostych treści tekstowych i multimedialnych (grafika, dźwięk, video) oraz nawigację między stronami przy użyciu hiperłączy. Użytkownik miał ponadto możliwość wysyłania danych do serwera przy użyciu formularzy.
2. Strony z wbudowaną funkcjonalnością, to kolejny etap ewolucji serwisów internetowych. Wraz z rozwojem przeglądarek internetowych i wprowadzeniem możliwości interpretacji języków programowania, do prezentacji treści została dodana możliwość wykonywania programów – skryptów po stronie klienta. Powstała możliwość tworzenia i wykorzystywania dynamicznych obiektów interfejsu użytkownika, np. list rozwijanych, układów menu, okien zawierających multimedia i innych elementów.
3. Aplikacje internetowe – oferują funkcjonalność aplikacji desktopowych (instalowanych na komputerze klienta), ale ich środowiskiem uruchomieniowym jest przeglądarka internetowa. Aplikacje takie przechowują i przetwarzają dane na serwerach w sieci Internet i dostarczają użytkownikowi treści w postaci statycznych i dynamicznych stron WWW (tzn. takich, w których zawartość może się zmieniać interaktywnie)

6.1.2 Funkcjonalność aplikacji internetowych.

W aplikacjach internetowych użytkownik, do tej pory występujący jedynie jako konsument treści, otrzymał możliwość generowania treści. W związku z tym najczęściej oferowane przez aplikacje webowe funkcjonalności to (Miłosz, 2008):

- manipulowanie danymi przez użytkowników (operacje CRUD),

- przechowywanie danych w sposób trwały,
- przetwarzanie i prezentacja danych,
- zarządzanie użytkownikami i ich rolami.

Można zauważyć, że aplikacje internetowe mają funkcjonalność aplikacji desktopowych – oprócz przedstawienia informacji, wymieniają dane z użytkownikiem oraz je przetwarzają, ale są od nich bardziej uniwersalne. Po pierwsze – nie ma konieczności ich instalacji na komputerze klienta – wystarczy działająca przeglądarka internetowa. Po drugie – każdy użytkownik korzysta zawsze z najbardziej aktualnej na dany czas wersji aplikacji (w przypadku aplikacji desktopowych to użytkownik decyduje czy chce bądź nie chce instalować aktualizacje). Aplikacje webowe cechują się uniwersalnością i elastycznością – mogą działać pod każdym systemem operacyjnym i na każdym urządzeniu obsługującym przeglądarkę internetową.

Wydawać by się mogło, że Aplikacje webowe to twór idealny, w praktyce okazuje się jednak, że posiadają one również znaczące wady. Przede wszystkim, aby móc z nich korzystać, należy mieć dostęp do Internetu, co nie zawsze jest możliwe. Ponadto użytkownik nie może dowolnie wybrać wersji aplikacji z której korzysta – musi używać takiej, jaką w danym czasie dostarcza producent. Kolejną wadą jest brak kontroli dostawcy aplikacji nad platformą, na której aplikacja jest uruchamiana. Ponieważ użytkownik może uruchomić aplikację na dowolnym urządzeniu wyposażonym w przeglądarkę internetową i dostęp do Internetu, dostawca aplikacji powinien zapewnić możliwość poprawnego działania aplikacji dla każdej konfiguracji sprzętowej.

Paradoksalnie - biorąc pod uwagę zarówno aplikacje webowe i desktopowe - te cechy, które z pewnego punktu widzenia są zaletami poszczególnych typów aplikacji, w innym przypadku mogą stanowić ich wady (aktualność, elastyczność). Czy zatem, podejmując w trakcie projektowania aplikacji decyzję o tym, czy ma to być aplikacja webowa, czy desktopowa - wybrać wersję webową? Zdaniem autora pracy, głównym czynnikiem decydującym jest w tym przypadku sposób wykorzystania aplikacji przez przyszłych użytkowników. Jeżeli użytkownik potrzebuje dostępu do funkcjonalności aplikacji na różnych urządzeniach, niezależnie od miejsca, w którym się znajduje (np. e-mail, portale społecznościowe, bankowość elektroniczna) – warto zdecydować się na aplikację internetową. Jeżeli natomiast przewidujemy, że użytkownik będzie korzystał z aplikacji w sposób stacjonarny, można wybrać aplikację desktopową (np. edytor tekstu, odtwarzacz plików multimedialnych).

6.1.3 Zastosowanie aplikacji internetowych

Aplikacje internetowe można podzielić w zależności od ich zastosowania (Zastosowanie aplikacji internetowych (webowych), 2020). Wspólną cechą wszystkich przypadków jest, co warto po raz kolejny zaakcentować, zorientowanie na użytkownika, jako jednocześnie konsumenta i dostawcy treści. Ta rewolucyjna zmiana w sposobie użytkowania sieci Internet została nazwana mianem Web 2.0.

W zależności od zastosowania aplikacji webowych można wyszczególnić:

1. Aplikacje internetowe dla firm
Systemy tej kategorii mają za zadanie usprawnienie realizowanych w firmie zadań, a także wymiany informacji na linii firma – klient. Są to wszelkiego rodzaju aplikacje finansowo-księgowe, systemy CRM, systemy ewidencji stanów magazynowych i portale pracownicze.
2. Aplikacje dla rynku e-handlu

Do tej kategorii zaliczają się aplikacje internetowe, realizujące funkcjonalność sklepów internetowych. Systemy e-sprzedaży, mają za zadanie usprawnienie obsługi procesu sprzedaży po stronie firmy, a dla klienta stanowią platformę służącą do atrakcyjnej prezentacji oferty handlowej, z możliwością wyboru i zakupu towarów, zazwyczaj oferując również możliwość płatności elektronicznych.

3. Aplikacje edukacyjne i informacyjne

Elektroniczne systemy nauczania są obecne w Internecie od długiego czasu. Platformy edukacyjne oferują funkcje interaktywnego nauczania, udostępniania multimedialnych materiałów szkoleniowych i sprawdzania wiedzy uczniów. Popularne platformy tego typu to chociażby Udacity czy Udemy. Część instytucji oświatowych, tradycyjnie realizujących nauczanie w trybie stacjonarnym, oferuje również możliwość nauki na odległość przy wykorzystaniu dedykowanej platformy e-learningowej. Jako przykład można wymienić platformę EDUX, która jest wykorzystywana w systemie kształcenia na odległość. W ostatnich czasach, po zamknięciu jednostek oświatowych w wielu krajach z powodu zagrożenia pandemią COVID-19, aplikacje edukacyjne okazały się szczególnie przydatne i umożliwiły płynne przeniesienie nauki ze szkół do domów zapobiegając paraliżowi systemu edukacji.

4. Aplikacje internetowe dla sektora usług

Są to systemy wspomagające działanie działalności usługowej. Można w tej kategorii umieścić wszelkiego typu systemy rezerwacji miejsc noclegowych w hotelach i ośrodkach, systemy rezerwacji biletów oraz inne systemy wspomagające realizację tego typu działalności. Często systemy tego typu zawierają podsystem sprzedaży, dzięki któremu klient może dokonać płatności elektronicznej za wykonaną usługę. Przykłady takich systemów to m.in. system e-monitoringu przesyłek Poczty Polskiej, system śledzenia przesyłek Inpost, system rezerwacji biletów e-bilet.pl i inne.

5. Portale społecznościowe, informacyjne i aplikacje rozrywkowe

Wspomniana wcześniej rewolucja Web 2.0 ugruntowała pozycję Internetu, jako głównego medium służącemu czerpaniu i wymianie informacji przez użytkowników. Portale informacyjne, oprócz dostarczania informacji, stały się platformą do komentowania wydarzeń przez użytkowników. Tematyczne fora internetowe są bazą wiedzy tworzoną przez zgromadzoną wokół nich społeczność. Portale społecznościowe i rozrywkowe, np. najpopularniejszy obecnie Facebook, przyczyniły się do nowego zjawiska jakim jest przeniesienie życia społecznego do Sieci.

6.1.4 Wyzwania dla dostawców aplikacji internetowych

Przed twórcami aplikacji internetowych stoi wiele wyzwań zarówno od strony technicznej jak i tych związanych ze stroną biznesową aplikacji. Parafrazując słowa G. McGoverna dotyczące systemów zarządzania treścią – twórcy aplikacji internetowych również muszą zadbać o to, aby „dostarczyć właściwej treści właściwej osobie we właściwym czasie i za właściwą cenę”. Kolejną ważną zasadą, którą warto się kierować przy tworzeniu aplikacji webowych jest zasada 4P, która zakłada, że informacje dostarczane przez aplikację powinny być (Miłosz, 2008):

1. **Prawdziwe**

Użytkownicy są w dużej mierze dostarczycielami treści – to od nich w dużej mierze zależy czy ten warunek zostanie spełniony.

2. **Personalizowane**

Przekazywane informacje powinny być dopasowane do potrzeb danego użytkownika lub informacje przeznaczone dla konkretnego użytkownika nie powinny być dostępne

dla innych (np. czytelnik powinien mieć wgląd jedynie do swojej karty bibliotecznej w aplikacji obsługującej bibliotekę).

3. **Przejrzyste**

Informacje powinny być czytelne i dostępne niezależnie od stopnia zaawansowania technicznego i umiejętności użytkownika. Aplikacja musi być projektowana tak, aby każdy użytkownik mógł z niej w łatwy sposób skorzystać.

4. **Punktualne**

Informacje muszą być dostępne dla użytkownika i poprawne w chwili, gdy ten je otrzymuje (np. w aplikacji obsługującej kino należy zadbać o to, aby nie dopuścić do sytuacji w której dwóch użytkowników mogłoby zarezerwować to samo miejsce na seansie filmowym).

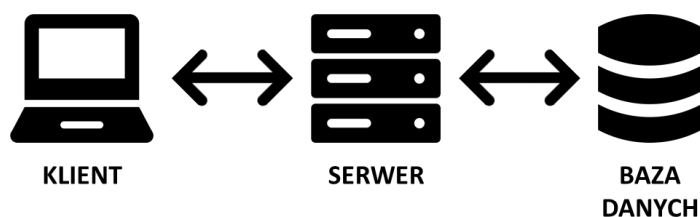
Jedną z najważniejszych kwestii od strony technicznej jest standaryzacja aplikacji internetowych. Twórcy aplikacji powinni zadbać o to, aby działała ona na różnorodnych konfiguracjach sprzętowo-programowych wykorzystywanych przez użytkowników.

Równie ważna jest kwestia bezpieczeństwa aplikacji internetowych. Ponieważ tego typu aplikacje, przez swoją obecność w Sieci, są powszechnie dostępne dla wszystkich, są one również podatne na różnego rodzaju ataki mające na celu sparaliżowanie pracy aplikacji, pozyskanie danych lub wykorzystanie aplikacji w nieprzewidziany sposób (np. umożliwienie zwykłemu użytkownikowi systemu VOD dostępu do treści zarezerwowanych wyłącznie dla abonentów).

Przed twórcami aplikacji webowych stoi również wyzwanie zapewnienia jej użyteczności (Usability). Interfejs użytkownika powinien być czytelny, prosty w obsłudze i dawać użytkownikowi satysfakcję z korzystania z aplikacji. Użytkownik mający do wyboru dwie aplikacje dostarczające takie same treści wybierze tę, z której przyjemniej się korzysta.

6.1.5 Budowa aplikacji internetowych

Budowa aplikacji internetowych oparta jest na architekturze warstwowej (Krawczyk, 2014; Sienkiewicz i Syty, 2008). W przeglądarce internetowej na komputerze użytkownika uruchamiane jest jedynie oprogramowanie interfejsu użytkownika, a oprogramowanie właściwej aplikacji oraz realizacja dostępu do danych leży po stronie serwera. Typowy schemat budowy aplikacji webowe, przedstawiony na rys. 9 zakłada istnienie trzech warstw funkcjonalnych – warstwy klienta, serwera i danych.



rys. 9. Budowa aplikacji internetowej

- warstwa klienta – odpowiedzialna za prezentację interfejsu, interakcję z użytkownikiem i komunikację z warstwą aplikacji. Może być zrealizowana w postaci tzw. „cienkiego” i „grubego” klienta,
- warstwa aplikacji – w zależności od przesyłanych z warstwy klienta żądań pobiera odpowiednie dane z warstwy danych, tworzy dokumenty dynamiczne i przekazuje do warstwy klienta,
- warstwa danych – służy przechowywaniu i udostępnianiu danych w komunikacji z warstwą aplikacji. Dane są przechowywane na serwerze bazy danych.

Dzięki zastosowaniu tego typu architektury, każda z jej składowych może być rozwijana niezależnie. Oczywistym faktem jest również to, że każda warstwa może być zlokalizowana na innym komputerze – komputerze klienta z przeglądarką internetową, serwerze aplikacji i serwerze baz danych. W architekturze trójwarstwowej wyraźnie rozdzielone są funkcje wizualizacji danych, operacji na danych i przechowywania danych, stąd aplikacje internetowe budowane są według wzorca Model-Widok-Kontroler (MVC)

„Cienki” i „Gruby” klient

W architekturze wielowarstwowej można wyróżnić dwa sposoby realizacji warstwy klienta. Może to być tak zwany „cienki” i „gruby” klient (Nowicki i Wrzosek, 2010).

„Cienki” klient zajmuje się jedynie prezentacją danych otrzymywanych z warstwy aplikacji i wysyłaniem uzyskanych od użytkownika danych w drugą stronę. Cała logika biznesowa aplikacji zlokalizowana jest po stronie serwera.

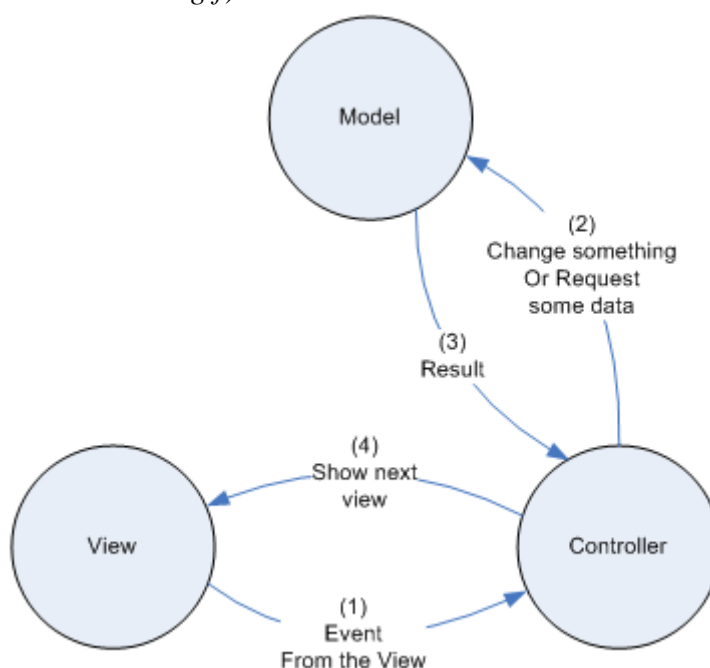
„Gruby” klient oprócz funkcji, które są charakterystyczne dla „cienkiego” klienta realizuje również (przynajmniej częściowo) funkcje logiki biznesowej aplikacji po stronie klienta.

Wzorzec projektowy MVC

Wzorzec projektowy to uniwersalne, abstrakcyjne (czyli bez skonkretyzowanej implementacji) rozwiązanie pewnego powtarzającego się problemu projektowego (Socha, 2012). Wzorzec Model – Widok – Kontroler, w skrócie MVC, jest jednym z częściej stosowanych wzorców projektowych w informatyce. Jego głównym założeniem jest rozdzielenie architektury aplikacji na 3 moduły (Zemczak, 2018):

- model – reprezentujący strukturę danych
- widok – odpowiada za prezentację danych
- kontroler – wykonuje operacje na danych

Schematycznie można przedstawić wzorzec MVC w postaci diagramu (*źródło grafiki: <https://i.stack.imgur.com/7RXh4.gif>*):



rys. 10. Diagram MVC

Najważniejszą zaletą wzorca MVC jest izolacja modułów od siebie i logiczny „podział obowiązków” między model widok i kontroler. Ponadto oddzielenie widoku od modelu pozawala na łatwe prezentowanie tych samych danych na różne sposoby. Główną wadą wzorca MVC jest brak elastyczności przy znaczących zmianach dokonywanych w modelu (Schalau, 2014). Zmiany w modelu powodują konieczność modyfikacji w kontrolerze i widoku. Z tego powodu, jeżeli przy budowie aplikacji ma być wykorzystywany ten wzorzec, należy starannie zaprojektować model danych.

Model

Model to warstwa która reprezentuje logikę biznesową aplikacji. W tej warstwie znajdują się obiekty, które służą do wykonywania operacji związanych z implementacją funkcjonalności aplikacji.

Widok

Widok to warstwa prezentacji danych, czyli to, co widzi użytkownik. Odpowiedzialny jest za prezentację wizualną użytkownikowi wybranych danych, które są wynikiem działań logiki biznesowej (czyli Modelu).

Kontroler

Kontroler to warstwa, która obsługuje żądania użytkownika. Żądania przekierowywane są do odpowiednich metod w Modelu, które z kolei przekażą odpowiednie dane do użytkownika – czyli do Widoku.

6.1.6 Podsumowanie

W rozdziale 6.1 przedstawiono definicję aplikacji internetowych, omówiono historię ich powstania. Następnie wyszczególniono najważniejsze obszary zastosowania aplikacji webowych, wskazano ich wady i zalety oraz opisano ich budowę i wymieniono najważniejsze pojęcia związane z ich architekturą.

6.2 Technologie używane w projektowaniu aplikacji internetowych

Projektant tworzący aplikację internetową ma do wyboru mnogość technologii, z których może w tym celu skorzystać. Jeżeli rozpatrujemy aplikację w architekturze trójwarstwowej to, poczynając od warstwy danych, istnieje na rynku wiele dostępnych rozwiązań. W zależności od złożoności projektu i typu przechowywanych danych, można wybrać relacyjne lub nierelacyjne bazy danych. Istnieją również gotowe rozwiązania dedykowane przechowywaniu dużej ilości plików. Również przy tworzeniu pozostałych warstw aplikacji przed projektantem stoi wybór jednej z wielu dostępnych ram programistycznych (Frameworków) (Nowak, 2019). W związku z powyższym w tym rozdziale autor pracy ograniczył się do przedstawienia podstawowych technologii i pojęć wiążących się z budową i działaniem aplikacji internetowych. Zostaną również tutaj opisane technologie wybrane do budowy tworzonej w ramach niniejszej pracy aplikacji.

6.2.1 Frontend i Backend

Te dwa pojęcia nierozzerwalnie łączą się z projektowaniem aplikacji internetowych (Stewart, 2020).

Technologie frontendowe to te, które odpowiedzialne są za wygląd strony internetowej, są na niej widzialne, działają w przeglądarce użytkownika. Do takich technologii zaliczamy m.in. HTML, CSS, JavaScript.

Drugi rodzaj technologii webowych – backendowe – działają na serwerze i służą do przetwarzania danych. Ich działanie jest dla użytkownika niewidzialne, widoczne są jedynie wprowadzone dane lub akcje wykonane na stronie oraz wynik działania backendu. Cały proces wykonawczy zachodzi więc poza przeglądarką.

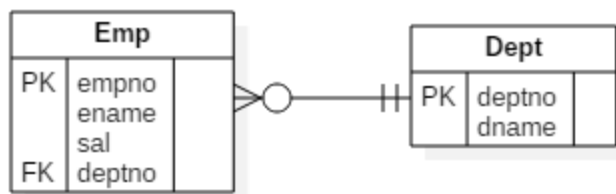
6.2.2 MySQL

MySQL to otwarty system zarządzania relacyjną bazą danych, który powstał w 1995 roku w Szwecji. Obecnie jest rozwijany i wspierany przez firmę Oracle. Jego nazwa powstała z połączenia imienia córki jednego z twórców – „My” i skrótu nazwy strukturalnego języka zapytań używanego do zarządzania danymi w relacyjnych bazach danych – SQL. (Vavatech - MySQL, 2020)

Główne zalety MySQL to:

1. **Uniwersalność** - jest dostępny jest dla wszystkich popularnych platform systemowych i różnorodnych architektur procesorów. Dostępny jest również w wersji źródłowej, co umożliwia kompilację dla w zasadzie dowolnej platformy.
2. **Skalowalność** - MySQL jest bardzo wydajny, dobrze radzi sobie z ogromnymi ilościami zapytań.
3. **Ochrona danych** – MySQL oferuje mechanizmy ochrony danych, takie jak: autentykacja użytkownika, wsparcie dla protokołów SSH i SSL, kontrola dostępu użytkowników i inne.

Szczegółowe definiowanie czym jest relacyjna baza danych wykracza poza ramy niniejszej pracy, warto natomiast zapamiętać, że w bazie relacyjnej dane przechowywane są w tabelach, które połączone są między sobą związkami logicznymi – relacjami. Przykład ilustrujący budowę prostej relacyjnej bazy danych przedstawiono na rysunku rys. 11.



rys. 11. Schemat prostej bazy danych z jedną relacją

6.2.3 JavaScript

JavaScript jest skryptowym językiem programowania, opracowanym w 1995 przez Netscape Corporation i Sun Microsystems z zamiarem wykorzystywania go przy tworzeniu stron WWW (Malik i Przewodnik, 2020). JavaScript jest językiem łatwym do opanowania, obsługiwanym przez wszystkie popularne przeglądarki internetowe. Skrypty napisane za pomocą JavaScript mogą być umieszczane bezpośrednio na stronach HTML bądź wczytywane z określonej lokalizacji. Język ten, używany we „frontendzie”, daje obszerne możliwości wzbogacania stron WWW o dodatkowe elementy. Obecnie, głównie dzięki środowiskom takim jak Node.js (którego opis przedstawiono dalej w tym rozdziale), jego znaczenie rośnie również przy wykorzystaniu w części backendowej.

6.2.4 Sequelize

Sequelize to maper obiektowo-relacyjny (ORM), czyli narzędzie zapewniające łatwy dostęp do baz danych, w tym wypadku opartych na SQL, poprzez odwzorowanie tabel i relacji między nimi na powiązane ze sobą obiekty. Typowy ORM zawiera interfejs programistyczny

do przeprowadzania operacji CRUD (tworzenie, odczytywanie, aktualizacja i usuwanie danych w bazie). Zastosowanie ORM wprowadza dodatkową warstwę abstrakcji nad bazą danych. Dzięki niemu można wykonywać operacje na danych w bazie traktując je jak obiekty w programowaniu obiektowym (Hoyos, 2018).

Sequelize jest biblioteką napisaną w języku JavaScript, opartą na mechanizmie tzw. „obietnic”, przeznaczoną dla środowiska Node.js (pojęcia te zostaną wyjaśnione w dalszej części tekstu). Sequelize oferuje wsparcie dla różnych dialektów języka SQL, w tym MySQL (Tyler, 2018).

Do tworzenia odwzorowań obiektowo-relacyjnych Sequelize wykorzystuje modele. Przykładowe definicje modeli, odwzorowujące tabele z rys. 11, przedstawiono na poniższych listingach:

```
sequelize.define("Emp", {
  empno: {
    type: Sequelize.INTEGER,
    allowNull: false,
    primaryKey: true,
    autoIncrement: true,
    unique: true
  },
  ename: {
    type: Sequelize.STRING(100),
    allowNull: false
  },
  sal: {
    type: Sequelize.DECIMAL(10, 2),
    allowNull: false
  }
})
```

Listing 1. Emp - model Sequelize

```
sequelize.define("Dept", {
  deptno: {
    type: Sequelize.INTEGER,
    allowNull: false,
    primaryKey: true,
    autoIncrement: true,
    unique: true
  },
  dname: {
    type: Sequelize.STRING(100),
    allowNull: false
  },
})
```

Listing 2. Dept - model Sequelize

Powiązania między obiektami dodaje się za pomocą wchodzących w skład Sequelize metod, w tym przypadku: `Dept.hasMany(emp)` i `Emp.belongsTo(Dept)`.

Mając tak przygotowany model można w prosty sposób wykonywać operacje CRUD na danych w bazie, np. odczyt:

```
Emp.findAll({
  where: {
    ename: "Kowalski",
  }
});
```

Dzięki modelom Sequelize w prosty sposób można zmienić dane w bazie przypisując nowe wartości pobranym obiektom, a następnie zapisując je w bazie przy użyciu metody `save()`.

6.2.5 Node.js

Node.js jest platformą typu open-source, która pozwala na uruchomienie kodu JavaScript poza przeglądarką internetową (vavatech.pl - Node.js, 2020). Dodatkowo, platforma oferuje wiele modułów stanowiących potężny zestaw narzędzi programistycznych.

Node.js jest jednym z niewielu rozwiązań serwerowych opartych na koncepcji programowania sterowanego zdarzeniami, która pozwala tworzyć wysoce skalowane serwery bez udziału wątków tak, jak ma to miejsce w tradycyjnych rozwiązaniach wielowątkowych wykorzystując tzw. pętlę zdarzeń.

Ponadto Node.js obsługuje wiele jednoczesnych równoległych żądań i operacji za pośrednictwem wywołania synchronicznych i nieblokujących operacji wejścia/wyjścia. Podstawą działania Node'a jest silnik V8, który został stworzony przez firmę Google. Dzięki niemu Node.js w szybki sposób jest w stanie wykonać kod JavaScript.

Node.js istnieje już od 2009 roku. Na samym początku był stworzony wyłącznie pod system Linux. Od 2011 Microsoft wspomógł platformę i pomógł stworzyć wersję dla Windows. Pomimo, że jest to projekt open-source, to czuwa nad nim fundacja Node.js będąca też członkiem fundacji Linux Foundation. Wraz ze swoimi członkami ma ona za zadanie promowanie i przyspieszenie rozwoju projektu i technologii Node.js.

Przed wprowadzeniem Node.js JavaScript używany był wyłącznie po stronie frontendowej, a przy programowaniu backendu wykorzystywane były inne języki programowania. Do popularności Node.js bez wątpienia przyczynił się fakt, że dzięki niemu można z użyciem JavaScriptu tworzyć zarówno części klienta i serwera aplikacji internetowej, przez co proces tworzenia jest łatwiejszy i bardziej efektywny.

6.2.6 NPM

NPM (Node Package Manager) jest managerem pakietów dla środowiska Node.js (Gutkowski, 2014). Spełnia dwie role:

- jest szeroko stosowanym repozytorium open-source'owych narzędzi dla Node.js, które każdy developer może dodawać i korzystać z nich,
- pełni funkcję narzędzia linii komend odpowiedzialnego za instalowanie i usuwanie pakietów, zarządzanie ich wersjami i zarządzanie zależnościami w projekcie.

W momencie pisania tej pracy liczba dostępnych w NPM pakietów wynosi ponad 1 350 000 (*źródło: <http://www.modulecounts.com/>*) i stale rośnie. Jest to największe i najszybciej rozwijające się narzędzie tego typu.

6.2.7 Express

Express to open-source'owa rama programistyczna dla Node.js zaprojektowana z myślą o ułatwieniu tworzenia aplikacji internetowych i API (ExpressJS - Overview, 2020):

- daje możliwość zdefiniowania routingu dla aplikacji w zależności od użytej metody żądania, HTTP i adresu URL
- ułatwia tworzenie backendu opartego na REST API,
- pozwala na wprowadzenie tzw. middleware w obsłudze żądań HTTP.

6.2.8 REST API

REST (Representational State Transfer) API jest to styl architektury oprogramowania zaprezentowany w 2000 roku przez Roya Fieldinga (Czech, 2017). Definiowany jest przez 6 reguł:

1. Klient-serwer

Przez rozdzielenie interfejsu użytkownika od aplikacji serwerowej uzyskujemy możliwość niezależnego rozwijania obu części i przenośność UI na różnorodne platformy.

2. Bezstanowość

Każde żądanie klienta musi zawierać wszystkie informacje niezbędne do jego obsłużenia. Na serwerze nie są przechowywane dane o stanie klienta.

3. Cache

Dane odpowiedzi na żądanie powinny być oznaczone jako buforowalne (i przez jaki czas będą aktualne) bądź niebuforowalne. W pierwszym przypadku klient może użyć pobranych danych ponownie bez konieczności powtórzenia połączenia z serwerem, o ile dane nie są jeszcze „przeterminowane”. Wprowadzenie tej zasady pozwala na znaczne zredukowanie ilości połączeń z serwerem, a co za tym idzie zmniejszenie wykorzystania zasobów.

4. Jednolity interfejs

Dzięki użyciu do komunikacji między klientem a serwerem jednolitego interfejsu możliwe jest uproszczenie integracji i architektury projektowanych rozwiązań. Interfejs taki cechują cztery zasady:

- interfejs jest oparty na zasobach – każdy zasób jest dostępny pod unikalnym adresem URI,
- realizacja manipulacji zasobami przez ich reprezentacje – klient nie wykonuje bezpośrednio operacji na zasobach. Otrzymuje od serwera ich reprezentacje i to na nich wykonuje działania i przesyła te reprezentacje do serwera,
- samoopisujące komunikaty – przesyłane komunikaty muszą zawierać wszystkie niezbędne informacje tak, aby odbiorca komunikatu był w stanie go zrozumieć,
- HATEOAS (Hypermedia as the Engine of Application State) - umożliwienie interfejsowi API na dostarczanie klientowi informacji na temat potencjalnych akcji jakie ten może wykonać.

5. Separacja warstw

Architektura powinna mieć postać warstwową, dana warstwa powinna mieć dostęp jedynie do warstw sąsiednich.

6. Code on demand (opcjonalnie)

Możliwość rozszerzenia funkcjonalności po stronie klienta przez pobieranie i wykonywanie kodu w postaci apletów i skryptów.

6.2.9 React.js

React.js jest JavaScriptową biblioteką służącą do tworzenia interaktywnych interfejsów użytkownika dla aplikacji internetowych i mobilnych (Morris, 2020). Została stworzona w 2013 na potrzeby Facebooka i obecnie jest najpopularniejszą biblioteką tego typu (Prajapati, 2020).

Cechy, które wyróżniają tę bibliotekę na tle konkurencji, to:

- łatwe tworzenie aplikacji dynamicznych – React ułatwia tworzenie dynamicznych aplikacji internetowych przy zastosowaniu mniejszej ilości kodu i oferuje większą funkcjonalność niż czysty JavaScript,
- wysoka wydajność – React wykorzystuje wirtualny DOM (model obiektowy dokumentu). W wirtualnym DOM porównuje się poprzedni stan komponentu i uaktualnia w faktycznym DOM jedynie te elementy, które zostały zmienione, w przeciwieństwie do uaktualniania wszystkich przy zmianie przynajmniej jednego jak ma to miejsce w konwencjonalnych aplikacjach webowych,
- komponenty wielokrotnego użytku – komponenty są budulcem aplikacji stworzonych z wykorzystaniem React. Komponenty zawierają logikę i elementy sterujące, mogą być użyte wielokrotnie w aplikacji, co znacząco skraca czas jej wytwarzania,
- jednokierunkowy przepływ danych – przy tworzeniu aplikacji z użyciem React często korzysta się z zagnieżdżania komponentów. Ponieważ przepływ danych występuje w kierunku rodzic – dziecko wyszukiwanie i debugowanie ewentualnych błędów jest znacznie ułatwione,
- React jest łatwy do przyswojenia – do opanowania tej biblioteki wystarczy znajomość podstawowych koncepty HTML i JavaScript,
- jest wyposażony w dedykowane narzędzia do łatwego wykrywania błędów – Facebook opublikował służące do tego celu rozszerzenie do przeglądarki Chrome,
- wykorzystuje JSX – jest to rozszerzenie składni języka JavaScript o elementy znane z XML/HTML, dzięki któremu można w prosty sposób tworzyć interfejsy użytkownika. Dzięki użyciu JSX można łączyć struktury HTML i JavaScript,
- jest elastyczny – można zwiększyć możliwości React.js przez zastosowanie dodatkowych rozszerzeń.

6.2.10 SOP i CORS

SOP (Same-Origin Policy) jest mechanizmem zabezpieczającym, zaimplementowanym w przeglądarkach internetowych (Niezabitowski, 2018). Mechanizm ten określa, że informacje mogą być wymieniane jedynie między stronami mającymi to samo pochodzenie (Origin). Przez to samo pochodzenie rozumie się ten sam protokół, domenę i port.

Mechanizm SOP został wprowadzony w celu obrony przed atakami typu CSRF (Cross-Site Request Forgery). Ataki tego typu polegają na skłonieniu zalogowanego w serwisie użytkownika do uruchomienia spreparowanego odnośnika, którego otwarcie wywoła w danym serwisie akcję, do której atakujący nie miałby w przeciwnym razie dostępu.

CORS (Cross-Origin Resource Sharing) jest to mechanizm umożliwiający wykonywanie asynchronicznych żądań HTTP do zasobów znajdujących się w innym położeniu (Origin). W praktyce realizowany jest on przez dodanie do odpowiedzi serwera nagłówka HTTP informującego o zezwoleniu na dostęp do zasobów przez klientów o określonym pochodzeniu.

6.2.11 React-router

React-router jest biblioteką dla React.js, która dodaje funkcjonalność obsługi routingu po stronie klienta (Maki, 2017). Dzięki jego zastosowaniu nawigacja w aplikacji jest bardziej intuicyjna, przez zróżnicowanie adresów URL tak jak w klasycznej stronie internetowej. Ponadto React-router daje użytkownikowi możliwość nawigacji w aplikacji przez wpisanie URL w pasku adresu przeglądarki, a także wykorzystania do nawigacji przycisków „wstecz” i „dalej” w przeglądarce.

6.2.12 HTTP

HTTP (HyperText Transfer Protocol) jest protokołem służącym do wymiany informacji w sieci WWW na zasadzie żądanie klienta - odpowiedź serwera (Mansweld, 2020). HTTP jest protokołem bezstanowym, tzn. nie przechowuje informacji o poprzednich żądaniach klienta. Ma to na celu zmniejszenie obciążenia serwera. W przypadku potrzeby zapamiętania stanu wykorzystuje się różne techniki, np. mechanizm ciasteczek (ang. Cookies) lub parametry w adresach URL.

HTTP jest protokołem warstwy aplikacji zgodnie z warstwowym modelem sieci OSI. Komunikacja w protokole HTTP odbywa się z użyciem protokołu TCP/IP, który gwarantuje niezawodność przesyłania.

Żądanie HTTP zawiera następujące elementy:

- metoda żądania – określa jaka operacja będzie wykonana na udostępnionym przez serwer zasobie,
- nagłówek – zawiera dodatkowe informacje o żądaniu,
- ścieżka dostępu do danego zasobu,
- ciało żądania – zawiera dane wysyłane na serwer.

Wyróżnia się 9 metod żądania HTTP. Do najczęściej wykorzystywanych należą:

- GET – pobranie zasobu z serwera. Żądanie GET nie zawiera ciała,
- POST – modyfikacja stanu zasobu na serwerze,
- PUT – umieszczenie zasobu na serwerze,
- DELETE – usunięcie zasobu z serwera.

Odpowiedź na żądanie zawiera:

- oznaczenie wersji protokołu HTTP,
- kod statusu odpowiedzi HTTP,
- wiadomość dla danego statusu,
- nagłówki odpowiedzi zawierające dodatkowe informacje,
- ciało odpowiedzi (opcjonalnie), w którym zawarty jest zasób pobrany z serwera.

Status odpowiedzi HTTP to trzycyfrowy kod informujący o sposobie realizacji zapytania, w którym pierwsza cyfra statusu oznacza grupę zaszeregowania:

1xx – kody informacyjne,

2xx – kody powodzenia,

- 3xx – kody przekierowania,
- 4xx – kody błędu klienta,
- 5xx – kody błędu serwera.

Wśród często spotykanych statusów można wymienić (HTTP status codes and their meaning, 2019):

- 200 – żądanie wykonane z sukcesem. Zasób został odnaleziony na serwerze i dostarczony do klienta,
- 301 – zasób trwale przeniesiony, dostępny pod nowym adresem,
- 302 – zasób przeniesiony tymczasowo,
- 403 – dostęp zabroniony – żądanie nie może zostać wykonane ze względu na brak dostępu klienta do zasobu,
- 404 – szukany zasób nie został odnaleziony,
- 500 – wewnętrzny błąd serwera,
- 503 – usługa niedostępna – serwer jest przeciążony i nie może zrealizować zapytania klienta.

6.2.13 HTML

HTML (HyperText Markup Language) jest hipertekstowym językiem znaczników służącym do prezentacji treści na stronach internetowych (Domantas, 2019). Opracował go Tim Berners-Lee w 1991 roku, a obecnie aktualna wersja oznaczona jest numerem 5 i jej specyfikacja została opublikowana jako rekomendacja W3C w 2014 roku.

W HTML używa się wspomnianych wcześniej znaczników w celu nadania znaczenia semantycznego poszczególnym fragmentom surowego tekstu. Dzięki temu przeglądarka internetowa wyświetlając zawartość wczytanego dokumentu HTML interpretuje napotkane w dokumencie znaczniki i odpowiednio formatuje wyświetlany tekst. Znacznik może określać np. paragraf, czy akapit tekstu, tabelę (wiersze, kolumny), listę numerowaną i wypunktowaną i inne. Istnieją również znaczniki dzięki którym można zagnieżdżać w tekście multimedia, a także wprowadzające możliwość interakcji z użytkownikiem przy użyciu formularzy. Szczegółowy opis znaczników dostępny jest w specyfikacji języka. Hipertekst oznacza, że wyświetlany dokument zawiera wyróżnione wyrażenia, które są odsyłaczami (linkami) do innych dokumentów. Przykładowy dokument HTML przedstawiono na Listing 3:

```
<!DOCTYPE html>
<html>
  <body>
    <p>To jest paragraf tekstu</p>
  </body>
</html>
```

Listing 3. Prosty dokument HTML

6.2.14 CSS

CSS (Cascading Style Sheets), czyli kaskadowe arkusze stylów, to język służący do opisu formy prezentowania dokumentów HTML. Tak jak HTML opisuje semantykę tekstu (czyli np. za pomocą znaczników można skategoryzować tekst jako listę wypunktowaną), tak przy użyciu CSS określa się sposób formatowania rzeczony listy (rodzaj punktora, wcięcia, położenie względem innych elementów, kolor czcionki i inne właściwości). Specyfikacja pierwszej wersji CSS została opublikowana w 1996 roku. Jej wprowadzenie miało na celu

odseparowanie warstwy danych (czyli tego co chcemy wyświetlić) od warstwy prezentacji (w jaki sposób ma to być wyświetlone). Dzięki temu można np. raz zdefiniowanego stylu użyć przy wyświetlaniu różnych danych, zwiększa się również czytelność dokumentów HTML. Przykładowy styl CSS, ustalający kolor wyświetlanego tekstu w paragrafie z Listing 34 przedstawiono na listingu poniżej:

```
p {  
  color: red;  
}
```

Listing 4. Przykładowy styl CSS

6.2.15 CSS Frameworks – Skeleton CSS

Frameworki CSS to gotowe biblioteki stylów CSS, ułatwiające prace nad formą prezentacji plików HTML. Do najpopularniejszych tego typu bibliotek należą Bootstrap i Materialize (Lazaris, 2019). Przy wykonywaniu aplikacji będącej przedmiotem niniejszej pracy wykorzystano lekki, bo liczący zaledwie 400 linii tekstu framework Skeleton.CSS. Skeleton zawiera gotowe definicje często używanych elementów – tekstu, list, tabel, formularzy, przycisków i nagłówków, a także definiuje układ elementów w postaci dwunastokolumnowej siatki (*źródło: getskeleton.com*).

6.2.16 JSON

JSON (JavaScript Object Notation) jest lekkim, tekstowym formatem wymiany danych zdefiniowanym w 2001 roku (Mansfeld, 2020). JSON powstał w oparciu o dwie struktury danych: zbiór par nazwa-wartość, oraz uporządkowana lista wartości. Notacja JSON jest zbieżna z notacją obiektów w JavaScript. Prostota tego formatu zapisu danych, a także to, że zapisane w postaci tekstu w JSON dane są czytelne dla ludzi sprawiają, że jest on chętnie wykorzystywany.

6.2.17 Fetch API

Fetch API jest interfejsem języka JavaScript, który umożliwia asynchroniczne wykonywanie żądań HTTP z wykorzystaniem tzw. obietnic (promises) (Wrzesień, 2018). Obietnica jest obiektem, który zwraca wynik jakiejś operacji, która może zakończyć się w dowolnym czasie. Obietnica może znajdować się w jednym z trzech stanów:

- spełniona/rozwiązana (fulfilled/resolved) – gdy powiązane z nią zadanie zwraca żadaną wartość,
- odrzucona (rejected) – gdy zadanie nie zwraca wartości, np. z powodu wyjątku lub zwrócona wartość jest nieprawidłowa,
- oczekująca (pending) – to stan od rozpoczęcia żądania do otrzymania wyniku.

6.2.18 JWT

JWT (JSON Web Token) jest otwartym standardem, który umożliwia bezpieczną wymianę informacji między dwiema stronami z użyciem JSON (Sajdak, 2018). Wiarygodność przekazywanych informacji jest gwarantowana, ponieważ są one podpisane cyfrowo. Mogą być podpisane z użyciem klucza tajnego (algorytm HMAC) lub jawnie przy użyciu klucza publicznego/prywatnego RSA lub ECDSA.

JWT jest wykorzystywany w dwóch sytuacjach: przy autoryzacji dostępu (po zalogowaniu do systemu użytkownik uzyskuje token, który będzie przysyłał z kolejnymi żadaniami HTTP) oraz w celu bezpiecznej wymiany informacji.

JWT składa się z trzech elementów:

1. Nagłówek (Header) – są w nim dwie części: rodzaj tokenu (JWT) i rodzaj algorytmu użytego do podpisu tokenu.
2. Zawartość (Payload) – tutaj umieszczane są dane do przesłania oraz metadane (najczęściej czas wydania tokenu, wydawca, czas wygaśnięcia ważności tokenu)
3. Sygnatura (Signature) – utworzona na podstawie zawartości Header i Payload z użyciem jednej z wymienionych wcześniej metod kryptograficznych. Sygnatura jest używana do sprawdzenia czy dane w Zawartości JWT nie zostały zmienione oraz do potwierdzenia tożsamości dostawcy.

JWT ma postać potrójnego ciągu znaków zakodowanego Base64, oddzielonego kropkami, który można w łatwy sposób przesyłać przez HTTP. Jest szeroko rozpowszechnionym standardem wykorzystywanym do autoryzacji.

6.2.19 JWT-decode

JWT-decode jest niewielką biblioteką służącą do dekodowania - odczytywania zawartości tokenu JWT, bez jego walidacji (*źródło: <https://www.npmjs.com/package/jwt-decode>*).

6.2.20 Bcrypt

Bcrypt jest funkcją skrótu kryptograficznego, która została stworzona głównie w celu hashowania haseł statycznych (Rodwald i Biernacik, 2018). Należy do grupy funkcji iteracyjnych z tzw. „solą” (dodany do hasła krótki ciąg znaków). Jej wewnętrzna struktura została oparta na algorytmie Blowfish. Bcrypt jest używany np. do szyfrowania haseł użytkowników przechowywanych w bazie danych. Dzięki temu nawet w przypadku wycieku danych z bazy włamywacz nie będzie miał wiedzy na temat haseł użytkowników. W aplikacji powstałej na potrzeby tej pracy została wykorzystana gotowa biblioteka bcrypt środowiska Node.js.

7 Koncepcja i budowa systemu

W rozdziale 7 przedstawiono koncepcję budowy systemu obsługi zleceń produkcyjnych oraz opisano specyfikację wymagań jakie powinien spełnić projektowany system aby zapewnić realizację procesu obsługi zlecenia według podanego w rozdziale 5 nowego modelu.

7.1 Budowa systemu

System powinien być zbudowany w architekturze trójwarstwowej, podzielonej na część frontendową i backendową.

7.1.1 Frontend

Część frontendowa aplikacji zostanie wykonana z wykorzystaniem React.js. Powinna być jednocześnie możliwie prosta i funkcjonalna. Wymagane jest zapewnienie określonych w dalszej części rozdziału funkcjonalności dla poszczególnych grup użytkowników oraz uniemożliwianie nieupoważnionym użytkownikom dostępu do funkcjonalności dla nich nieprzeznaczonych. Komunikacja z częścią backendową zostanie wykonana przy użyciu interfejsu Fetch API.

7.1.2 Backend

Część backendowa aplikacji będzie składać się z dwóch elementów:

1. Bazy danych MySQL, w której przechowywane będą dane dotyczące użytkowników i zleceń. Ponieważ w bazie będą przechowywane również hasła użytkowników, powinny być one w odpowiedni sposób zabezpieczone na wypadek wycieku danych z bazy. W tym celu zostanie zastosowane szyfrowanie haseł z użyciem bcrypt.
2. Interfejs REST API, zajmujący się obsługą żądań przychodzących z części frontendowej z zapewnieniem obsługi żądań CORS. Ważnym wymaganiem jest zapewnienie kontroli dostępu do udostępnianych zasobów (danych z bazy) przez autentykację i autoryzację użytkowników wysyłających żądania. Postulat ten zostanie spełniony przez zastosowanie technologii JSON Web Token.

7.2 Przechowywane dane

Kolejnym ważnym krokiem w budowie systemu jest określenie jakie dane będą przechowywane w bazie. Przyjęto założenie, że ilość przechowywanych danych powinna być jak najmniejsza, ale taka, aby zapewnić wszystkie wymagane informacje o użytkownikach systemu i zleceniach. System powinien więc przechowywać dane dotyczące:

1. Użytkowników:

- ID użytkownika,
- Login użytkownika,
- Hasło użytkownika (w postaci zaszyfrowanej),

- Rola użytkownika,
- Imię użytkownika,
- Nazwisko użytkownika,
- Nazwa firmy użytkownika,
- NIP firmy użytkownika,
- Stan użytkownika (aktywny / zablokowany).

2. Zlecen:

- ID zlecenia,
- ID zlecającego,
- Nazwa surowca,
- Data złożenia zlecenia,
- Masa surowca,
- Liczba palet z surowcem,
- Masa uzyskanego produktu,
- Liczba palet uzyskanego produktu,
- Status zlecenia.

3. Historii zmian statusów zlecenia:

- ID historii,
- ID zlecenia,
- ID statusu,
- ID zmieniającego.

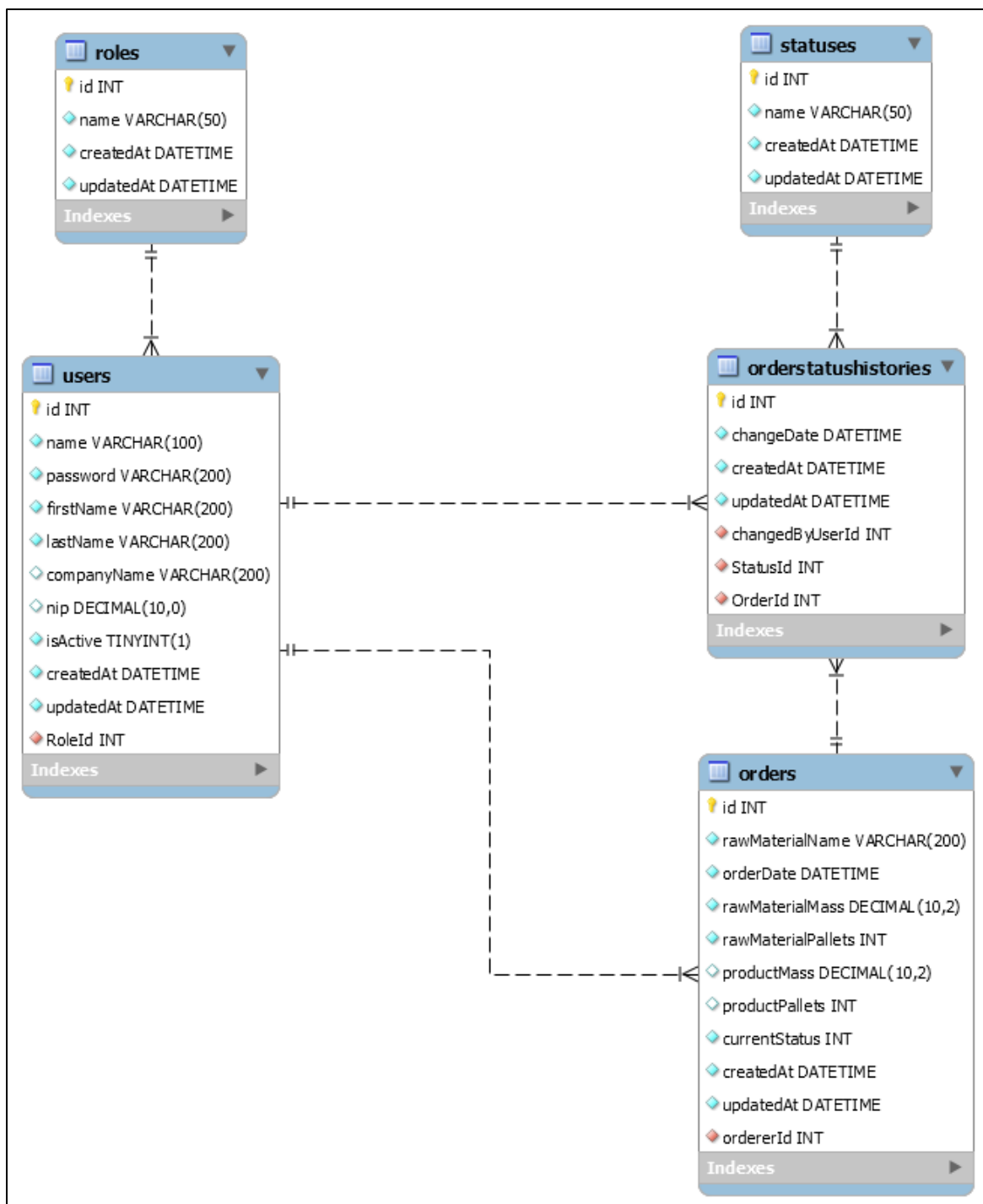
4. Statusów zlecenia, zgodnie z rys. 8:

- ID statusu,
- Nazwa statusu.

5. Ról użytkowników, zgodnie z rozdziałem 5.1:

- ID roli,
- Nazwa roli.

Schemat bazy danych systemu obsługi zleceń, przechowującej podane wyżej informacje przedstawiono na rys. 12. Dodatkowo każda tabela ma dodane kolumny `createdAt` i `updatedAt`, wygenerowane automatycznie przez ORM Sequelize.



rys. 12. Schemat bazy danych Systemu Obsługi Zleceń

7.3 Wymagania dla użytkowników

Po określeniu jakie dane mają być w systemie przechowywane pozostało jeszcze wskazać jakie funkcjonalności projektowany system powinien zapewniać. W rozdziale 7.3 opisano funkcjonalności z podziałem na poszczególne role użytkowników.

1. Użytkownik niezalogowany

- Rejestracja w systemie jako klient,

- Logowanie do systemu.

2. Klient

- Podgląd wszystkich zleceń danego klienta z możliwością filtrowania wyświetlania względem statusu,
- Podgląd historii zmiany statusów danego zlecenia,
- Zmiana statusu zlecenia zgodnie z Tabela 1,
- Podgląd danych swojego konta,
- Dodawanie nowych zleceń,
- Wylogowanie z systemu.

3. Technolog / Magazynier

- Podgląd wszystkich zleceń klientów z możliwością filtrowania wyświetlania względem statusu,
- Podgląd historii zmiany statusów danego zlecenia,
- Zmiana statusu zlecenia zgodnie z Tabela 1,
- Podgląd danych swojego konta,
- Podgląd listy klientów,
- Wylogowanie z systemu.

4. Administrator

- Podgląd wszystkich zleceń klientów z możliwością filtrowania wyświetlania względem statusu,
- Podgląd historii zmiany statusów danego zlecenia,
- Zmiana statusu zlecenia na dowolny,
- Podgląd danych swojego konta,
- Podgląd listy klientów,
- Blokowanie / odblokowanie kont klientów,
- Wylogowanie z systemu.

W pierwszej wersji Systemu zakładane jest istnienie pojedynczych kont Technologa, Magazyniera i Administratora, należy jednak zapewnić możliwość dodania większej ilości takich kont w kolejnych wersjach systemu, jeżeli zajdzie taka potrzeba.

Mając już do dyspozycji wszystkie niezbędne dane można było przystąpić do wykonania aplikacji.

8 Implementacja

W rozdziale 8 przedstawiono sposób implementacji Systemu Obsługi Zleceń Produkcyjnych, z podziałem na część frontendową i backendową.

8.1 Frontend

Część frontendowa składa się z komponentów React.js i stanowi warstwę prezentacji. Użytkownik ma możliwość interakcji z aplikacją dzięki użyciu interaktywnych elementów HTML (formularzy, przycisków, odnośników, list wyboru). Wykonanie akcji przez użytkownika skutkuje wysłaniem odpowiedniego żądania HTTP do części backendowej i reakcji frontendu na otrzymaną odpowiedź.

8.1.1 Proste komponenty

W części frontendowej aplikacji jest kilka komponentów, których jedyną rolą jest prezentacja statycznych treści. Są to:

1. Komponent HeaderPanel – wyświetla treść nagłówka aplikacji

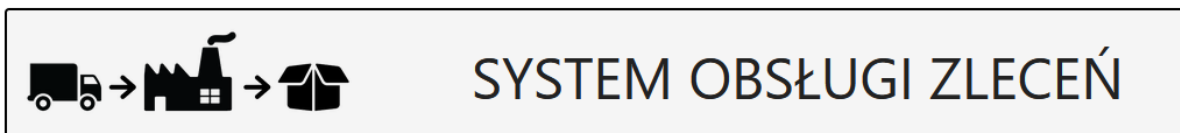
```
import React from 'react';
import logo from './logo.png';
import './HeaderPanel.css';

function HeaderPanel(props) {
  return (
    <div className="top-bar row">
      <div className="four columns u-max-full-width logoimg">
        <img src={logo} alt="Logo" height="75"/>
      </div>
      <div className="eight columns titlediv">
        System Obsługi Zleceń
      </div>
    </div>
  )
}

export default HeaderPanel
```

Listing 5. Komponent HeaderPanel

Komponent HeaderPanel wyświetlony w przeglądarce przedstawiono na poniższym rysunku:



rys. 13. Widok komponentu HeaderPanel

2. Komponent FooterPanel – wyświetla treść stopki aplikacji

```
import React from 'react';
import './FooterPanel.css';

function FooterPanel(props) {
  return (
    <div className="footer row">
      <div className="twelve columns">
        PJATK 2020 - Projekt i wykonanie: Rafał Reszczyński
      </div>
    </div>
  )
}

export default FooterPanel
```

Listing 6. Komponent FooterPanel

A tak wygląda efekt wyświetlenia komponentu FooterPanel w przeglądarce:

PJATK 2020 - Projekt i wykonanie: Rafał Reszczyński

rys. 14. Widok komponentu FooterPanel

3. Komponent NotFound – wyświetla się po wprowadzeniu nieprawidłowego URL

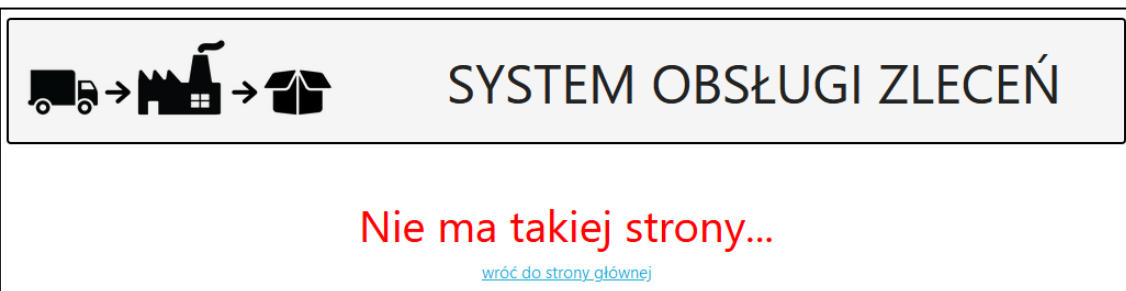
```
import React from 'react';
import { Link } from 'react-router-dom';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import './NotFound.css'

function notFound(props) {
  return (
    <div>
      <HeaderPanel />
      <div className="nf-message">Nie ma takiej strony...</div>
      <div className="nf-link">
        <Link to={'/'}>wróć do strony głównej</Link>
      </div>
    </div>
  )
}

export default notFound
```

Listing 7. Komponent NotFound

Na poniżej przedstawionym obrazie komponentu NotFound można zauważyć, że jedną z jego części składowych jest osadzony komponent HeaderPanel:



rys. 15. Widok komponentu NotFound

4. Komponenty Dashboard i osadzony w nim MainPanel – strona powitalna

```
import React, { Component } from 'react';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import FooterPanel from '../FooterPanel/FooterPanel.js';
import MenuPanel from '../MenuPanel/MenuPanel.js';
import MainPanel from '../MainPanel/MainPanel.js';

class Dashboard extends Component {
  render() {
    return (
      <div>
        <HeaderPanel />
        <MenuPanel />
        <MainPanel />
        <FooterPanel />
      </div>
    )
  }
}

export default Dashboard
```

Listing 8. Komponent Dashboard

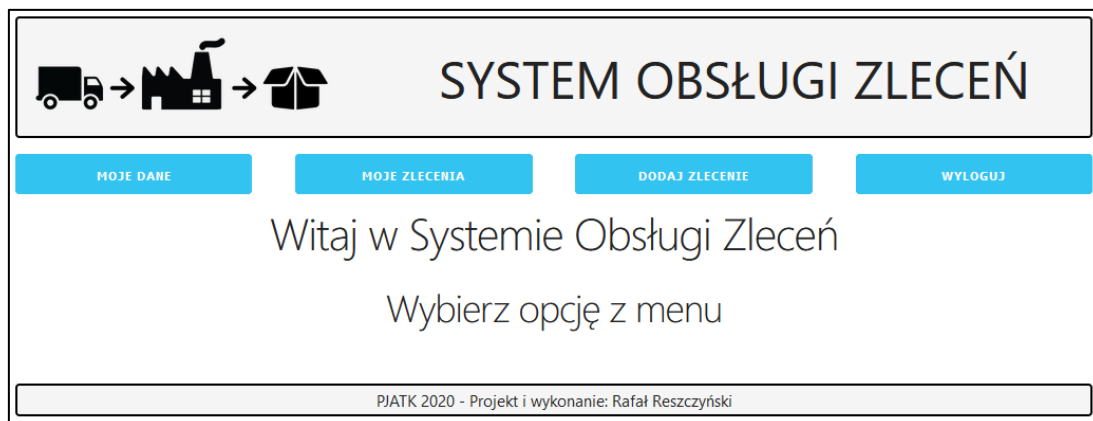
```
import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import './MainPanel.css';

class MainPanel extends Component {
  render() {
    return (
      <div className="main-container row">
        <h2 className="mp-info">Witaj w Systemie Obsługi Zleceń</ h2>
        <h3 className="mp-info">Wybierz opcję z menu</ h3>
      </div>
    )
  }
}

export default withRouter(MainPanel)
```

Listing 9. Komponent MainPanel

Widok komponentu Dashboard przedstawiono na rysunku poniżej. Osadzony w nim komponent MenuPanel zawiera menu użytkownika z opcjami wyświetlanymi w zależności od jego roli.



rys. 16. Widok komponentu Dashboard

8.1.2 Plik *index.js* i komponent `ProtectedRoute`

W pliku *index.js* znajduje się definicja routingu aplikacji klienta. Komponent `ProtectedRoute` blokuje dostęp do danej ścieżki dla niezalogowanych użytkowników. Jak można zauważyć użytkownik niezalogowany będzie miał dostęp wyłącznie do logowania i rejestracji:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import { Route, BrowserRouter as Router, Switch } from 'react-router-dom'

import NotFound from './NotFound/NotFound.js'
import Login from './Login/Login.js'
import Dashboard from './Dashboard/Dashboard.js'
import AboutMe from './AboutMe/AboutMe.js'
import AddUser from './AddUser/AddUser.js'
import AddOrder from './AddOrder/AddOrder.js'
import OrdersList from './OrdersList/OrdersList.js'
import ClientsList from './ClientsList/ClientsList.js'
import OrderHistory from './OrderHistory/OrderHistory.js'
import ProtectedRoute from './ProtectedRoute/ProtectedRoute.js'

const routing = (
  <Router>
    <Switch>
      <Route exact path="/registerUser" component={AddUser} />
      <Route exact path="/login" component={Login} />

      <ProtectedRoute exact path="/" component={Dashboard} />
      <ProtectedRoute exact path="/aboutme" component={AboutMe} />
      <ProtectedRoute exact path="/addorder" component={AddOrder} />
      <ProtectedRoute exact path="/myorders" component={OrdersList} />
      <ProtectedRoute exact path="/history" component={OrderHistory} />
      <ProtectedRoute exact path="/clients" component={ClientsList} />
      <Route component={NotFound} />

    </Switch>
  </Router>
)

ReactDOM.render(routing, document.getElementById('root'))
```

Listing 10. Plik *index.js*

Komponent `ProtectedRoute`, przedstawiony na Listing 11 odpowiada za blokowanie dostępu niezalogowanych użytkowników do niektórych adresów URL. Logika jego działania jest stosunkowo prosta i polega na sprawdzeniu, czy wśród zmiennych sesyjnych znajduje się `accessToken` – token JWT. Jeżeli tak – oznacza to, że użytkownik jest zalogowany i powinien mieć dostęp do danej ścieżki. Jeżeli nie – użytkownik nie jest zalogowany i nastąpi przekierowanie na stronę logowania. W jaki sposób użytkownik pozyskuje token – zostanie wyjaśnione przy okazji omawiania komponentu odpowiedzialnego za logowanie.

```

import React from 'react';
import { Route, Redirect } from 'react-router-dom';

const ProtectedRoute = ({ component: Component, ...rest }) => {
  const userLoggedIn = sessionStorage.getItem('accessToken');
  return (
    <Route {...rest} render={
      props => {
        if (userLoggedIn) {
          return <Component {...rest} {...props} />
        } else {
          return <Redirect to='/login' />
        }
      }
    } />
  )
}

export default ProtectedRoute;

```

Listing 11. Komponent ProtectedRoute

8.1.3 Dodawanie nowego klienta – komponent AddUser

Komponent AddUser odpowiada za wyświetlenie formularza umożliwiającego wprowadzenie danych nowego klienta, walidację tych danych po stronie klienta, przekazanie danych do backendu za pomocą Fetch API i reakcję na otrzymaną odpowiedź – wyświetlenie użytkownikowi komunikatu błędu w przypadku niepowodzenia operacji lub przekierowanie na stronę główną klienta po pomyślnym dodaniu nowego konta.

```

import React, { Component } from 'react';
import { Redirect } from 'react-router-dom';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import FooterPanel from '../FooterPanel/FooterPanel.js';
import add_user_picture from './adduser.png';

class AddUser extends Component {
  constructor(props) {
    super(props);
    this.state = {
      username: '',
      password: '',
      passwordConfirmed: '',
      firstName: '',
      lastName: '',
      companyName: '',
      nip: '',

      userIsloggedIn: false,
      errorMessage: ''
    };
  }

  this.handleChange = this.handleChange.bind(this);
  this.validateNip = this.validateNip.bind(this);
}

componentDidMount() {
  if (sessionStorage.getItem('accessToken')) {
    this.setState({userIsloggedIn: true})
  }
}

```

```

validateNip(nip) {
  var weights = [6, 5, 7, 2, 3, 4, 5, 6, 7];
  nip = nip.replace(/[s-]/g, '');

  if (nip.length === 10 && parseInt(nip, 10) > 0) {
    var sum = 0;
    for(var i = 0; i < 9; i++){
      sum += nip[i] * weights[i];
    }
    return (sum % 11) === Number(nip[9]);
  }
  return false;
}

addUser = (e) => {
  e.preventDefault();
  if(!this.validateNip(this.state.nip)) {
    this.setState( {errorMessage: "Podano niepoprawny NIP."})
  }
  else if(this.state.password !== this.state.passwordConfirmed) {
    this.setState( {errorMessage: "Hasła różnią się."})
  }
  else {
    fetch('http://localhost:5000/users/addClient', {
      method: 'POST',
      headers: { "Content-type": "application/json; charset=UTF-8" },
      body: JSON.stringify({
        username: this.state.username,
        password: this.state.password,
        firstName: this.state.firstName,
        lastName: this.state.lastName,
        companyName: this.state.companyName,
        nip: this.state.nip
      })
    })
    .then(response => {
      if (response.status !== 200) {
        response.text()
        .then(text => (this.setState({ errorMessage: text })));
      }
      else {
        response.json()
        .then(json => {
          sessionStorage.setItem('accessToken', json.accessToken)
          this.setState( {userIsloggedIn: true});
        });
      }
    })
  }
}

```

Listing 12. Komponent AddUser - bez metody render()

Widoczna na Listing 12 metoda validateNip(nip), odpowiadająca za sprawdzenie, czy użytkownik wprowadził do formularza poprawny numer NIP została pobrana ze źródła:

https://pl.wikibooks.org/wiki/Kody_%C5%BAr%C3%B3d%C5%82owe/Implementacja_NIP#Implementacja_alorytmu_w_j%C4%99zyku_JavaScript

```

render() {
  const errorMessage = this.state.errorMessage;
  if(!this.state.userIsLoggedIn) {
    return (
      <div>
        <HeaderPanel />
        <div className="row u-center-abs">
          <img id="user-image" src={add_user_picture} height="100"
            alt="Obraz - dodaj użytkownika" />

          <form onSubmit={this.addUser}>
            <div className="u-full-width">
              <label htmlFor="username">Użytkownik</label>
              <input className="u-full-width" type="text" placeholder="Użytkownik"
                name="username" required onChange={this.handleChange}/>
            </div>
            <div className="u-full-width">
              <label htmlFor="password">Hasło</label>
              <input className="u-full-width" type="password" placeholder="Hasło"
                name="password" required onChange={this.handleChange}/>
            </div>
            <div className="u-full-width">
              <label htmlFor="password">Potwierdź Hasło</label>
              <input className="u-full-width" type="password"
                placeholder="Potwierdź Hasło" name="passwordConfirmed" required
                onChange={this.handleChange}/>
            </div>
            <div className="u-full-width">
              <label htmlFor="firstName">Imię</label>
              <input className="u-full-width" type="text" placeholder="Imię"
                name="firstName" required onChange={this.handleChange}/>
            </div>
            <div className="u-full-width">
              <label htmlFor="lastName">Nazwisko</label>
              <input className="u-full-width" type="text" placeholder="Nazwisko"
                name="lastName" required onChange={this.handleChange}/>
            </div>
            <div className="u-full-width">
              <label htmlFor="companyName">Nazwa firmy</label>
              <input className="u-full-width" type="text" placeholder="Nazwa firmy"
                name="companyName" required onChange={this.handleChange}/>
            </div>
            <div className="u-full-width">
              <label htmlFor="lastName">NIP firmy</label>
              <input className="u-full-width" type="text" placeholder="NIP firmy"
                name="nip" required onChange={this.handleChange}/>
            </div>
            <div className="u-pull-right">
              <input className="button-primary" type="submit" value="Zarejestruj" />
            </div>
          </form>


          {errorMessage ? ( <div className="errorMessage"> {this.state.errorMessage}
            </div> ) : (null)}
        </div>
        <FooterPanel />
      </div>
    )
  }
  else { return(<Redirect to="/dashboard" />) }
}
}
export default AddUser




```

Listing 13. Komponent AddUser - metoda render

Sprawdzanie, czy użytkownik wypełnił wszystkie wymagane pola formularza wykonywane jest z użyciem odpowiednich atrybutów HTML: `required` dla pól formularza. Dodatkowo przy wysyłaniu formularza przy użyciu opisaną wcześniej metody `validateNip(nip)` sprawdzana jest poprawność wpisanego numeru NIP oraz następuje sprawdzenie, czy poprawnie wpisano dwukrotnie hasło. Dopiero po walidacji do backendu wysyłane jest żądanie zawierające dane nowego użytkownika. W wersji produkcyjnej aplikacji wszystkie żądania będą wykonywane z użyciem protokołu HTTPS, który jest szyfrowaną wersją protokołu HTTP tak, aby przesyłane dane były bezpieczne. W odpowiedzi na poprawnie wykonane żądanie nowo zarejestrowany klient uzyskuje token JWT i jest przekierowywany na stronę główną.

Widok ekranu rejestracji nowego klienta przedstawiono na rysunku rys. 17:





SYSTEM OBSŁUGI ZLECEŃ



Użytkownik

Hasło

Potwierdź Hasło

Imię

Nazwisko

Nazwa firmy

NIP firmy

ZAREJESTRUJ

PJATK 2020 - Projekt i wykonanie: Rafał Reszczyński

rys. 17. Widok ekranu rejestracji nowego klienta

8.1.4 Logowanie użytkownika – komponent Login

Komponent Login odpowiada za logowanie użytkownika do systemu. Po wypełnieniu pól wyświetlonego formularza i zatwierdzeniu formularza (sprawdzany jest warunek, czy pola zostały uzupełnione, za pomocą atrybutu HTML: required) do backendu wysyłane jest żądanie zawierające login i hasło użytkownika. Jeżeli żądanie zostanie wykonane pomyślnie użytkownik zostanie zalogowany, uzyska token JWT i będzie przekierowany na stronę główną. W przypadku błędu w odpowiedzi zostanie zwrócony komunikat opisujący błąd.

```
import React, { Component } from 'react';
import { Link, Redirect } from 'react-router-dom';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import FooterPanel from '../FooterPanel/FooterPanel.js';
import user_picture from './user.png';
import './Login.css';

class Login extends Component {
  constructor(props) {
    super(props);
    this.state = {
      username: '',
      password: '',
      userIsloggedIn: false,
      errorMessage: ''
    };
    this.handleChange = this.handleChange.bind(this);
  }

  componentDidMount() {
    if (sessionStorage.getItem('accessToken')) {
      this.setState({userIsloggedIn: true})
    }
  }

  handleChange(event) {
    this.setState({ [event.target.name]: event.target.value });
  }

  loginUser = (e) => {
    e.preventDefault();
    fetch('http://localhost:5000/users/login', {
      method: 'POST',
      headers: { "Content-type": "application/json; charset=UTF-8" },
      body: JSON.stringify({
        username: this.state.username,
        password: this.state.password
      })
    })
    .then(response => {
      if (response.status !== 200) {
        response.text()
        .then(text => this.setState({ errorMessage: text }));
      }
      else {
        response.json()
        .then(json => sessionStorage.setItem('accessToken', json.accessToken))
        .then(res => this.setState({ userIsloggedIn: true}))
      }
    })
    .catch(error => this.setState(
      { errorMessage: 'Błąd serwera - spróbuj ponownie za chwilę' }));
  }
}
```

```

render() {
  const errorMessage = this.state.errorMessage;

  if(!this.state.userIsLoggedIn) {
    return (
      <div className="login-container">
        <HeaderPanel />
        <div className="row u-center-abs">
          <img id="user-image" src={user_picture} alt="Obraz - użytkownik" />

          <form onSubmit={this.loginUser}>
            <div className="u-full-width">
              <label htmlFor="username">Użytkownik</label>
              <input className="u-full-width" type="text" placeholder="Użytkownik"
                name="username" required onChange={this.handleChange}/>
            </div>

            <div className="u-full-width">
              <label htmlFor="password">Hasło</label>
              <input className="u-full-width" type="password" placeholder="Hasło"
                name="password" required onChange={this.handleChange}/>
            </div>

            <div className="u-pull-right">
              <input className="button-primary" type="submit" value="Zaloguj" />
            </div>
          </form>
          {errorMessage ? (
            <div className="errorMessage">
              {this.state.errorMessage}
            </div> ) : (null)}
        <div className="registration-info">
          Nie jesteś jeszcze naszym klientem?&nbsp;
          <Link to={"/registerUser"}>Zarejestruj się.</Link>
        </div>




        </div>
        <FooterPanel />
      </div>
    )
  }
  else {
    return(<Redirect to="/" />)
  }
}
}

export default Login


```

Listing 14. Komponent Login

Na rys. 18 przedstawiono widok komponentu Login po wykonaniu przez użytkownika nieudanej próby logowania (podanie błędnego hasła):



SYSTEM OBSŁUGI ZLECEŃ



Użytkownik

admin

Hasło

.....

ZALOGUJ

Nieprawidłowe hasło

Nie jesteś jeszcze naszym klientem? [Zarejestruj się.](#)

PJATK 2020 - Projekt i wykonanie: Rafał Reszczyński

rys. 18. Widok komponentu Login z komunikatem błędu

8.1.5 Komponenty MenuPanel, ClientMenuPanel i CompanyMenuPanel

Trzy wymienione w tytule komponenty odpowiadają za wyświetlenie odpowiedniego menu aplikacji w zależności od roli użytkownika – użytkownicy wewnętrzni firmy mają do dyspozycji inne opcje, niż klienci. Rola użytkownika jest pobierana z zawartości JWT, przechowywanego w zmiennej sesyjnej accessToken. Ponieważ po stronie klienta nie ma możliwości walidacji tokenu (wtedy po stronie klienta musiałby być przechowywany klucz tajny, do którego bardziej złośliwi użytkownicy mogliby uzyskać nieuprawniony dostęp), użytkownik może podmienić wartość zmiennej określającej jego rolę, przechowywaną w JWT i przypisać sobie np. rolę administratora. Przed takim postępowaniem po stronie klienta niestety nie da się zabezpieczyć, ale ponieważ jest to jedynie warstwa prezentacji nieuprawniony użytkownik zobaczy co najwyżej opcje menu dla niego nieprzeznaczone. Końcowa autentykacja i autoryzacja odbywa się przecież po stronie backendu.

Powyższa uwaga dotyczy się również pozostałych komponentów, w których występuje zróżnicowanie opcji względem roli użytkownika.

Komponent MenuPanel, przedstawiony na Listing 15 ma za zadanie pobranie roli użytkownika z JWT i na tej podstawie wyświetlenie odpowiednich opcji menu. Plik roles.js zawiera mapę przyporządkowującą nazwie roli odpowiednią wartość liczbową. Dzięki temu

zamiast niewiele mówiącego zapisu, np. `role === 4`, można użyć czytelniejszego: `role === ROLES.KLIENT`

```
import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import ClientMenuPanel from '../ClientMenuPanel/ClientMenuPanel.js';
import CompanyMenuPanel from '../CompanyMenuPanel/CompanyMenuPanel.js';
import jwtDecode from 'jwt-decode';
import ROLES from '../helpers/roles.js';

class MenuPanel extends Component {
  constructor(props) {
    super(props);

    this.state = {
      role: ''
    }
  }

  componentDidMount() {
    // pobranie roli klienta z JWT
    const tokenData = jwtDecode(sessionStorage.getItem('accessToken'));
    const role = tokenData.role;
    this.setState({role : role});
  }

  render() {
    const role = this.state.role;
    return (
      <div>
        {(role === ROLES.KLIENT) ? (<ClientMenuPanel />) : (<CompanyMenuPanel />)}
      </div>
    )
  }
}

export default withRouter(MenuPanel)
```

Listing 15. Komponent MenuPanel

Komponenty `ClientMenuPanel` i `CompanyMenuPanel` odpowiadają za wyświetlenie zróżnicowanych przycisków menu dla klientów firmy i użytkowników wewnętrznych i po wybraniu opcji menu przekierowanie użytkownika na odpowiedni adres URL, lub w przypadku wybrania opcji wylogowania – usunięcie JWT z session storage. Ponieważ kod komponentów `ClientMenuPanel` i `CompanyMenuPanel` jest podobny, przedstawiono jedynie listing komponentu `ClientMenuPanel`:

```

import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';

class ClientMenuPanel extends Component {
  constructor(props) {
    super(props);

    this.logout = this.logout.bind(this);
    this.showAboutMe = this.showAboutMe.bind(this);
    this.showAddOrder = this.showAddOrder.bind(this);
    this.showClientOrders = this.showClientOrders.bind(this);
  }

  logout() {
    sessionStorage.removeItem('accessToken');
    this.props.history.push("/login");
  }

  showAboutMe() {
    this.props.history.push("/aboutme");
  }

  showAddOrder() {
    this.props.history.push("/addorder");
  }

  showClientOrders() {
    this.props.history.push("/myorders");
  }

  render() {
    return (
      <div className="row">
        <button className="three columns button-primary" onClick={this.showAboutMe}>
          Moje dane
        </button>
        <button className="three columns button-primary"
          onClick={this.showClientOrders}>
          Moje zlecenia
        </button>
        <button className="three columns button-primary" onClick={this.showAddOrder}>
          Dodaj zlecenie
        </button>
        <button className="three columns button-primary" onClick={this.logout}>
          Wyloguj
        </button>
      </div>
    )
  }
}

export default withRouter(ClientMenuPanel)

```

Listing 16. Komponent ClientMenuPanel

Widok menu dla różnych użytkowników przedstawiono na rys. 19 i rys. 20:



rys. 19. Widok menu klienta - komponent ClientMenuPanel



rys. 20. Widok menu użytkownika wewnętrznego Zakładu - komponent CompanyMenuPanel

8.1.6 Komponent AboutMe

Komponent AboutMe jest odpowiedzialny za prezentację danych konta użytkownika. Po zamontowaniu komponentu wykonywana jest metoda `getUser()`, która wysyła żądanie http do backendu zawierające przechowywany w zmiennej sesyjnej JWT. W odpowiedzi na żądanie zwracane są dane użytkownika zidentyfikowanego na podstawie przesłanego tokenu.

Kod komponentu AboutMe przedstawiono na poniższym listingu:

```
import React, { Component } from 'react';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import FooterPanel from '../FooterPanel/FooterPanel.js';
import MenuPanel from '../MenuPanel/MenuPanel.js';
import './AboutMe.css';

class AboutMe extends Component {
  constructor(props) {
    super(props);
    this.state = {
      user: '',
    };
    this.getUser = this.getUser.bind(this);
  }

  getUser() {
    fetch('http://localhost:5000/users/getUser', {
      method: 'GET',
      headers: {
        'Authorization': `Bearer ${localStorage.getItem('accessToken')}`,
        'Content-type': 'application/json'
      }
    })
    .then(res => res.json())
    .then(json => {
      json.createdAt = json.createdAt.split('T')[0];
      this.setState({ user: json })
    })
    .catch(error => console.log(error));
  }

  componentDidMount() {
    this.getUser();
  }

  render() {
    return (
      <div>
        <HeaderPanel />
        <MenuPanel />
        <div className="row"><h1 className="u-full-width">Profil użytkownika</h1></div>

        <div className="row row-highlight">
          <div className="four columns description">Login: </div>
          <div className="eight columns value">{this.state.user.name}</div>
        </div>

        <div className="row row-highlight">
          <div className="four columns description">Firma: </div>
          <div className="eight columns value">{this.state.user.companyName}</div>
        </div>

        <div className="row row-highlight">
          <div className="four columns description">NIP: </div>
          <div className="eight columns value">{this.state.user.nip}</div>
        </div>
      </div>
    );
  }
}
```

```

    <div className="row row-highlight">
      <div className="four columns description">Imię: </div>
      <div className="eight columns value">{this.state.user.firstName}</div>
    </div>




    <div className="row row-highlight">
      <div className="four columns description">Nazwisko: </div>
      <div className="eight columns value">{this.state.user.lastName}</div>
    </div>

    <div className="row row-highlight">
      <div className="four columns description">Data rejestracji: </div>
      <div className="eight columns value">{this.state.user.createdAt}</div>
    </div>
    <FooterPanel />
  </div>
)
}
}
export default AboutMe

```

Listing 17. Komponent AboutMe

Widok komponentu AboutMe przedstawiono na rys. 21:




SYSTEM OBSŁUGI ZLECEŃ

MOJE DANE
MOJE ZLECENIA
DODAJ ZLECENIE
WYLOGUJ

Profil użytkownika

Login:	jank
Firma:	PPHU Janex
NIP:	4964676764
Imię:	Jan
Nazwisko:	Kowalski
Data rejestracji:	2020-08-03

PJATK 2020 - Projekt i wykonanie: Rafał Reszczyński

rys. 21. Komponent AboutMe

8.1.7 Lista klientów – komponenty ClientsList, Client i ClientAction

Komponenty ClientsList i Client służą wyświetleniu dostępnej dla użytkowników wewnętrznych firmy listy klientów. Dodatkowy komponent ClientAction odpowiada za wyświetlenie przycisku zablokowania/odblokowania klienta i przeprowadzenie tej operacji, przez wysłanie odpowiedniego żądania do backendu. Opcja ta jest dostępna wyłącznie dla użytkownika będącego administratorem.

Kod komponentu ClientsList został przedstawiony na listingu poniżej:

```

import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import FooterPanel from '../FooterPanel/FooterPanel.js';
import Client from '../Client/Client.js';
import CompanyMenuPanel from '../CompanyMenuPanel/CompanyMenuPanel.js';
import jwtDecode from 'jwt-decode';
import ROLES from '../helpers/roles.js';
import './ClientsList.css';

class ClientsList extends Component {
  constructor(props) {
    super(props);

    this.state = {
      clients: [],
      errorMessage: '',
      role: '',
    }
  }

  componentDidMount(){
    const tokenData = jwtDecode(sessionStorage.getItem('accessToken'));
    const role = tokenData.role;

    if(role === ROLES.KLIENT) {
      this.props.history.push('/');
    }

    this.setState({ role: role});

    fetch('http://localhost:5000/users/getallclients', {
      method: 'GET',
      headers: {
        'Authorization': `Bearer ${sessionStorage.getItem('accessToken')}`,
        'Content-type': 'application/json'
      }
    })
    .then(response => {
      if (response.status !== 200) {
        response.text()
        .then(text => this.setState({ errorMessage: text }));
      }
      else {
        response.json()
        .then(json => {
          this.setState({ clients: json })
        })
      }
    })
    .catch(error => this.setState({ errorMessage: 'Wystąpił błąd' }));
  }

  render() {
    const role = this.state.role;
    const errorMessage = this.state.errorMessage;
    return (
      <div>
        <HeaderPanel />
        <CompanyMenuPanel />
        <h3>Lista klientów:</h3>
        <table className="oh-table">
          <thead>
            <tr>
              <th>ID</th>

```

```

        <th>Firma</th>
        <th>NIP</th>
        <th>Imię</th>
        <th>Nazwisko</th>
        <th>Status</th>
        {(role === ROLES.ADMIN) ? (<th>Akcja</th>) : (null)}
    </tr>
</thead>
<tbody>
    {
        this.state.clients.map((client, index) => {
            return (
                <Client
                    key = {client.id}
                    id = {client.id}
                    companyName = {client.companyName}
                    nip = {client.nip}
                    firstName = {client.firstName}
                    lastName = {client.lastName}
                    isActive = {client.isActive}
                    userRole = {role}
                />
            )
        })
    }
</tbody>
</table>
{errorMessage ? (
    <div className="errorMessage">
        {this.state.errorMessage}
    </div> ) : (null)}
<FooterPanel />
</div>
)
}
}
export default withRouter(ClientsList)

```

Listing 18. Komponent ClientsList

Głównym zadaniem komponentu ClientsList jest pobranie listy klientów z backendu aplikacji. W tym celu wysyłane jest odpowiednie żądanie HTTP, z umieszczonym w nim JWT. W odpowiedzi na żądanie przesyłana jest lista klientów w postaci obiektu JSON. W metodzie render() komponentu ClientsList lista klientów jest mapowana, dla każdego klienta renderowany jest wiersz tabeli – komponent Client, do którego są przesyłane dane klienta w postaci odpowiednich props. Dodatkowo, jeżeli użytkownik ma rolę administratora, lista klientów będzie zawierać dodatkową kolumnę „Akcja”. To w niej znajdzie się przycisk realizujący blokowanie/odblokowanie klienta.

Kod komponentu Client odpowiedzialnego za wyświetlenie wiersza tabeli, zawierającego dane klienta przedstawiono na poniższym listingu:

```
import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import ClientAction from '../ClientAction/ClientAction.js';
import ROLES from '../helpers/roles.js';

class Client extends Component {

  constructor(props) {
    super(props);
    this.state = {
      ...props
    }
    this.setStateInClient = this.setStateInClient.bind(this);
  }

  setStateInClient(key, value) {
    this.setState({[key] : value});
    console.log(key, value);
  }

  render() {
    return (
      <tr>
        <td>{this.state.id}</td>
        <td>{this.state.companyName}</td>
        <td>{this.state.nip}</td>
        <td>{this.state.firstName}</td>
        <td>{this.state.lastName}</td>
        <td>{(this.state.isActive) ? ('Aktywny') : ('Zablokowany')}</td>

        {(this.state.userRole === ROLES.ADMIN) ?
          ( <td><ClientAction
              isActive = {this.state.isActive}
              clientId = {this.state.id}
              setStateInClient = {this.setStateInClient}
              userRole = {this.state.userRole}
            />
          </td>
          ) : (null)}
      </tr>
    )
  }
}

export default withRouter(Client)
```

Listing 19. Komponent Client

Komponent Client odpowiada za wyświetlenie danych pojedynczego klienta oraz, jeżeli użytkownik jest administratorem prezentację komponentu ClientAction do którego przekazywane są w postaci props dane klienta i metoda aktualizująca wartość wyświetlaną w polu statusu klienta komponentu Client.

Ostatnim komponentem, dostępnym jedynie dla Administratora i służącym do blokowania/odblokowania klientów jest komponent ClientAccion. Listing komponentu przedstawiono poniżej:

```
import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import ROLES from '../helpers/roles.js';
import './ClientAction.css';

class ClientAction extends Component {
  constructor(props) {
    super(props);

    this.state = {
      isActive: '',
      errorMessage: '',
    };

    this.getActionText = this.getActionText.bind(this);
    this.changeStatus = this.changeStatus.bind(this);
  }

  getActionText(isActive) {
    if (this.state.isActive) {
      return 'Zablokuj'
    } else {
      return 'Aktywuj'
    }
  }

  changeStatus() {
    const clientId = this.props.clientId;

    fetch('http://localhost:5000/users/changeclientstatus', {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${sessionStorage.getItem('accessToken')}`,
        'Content-type': 'application/json'
      },
      body: JSON.stringify({
        clientId: clientId
      })
    })
    .then(response => {
      if (response.status !== 200) {
        response.text()
        .then(text => this.setState({ errorMessage: text }));
      }
      else {
        this.setState({errorMessage : ''});
        const newStatus = !this.state.isActive;
        this.setState({isActive : newStatus});
        console.log(newStatus);

        this.props.setStateInClient('isActive', newStatus);
      }
    })
    .catch(error => this.setState({ errorMessage: 'Wystąpił błąd' }));
  }

  componentDidMount() {
    this.setState({ isActive : this.props.isActive });
  }
}
```



```

render() {
  const action = this.getActionText(this.state.action);
  return (
    <div>
      {(this.props.userRole === ROLES.ADMIN) ?
        (<button className="ca-button" onClick={this.changeStatus}>{action}</button>)
        : (null)}
      <div className="ca-error">{this.state.errorMessage}</div>
    </div>
  )
}
}
export default withRouter(ClientAction)


```

Listing 20. Komponent ClientAction

Warte krótkiego omówienia są zawarte w komponencie ClientAction metody `getActionText(isActive)` i `changeStatus()`. Metoda `getActionText(isActive)` odpowiada za przypisanie tekstu do przycisku akcji na podstawie wartości zmiennej typu boolean `isActive` oznaczającej status użytkownika.

Metoda jest wywoływana po kliknięciu przycisku akcji zmiany statusu przez administratora. Metoda wysyła żądanie do backendu, zawierające JWT i id Klienta, dla którego ma być wykonana zmiana statusu. Jeżeli przy obsłudze żądania wystąpi błąd zostanie wyświetlony odpowiedni komunikat. Jeżeli status odpowiedzi ma kod 200 – tzn. nastąpiła zmiana statusu klienta w bazie danych i po stronie klienta również następuje zmiana wyświetlanego statusu – wykorzystywana jest do tego celu metoda `setStateInClient()` przekazana do komponentu jako prop.

Widok listy klientów wyświetlony dla użytkownika – administratora przedstawiono na rys. 22:



SYSTEM OBSŁUGI ZLECEŃ

MOJE DANE
ZLECENIA
KLIENTY
WYLOGUJ

Lista klientów:

ID	Firma	NIP	Imię	Nazwisko	Status	Akcja
4	PPHU Janex	4964676764	Jan	Kowalski	Zablokowany	AKTYWUJ
5	Botanical experts	8122933518	Adam	Nowak	Aktywny	ZABLOKUJ
6	EKO materials	5680832741	Jolanta	Depta	Aktywny	ZABLOKUJ

PJATK 2020 - Projekt i wykonanie: Rafał Reszczyński

rys. 22. Widok listy klientów dla administratora

8.1.8 Komponent OrdersList

Komponent OrdersList odpowiada za wyświetlenie listy zleceń produkcyjnych. W zależności od roli zalogowanego użytkownika zostaną wyświetlone wszystkie zlecenia (użytkownicy wewnętrzni firmy), bądź jedynie zlecenia złożone przez zalogowanego użytkownika (klient).

Kod komponentu OrdersList przedstawiono na poniższym listingu:

```
import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import FooterPanel from '../FooterPanel/FooterPanel.js';
import MenuPanel from '../MenuPanel/MenuPanel.js';
import Order from '../Order/Order.js';
import statusesMap from '../helpers/statusesMap';
import './OrderList.css';

class OrdersList extends Component {
  constructor(props) {
    super(props);
    this.state = {
      role: null, // rola użytkownika - w zależności od niej będzie wyświetlane różne menu
      orders: [],
      errorMessage: '',
      statusToShow: 0, // filtrowany status, wybór z listy rozwijanej
    };
    this.getOrders = this.getOrders.bind(this);
    this.statusSelected = this.statusSelected.bind(this);
  }

  componentDidMount() {
    this.getOrders();
  }

  statusSelected(e) {
    const status = e.target.selectedOptions[0].index;
    this.setState({ statusToShow: status });
  }

  getOrders = () => {
    fetch('http://localhost:5000/orders/getorders', {
      method: 'GET',
      headers: {
        "Content-type": "application/json; charset=UTF-8",
        'Authorization': `Bearer ${sessionStorage.getItem('accessToken')}`
      },
    })
    .then(response => {
      if (response.status !== 200) {
        response.text()
          .then(text => (this.setState({ errorMessage: text })));
      }
      else {
        response.json()
          .then(json => {
            //console.log(json)
            this.setState({ orders: json });
          });
      }
    })
  }

  render() {
    const errorMessage = this.state.errorMessage;
    const statusesFromMap = Array.from(statusesMap.values());
  }
}
```

```

const statuses = ['Wszystkie', ...statusesFromMap];

const statusToShow = this.state.statusToShow;
return(
  <div>
    <HeaderPanel />
    <MenuPanel />
    <div className="row">
      <h3 className="u-full-width orl-header">Lista zleceń:</h3>
    </div>
    <div className="row u-center-abs">

      {this.state.orders[0] ?
      (
        <table className="u-full-width">
          <thead>
            <tr>
              <th className="th-group orl-th" colSpan="3">Surowiec</th>
              <th className="th-group orl-th" colSpan="2">Produkt</th>
              <th className="th-group orl-th" colSpan="5">Zlecenie</th>
            </tr>
            <tr>
              <th className="orl-th">Nazwa</th>
              <th className="orl-th">Masa<br />[kg]</th>
              <th className="orl-th">Palety<br />[szt]</th>
              <th className="orl-th">Masa<br />[kg]</th>
              <th className="orl-th">Palety<br />[szt.]</th>
              <th className="orl-th">ID</th>
              <th className="orl-th">Zlecający</th>
              <th className="orl-th">Data<br />złożenia</th>
              <th className="orl-th">Status<br />

              <select className="orl-select" onChange={this.statusSelected}>
                {
                  statuses.map((element, index) => {
                    return (
                      <option key={index}>{element}</option>
                    )
                  })
                }
              </select>
            </th>
              <th className="orl-th">Akcja</th>
            </tr>
          </thead>
          <tbody>
            {
              this.state.orders
                .filter((order, index) => {
                  if(statusToShow === 0) {
                    return order;
                  } else if (statusToShow === order.currentStatus){
                    return order;
                  } else {
                    return null;
                  }
                })
                .map((order, index) => {
                  return(
                    <Order
                      key = {order.id}
                      id = {order.id}
                      rawMaterialName = {order.rawMaterialName}
                      orderDate = {order.orderDate}
                      companyName = {order.User.companyName}
                      rawMaterialMass = {order.rawMaterialMass}
                      rawMaterialPallets = {order.rawMaterialPallets}

```

```

        productMass = {order.productMass}
        productPallets = {order.productPallets}
        currentStatus = {order.currentStatus}
      />
    )
  })
}
</tbody>
</table>
) :
(
  <h3 className="or-h3">Brak zleceń...</h3>
)
}
</div>
{errorMessage ? (
  <div className="errorMessage">
    {this.state.errorMessage}
  </div> ) : (null)
}
<FooterPanel />
</div>
)
}
}

export default withRouter(OrdersList)

```

Listing 21. Komponent OrdersList

Po zamontowaniu komponentu wykonywana jest metoda `getOrders()` która odpowiada za pobranie z backendu odpowiedniej dla zalogowanego użytkownika (identyfikacja na podstawie JWT) listy zleceń. Jeżeli przy obsłudze żądania wystąpi błąd użytkownikowi zostanie wyświetlony odpowiedni komunikat. W odpowiedzi na poprawnie wykonane żądanie przesyłana jest lista zleceń w postaci JSON. W metodzie `render()` lista zleceń jest mapowana i dla każdego z nich wyświetlany jest komponent `Order`, do którego dane przekazane są za pomocą props.

Dodatkowo w komponencie `OrdersList` zaimplementowana jest funkcjonalność filtrowania zleceń na podstawie wybranego statusu. Status do wyświetlenia jest wybierany przy użyciu listy rozwijanej – elementu `select` HTML. Wybrany z listy status, za pomocą wywoływanej przy wyborze pozycji z listy metody `statusSelected(e)` jest przypisywany do zmiennej stanu komponentu: `statusToShow`. W metodzie `render()` komponentu `OrdersList` przeznaczone do wyświetlenia zlecenia są filtrowane na podstawie wybranego stanu.

8.1.9 Komponent Order

Zadanie komponentu `Order` jest stosunkowo proste: przyjmuje dane dotyczące zlecenia z komponentu `OrdersList` jako props i wyświetla je jako wiersz w tabeli. Warto zwrócić uwagę na umieszczony w komponencie `Order` przycisk, po naciśnięciu którego wywoływana jest metoda `showHistory()` służąca do przejścia do podglądu historii zmiany statusów zlecenia. W komponencie `Order` został również zagnieżdżony komponent `StatusAction`. Listing komponentu `Order` został podany poniżej:

```

import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import StatusAction from '../StatusAction/StatusAction.js';
import statusesMap from '../helpers/statusesMap.js';
import historyBtnImg from './history.png';
import './Order.css';

class Order extends Component {
  constructor(props) {
    super(props);
    this.state = {
      ...props
    }

    this.setStateInOrder = this.setStateInOrder.bind(this);
    this.showHistory = this.showHistory.bind(this);
  }

  setStateInOrder(key, value) {
    this.setState({[key] : value});
  }

  showHistory() {
    this.props.history.push({
      pathname: '/history',
      orderId: this.state.id,
    });
  }

  render() {
    return (
      <tr>
        <td>{this.state.rawMaterialName}</td>
        <td>{this.state.rawMaterialMass}</td>
        <td>{this.state.rawMaterialPallets}</td>
        <td>{this.state.productMass}</td>
        <td>{this.state.productPallets}</td>
        <td>{this.state.id}&nbsp;
          <button className="or-button">
            <img className="or-image" src={historyBtnImg} alt="H"
              onClick={this.showHistory} />
          </button>
        </td>
        <td>{this.state.companyName}</td>
        <td>{this.state.orderDate.split('T')[0]}</td>
        <td>{statusesMap.get(this.state.currentStatus)}</td>
        <td><StatusAction currentStatus={this.state.currentStatus}
          setStateInOrder={this.setStateInOrder} orderId={this.state.id}/>
        </td>
      </tr>
    )
  }
}

export default withRouter(Order)

```

Listing 22. Komponent Order

8.1.10 Komponent StatusAction

Komponent StatusAction spełnia bardzo istotną rolę w części frontendowej systemu, ponieważ to w nim wykonywana jest główna funkcjonalność systemu, czyli zmiana statusu zlecenia. Omówienie sposobu działania komponentu należy rozpocząć od operacji wykonywanych po jego zamontowaniu. Po pierwsze na podstawie obecnego statusu zlecenia i roli użytkownika określana jest akcja lub akcje jakie użytkownik może wykonać (zgodnie z Tabela 1 i przygotowaną na jej podstawie metodą `statusAndRoleToAction(status, role)`, przyporządkowującą roli użytkownika możliwe do wykonania akcje dla danego statusu zlecenia). Kolejnym krokiem jest wyświetlenie odpowiednich przycisków akcji oraz w szczególnych przypadkach dodatkowych elementów (administrator dostaje listę wyboru statusów do ustawienia, a technolog w przypadku zdawania wykonanego produktu na magazyn będzie widział formularz, w którym musi podać odpowiednie dane produktu przed zdaniem). Samo wykonanie akcji zmiany statusu polega na wysłaniu do backendu żądania HTTP zawierającego JWT i odpowiednie dane dotyczące zlecenia. Po uzyskaniu pozytywnej odpowiedzi na żądanie status zlecenia jest uaktualniany, w przypadku błędu wyświetlany jest odpowiedni komunikat. Listing komponentu StatusAction przedstawiono poniżej:

```
import React, { Component } from 'react';
import ROLES from '../helpers/roles.js';
import ACTIONS from '../helpers/actions.js';
import actionToNewStatus from '../helpers/actionToNewStatus.js';
import statusesMap from '../helpers/statusesMap.js';
import statusesToIntMap from '../helpers/statusesToIntMap.js';
import statusAndRoleToAction from '../helpers/statusAndRoleToAction.js';
import jwtDecode from 'jwt-decode';
import './StatusAction.css';

class StatusAction extends Component {
  constructor(props) {
    super(props);

    this.state = {
      role: '',
      actions: [],
      errorMessage: '',
      statusToSet: 'DEFAULT'
    };

    this.setNewStatus = this.setNewStatus.bind(this);
    this.handleChange = this.handleChange.bind(this);
    this.putProductToStorage = this.putProductToStorage.bind(this);
  }

  putProductToStorage(event) {
    event.preventDefault();
    this.setNewStatus(event);
  }

  handleChange(event) {
    this.setState({ [event.target.name]: event.target.value });
  }

  setNewStatus(event) {
    var action;
    if(event.target[2]) {
      console.log(event.target[2].value);
      action = event.target[2].value;
    } else {
```

```

    action = event.target.textContent;
  }

  var newStatus = actionToNewStatus.get(action);
  if(this.state.role === ROLES.ADMIN && this.state.statusToSet !== '') {
    const newStatusText = this.state.statusToSet;
    newStatus = statusesToIntMap.get(newStatusText);
    console.log(newStatus + newStatus);
  } else if(this.state.role === ROLES.ADMIN && this.state.statusToSet === '') {
    this.setState({ errorMessage: 'Wybierz opcję' });
    return;
  }

  fetch('http://localhost:5000/statuses/changestatus', {
    method: 'POST',
    headers: {
      "Content-type": "application/json; charset=UTF-8",
      'Authorization': `Bearer ${sessionStorage.getItem('accessToken')}`
    },
    body: JSON.stringify({
      orderId: this.props.orderId,
      oldStatus: this.props.currentStatus,
      newStatus: newStatus,
      productMass: this.state.productMass,
      productPallets: this.state.productPallets
    })
  })
  .then(response => {
    if (response.status !== 200) {
      response.text()
        .then(text => this.setState({ errorMessage: text }));
    }
    else {
      this.props.setStateInOrder('currentStatus', newStatus);
      this.props.setStateInOrder('productMass', this.state.productMass);
      this.props.setStateInOrder('productPallets', this.state.productPallets);
      const actions = statusAndRoleToAction(newStatus, this.state.role);
      this.setState({ actions : actions });
      this.setState({ errorMessage: '' });
      this.setState({ statusToSet: 'DEFAULT' });
      this.setState({ productMass: null });
      this.setState({ productPallets: null });
    }
  })
  .catch(error => console.log(error))
}

componentDidMount() {
  const status = this.props.currentStatus;

  const tokenData = jwtDecode(sessionStorage.getItem('accessToken'));
  const role = tokenData.role;
  var actions = statusAndRoleToAction(status, role);
  this.setState({role: role});
  this.setState({ actions: actions });
}

render() {
  const statuses = Array.from(statusesMap.values());
  const role = this.state.role;
  return (
    <div>
      {
        (role !== ROLES.ADMIN) ? (
          this.state.actions.map((action, index) => {
            return(

```

```

        <div key="index">
          {(role===ROLES.TECHNOLOG && action===ACTIONS.ZDAJ_PRODUKT) ? (
            <form className="sa-form"
              onSubmit={this.putProductToStorage}>
              <label className="sa-label" htmlFor="productMass">
                kg produktu
              </label>
              <input className="sa-input" type="number" min="0"
                step="0.01" placeholder="il. produktu" name="productMass"
                required onChange={this.handleChange} />
              <label className="sa-label" htmlFor="productPallets">
                il. palet produktu
              </label>
              <input className="sa-input" type="number" min="0" step="1"
                placeholder="palet" name="productPallets" required
                onChange={this.handleChange}
              />
              <input className="sa-button" type="submit"
                value={action} />
            </form>
          ) : (
            <div key={index}><button className="sa-input sa-button"
              onClick={this.setNewStatus}>{action}</button><br/>
            </div>
          )
        }
      </div>
    )
  })
) : (
  <div className="sa-admin-actions">
    <select className="sa-select" value={this.state.statusToSet}
      name="statusToSet" onChange={this.handleChange}>
      <option value="DEFAULT" disabled> -- wybierz -- </option>>
      { statuses.map((element, index) => {
        return (<option key={index}>{element}</option>) }}
    </select><br />
    <button className="sa-button" type="submit"
      onClick={this.setNewStatus}>Ustaw status
    </button>
  </div>
)
}
<div className="sa-error">{this.state.errorMessage}</div>
</div>
)
}
}
export default StatusAction

```

Listing 23. Komponent StatusAction

Przykładowa lista zleceń, wyświetlana dla użytkownika zalogowanego jako Technolog została zamieszczona na rys. 23. Dla każdego zlecenia wyświetlany jest przycisk podglądu jego historii. W nagłówku kolumny Status widoczne jest pole filtrowania statusów. Można również zauważyć dostępne dla danych zleceń i zalogowanego użytkownika, w tym wypadku mającego rolę Technologa, akcje. Warto zaznaczyć, że po zakończeniu produkcji, przy zdawaniu wykonanego produktu do magazynu Technolog musi wprowadzić odpowiednie dane – masę produktu i ilość palet z produktem. Jeżeli te dane nie zostaną podane – akcja nie będzie mogła zostać wykonana.



SYSTEM OBSŁUGI ZLECEŃ

MOJE DANE

ZLECENIA

KLIENCI

WYLOGUJ

Lista zleceń:

SUROWIEC			PRODUKT		ZLECENIE				
Nazwa	Masa [kg]	Palety [szt]	Masa [kg]	Palety [szt.]	ID	Zlecający	Data złożenia	Status	Akcja
Marchew	2500.00	2			1	PPHU Janex	2020-08-02	Złożone	<div>ZATWIERDŹ</div> <div>ANULUJ</div>
Liście mięty	3500.00	10			2	PPHU Janex	2020-08-03	Złożone	<div>ZATWIERDŹ</div> <div>ANULUJ</div>
Nasiona truskawki	5470.00	13	523.00	10	3	PPHU Janex	2020-08-03	Zakończzone	
Śruta kukurydziana	12000.00	15			4	PPHU Janex	2020-08-03	W Produkcji	<div>kg produktu</div> <div>il. produktu</div> <div>il. palet produktu</div> <div>palet</div> <div>ZDAJ PRODUKT</div>
Kwiaty róży	3000.00	15	12.00	1	5	PPHU Janex	2020-08-03	Na Magazynie	<div>POBIERZ SUROWIEC</div>

rys. 23. Widok listy zleceń dla zalogowanego Technologa

Użytkownik zalogowany jako administrator może dowolnie zmieniać status zlecenia (możliwość ta została wprowadzona w celu ewentualnej korekty pomyłek użytkowników przy zmianie stanów zleceń):

Marchew	2500.00	2	1	PPHU Janex	2020-08-02	Złożone	-- wybierz -- USTAW STATUS
---------	---------	---	---	------------	------------	---------	-------------------------------

rys. 24. Widok pojedynczego zlecenia dla zalogowanego Administratora

Pozostali użytkownicy systemu dla danego zlecenia będą mieli wyświetlane odpowiednie akcje dla nich dostępne.

8.1.11 Komponent OrderHistory

Ostatnim pozostałym do omówienia komponentem jest OrderHistory. Odpowiada on za przedstawienie historii zmian statusów danego zlecenia i jest wywoływany po naciśnięciu odpowiedniego przycisku. Jego działanie jak w przypadku innych komponentów polega na wysłaniu odpowiedniego żądania HTTP, zawierającego JWT i ID zlecenia oraz wyświetleniu uzyskanej w odpowiedzi listy, a w przypadku błędu przy obsłudze zlecenia – wyświetleniu odpowiedniego komunikatu. Listing komponentu OrderHistory przedstawiono poniżej:

```
import React, { Component } from 'react';
import { withRouter } from 'react-router-dom';
import HeaderPanel from '../HeaderPanel/HeaderPanel.js';
import FooterPanel from '../FooterPanel/FooterPanel.js';
import statusesMap from '../helpers/statusesMap.js';
import './OrderHistory.css';

class OrderHistory extends Component {
  constructor(props) {
    super(props);
    this.state = {
      orderId: this.props.location.orderId,
      orderHistory: [],
      rawMaterialName: '',
      errorMessage: ''
    }
    this.goBack = this.goBack.bind(this);
  }

  componentDidMount(){
    if(!this.props.location.orderId) {
      this.props.history.push('/');
    }
    fetch('http://localhost:5000/orders/getorderhistory', {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${sessionStorage.getItem('accessToken')}`,
        'Content-type': 'application/json'
      },
      body: JSON.stringify({
        orderId: this.state.orderId
      })
    })
    .then(response => {
      if (response.status !== 200) {
        response.text()
          .then(text => this.setState({ errorMessage: text }));
      }
      else {
        response.json()
          .then(json => {
            this.setState({ orderHistory: json.orderHistory });
            this.setState({ rawMaterialName: json.rawMaterialName });
          })
      }
    })
    .catch(error => this.setState({ errorMessage: 'Wystąpił błąd' }));
  }

  goBack() {
    this.props.history.push('/myorders');
  }
}
```

```

render() {
  const errorMessage = this.state.errorMessage;
  return (
    <div>
      <HeaderPanel />
      <div className="row">
        <button className="four columns button-primary" onClick={this.goBack}>
          Wróć do zleceń
        </button>
      </div>
      <h3>Historia zlecenia nr {this.state.orderId} ({this.state.rawMaterialName}):</h3>
      <table className="oh-table">
        <thead>
          <tr>
            <th>Status</th>
            <th>Ustawiono</th>
          </tr>
        </thead>
        <tbody>
          {
            this.state.orderHistory.map((element, index) => {
              return (
                <tr key={index}>
                  <td>{statusesMap.get(element.StatusId)}</td>
                  <td>{element.changeDate.split('T')[0]}</td>
                </tr>
              )
            })
          }
        </tbody>
      </table>
      {errorMessage ? (
        <div className="errorMessage">
          {this.state.errorMessage}
        </div> ) : (null)}
      <FooterPanel />
    </div>
  )
}
}
export default withRouter(OrderHistory)

```

Listing 24. Komponent OrderList

Przykładowa historia zmian statusów zlecenia została przedstawiona poniżej:

SYSTEM OBSŁUGI ZLECEŃ

WRÓĆ DO ZLECEŃ

Historia zlecenia nr 10 (porzeczka):

Status	Ustawiono
Złożone	2020-08-04
Anulowane	2020-08-04

PJATK 2020 - Projekt i wykonanie: Rafał Reszczyński

Listing 25. Widok historii zmian statusów zlecenia

8.2 Backend

Na część backendową aplikacji składa się interfejs REST API zajmujący się obsługą żądań przychodzących z części frontendowej, baza danych MySQL oraz łącznik między bazą, a kontrolerem – wykonane z użyciem ORM Sequelize modele danych.

8.2.1 Index.js

Plik Index.js zawiera skrypt uruchamiający backend. Wykonywane jest w nim połączenie z bazą danych, deklarowane użycie w aplikacji CORS i środowiska Express.js oraz ustalane są ścieżki routingu. Listing pliku Index.js przedstawiono poniżej:

```
const express = require('express');
const cors = require('cors');
const db = require('./database/db');
const app = express();

app.use(cors());
app.use(express.json());

db.sequelize.authenticate()
  .then(() => {
    console.log('Connection with DB has been established successfully.');
```

```
  })
  .catch(err => {
    console.error('Unable to connect to the database:', err);
  });

app.get('/', (req, res) => {
  res.send('Backend server working...');
})

app.use('/users', require('./routes/users'));
app.use('/orders', require('./routes/orders'));
app.use('/statuses', require('./routes/statuses'));

app.listen(5000, () => {
  console.log('Backend server listening on port 5000')
});
```

Listing 26. Plik Index.js

8.2.2 Db.js

Plik db.js odpowiada za konfigurację bazy danych. To tutaj do bazy podpinane są modele Sequelize, tworzone asocjacje między modelami oraz przypisywane do modeli dodatkowe metody. Wartą szczególnej uwagi jest metoda `checkPassword(password)`. Ponieważ hasła użytkowników są zapisywane w bazie w postaci zaszyfrowanej, a użytkownicy przy logowaniu będą dostarczać hasła w postaci jawnego tekstu – do porównania, czy hasło użytkownika jest zgodne z zapisanym w bazie należy je najpierw zaszyfrować i dopiero wtedy porównać wartości. Tym właśnie zajmuje się metoda `checkPassword(password)`.

```
const dbConnectionParams = require('./dbConnectionParams.json');
const Sequelize = require('sequelize');
const bcrypt = require('bcrypt');
const db = {};

const sequelize = new Sequelize(
  dbConnectionParams.dbName,
```

```

    dbConnectionParams.dbUser,
    dbConnectionParams.dbPassword,
    dbConnectionParams.dbOptions
  );

  db.sequelize = sequelize;
  db.Sequelize = Sequelize;

  db.User = require('../models/User')(sequelize, Sequelize);
  db.Role = require('../models/Role')(sequelize, Sequelize);
  db.Status = require('../models/Status')(sequelize, Sequelize);
  db.Order = require('../models/Order')(sequelize, Sequelize);
  db.OrderStatusHistory = require('../models/OrderStatusHistory')(sequelize, Sequelize);

  db.User.prototype.checkPassword = async function(password) {
    return await bcrypt.compare(password, this.password);
  }

  db.User.prototype.validateNip = function(nip) {
    var weights = [6, 5, 7, 2, 3, 4, 5, 6, 7];
    nip = nip.replace(/[\s-]/g, '');

    if (nip.length === 10 && parseInt(nip, 10) > 0) {
      var sum = 0;
      for(var i = 0; i < 9; i++){
        sum += nip[i] * weights[i];
      }
      return (sum % 11) === Number(nip[9]);
    }
    return false;
  }

  db.Role.hasMany(db.User, {
    foreignKey: {
      allowNull: false
    }
  });

  db.OrderStatusHistory.belongsTo(db.User, {
    foreignKey: {
      name: 'changedByUserId',
      allowNull: false
    }
  });

  db.OrderStatusHistory.belongsTo(db.Status, {
    foreignKey: {
      allowNull: false
    }
  });

  db.OrderStatusHistory.belongsTo(db.Order, {
    foreignKey: {
      allowNull: false
    }
  });

  db.Order.belongsTo(db.User, {
    foreignKey: {
      name: 'ordererId',
      allowNull: false
    }
  });
}

module.exports = db;

```

Listing 27. Plik db.js

8.2.3 Modele Sequelize

Dzięki zastosowaniu ORM Sequelize i przygotowaniu odpowiednich modeli można w prosty sposób za pomocą działań na obiektach reprezentujących modele wykonywać operacje na danych przechowywanych w bazie. Dodatkowo przy definiowaniu modelu na poszczególne atrybuty zostały nałożone odpowiednie ograniczenia, przez co został spełniony warunek walidacji danych. Listingi poszczególnych modeli zostały przedstawione poniżej:

```
const Sequelize = require('sequelize');
module.exports = function(sequelize, DataTypes) {
  return sequelize.define("Order", {
    id: {
      type: Sequelize.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true
    },
    rawMaterialName: {
      type: Sequelize.STRING(200),
      allowNull: false
    },
    orderDate: {
      type: Sequelize.DATE,
      allowNull: false,
      defaultValue: Sequelize.NOW
    },
    rawMaterialMass: {
      type: Sequelize.DECIMAL(10, 2).UNSIGNED,
      allowNull: false
    },
    rawMaterialPallets: {
      type: Sequelize.INTEGER,
      allowNull: false
    },
    productMass: {
      type: Sequelize.DECIMAL(10, 2).UNSIGNED,
      allowNull: true
    },
    productPallets: {
      type: Sequelize.INTEGER.UNSIGNED,
      allowNull: true
    },
    currentStatus: {
      type: Sequelize.INTEGER.UNSIGNED,
      min: 1,
      max: 9,
      allowNull: false
    },
  });
};
```

Listing 28. Model Order

```

const Sequelize = require('sequelize');

module.exports = function(sequelize, DataTypes) {
  return sequelize.define("OrderStatusHistory", {
    id: {
      type: Sequelize.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true
    },
    changeDate: {
      type: Sequelize.DATE,
      allowNull: false,
      defaultValue: Sequelize.NOW
    },
  });
};

```

Listing 29. Model OrderStatusHistory

```

const Sequelize = require('sequelize');

module.exports = function(sequelize, DataTypes) {
  return sequelize.define("Role", {
    id: {
      type: Sequelize.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true
    },
    name: {
      type: Sequelize.STRING(50),
      allowNull: false
    },
  });
};

```

Listing 30. Model Role

```

const Sequelize = require('sequelize');

module.exports = function(sequelize, DataTypes) {
  return sequelize.define("Status", {
    id: {
      type: Sequelize.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true
    },
    name: {
      type: Sequelize.STRING(50),
      allowNull: false
    },
  });
};

```

Listing 31. Model Status

```

const Sequelize = require('sequelize');
const bcrypt = require('bcrypt');

module.exports = function(sequelize, DataTypes) {
  return sequelize.define("User", {
    id: {
      type: Sequelize.INTEGER,
      allowNull: false,
      primaryKey: true,
      autoIncrement: true,
      unique: true
    },

    name: {
      type: Sequelize.STRING(100),
      allowNull: false
    },

    password: {
      type: Sequelize.STRING(200),
      allowNull: false
    },

    firstName: {
      type: Sequelize.STRING(200),
      allowNull: false
    },

    lastName: {
      type: Sequelize.STRING(200),
      allowNull: false
    },

    companyName: {
      type: Sequelize.STRING(200),
      allowNull: true
    },

    nip: {
      type: Sequelize.DECIMAL(10, 0).UNSIGNED,
      allowNull: true
    },

    isActive: {
      type: Sequelize.BOOLEAN,
      allowNull: false
    }
  }, {
    hooks: {
      beforeSave: async function(user, options) {
        if (user.changed('password')) {
          const salt = await bcrypt.genSalt(10);
          user.password = await bcrypt.hash(user.password, salt);
        }
      }
    }
  });
};

```

Listing 32. Model User

W modelu User warto zwrócić uwagę na metodę `beforeSave`, która ma za zadanie zaszyfrowanie hasła użytkownika z użyciem `bcrypt`. Jest ona wywoływana przed zapisem użytkownika w bazie.

8.2.4 jwtAuth.js

Plik `jwtAuth.js` zawiera metodę wywoływaną jako middleware przy obsłudze żądań HTTP przychodzących z części frontendowej. Następuje w niej sprawdzenie, czy dostarczony w nagłówku żądania token JWT jest prawidłowy (ponieważ po stronie backend mamy dostęp do klucza tajnego – można zweryfikować token). W metodzie sprawdzane jest również czy użytkownik jest zablokowany. W przypadku stwierdzenia, że token jest nieprawidłowy lub użytkownik jest zablokowany wysyłana jest odpowiedź ze statusem 401 i informacją o nieautoryzowanym dostępie. Jeżeli dostarczony token jest prawidłowy dane użytkownika są pobierane z bazy i dodawane do żądania (będą wykorzystywane w metodach obsługujących żądania). Listnig pliku `jwtAuth` został przedstawiony poniżej:

```
const jwt = require('jsonwebtoken')
const secretKey = require('../secret/key');
const db = require('../database/db');

const jwtAuth = async(req, res, next) => {
  try {
    const token = req.header('Authorization').split(' ')[1];
    const data = jwt.verify(token, secretKey);
    const user = await db.User.findByPk(data.id);

    if (!user || !user.isActive) {
      throw new Error();
    }
    req.user = user;
    next();
  } catch (error) {
    res.status(401).send('Nieautoryzowany dostęp')
  }
}
module.exports = jwtAuth
```

Listing 33. Plik `jwtAuth.js`

8.2.5 Helpers

Backend zawiera kilka pomocniczych skryptów js. Są to dwie mapy: przyporządkowująca statusowi zlecenia odpowiednią wartość numeryczną i przyporządkowująca roli użytkownika odpowiednią wartość numeryczną oraz metoda `canStatusBeChanged` przyjmująca trzy argumenty: rolę użytkownika, stary status zlecenia i nowy status zlecenia, której listing zamieszczono poniżej, sprawdzająca, czy dany użytkownik może dokonać odpowiedniej zmiany statusu.

```
const STATUSES = require('../helpers/statuses')
const ROLES = require('../helpers/roles')

const canStatusBeChanged = function (role, oldStatus, newStatus){
  switch (role) {
    case ROLES.ADMIN:
      return true;
      break;

    case ROLES.MAGAZYNIER:
      if (oldStatus === STATUSES.WYSLANE && newStatus === STATUSES.NA_MAGAZYNIE)
        return true;
      if (oldStatus === STATUSES.WYKONANE && newStatus === STATUSES.DO_ODBIORU)
```

```

        return true;
        if (oldStatus === STATUSES.DO_ODBIORU && newStatus === STATUSES.ZAKONCZONE)
            return true;
        break;

    case ROLES.TECHNOLOG:
        if (oldStatus === STATUSES.ZLOZONE && newStatus == STATUSES.ZATWIERDZONE)
            return true;
        if (oldStatus === STATUSES.ZLOZONE && newStatus == STATUSES.ANULOWANE)
            return true;
        if (oldStatus === STATUSES.ZATWIERDZONE && newStatus === STATUSES.ANULOWANE)
            return true;
        if (oldStatus === STATUSES.NA_MAGAZYNIE && newStatus === STATUSES.W_PRODUKCJI)
            return true;
        if (oldStatus === STATUSES.W_PRODUKCJI && newStatus === STATUSES.WYKONANE)
            return true;
        break;

    case ROLES.KLIENT:
        if (oldStatus === STATUSES.ZLOZONE && newStatus === STATUSES.ANULOWANE)
            return true;
        if (oldStatus === STATUSES.ZATWIERDZONE && newStatus === STATUSES.ANULOWANE)
            return true;
        if (oldStatus === STATUSES.ZATWIERDZONE && newStatus === STATUSES.WYSLANE)
            return true;
        break;

    default:
        return false;
        break;
}

}

module.exports = canStatusBeChanged

```

Listing 34. Metoda CanStatusBeChanged

8.2.6 Kontroler orders.js

Plik orders.js zawiera metody obsługi żądań HTTP dla endpointów związanych z zamówieniami. Zawarte są w nim trzy metody odpowiedzialne za:

- `getOrders`: pobranie z bazy listy zleceń i przekazanie jej w odpowiedzi,
- `getOrderHistory`: pobranie z bazy historii zlecenia i przekazanie jej w odpowiedzi,
- `addOrder`: dodanie nowego zamówienia do bazy

Każda metoda zawiera logikę odpowiedzialną za autoryzację operacji dla użytkownika wysyłającego żądanie. W przypadku niepowodzenia autoryzacji wysyłana jest odpowiedź ze statusem 500. Listing pliku orders.js zamieszczono poniżej:

```

const express = require('express');
const router = express.Router();
const db = require('../database/db');
const jwtAuth = require('../middleware/jwtAuth');
const STATUSES = require('../helpers/statuses');
const ROLES = require('../helpers/roles');

const addOrder = async (req, res) => {
  try {
    if (req.user.RoleId !== ROLES.KLIENT) {
      return res.status(500).send('Wyłącznie klient może dodać zlecenie!')
    }
    const newOrder = await db.Order.create({
      ordererId: req.user.id,
      rawMaterialName: req.body.rawMaterialName,
      orderDate: Date.now(),
      currentStatus: STATUSES.ZLOZONE,
      rawMaterialMass: req.body.rawMaterialMass,
      rawMaterialPallets: req.body.rawMaterialPallets
    });
    await db.OrderStatusHistory.create({
      changedByUserId: req.user.id,
      changeDate: Date.now(),
      OrderId: newOrder.id,
      StatusId: newOrder.currentStatus
    });
    return res.status(200).send('Pomyślnie dodano zamówienie');
  } catch (error) {
    return res.status(500).send(error);
  }
}

const getOrders = async (req, res) => {
  try {
    if (req.user.RoleId === ROLES.KLIENT) {
      const orders = await db.Order.findAll({
        where: {
          ordererId: req.user.id
        },
        attributes: {
          exclude: ['createdAt', 'updatedAt']
        },
        include: [{ model: db.User, as: 'User', attributes: ['companyName'] }]
      });
      return res.status(200).send(orders);
    }
    else {
      const orders = await db.Order.findAll({
        attributes: {
          exclude: ['createdAt', 'updatedAt']
        },
        include: [{ model: db.User, as: 'User', attributes: ['companyName'] }]
      });
      return res.status(200).send(orders);
    }
  } catch (error) {
    return res.status(500).send('Nie udało się pobrać listy zleceń');
  }
}

const getOrderHistory = async (req, res) => {
  try {
    orderToCheck = await db.Order.findOne({ where: { id: req.body.orderId } })
    if (req.user.RoleId === ROLES.KLIENT && orderToCheck.ordererId !== req.user.id) {
      return res.status(500).send('Nie masz dostępu do historii tego zamówienia');
    }
  }
}

```

```

    const orderHistory = await db.OrderStatusHistory.findAll({
      where: {
        orderId: req.body.orderId
      },
      attributes: {
        exclude: ['createdAt', 'updatedAt', 'changedByUserId']
      }
    });
    return res.status(200).send(
      {orderHistory: orderHistory,
       rawMaterialName: orderToCheck.rawMaterialName
      });
  } catch (error) {
    return res.status(500).send(error);
  }
}

router.get('/getorders', jwtAuth, getOrders);
router.post('/addOrder', jwtAuth, addOrder);
router.post('/getorderhistory', jwtAuth, getOrderHistory);

module.exports = router;

```

Listing 35. orders.js

8.2.7 Kontroler Statuses.js

Plik statuses.js zawiera metodę obsługi żądań HTTP dla endpointu /changestatus odpowiedzialnego za zmianę statusu zlecenia.

Metoda changeStatus została przedstawiona na listingu poniżej. W celu sprawdzenia czy dany użytkownik może wykonać zmianę statusu wykorzystywana jest metoda pomocnicza canStatusBeChanged opisana w punkcie 8.2.5. Sprawdzane są również dodatkowe warunki:

- jeżeli użytkownik jest klientem, to może zmienić jedynie status swojego zlecenia,
- jeżeli użytkownik jest technologiem i zmienia status zlecenia na „wykonane” (zdaje wytworzony produkt do magazynu), to czy zostały podane w ciele żądania wymagane dane dotyczące produktu.

```

const express = require('express');
const router = express.Router();
const db = require('../database/db');
const jwtAuth = require('../middleware/jwtAuth');
const STATUSES = require('../helpers/statuses');
const ROLES = require('../helpers/roles');
const canStatusBeChanged = require('../helpers/canStatusBeChanged');

const changeStatus = async (req, res) => {
  try {
    const oldStatus = req.body.oldStatus;
    const newStatus = req.body.newStatus;
    const user = req.user;
    const orderToChange = await db.Order.findOne({ where: { id: req.body.orderId } });

    if (user.RoleId === ROLES.KLIENT && orderToChange.ordererId !== user.id) {
      return res.status(500).send('Zabronione');
    }

    if(canStatusBeChanged(user.RoleId, oldStatus, newStatus)) {
      orderToChange.currentStatus = newStatus;
    }
  }
}

```

```

    if(newStatus === STATUSES.WYKONANE) {
      if(req.body.productMass && req.body.productPallets) {
        orderToChange.productMass = req.body.productMass;
        orderToChange.productPallets = req.body.productPallets;
      } else {
        throw new error();
      }
    }
    await orderToChange.save();

    await db.OrderStatusHistory.create({
      changedByUserId: user.id,
      changeDate: Date.now(),
      OrderId: orderToChange.id,
      StatusId: newStatus
    });
    return res.sendStatus(200);
  } else {
    return res.status(500).send('Nieduane');
  }
} catch (error) {
  return res.status(500).send('Błąd');
}
}

router.post('/changestatus', jwtAuth, changeStatus);

module.exports = router;

```

Listing 36. statuses.js

8.2.8 Kontroler Users.js

Plik users.js zawiera metody obsługi żądań HTTP dla endpointów związanych z użytkownikami.

Zawarte jest w nim pięć metod odpowiedzialnych za:

- **addClient**: dodanie nowego klienta do bazy i wysłanie jego danych w ciele odpowiedzi. Wartym zaznaczenia warunkiem sprawdzanym w metodzie jest unikatowość loginu nowego klienta,
- **loginUser**: logowanie użytkownika w systemie i przesłanie w ciele odpowiedzi tokenu JWT. W tej metodzie sprawdzana jest poprawność danych logowania (porównanie hasła przesłanego z formularza we frontendzie po zahashowaniu z hasłem przechowywanym w bazie). Dodatkowo sprawdzane jest, czy użytkownik nie jest zablokowany. Jeżeli logowanie odbędzie się poprawnie w odpowiedzi na żądanie zostanie przesłany token JWT, którym użytkownik będzie potwierdzał swoją tożsamość przy wysyłaniu kolejnych żądań,
- **getAllClients**: pobranie listy klientów z bazy i przesłanie w ciele odpowiedzi. Pobrać listę klientów może jedynie użytkownik nie będący klientem,
- **getUser**: pobranie danych użytkownika z bazy i przesłanie ich w ciele odpowiedzi.
- **changeClientStatus**: zmianę statusu klienta (zablokowany/aktywny). Może to zrobić jedynie administrator.

Kod pliku users.js został przedstawiony na Listing 37:

```

const express = require('express');
const router = express.Router();
const db = require('../database/db');
const jwt = require('jsonwebtoken');
const secretKey = require('../secret/key');
const jwtAuth = require('../middleware/jwtAuth');
const ROLES = require('../helpers/roles');

const addClient = async (req, res) => {
  try {
    if (await db.User.findOne({ where: { name: req.body.username } })) {
      return res.status(500).send('Nazwa użytkownika jest już zajęta')
    }
    const newClient = await db.User.create({
      name: req.body.username,
      password: req.body.password,
      RoleId: 4,
      firstName: req.body.firstName,
      lastName: req.body.lastName,
      companyName: req.body.companyName,
      nip: req.body.nip,
      isActive: true
    });
    return res.status(200).send(newClient);
  } catch (error) {
    return res.status(500).send(error.message);
  }
}

const loginUser = async (req, res) => {
  const username = req.body.username;
  const password = req.body.password;
  const user = await db.User.findOne({ where: { name: username } });

  if(user) {
    const correct = await user.checkPassword(password);
    if (!user.isActive) {
      return res.status(500).send('Użytkownik zablokowany');
    }
    if(correct && user.isActive) {
      const accessToken = jwt.sign({ id: user.id, role: user.RoleId, name: user.name },
secretKey);
      return res.json({ accessToken });
    } else {
      return res.status(500).send('Nieprawidłowe hasło');
    }
  } else {
    return res.status(500).send('Użytkownik nie istnieje');
  }
}

const getUser = async (req, res) => {
  const userData = {
    name: req.user.name,
    RoleId: req.user.RoleId,
    firstName: req.user.firstName,
    lastName: req.user.lastName,
    companyName: req.user.companyName,
    nip: req.user.nip,
    createdAt: req.user.createdAt
  }
  return res.json(userData);
}

const getAllClients = async (req, res) => {
  try {

```

```

    if (req.user.RoleId === ROLES.KLIENT) {
      return res.status(500).send('Dostęp zabroniony');
    }

    const clients = await db.User.findAll({
      where: {
        RoleId: ROLES.KLIENT
      },
      attributes: {
        exclude: ['password', 'RoleId', 'createdAt', 'updatedAt']
      }
    });
    return res.status(200).send(clients);
  } catch (error) {
    return res.status(500).send('Wystąpił błąd przy pobieraniu listy klientów.');
```

```

  }
}

const changeClientStatus = async (req, res) => {
  try {
    if (req.user.RoleId !== ROLES.ADMIN) {
      return res.status(500).send('Dostęp zabroniony');
    }

    const client = await db.User.findOne({ where: { id: req.body.clientId } })
    client.isActive = ! client.isActive;
    client.save();
    return res.status(200).send('status zmieniony na: ' + client.isActive);
  } catch (error) {
    return res.status(500).send('Błąd serwera');
  }
}

router.post('/addClient', addClient);
router.get('/getallclients', jwtAuth, getAllClients);
router.post('/login', loginUser);
router.get('/getUser', jwtAuth, getUser);
router.post('/changeclientstatus', jwtAuth, changeClientStatus);

module.exports = router;

```

Listing 37. users.js

8.3 Podsumowanie rozdziału

W rozdziale 8 został przedstawiony sposób implementacji systemu. Opisano poszczególne składniki frontendowej i backendowej części aplikacji. Objąsnilo sposób rozwiązania problemu autentykacji i autoryzacji użytkowników oraz przedstawiono sposób działania aplikacji zamieszczając przykładowe zrzuty ekranów prezentujące realizację poszczególnych wymaganych funkcjonalności.

Podsumowanie

W pracy w kompleksowy przedstawiono sposób proces projektowania i implementacji systemu typu CRM dla firmy świadczącej działalność usługową. System został wykonany jako aplikacja internetowa z uwagi na cechujące ten rodzaj aplikacji zalety:

- brak konieczności instalacji na komputerze klienta. Jest ona uruchamiana z poziomu przeglądarki internetowej,
- gwarancję, że każdy użytkownik korzysta zawsze z najbardziej aktualnej na dany czas wersji aplikacji, co jest szczególnie ważne biorąc pod uwagę planowaną rozbudowę aplikacji o nowe funkcjonalności,
- aplikacje internetowe są niezależne od platformy sprzętowej - mogą działać pod każdym systemem operacyjnym i na każdym urządzeniu obsługującym przeglądarkę internetową.

Szczegółowa analiza istniejącego dotychczas w Zakładzie modelu obsługi zlecenia produkcyjnego umożliwiła zidentyfikowanie obszarów wymagających usprawnienia. Poprawiony i uproszczony nowy model obsługi zlecenia ułatwił specyfikację wymagań zaprojektowanego Systemu. Dzięki dokładnej specyfikacji wymagań i funkcjonalności Systemu późniejszy proces implementacji przebiegł bezproblemowo i sprawnie, co tylko potwierdza jak ważny jest etap projektowania w pracy programisty.

Duży nacisk został położony na bezpieczeństwo aplikacji – zaimplementowano autentykację użytkowników z użyciem JSON Web Token. Wprowadzono autoryzację na podstawie roli użytkownika. Szczególną uwagę poświęcono bezpieczeństwu przechowywanych w bazie danych haseł, rozwiązując ten problem przez szyfrowanie ich przy użyciu bcrypt.

Końcowym efektem pracy jest działająca aplikacja internetowa oferująca wymagane funkcjonalności, która po wdrożeniu do użytkowania powinna usprawnić pracę Zakładu.

W zakresie pracy leżało opracowanie aplikacji oferującej podstawowe funkcjonalności. Zastosowane technologie umożliwiają łatwą rozbudowę aplikacji o dodatkowe możliwości, w kolejnych iteracjach systemu można na przykład:

- dodać bardziej zaawansowane możliwości filtrowania i wyszukiwania zleceń na liście zleceń,
- wprowadzić możliwość interaktywnej komunikacji na linii Klient – Zakład, dodając do aplikacji moduł chatu,
- jeżeli wystąpi taka konieczność – wprowadzić możliwość dodawania kolejnych użytkowników o rolach Technolog i Magazynier do systemu z poziomu konta Administratora (Z uwagi na to, że obecnie w Zakładzie do obsługi zleceń są wyznaczone pojedyncze osoby pełniące te funkcje, jedno konto na rolę, tworzone przy zakładaniu bazy danych, jest wystarczające).

Po wdrożeniu aplikacji do użytku w obecnej formie i jej faktycznym wykorzystywaniu i testowaniu przez poszczególnych użytkowników z pewnością, na zasadzie sprzężenia zwrotnego, dostarczą oni informacji o możliwościach dalszej rozbudowy.

Bibliografia

- Czech, S. (2017, 02 20). *Cechy dobrego Restful API*. Pobrano z lokalizacji sebastianczech.com: <http://sebastianczech.com/2017/02/20/cechy-dobrego-restful-api/>
- Domantas, G. (2019, 11 25). *What is HTML? The Basics of Hypertext Markup Language Explained*. Pobrano z lokalizacji hostinger.com: <https://www.hostinger.com/tutorials/what-is-html>
- ExpressJS - Overview*. (2020). Pobrano z lokalizacji tutorialspoint.com: https://www.tutorialspoint.com/expressjs/expressjs_overview.htm
- Gutkowski, J. (2014, 05 26). *Co to jest NPM?* Pobrano z lokalizacji blog.gutepk.pl: <https://blog.gutepk.pl/2014/05/26/co-to-jest-npm/>
- Hoyos, M. (2018, 12 24). *What is an ORM and Why You Should Use it*. Pobrano z lokalizacji blog.bitscr.io: <https://blog.bitscr.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a>
- HTTP status codes and their meaning*. (2019, 05 19). Pobrano z lokalizacji ionos.com: <https://www.ionos.com/digitalguide/hosting/technical-matters/the-most-important-http-status-codes-at-a-glance/>
- Kobylińska, U. (2014). Ewolucja czy rewolucja? Zmiany w standardzie ISO 9001:2015. *Economics and Management*(1), 205 - 219.
- Krawczyk, H. (2014, 11 01). *Projektowanie aplikacji użytkowych*. Pobrano z lokalizacji repository.os.niwa.gda.pl: https://repository.os.niwa.gda.pl/bitstream/handle/niwa_item/106/Projektowanie%20aplikacji%20u%C5%BCytkowych.pdf?sequence=1&isAllowed=y
- Lazaris, L. (2019, 09 10). *10 best CSS frameworks in 2020*. Pobrano z lokalizacji creativebloq.com: <https://www.creativebloq.com/features/best-css-frameworks>
- Maki, M. (2017, 12 04). *A Brief Overview of React Router and Client-Side Routing*. Pobrano z lokalizacji medium.com: <https://medium.com/@marcellamaki/a-brief-overview-of-react-router-and-client-side-routing-70eb420e8cde>
- Malik, S. i Przewodnik, K. (2020, 05 12). *Encyklopedia Zarządzania - JavaScript*. Pobrano z lokalizacji mfiles.pl: <https://mfiles.pl/pl/index.php/JavaScript>
- Mansfeld, P. (2020, 05 07). *JSON – co to jest i dlaczego warto używać?* Pobrano z lokalizacji mansfeld.pl: <https://mansfeld.pl/programowanie/json-co-to-jest-czy-warto-uzywac/>
- Mansweld, P. (2020, 02 17). *Co to jest: Protokół HTTP*. Pobrano z lokalizacji mansfeld.pl: <https://mansfeld.pl/co-to-jest/protokol-http/>
- Miłosz, M. (2008). *Aplikacje internetowe - od teorii do praktyki*. Pobrano z lokalizacji http://pti.cs.pollub.pl: <http://pti.cs.pollub.pl/wp-content/uploads/2018/06/Aplikacje-internetowe-2008.pdf>
- Morris, S. (2020). *Tech 101: What is React JS?* Pobrano z lokalizacji skillcrush.com: <https://skillcrush.com/blog/what-is-react-js/>
- Niezabitowski, M. (2018, 12 19). *Czym jest CORS (Cross-Origin Resource Sharing) i jak wpływa na bezpieczeństwo*. Pobrano z lokalizacji sekurak.pl: <https://sekurak.pl/czym-jest-cors-cross-origin-resource-sharing-i-jak-wplywa-na-bezpieczenstwo/>

- Nowak, M. (2019, 06 05). *Choosing a Technology Stack for Building Your Web Application*. Pobrano z lokalizacji www.monterail.com: <https://www.monterail.com/blog/web-development-technology-stack>
- Nowicki, T. i Wrzosek, E. (2010). Modelowanie, symulacja i analiza systemów klasy klient-serwer. *Symulacja w Badaniach i Rozwoju*, strony 265 - 277.
- Prajapati, B. (2020, 04 24). *Why is ReactJS Gaining So Much Popularity?* Pobrano z lokalizacji medium.com: <https://medium.com/devtechtoday/why-is-reactjs-gaining-so-much-popularity-6af4c43a3236>
- Rodwald, P. i Biernacik, B. (2018). Zabezpieczanie haseł w systemach informatycznych. *Biuletyn WAT, Vol. IXVii, nr 1*, strony 73- 92.
- Sajdak, M. (2018, 06). *(Nie) bezpieczeństwo JWT (JSON Web Token)*. Pobrano z lokalizacji sekurak.pl: <https://sekurak.pl/jwt-security-ebook.pdf>
- Schalau, M. (2014). *ASP.NET MVC Model-View-Controller*. Pobrano z lokalizacji http://fizyka.umk.pl/~jacek/dydaktyka/net/referaty/2014L_asp_net_mvc/2014-03-14_MSchalau_MVC4_KontrolerWidok.pdf
- Sienkiewicz, J. i Syty, P. (2008). *Architektura warstwowa aplikacji internetowych*. Pobrano z lokalizacji www.mif.pg.gda.pl: http://www.mif.pg.gda.pl/homepages/sylas/articles/art_2008_elblog.pdf
- Socha, Ł. (2012, 07 15). *Wzorce projektowe*. Pobrano z lokalizacji Łukasz Socha Web Developer: <https://lukasz-socha.pl/php/wzorce-projektowe-spis-tresci/>
- Stewart, L. (2020, 08 12). *courseport.com*. Pobrano z lokalizacji Front End vs Back End Development: <https://www.coursereport.com/blog/front-end-development-vs-back-end-development-where-to-start>
- Turlejska, H. (2014). *Zasady systemu HACCP oraz GMP/GHP w zakładach produkcji i obrotu żywnością oraz żywnienia zbiorowego*. Pobrano z lokalizacji psse.lubin.ibip.wroc.pl: <http://psse.lubin.ibip.wroc.pl/public/?id=70857>
- Tyler, J. (2018, 11 15). *What is Sequelize?* Pobrano z lokalizacji [@julianam.tyler/what-is-sequelize-a5cbb2d263c3](https://medium.com)
- Vavatech - MySQL. (2020). Pobrano z lokalizacji vavatech.pl: <https://vavatech.pl/technologie/bazy-danych/mysql>
- vavatech.pl - Node.js. (2020). Pobrano z lokalizacji vavatech.pl: <https://vavatech.pl/technologie/frameworki/nodejs>
- Wawak, S. (2020, 05 19). *Aplikacja internetowa*. Pobrano z lokalizacji <https://mfiles.pl>: https://mfiles.pl/pl/index.php?title=Aplikacja_internetowa&oldid=108883
- Wrzesień, D. (2018, 05 21). *Fetch API*. Pobrano z lokalizacji devenv.pl: <https://devenv.pl/fetch-api/>
- Zastosowanie aplikacji internetowych (webowych)*. (2020). Pobrano z lokalizacji WebDevExperts: <https://stronywww.pro/zastosowanie-aplikacji-internetowych-webowych>
- Zemczak, K. (2018, 11 22). *Wzorzec projektowy MVC*. Pobrano z lokalizacji Biegający Programista: <https://biegajacyprogramista.pl/2018/11/22/wzorzec-projektowy-mvc/>

Spis ilustracji

rys. 1. Wymiana informacji w dotychczasowym modelu	10
rys. 2. Atinea - dodawanie nowego zlecenia	11
rys. 3. Atinea - lista zleceń	12
rys. 4. System obsługi zleceń PL+ - ewidencja zleceń.....	13
rys. 5. System obsługi zleceń PL+ - edycja zlecenia	13
rys. 6. InPost - śledzenie przesyłki	14
rys. 7. Wymiana informacji w nowym modelu	15
rys. 8. Diagram stanów zlecenia.....	17
rys. 9. Budowa aplikacji inernetowej	22
rys. 10. Diagram MVC	23
rys. 11. Schemat prostej bazy danych z jedną relacją	25
rys. 12. Schemat bazy danych Systemu Obsługi Zleceń.....	36
rys. 13. Widok komponentu HeaderPanel.....	38
rys. 14. Widok komponentu FooterPanel	39
rys. 15. Widok komponentu NotFound	39
rys. 16. Widok komponentu Dashboard.....	40
rys. 17. Widok ekranu rejestracji nowego klienta	45
rys. 18. Widok komponentu Login z komunikatem błędu	48
rys. 19. Widok menu klienta - komponent ClientMenuPanel	50
rys. 20. Widok menu użytkownika wewn. Zakładu - komponent CompanyMenuPanel	50
rys. 21. Komponent AboutMe.....	52
rys. 22. Widok listy klientów dla administratora	57
rys. 23. Widok listy zleceń dla zalogowanego Technologa	65
rys. 24. Widok pojedynczego zlecenia dla zalogowanego Administratora.....	65

Spis tabel

Tabela 1. Przejścia między stanami zlecenia	18
---	----

Spis listingów

Listing 1. Emp - model Sequelize	26
Listing 2. Dept - model Sequelize	26
Listing 3. Prosty dokument HTML	31
Listing 4. Przykładowy styl CSS	32
Listing 5. Komponent HeaderPanel	38
Listing 6. Komponent FooterPanel	39
Listing 7. Komponent NotFound	39
Listing 8. Komponent Dashboard	40
Listing 9. Komponent MainPanel	40
Listing 10. Plik index.js	41
Listing 11. Komponent ProtectedRoute	42
Listing 12. Komponent AddUser - bez metody render()	43
Listing 13. Komponent AddUser - metoda render	44
Listing 14. Komponent Login	47
Listing 15. Komponent MenuPanel	49
Listing 16. Komponent ClientMenuPanel	50
Listing 17. Komponent AboutMe	52
Listing 18. Komponent ClientsList	54
Listing 19. Komponent Client	55
Listing 20. Komponent ClientAction	57
Listing 21. Komponent OrdersList	60
Listing 22. Komponent Order	61
Listing 23. Komponent StatusAction	64
Listing 24. Komponent OrderList	67
Listing 25. Widok historii zmian statusów zlecenia	67
Listing 26. Plik Index.js	68
Listing 27. Plik db.js	69
Listing 28. Model Order	70
Listing 29. Model OrderStatusHistory	71
Listing 30. Model Role	71
Listing 31. Model Status	71
Listing 32. Model User	72
Listing 33. Plik jwtAuth.js	73
Listing 34. Metoda CanStatusBeChanged	74
Listing 35. orders.js	76
Listing 36. statuses.js	77
Listing 37. users.js	79