

ZACHODNIOPOMORSKI UNIWERSYTET TECHNOLOGICZNY
WYDZIAŁ ELEKTRYCZNY
INSTYTUT AUTOMATYKI I ROBOTYKI

MACIEJ DYSPUT

PRACA DYPLOMOWA INŻYNIERSKA

Aplikacja webowa do komunikacji tekstowej człowiek-maszyna

Promotor: dr Przemysław Włodarski

Szczecin, 2023

Spis treści

1. Wstęp	3
1.1. Omówienie	3
1.2. Abstract	3
1.3. Wstęp	3
1.4. Historia	4
1.5. Streszczenie pracy	4
2. Maszyna do komunikacji	6
2.1. Raspberry Pi 2 Model B	6
2.2. Złącze GPIO	7
2.3. Płytki Prototypowa	10
3. Konfiguracja Raspberry Pi z komputerem	13
3.1. Statyczny adres IP	13
3.2. Protokół SSH	13
4. Komunikacja klient-serwer	15
4.1. Socket.IO	15
4.2. Klient	15
4.3. Serwer	16
4.4. Komunikacja klient-server	16
5. Implementacja	18
5.1. Środowisko programistyczne	18
5.2. Język JavaScript	18
5.3. Node.js	19
5.4. React.js	20
5.5. Wdrożenie aplikacji	21
5.6. Model drzewa programu	21
5.7. Implementacja strony klienta	23
5.7.1. Komponent logowania	24
5.7.2. Komponent komunikatora	26
5.8. Zarządzanie ścieżką nawigacji	31
5.9. Implementacja strony serwera	32
5.10. Obsługa zdarzeń	34
5.10.1. Zdarzenia nasłuchiwanie	34
5.11. Translacja komend do komunikacji z mikrokontrolerem	37

5.12.	Implementacja działania elementów	39
5.12.1.	Włączenie diody LED	39
5.12.2.	Wyłączenie diody LED	40
5.12.3.	Włączenie wszystkich diod	40
5.12.4.	Wyłączenie wszystkich diod	41
5.12.5.	Naprzemienne włączanie diod	41
6.	Obsługa	42
6.1.	Obsługa aplikacji	42
6.2.	Zaimplementowane polecenia	42
6.3.	Zarządzanie komunikatorem	44
6.4.	Przedstawienie uzyskanych wyników	52
7.	Zakończenie	59
7.1.	Zakończenie	59

1. Wstęp

1.1. Omówienie

Przedmiotem niniejszej pracy jest implementacja aplikacji webowej do komunikacji tekstowej człowieka z maszyną. Kluczowym jej elementem jest możliwość translacji komend wpisanych w języku naturalnym na odpowiednie funkcje na mikrokontrolerze w aplikacji będącej komunikatorem. W ramach wykonania pracy stworzono odpowiedni obwód na płytce prototypowej zawierająca potrzebne elementy elektroniczne takie jak diody czy serwomechanizm oraz możliwość wykonywania operacji na tych elementach za pomocą aplikacji.

1.2. Abstract

The subject of this thesis is the implementation of a web application for human-machine text communication. The key feature is the ability to translate commands written in a natural language into appropriate functions on the microcontroller in the communicator application. As part of the work, an appropriate circuit was created on the breadboard containing the necessary electronic elements such as diodes or a servo and the ability to perform operations on these elements using the application.

1.3. Wstęp

Komunikacja tekstowa pomiędzy człowiekiem, a maszyną to proces, który polega na komunikowaniu się poprzez wysyłanie i odbieranie tekstu przez obie strony. Komunikacja tekstowa w zależności od narzędzia może odbywać się na różne sposoby. W przypadku stosowania komputerów bądź laptopów, komunikacja tekstowa może odbywać się za pomocą klawiatury. Komunikacja tekstowa pomiędzy człowiekiem, a maszyną może być używana do różnych celów, takich jak wprowadzenie danych do systemu, komunikowanie się z innymi użytkownikami, a także do kontrolowania urządzeń. W komunikacji tekstowej z maszyną, człowiek może wysyłać polecenia, a maszyna może odpowiadać na nie za pomocą odpowiednich działań. Ukazane rozwiązanie zakłada otrzymanie odpowiedzi w

postaci odpowiednich działań z strony maszyny jaką jest Raspberry Pi poprzez wpisanie odpowiednich komend przez człowieka w aplikacji webowej.

1.4. Historia

Początki komunikacji tekstowej pomiędzy człowiekiem, a maszyną sięgają kilkudziesięciu lat wstecz. Ich długa historia sięga od czasów początków komputera kiedy to maszyny te były obsługiwane przez operatorów. Komunikowali się oni z komputerem poprzez wpisywanie różnych danych za pomocą klawiatury. W późniejszych latach XX wieku pojawiły się różnego rodzaju systemy komunikacji tekstowej jak poczta elektroniczna, która zastępowała tradycyjną pocztę, a także komunikatory tekstowe, które ułatwiały komunikację pomiędzy ludźmi. Również komunikacja tekstowa pomiędzy człowiekiem, a maszyną znalazła swoje zastosowanie w różnych działach automatyki. Obejmowało to różnego rodzaju komunikacji, takie jak polecenia, pytania, odpowiedzi i komunikaty błędów. W automatyce, komunikację tekstową zaczęto stosować przede wszystkim by umożliwić komunikację ludziom z maszynami, które brały udział w różnych procesach przemysłowych czy wytwórczych. Do przesłanej wiadomości bądź żądania do maszyny, człowiek uzyskiwał określoną informację zwrotną. Komunikacja ta najczęściej stosowana była za pomocą interfejsów tekstowych lub graficznych. W automatyce przemysłowej komunikacja tekstowa jest ważnym narzędziem do zarządzania i sterowania procesami, monitorowania stanu maszyn a także samej kalibracji urządzeń i dokonywania wymaganych prac diagnostycznych. Informacje te przeważnie przekazywane są w postaci języka naturalnego. Za pomocą komunikacji tekstowej, maszyna jest w stanie rozpoznać i zrozumieć wprowadzony przez nas tekst i zamienić go na konkretne działania. Komunikacja tekstowa pomiędzy człowiekiem, a maszyną do dnia dzisiejszego ma szerokie zastosowanie. Umożliwia ona ludziom przesyłanie danych bądź korzystanie z różnych aplikacji i usług online takie jak czaty czy forum internetowe. Komunikacja tekstowa jest dostępna w każdym miejscu i o każdej porze tylko wymagany jest dostęp do urządzenia. Główną z zalet takiej komunikacji jest brak barier językowych. Komunikacja tekstowa umożliwia komunikację z maszynami, które są programowane do obsługi wielu języków przez co przekazywanie informacji może odbywać się z osobami posługującymi się innymi językami.

1.5. Streszczenie pracy

Aplikacja webowa do komunikacji tekstowej człowiek-maszyna jest aplikacją czasu rzeczywistego. Składa się ona z dwóch interfejsów użytkownika. Pierwszym

interfejsem jest okno dołączenia do czatu. Składa się z dwóch pól tekstowych do których możemy wprowadzić naszą nazwę użytkownika i pokój do którego chcemy dołączyć. Również mamy przycisk dołączenia, który po wprowadzeniu danych przeniesie nas do wybranego przez nas pokoju. Nie jest ograniczona liczba użytkowników korzystających z aplikacji. Drugim interfejsem użytkownika jest komunikator. Możemy tam wpisywać komendy, które pozwolą nam na komunikację z maszyną jaką jest Raspberry Pi. Za pomocą odpowiednich działań maszyna będzie nam odpowiadać na nasze żądania napisane w języku naturalnym. Interfejs ten składa się z takich komponentów jak pole tekstowe umożliwiające wpisanie wiadomości, okienko czatu w którym widzimy wpisane przez nas komendy, a także przycisk wysyłania komend.

2. Maszyna do komunikacji

Celem rozdziału jest przedstawienie stanowiska, które pozwoli nam na rozwiązanie problemu zawartego w temacie pracy.

2.1. Raspberry Pi 2 Model B

W aplikacji jako maszynę przyjęto mikrokomputer Raspberry Pi 2 Model B. Model B to druga generacja Raspberry Pi, który opiera się na układzie system-on-chip BCM2836. Raspberry Pi 2 może być używany do przetwarzania tekstu czy programowania. W porównaniu do starszej wersji Modelu B+ ma procesor ARMv7 co oznacza możliwość obsługi pełnej gamy dystrybucji ARM GNU/Linux, w tym Snappy Ubuntu Core, a także Windows 10. Pi 2 wykorzystuje płytkę drukowaną rozmiaru 3,37 x 2,13 x 0,67 cala ważącą tylko 35 gramów. Są to mniej więcej rozmiary karty kredytowej. Obsługa Raspberry Pi ogranicza korzystanie pod względem systemu operacyjnego. Dostępny jest tylko Linux. Obecnie powszechnym i stosowanym systemem operacyjnym jest Windows, dlatego Raspberry Pi zmusza nas do poznania tajników wiersza poleceń Linuxa. W tym celu Fundacja Raspberry Pi stworzyła menadżer instalacji o nazwie NOOBS – New Out Of the Box Software. Instalator NOOBS zawiera również różne systemy operacyjne innych firm, w tym OpenELEC oraz Windows 10 IoT Core firmy Microsoft. Ra-



Rysunek 2.1: Raspberry Pi 2 Model B.

Tabela 2.1: Cechy urządzenia Raspberry Pi





















Czterordzeniowy procesor ARM Cortex-A7 900 MHz
1GB RAM
Rdzeń graficzny VideoCore IV 3D
Port Ethernet
Cztery porty USB i możliwość podłączenia klawiatury i myszki
Pełnowymiarowe wyjście HDMI i możliwość podłączenia się do monitora
Czterobiegunowe gniazdo 3,5mm z wyjściem audio i kompozytowym wyjściem wideo
40 stykowe złącze GPIO z męskimi stykami, które są kompatybilne z żeńskimi złączami
Interfejs kamery (CSI)
Interfejs wyświetlacza (DS1)
Gniazdo karty micro SD

Raspberry Pi 2 nie ma wbudowanej pamięci co wiąże się z ograniczeniami do karty microSD i dowolnej dołączonej pamięci. Raspberry Pi 2 ma tylko jeden czytnik karty microSD przez co jest ograniczony do korzystania z tylko jednego systemu operacyjnego.

2.2. Złącze GPIO

Raspberry Pi jest przystosowane do pracy z napięciem 3,3V oraz z maksymalnym obciążeniem prądowym równym 16mA. Podanie wyższego napięcia może wiązać się z uszkodzeniem mikrokomputera. Płytkę zawiera 40-pinowe złącze. Piny numer 2 i 4 podłączone są do 5V. Jeżeli dokonamy zwarcia tych pinów z innymi pinami możemy doprowadzić do uszkodzenia płytki Raspberry Pi. Piny 6,9,14,20,25,30,34 oraz 39 podłączone są do masy. Raspberry Pi posiada dwukierunkowe piny wejście/wyjście. Sterowanie liniami wejście/wyjście wymaga zaprogramowania w wybranym przez nas języku. Istnieją dwa różne schematy numeracji pinów. Piny GPIO są cyfrowymi pinami co znaczy, że posiadają stan on – włączenia oraz off – wyłączenia. Za pomocą pinów masy – GND, jesteśmy w stanie zmierzyć wszystkie napięcia i zakończyć obwód elektryczny. Pin GND traktuje się jako nasz punkt zerowy i podczas projektowania układów ważną jest rzeczą skorzystanie z pinu uziemiającego przed podaniem napięcia na obwód. Dzięki takiej operacji unikniemy uszkodzenie najbardziej wrażliwych elementów w układzie. Przy podłączeniu dowolnego komponentu ze źródłem zasilania i uziemieniem, element staje się częścią naszego obwodu dzięki czemu poprzez przepływ jesteśmy uzyskać wymagany przez nas efekt. W przypadku podłączenia diody LED wytworzymy światło. Piny 3V zapewniają zasilanie naszego układu

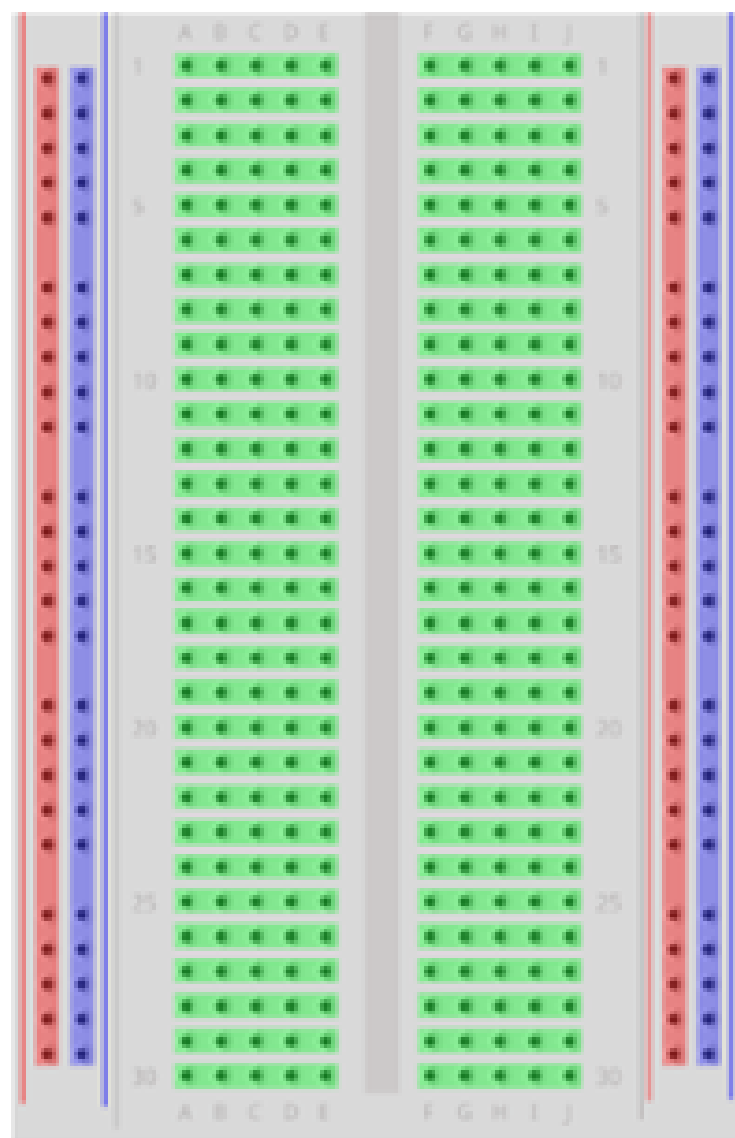
napięciem 3,3V. Piny te mają specjalne zastosowanie. Przy podłączeniu diody LED do GPIO 3V możemy sprawdzić czy nasz układ został prawidłowo podłączony i czy dioda świeci. Należy również brać pod uwagę fakt, że pin 3V ma ograniczone możliwości prądowe. Wartość prądu jaką można pobrać z tego pinu to około 50mA co oznacza, że nie jesteśmy w stanie obsłużyć naszą płytkę dużymi prądami. Jeżeli chcemy podłączyć elementy, które wymagają większej wartości prądu, należy pomyśleć nad innym sposobem zasilania. Piny 3V zawierają również ochronę przeciwzwarciovą co oznacza, że jeżeli zostanie podłączony element, który pobiera większy prąd niż jest wymagany to zabezpieczenie automatycznie zareaguje i odciąży pin przez co unikniemy uszkodzenia naszej płytki Raspberry Pi. Pin GPIO 3V jest zasilany napięciem stałym, a nie zmiennym, dlatego nie nadaje się do sterowania takich urządzeń jak silniki czy przekaźniki. Piny 3V najlepiej sprawdzają się przy zasilaniu takich elementów jak diody LED czy czujniki, ponieważ elementy te charakteryzują się niskim poborem mocy. Piny 5V są typem pinów, które pozwalają na zasilanie urządzeń zewnętrznych za pośrednictwem płytki Raspberry Pi. Piny GPIO 5V są zasilane napięciem stałym o wartości nieprzekraczającej 5V i może dostarczać maksymalnie do 2,5A prądu. Piny 5V są również zasilane napięciem stałym, a nie zmiennym. Za pomocą tych pinów możemy zasilać urządzenia o wyższym poborze mocy. Również w złączu GPIO Raspberry Pi znajdują się piny korzystające z protokołów I2C, SPI lub UART. Magistrala I2C to cyfrowy interfejs komunikacyjny, która pozwala na przesyłanie danych pomiędzy urządzeniami za pomocą linii SDA (Serial Data) oraz SCL (Serial Clock). Pin 3 i pin 5 to dwa piny I2C. Pin SCL jest pinem zegarowym, który umożliwia synchronizację przesyłania danych pomiędzy urządzeniami, zaś pin SDA jest pinem danych, który przesyła dane pomiędzy urządzeniami za pomocą magistrali I2C. Głównym zastosowaniem tych pinów jest możliwość podłączenia takich urządzeń jak ekrany LCD. Magistrala I2C wymaga obecności przynajmniej dwóch urządzeń na magistrali, które jedno będzie jako master, a drugie jako slave. Urządzenie Master będzie wysyłało komendy do slave'a. Pin SPI umożliwia komunikację między urządzeniami za pomocą magistrali SPI. Raspberry Pi gwarantuje nam cztery piny SPI. Są to MOSI, MISO, SCLK oraz CE0 lub CE1. Pin MOSI umożliwia nam przesyłanie danych od mastera do slave'a. Pin MISO przesyła dane z slave'a do mastera. Pin SCLK jest pinem zegarowym, który umożliwia synchronizację przesyłania danych pomiędzy urządzeniami, zaś pin CE0 lub CE1 służy do włączania lub wyłączania slave'a. Pin SPI znajduje swoje zastosowanie przy zasilaniu ekranów OLED lub modułów GPS. Pin UART pozwala na komunikację między urządzeniami za pomocą portu szeregowego UART. W płytce Raspberry Pi znajdują się dwa piny UART. Pierwszy to RX (Receive), a drugi to TX (Transmit). Pin RX służy do odbierania danych z urządzenia zewnętrznego, zaś pin TX pozwala na wysyłanie danych do urządzenia zewnętrznego.

Pin#	NAME		NAME	Pin#
01	3.3v DC Power		DC Power 5v	02
03	GPIO02 (SDA1 , I2C)		DC Power 5v	04
05	GPIO03 (SCL1 , I2C)		Ground	06
07	GPIO04 (GPIO_GCLK)		(TXD0) GPIO14	08
09	Ground		(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)		(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)		Ground	14
15	GPIO22 (GPIO_GEN3)		(GPIO_GEN4) GPIO23	16
17	3.3v DC Power		(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)		Ground	20
21	GPIO09 (SPI_MISO)		(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)		(SPI_CE0_N) GPIO08	24
25	Ground		(SPI_CE1_N) GPIO07	26
27	ID_SD (I2C ID EEPROM)		(I2C ID EEPROM) ID_SC	28
29	GPIO05		Ground	30
31	GPIO06		GPIO12	32
33	GPIO13		Ground	34
35	GPIO19		GPIO16	36
37	GPIO26		GPIO20	38
39	Ground		GPIO21	40

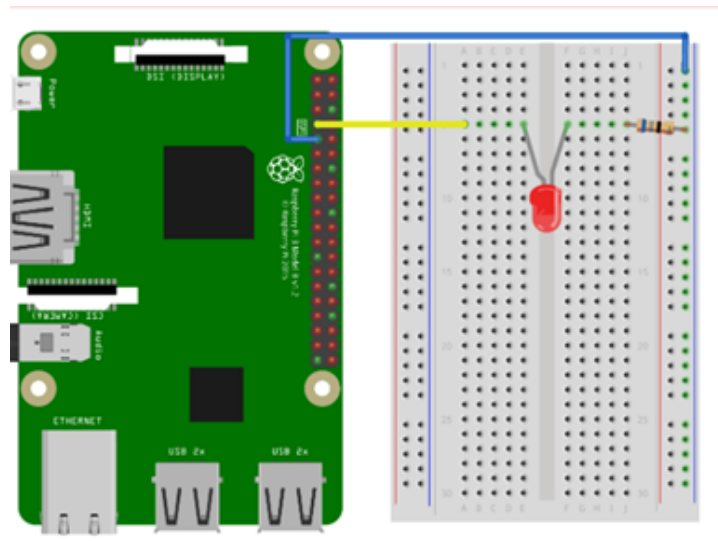
Rysunek 2.2: 40stykowe złącze GPIO Raspberry Pi 2 Model B.

2.3. Płytki Prototypowa

Czerwonym kolorem oznaczona jest szyna zasilania. Zazwyczaj jest ona używana do podłączenia zasilania do płytki prototypowej. Ze względu na to, że cała kolumna jest podłączona, możliwe jest podłączenie zasilania do dowolnie wybranego punktu w kolumnie. Linia ta służy do dostarczania zasilania do elementów elektronicznych umieszczonych na płytce, a także urządzeń zewnętrznych, w tym Raspberry Pi. W płytkach prototypowych szyny zasilania zazwyczaj oznaczone są literami „VCC” lub „+”. Raspberry Pi posiada specjalne gniazdo zasilania przeznaczone dla zasilaczy sieciowych. Następnie po podłączeniu zasilacza dostarczany jest prąd do płyty za pomocą szyn zasilania, które są umieszczone w gnieździe zasilania. Przed podłączeniem ważne jest, żeby się upewnić czy napięcie zasilania na szynie zasilania płytki prototypowej jest zgodne z napięciem zasilania Raspberry Pi oraz czy prąd zasilania nie przekracza maksymalnych wartości dopuszczalnych dla linii zasilania w Raspberry Pi. Niebieskim kolorem oznaczona jest szyna naziemna. Zazwyczaj jest ona używana do podłączenia uziemienia do płytki prototypowej. Ze względu na to, że cała kolumna jest podłączona, możliwe jest podłączenie zasilania do dowolnie wybranego punktu w kolumnie. Szyna naziemna zwykle oznaczona jest symbolem „GND” lub „-”. W płytce prototypowej, szyna naziemna jest używana do połączenia elementów elektronicznych z masą co pozwala im działać prawidłowo. W momencie gdy rezystor zostanie podłączony do szyny naziemnej i do pinu wejściowego mikrokontrolera, będzie on działał jako dzielnik napięcia co pozwoli mikrokontrolerowi odczytać wartość napięcia na wejściu. Przed podłączeniem ważne jest, żeby się upewnić czy szyna naziemna jest poprawnie podłączona, ponieważ błędne podłączenie może doprowadzić do nieprawidłowego działania elementów elektronicznych lub nawet do ich uszkodzenia. Zielonym kolorem oznaczono rzędy połączonych punktów wiązania. Wiazania każdego z tych rzędów są połączone, ale nie cały rząd. Punkty wiązania używana są najczęściej do łączenia elementów i obwodów elektronicznych. Rzędy połączonych punktów wiązania są oznaczone kolumnami liter i cyfr. Za pomocą płytki prototypowej z zastosowaniem Raspberry Pi możemy połączyć prosty układ składający się z diody LED i rezystora. Jeśli dioda jest połączona szeregowo z rezystorem, może ona służyć do ograniczenia prądu płynącego przez obwód. Raspberry Pi jest przystosowane do pracy z niskim obciążeniem prądowym. W takim przypadku dioda będzie przepuszczać prąd tylko w jednym kierunku co oznacza, że jeżeli wartość prądu będzie zbyt wysoka to dioda się zablokuje i ograniczy przepływ prądu, chroniąc pozostałe elementy obwodu przed uszkodzeniem. Ważne jest dobranie odpowiedniej wartości rezystora, aby zapobiec uszkodzeniu się diody. Wartość rezystora zależy od napięcia zasilania, napięcia przewodzenia diody LED i jej prądu. Podczas sterowania diodą LED, Raspberry Pi wysyła sygnał sterujący



Rysunek 2.3: Płytki prototypowa



Rysunek 2.4: Proste połączenie układu diody LED z rezystorem

do pinu, który jest połączony z anodą diody LED. Sygnał ten przechodzi przez rezystor ograniczający prąd, który płynie przez diodę LED. Następnie sygnał dotrze do anody diody LED co spowoduje jej zaświecenie. Kolejnym podstawowym elementem możliwym do połączenia za pomocą płytki prototypowej z zastosowaniem Raspberry Pi jest serwomechanizm. Element ten służy do precyzyjnego obracania się o określoną wartość kąta. Raspberry Pi wysyła sygnał sterujący do serwomechanizmu za pomocą odpowiedniego interfejsu. Następnie interfejs odbiera sygnał i przetwarza go na sygnał elektryczny, który wysyłany jest do serwomechanizmu dzięki któremu zaczyna się obracać o określony kąt za pomocą silnika elektrycznego i przekładni. W momencie osiągnięcia odpowiedniej wartości kąta, serwomechanizm zatrzymuje się i czeka na kolejny sygnał sterujący.

3. Konfiguracja Raspberry Pi z komputerem

Celem rozdziału jest przedstawienie możliwości konfiguracji Raspberry Pi z komputerem potrzebnej do zdalnej komunikacji między urządzeniami.

3.1. Statyczny adres IP

Adres IP jest adresem, który przypisuje do dowolnego urządzenia komputer, który się z nim łączy. Tendencją tego adresu IP jest to, że się zmienia między urządzeniami z biegiem czasu w zależności jakiego urządzenia się podłączyło. Adres IP jest potrzebny do identyfikacji urządzenia, które się podłączyło do sieci, aby ułatwić mu dostęp do korzystania z usługi. Podczas podłączenia, urządzenie otrzymuje unikalny adres IP w sieci. Naszym celem jest ustawienie takiego adresu IP urządzenia by te przy podłączeniu z routerem nie ulegało zmianie i się nie rozłączało. Do ustawienia trwałego połączenia sieciowego między Raspberry Pi, a komputerem potrzebne będzie ustawienie statycznego adresu IP. Stacyjny adres IP jest używany, aby umożliwić stałe dostęp do urządzenia z sieci lokalnej lub Internetu. Może być używany w przypadku urządzenia takiego jak Raspberry Pi, które może być uznawane jako serwer lub do innych celów wymagających stałego dostępu do sieci. Ustawianie statycznego adresu IP może być szczególnie przydatne kiedy chcemy zdalnie obsługiwać Raspberry Pi za pomocą protokołu SSH oraz udostępniać dane między tymi urządzeniami.

3.2. Protokół SSH

Jest to protokół sieciowy, który umożliwia bezpieczne połączenie się z komputerem i wykonywanie na nim zdalnie poleceń. Jest to niezbędny element przy konfiguracji systemu operacyjnego na Raspberry Pi, ponieważ umożliwia on zdalne zarządzanie urządzeniem za pomocą komputera, w którym zainstalowany jest inny system operacyjny. Za pomocą zdalnego dostępu do urządzenia jesteśmy w stanie dokonywać regularnej aktualizacji. Potrzebujemy do tego nawiązywania połączenia z urządzeniem przez sieć co protokół SSH nam zapewnia. W tej sa-

mej sieci korzystając z komputera jesteśmy w stanie uzyskać dostęp do wiersza poleceń Raspberry Pi. Zagwarantuje nam to zdalne zarządzanie mikrokomputerem. Protokół SSH jest bardzo przydatny w przypadku konfiguracji i zarządzania urządzeniami zdalnie, ponieważ umożliwia bezpieczne połączenie się z urządzeniem i wykonywanie poleceń na nim, unikając konieczności fizycznego dostępu do urządzenia. Jest również często używany do zabezpieczania połączeń sieciowych, ponieważ protokół SSH szyfruje dane przesyłane między komputerami.

4. Komunikacja klient-serwer

Celem rozdziału jest przedstawienie zależności komunikacji klient-serwer oraz ukazanie ich przeznaczenia do tworzenia aplikacji webowych.

4.1. Socket.IO

Socket.IO jest to biblioteka Javascript, która pozwala na realizację połączeń z serwerem przy użyciu WebSocket. Ten protokół sieciowy umożliwia przesyłanie danych w czasie rzeczywistym. Dane mogą być przesyłane z strony klienta na stronę serwera oraz na odwrót. Przy zastosowaniu biblioteki Socket.IO jesteśmy w stanie tworzyć aplikacje, które potrzebują wymianę danych w czasie rzeczywistym, a także wymagają nieprzerwanej komunikacji pomiędzy obiema stronami. Obie te cechy idealnie pasują do stworzenia komunikatora, w którym przesyłanie danych powinno odbywać się w czasie rzeczywistym, a także umożliwi nam nieprzerwaną komunikację z drugą stroną. Socket.IO może korzystać z metody emitowania komunikatów, które pozwolą na wysyłanie i odbieranie danych, a także nasłuchiwanie takie jak połączenie bądź rozłączenie klienta. Jeżeli chcemy by nasza aplikacja umożliwiała przesyłanie danych w czasie rzeczywistym musimy korzystać z Socket, ponieważ żądania HTTP są bardzo powolne przez co uniemożliwiają nam komunikację w czasie rzeczywistym.

4.2. Klient

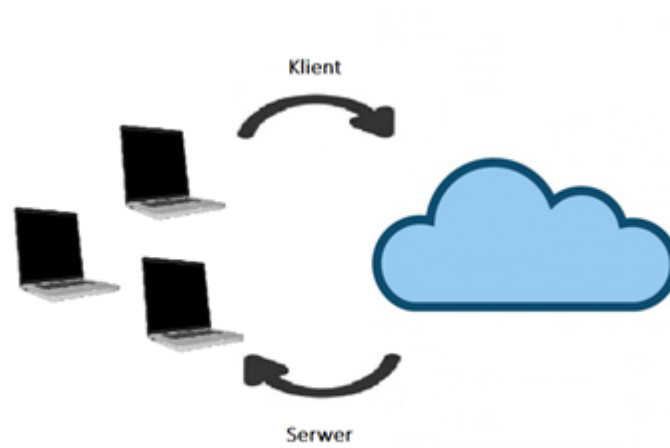
W komunikacji klient-serwer z użyciem Socket.IO, klient odpowiedzialny jest za nawiązanie połączenia z serwerem i wysyłanie oraz odbieranie danych za pomocą Socket.IO. W aplikacji może być to realizowane za pomocą skryptów Javascript, które są uruchamiane po stronie klienta. Klient może nawiązać połączenie z serwerem, ustawić odpowiednie obsługi zdarzeń czy również odbierać dane z serwera i je wysyłać. Oprócz nawiązywania połączenia i przesyłania danych, klient jest również odpowiedzialny za obsługę błędów połączenia oraz zamknięcia połączenia. Podczas komunikacji każdy klient posiada swój identyfikator gniazda.

4.3. Serwer

W komunikacji klient-serwer z użyciem Socket.IO, serwer odpowiedzialny jest za utrzymanie połączenia sieciowego z klientem i przesyłanie oraz odbieranie danych za pomocą Socket.IO. W aplikacji może być to realizowane za pomocą skryptów Javascript, które są uruchamiane na serwerze. Serwer może utworzyć instancję Socket.IO i ustawić ją na odpowiednim adresie URL, a następnie ustawić obsługę zdarzeń dla połączeń klientów. Serwer również może odbierać dane od klienta, a także wysyłać dane do klienta. Oprócz utrzymywania połączenia i przesyłania danych, serwer jest również odpowiedzialny za obsługę błędów połączenia oraz zamknięcia połączenia. Serwer może odpowiadać na żądania i przetwarzać dane przesyłane przez klienta. Każdy serwer posiada listę podłączonych gniazd, które korzystają z jego usług. Za pomocą tej listy jest w stanie wybrać identyfikator klienta, który się z nim komunikuje i przesłać mu dane.

4.4. Komunikacja klient-serwer

Komunikacja klient-serwer z użyciem Socket.IO jest stosowana w aplikacji, które wymagają szybkiej i bezstanowej wymiany danych pomiędzy klientem, a serwerem. Może to być przydatne w przypadku aplikacji, w których dane muszą być wymieniane w czasie rzeczywistym, aby zapewnić płynne działanie aplikacji. Socket.IO umożliwia tworzenie połączeń sieciowych co pozwala na elastycznie dostosowywanie sposobu komunikacji do potrzeb aplikacji. Aplikacja webowa do komunikacji tekstowej człowiek-maszyna korzysta z biblioteki Socket.IO. Umożliwia ona tworzenie aplikacji klient-serwer za pomocą połączeń sieciowych takich jak WebSockets. Biblioteka Socket.IO może być używana zarówno po stronie klienta jak i na stronie serwera. Model komunikacji klient-serwer umożliwia przesyłanie danych na różne komputery połączone w sieci. Czynność ta jest możliwa w momencie kiedy istnieją procesy wysyłające żądania czyli klienci, a także procesy, które je obsługują czyli serwery. Przy takim podziale obowiązków, aplikacja umożliwia użytkownikom dostęp do przesłanych i udostępnionych danych. Główną zaletą takiej komunikacji jest przede wszystkim obsługa dowolnej liczby klientów niezależnie od ich lokalizacji. Serwer może zostać postawiony na dowolnej maszynie w sieci i stąd będzie udostępniał wszystkie dane klientom. W aplikacji została zastosowana architektura dwuwarstwowa. Umożliwia ona korzystanie z serwera dowolnej liczbie klientów. Dwukierunkową komunikację w czasie rzeczywistym umożliwia biblioteka Socket.IO, która asynchronicznie aktualizuje przesyłane przez nas żądania do serwera.



Rysunek 4.1: Żądanie i odpowiedź komunikacji Socket.IO

5. Implementacjar

Celem rozdziału jest przedstawienie środowiska i bibliotek umożliwiających programowanie z użyciem urządzenia Raspberry Pi.

5.1. Środowisko programistyczne

W celu realizacji założeń projektowych, program został wykonany przy użyciu środowiska programistycznego Visual Studio Code. JavaScript jest językiem programowym, który jest popularny wśród programistów i istnieje wiele gotowych bibliotek i frameworków, które można wykorzystać do tworzenia aplikacji. Visual Studio Code jest środowiskiem, które ma dobrą obsługę JavaScriptu i Node.js, gwarantując narzędzia do podpowiadania składni i automatycznego uzupełniania kodu IntelliSense, a także narzędzia do debugowania i testowania aplikacji. Visual Studio Code również umożliwia korzystanie z niego na różnych systemach operacyjnych takich jak Linux czy Windows. Raspberry Pi ma wbudowany system operacyjny Linux dzięki czemu środowisko to idealnie się wkomponuje w tworzeniu aplikacji na tym urządzeniu. Aplikacja składa się z dwóch głównych elementów. Jest to serwer i klient. Część serwerowa to część aplikacji, która działa na serwerze i odpowiada za przetwarzanie i przechowywanie danych oraz obsługę żądań użytkowników. Została napisana w języku programowania JavaScript przy użyciu Node.js. Część klienta to część aplikacji, która działa na urządzeniu użytkownika i odpowiada za wyświetlenie interfejsu użytkownika oraz komunikację z serwerem. Została ona napisana również w języku programowania JavaScript przy użyciu biblioteki React.js. Visual Studio Code pozwala na łatwą obsługę i zarządzanie strukturą plików co będzie pomocne przy tak dużych zasobach drzewa programu.

5.2. Język JavaScript

JavaScript jest językiem skryptowym co oznacza, że jest on interpretowany przez przeglądarkę internetową, a nie kompilowany do binarnego kodu. JavaScript umożliwia tworzenie dynamicznych stron internetowych, a także pozwala na projektowanie interfejsów użytkownika. JavaScript można używać w różnych aplikacjach gdzie wymagane jest sterowanie sprzętem czy serwerem. Za pomocą

szerokiego asortymentu frameworków, programiści mogą tworzyć aplikacje mobilne i internetowe. Najpopularniejszymi stosowanymi bibliotekami są React.js i Node.js. JavaScript jest językiem programowania, który może być stosowany po stronie klienta gdzie znajduje się cały front-end aplikacji, ale również dzięki Node.js, JavaScript może być stosowany po stronie serwera czyli back-endu. Te dwie główne cechy sprawiają, że przy stosowaniu JavaScriptu możemy tworzyć dynamiczne i interaktywne aplikacje internetowe. Dzięki możliwości programowania Raspberry Pi za pomocą języka Javascript, jesteśmy w stanie tworzyć aplikacje i skrypty, które będą działać na tym urządzeniu. Istnieje wiele gotowych bibliotek, które pozwalają na połączenie i obsługę różnych elementów i urządzeń zewnętrznych podłączonych do Raspberry Pi. Sterowanie pinami GPIO przy użyciu języka programowania Javascript wymaga zaimportowania odpowiednich bibliotek. JavaScript również jest zgodny z innymi systemami operacyjnymi w tym również z Linuxem, który jest domyślnym systemem operacyjnym dla Raspberry Pi.

5.3. Node.js

Node.js to platforma oparta na języku JavaScript, która służy do budowy back-endu aplikacji oraz umożliwia uruchamianie kodu JavaScript poza przeglądarką internetową, np. na serwerze. Idealnie odnajduje swoje zastosowanie przy tworzeniu aplikacji sieciowych takich jak serwery internetowe. Node.js jest używany głównie do uzyskiwania dostępu do baz danych czy obsługi żądań. Programowanie stosowane w Node.js jest lekkie i wydajne co sprawia, że idealnie się odnajduje w programowaniu urządzeń o niewielkich zasobach sprzętowych jak Raspberry Pi. Node.js został zaprojektowany do wykonania optymalizacji przepustowości i skalowalności w aplikacjach internetowych przez co jest idealnym rozwiązaniem dla tworzenia aplikacji internetowych czasu rzeczywistego. Swoje szerokie zastosowanie może zawdzięczać menadżerowi pakietów węzłów npm, który zapewnia dostęp do setek tysięcy pakietów i bibliotek. Express.js to biblioteka oparta na platformie Node.js i pozwala na tworzenie aplikacji serwerowych. Jej głównym zadaniem jest udostępnianie szeregu narzędzi do obsługi protokołów sieciowych oraz umożliwienie łatwego tworzenia aplikacji serwerowych z wykorzystaniem technologii middleware. Express.js może być używany z Raspberry Pi do tworzenia aplikacji serwerowych, który umożliwia dostęp do zasobów za pośrednictwem odpowiedniego protokołu. Również express.js stosuje się do tworzenia serwera sieciowego, który umożliwi komunikację między urządzeniami zewnętrznymi. Główną funkcją, którą nam udostępnia biblioteka Express.js jest możliwość tworzenia aplikacji, które wykorzystują funkcję Raspberry Pi, takie jak porty GPIO do obsługi urządzeń zewnętrznych. Platforma Node.js udostępnia nam kilka modułów potrzebnych do obsługi urządzenia Raspberry Pi:

- **Moduł on-off**, umożliwia nam obsługę wejść/wyjść cyfrowych dla urządzeń zewnętrznych. Może być używany do sterowania np. diodami LED czy przełącznikami poprzez interfejsy takie jak GPIO lub inne interfejsy cyfrowe jak I2C lub SPI. Moduł on-off umożliwia programowi dostęp do funkcji wejść/wyjść cyfrowych systemu operacyjnego. Może być używana z systemem operacyjnym Windows czy Linux dzięki czemu idealnie odnajduje się przy programowaniu z użyciem Raspberry Pi. Moduł on-off zapewnia również rejestrowanie zdarzeń zmiany stanu wejścia i wyjścia i reagować na nie. Dzięki temu modułowi jesteśmy w stanie integrować urządzenia zewnętrzne z aplikacjami Node.js, a także tworzyć projekty z wykorzystaniem Raspberry Pi i innych urządzeń z interfejsem GPIO.
- **Moduł pigpio**, biblioteka ta korzysta z frameworka Node.js. Moduł pigpio może być używana w celu kontrolowania i monitorowania sygnałów z wejść/wyjść cyfrowych oraz modulację szerokości impulsów na Raspberry Pi. Dzięki tej bibliotece jesteśmy w stanie tworzyć projekty z użyciem urządzeń elektronicznych takie jak diody LED czy serwomechanizmy, które są podłączone do portów GPIO urządzenia Raspberry Pi. Biblioteka pigpio znalazła szerokie zastosowanie w aplikacjach gdzie wymagane jest sterowanie urządzeniami elektrycznymi lub elektronicznymi.

5.4. React.js

React.js to biblioteka Javascript, która swoje główne zastosowanie ma przy tworzeniu interfejsów użytkownika po stronie front-endu. Pozwala tworzyć dynamiczne aplikacje, w których interfejs użytkownika reaguje na zmiany wprowadzone w aplikacji. Swoje szerokie zastosowanie znajduje tam gdzie aplikacje opierają swoje funkcjonowanie na interakcjach użytkownika z elementami interfejsu użytkownika. React.js może idealnie współpracować z urządzeniem Raspberry Pi poprzez kontrolowanie urządzeń zewnętrznych podłączonych do tego urządzenia lub wyświetlania danych za pomocą odpowiedniego interfejsu użytkownika. Głównym warunkiem korzystania React.js na Raspberry Pi jest zainstalowanie odpowiedniej wersji Node.js. React.js umożliwia tworzenie interfejsu użytkownika za pomocą komponentów, które mogą być stosowane wielokrotnie przy projektowaniu różnych elementów. Główną zaletą stosowania React.js przy programowaniu z użyciem urządzenia Raspberry Pi jest tworzenie aplikacji, które muszą reagować na zmiany danych w czasie rzeczywistym.

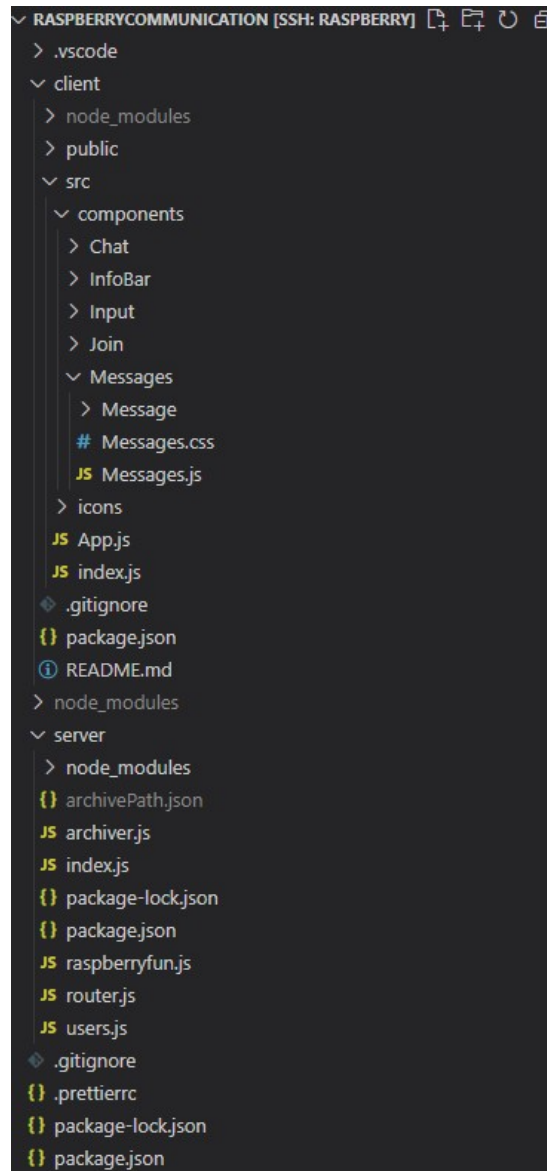
5.5. Wdrożenie aplikacji

Relacja klient-serwer jest podstawowym modelem architektury systemów komputerowych, w którym oba te podmioty są różnymi urządzeniami bądź programami. Współpracują one ze sobą w celu umożliwienia użytkownikowi wymiany danych bądź korzystania z określonych usług. Aplikacje czasu rzeczywistego mają to do siebie, że serwer zazwyczaj odpowiada za przetwarzanie i przechowywanie danych oraz udostępnianie ich klientom. Klienci są odpowiedzialni za wyświetlenie tych danych i wykorzystanie ich według własnych założeń. Strona klienta i serwera komunikują się ze sobą za pomocą protokołów sieciowych takich jak WebSockets, w celu umożliwienia przepływu danych pomiędzy nimi. Aplikacje czasu rzeczywistego są często wykorzystywane w komunikacji tekstowej, aby umożliwić użytkownikom szybką i efektywną wymianę informacji. Użytkownicy mogą szybko uzyskać oczekiwaną odpowiedź z strony serwera po przesłaniu danego żądania, nie tracąc czasu na opóźnienia w przesyłaniu informacji. Takie aplikacje często znajdują swoje zastosowanie w sytuacjach, w których szybka i dokładna wymiana danych jest istotna. W stworzonej aplikacji, kluczowym elementem przesyłania żądania do mikrokontrolera jest uzyskanie jak najszybszej odpowiedzi urządzenia w postaci odpowiedniego działania elementów zainstalowanych po stronie mikrokomputera. Komunikacja przy pomocy relacji klient-serwer idealnie odnajduje swoje zastosowanie.

5.6. Model drzewa programu

Struktura programu składa się z dwóch katalogów:

- **Klient**, odpowiedzialny za nawiązanie połączenia z serwerem oraz odbieranie i wysyłanie danych za pomocą Socket.IO. Po stronie klienta zaimplementowane są interfejsy użytkownika umożliwiające komunikację z urządzeniem.
- **Serwer**, odpowiedzialny za utrzymanie połączenia sieciowego z klientem i przesyłanie oraz odbieranie danych za pomocą Socket.IO. Serwer umożliwia odpowiadanie na żądania i przetwarzanie danych na odpowiednie czynności. Po stronie serwera zaimplementowany jest cały mechanizm umożliwiający komunikację tekstową.



Rysunek 5.1: Struktura programu

5.7. Implementacja strony klienta

Strona klienta została zaimplementowana przy użyciu biblioteki React.js, która służy przede wszystkim do tworzenia interfejsu użytkownika dla aplikacji webowych. Główną zaletą korzystania z tego modułu jest możliwość tworzenia komponentów. Odpowiadają one za wyrenderowanie i zarządzanie fragmentem stworzonego widoku. Komponenty są najważniejszym elementem projektowania oprogramowania, które opisują jak dana aplikacja ma działać i wyglądać. Są one w pełni niezależne od siebie co umożliwia ich wielokrotne użytkowanie w różnych miejscach aplikacji. Dzięki temu kod staje się bardziej modułowy i łatwiejszy do zarządzania. Komponenty są tworzone za pomocą składni JSX, która umożliwia zapisanie kodu HTML w plikach JavaScript. W celu łatwiejszej implementacji interfejsów użytkownika zostało stworzone 6 komponentów:

- **Logowanie**, komponent ten zwraca widok dołączenia użytkownika do komunikatora tekstowego. Jest on pierwszym interfejsem użytkownika podczas włączenia aplikacji.
- **Czat**, komponent ten zwraca widok komunikatora tekstowego odpowiedzialnego za wysyłanie żądań do urządzenia. Komponent ten zwraca funkcję, która zawiera elementy nawigacji, zbioru wiadomości i komunikacji i renderują widok drugiego interfejsu użytkownika po wprowadzeniu danych do dołączeniu do komunikatora
- **Nawigacja**, komponent ten zwraca widok zawierający informacje o aktualnie wybranym pokoju przez użytkownika. Wchodzi on w skład drugiego interfejsu użytkownika, który renderuje widok komunikatora
- **Komunikacja**, komponent ten zwraca widok zawierający pole tekstowe oraz przycisk wysyłania wiadomości w aplikacji. Wchodzi on w skład drugiego interfejsu użytkownika, który renderuje widok komunikatora
- **Zbiór wiadomości**, komponent zwraca widok wyrenderowanej listy wiadomości
- **Wiadomość**, komponent ten zwraca widok wyrenderowania pojedynczej wiadomości



Rysunek 5.2: Interfejs logowania

5.7.1. Komponent logowania

Komponent Logowania wyświetla formularz, w którym użytkownik może wprowadzić swoją nazwę i nazwę pokoju do którego chce dołączyć. Za pomocą przycisku „Sign In” może on przejść do komunikatora tekstowego. Nazwa użytkownika jest dowolna. Klient może dołączyć do pokoju, z którego korzystają już inni użytkownicy, a także może stworzyć swój własny oddzielny pokój.

```

1 export default function SignIn() {
2   const [name, setName] = useState('')
3   const [room, setRoom] = useState('')
4
5   return (
6     <div className="joinOuterContainer">
7       <div className="joinInnerContainer">
8         <h1 className="heading">Chat</h1>
9         <div>
10          <input
11            placeholder="Name"
12            className="joinInput "
13            type="text "
14            onChange={(event) => setName(event.target.value)}
15          />
16        </div>
17        <div>
18          <input
19            placeholder="Room"
20            className="joinInput mt-20"
21            type="text "
22            onChange={(event) => setRoom(event.target.value)}
23          />
24        </div>

```

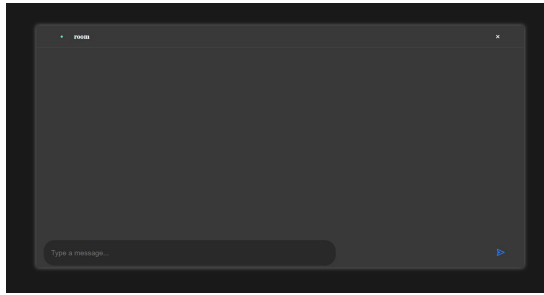
```

25         {!name || !room ? null : (
26             <Link
27                 onClick={(e) =>
28                     !name || !room ? e.preventDefault() : null
29                 }
30                 to={"/chat?name=${name}&room=${room}`}
31             >
32                 <button className={'button mt-20'} type="submit">
33                     Sign In
34                 </button>
35             </Link>
36         )}
37     </div>
38 </div>
39 )}

```

Listing 5.1: Implementacja komponentu logowania

Komponent Logowania składa się z dwóch funkcji `useState` oraz metody, która zwraca widok interfejsu użytkownika umożliwiającego dołączenie do czatu. Poprzez zastosowanie funkcji `useState`, jesteśmy zdolni do przechowywania stworzonych parametrów nazwy użytkownika i pokoju. Wartości stanów są zmienne i mogą być używane do przechowywania danych wewnątrz komponentu i reagowania na zmiany tych danych. Funkcja `useState` zwraca tablicę z dwoma elementami, pierwszym z nich pełni rolę aktualnego stanu, a drugim jest wartość funkcji do jego zmiany. W przypadku ustalenia nazwy użytkownika, aktualny stan zmiennej przechowujemy w elemencie `name` – nazwa użytkownika. Za pomocą elementu `setName` – jesteśmy w stanie zmieniać nazwę użytkownika, której zmienna przyjmuje typ `string`. Adekwatna sytuacja występuje również w przypadku funkcji `useState` przy przypisaniu nazwy pokoju. Komponent ten zwraca funkcję, która renderuje formularz z dwiema sekcjami tekstowymi umożliwiającymi wprowadzenie nazwy użytkownika i nazwy pokoju komunikatora, a także przycisk do dołączenia. Kiedy użytkownik wprowadzi swoją nazwę to stan „`name`” funkcji `useState` jest aktualizowana. Adekwatna sytuacja występuje przy wprowadzeniu nazwy pokoju. W komponencie zaimplementowano również mechanizm, który umożliwia przekierowanie użytkownika do strony komunikatora poprzez kliknięcie przycisku. Do spełnienia tej czynności, wymagane jest wypełnienie sekcji z nazwą użytkownika i pokoju. Jeżeli jedno z pól tekstowych jest niewypełnione, niemożliwe jest przejście do komunikatora. Wymagane jest wpisanie obu danych w celu uzyskania dostępu do komponentu czatu.



Rysunek 5.3: Widok komponentu komunikatora

5.7.2. Komponent komunikatora

Komponent czatu wyświetla komunikator tekstowy, w którym użytkownik może wprowadzać swoje żądania i komunikować się z maszyną. W jego skład wchodzi takie komponenty jak:

- **Nawigacja**
- **Komunikacja**
- **Zbiór wiadomości**
- **Wiadomość**

Każdy z tych fragmentów interfejsu spełnia swoje założenia co pozwala na wyrenderowanie widoku komunikatora. Komponent Nawigacji renderuje widok paska zawierającego nazwę pokoju oraz ikony z aktywnością pokoju i zamknięcia. Pierwsza z tych ikon wyświetlana jest zawsze i oznacza status dostępny w postaci zielonego koła. Po kliknięciu ikony zamknięcia zostaniemy ponownie przekierowany do komponentu logowania.

```

1 const InfoBar = ({ room }) => (
2   <div className="infoBar">
3     <div className="leftInnerContainer">
4       <img className="onlineIcon" src={onlineIcon} alt="online icon" />
5       <h3>{room}</h3>
6     </div>
7     <div className="rightInnerContainer">
8       <a href="/"><img src={closeIcon} alt="close icon" /></a>
9     </div>
10  </div>
11 );

```

Listing 5.2: Implementacja komponentu nawigacji

Komponent Komunikacji służy do wysyłania wiadomości w aplikacji czatu. Element ten składa się z formularza zawierającego przycisk do wysyłania wiadomości i jedno pole tekstowe umożliwiające wpisanie wiadomości. Komponent otrzymuje 3 parametry : message, setMessage i sendMessage. Za pomocą zmiennej message ustalamy aktualny tekst wiadomości, funkcja setMessage pozwala nam na zmianę tekstu wiadomości w polu tekstowym, a parametr sendMessage jest wywoływany za pomocą przycisku, który umożliwia wysyłanie wiadomości. Pole tekstowe formularza kontrolowane jest przez zmienną message.

```
1 const Input = ({ message, setMessage, sendMessage }) => (  
2     <form className='form '>  
3         <input  
4             className='input '  
5             type="text "  
6             placeholder='Type a message... '  
7             value={message}  
8             onChange={(event) => setMessage(event.target.value)}  
9             onPress={(event) => event.key === 'Enter' ? sendMessage(event)  
10                <button className='sendButton '  
11                    onClick={(e) => sendMessage(e)}<BiSend className='send ' />  
12                </button>  
13            </form>  
14 )
```

Listing 5.3: Implementacja komponentu komunikacji

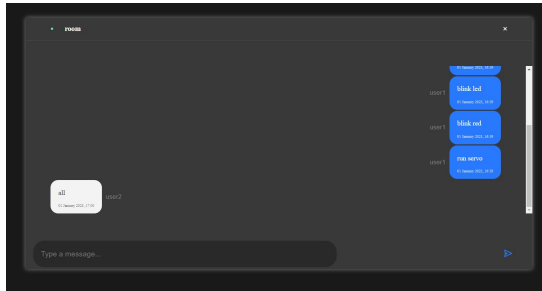
Komponent o nazwie Zbiór Wiadomości renderuje nam widok, który wyświetla listę wysłanych poleceń w aplikacji. Komponent otrzymuje 2 parametry: messages i name. W tym elemencie zaimplementowany został komponent ScrollToBottom, który automatycznie przewija stronę do dołu, gdy dodawane są nowe elementy do listy. Jest to wbudowana funkcja, która udostępniana jest przez bibliotekę React.js. Za pomocą funkcji mapowania tworzony jest kolejny komponent Wiadomość dla każdego nowego wysłanego polecenia. Każdy z tych komponentów otrzymuje jedną wiadomość, nazwę użytkownika, a także unikalny klucz id. Przypisywanie unikalnego klucza id dla elementu jest ważne, ponieważ pomaga systemowi w efektywnym zarządzaniu stanem aplikacji i wykrywaniu ewentualnych zmian w komponentach.

```
1 import ScrollToBottom from 'react-scroll-to-bottom';  
2 const Messages = ({ messages, name }) => (  
3  
4     <ScrollToBottom className="messages">  
5         {messages.map((message, i) => <div key={i}><Message message={message} name={na  
6     </ScrollToBottom>  
7 )};
```

Listing 5.4: Implementacja komponentu zbior wiadomosci

Komponent Wiadomość wyświetla pojedynczą wiadomość w aplikacji. Element ten otrzymuje 2 argumenty: message i name. Za pomocą funkcji useState tworzony jest stan, który przechowuje aktualną datę i godzinę wysłanej wiadomości. Za pomocą funkcji useEffect ustawiana jest wartość stanu daty i godziny po uruchomieniu komponentu.

```
1 const Message = ({ message: { text, user }, name }) => {
2   const [currentDate, setCurrentDate] = useState('')
3
4   useEffect(() => {
5     var date = new Date().getDate()
6     const monthNames = [
7       'January',
8       'February',
9       'March',
10      'April',
11      'May',
12      'June',
13      'July',
14      'August',
15      'September',
16      'October',
17      'November',
18      'December',
19    ]
20    let monthIndex = new Date().getMonth()
21    let monthName = monthNames[monthIndex]
22    var year = new Date().getFullYear()
23    var hours = new Date().getHours()
24    var min = new Date().getMinutes()
25    if (min < 10) {
26      min = '0' + min
27    }
28
29    if (date < 10) {
30      date = '0' + date
31    }
32    setCurrentDate(
33      date + '-' + monthName + '-' + year + 'T' + hours + ':' + min
34    )
35  }, [currentDate])
36  let isSentByCurrentUser = false
37  const trimmedName = name.trim().toLowerCase()
38  if (user === trimmedName) {
39    isSentByCurrentUser = true
```



Rysunek 5.4: Widok komunikatora po wpisaniu wiadomości przez dwóch użytkowników

```

40     }
41     return isSentByCurrentUser ? (
42         <div className="messageContainer justifyEnd">
43             <p className="sentText pr-10">{trimmedName}</p>
44             <div className="messageBox backgroundBlue">
45                 <p className="messageText colorWhite">{text}</p>
46                 <p className="currentDate">{currentDate}</p>
47             </div>
48         </div>
49     ) : (
50         <div className="messageContainer justifyStart">
51             <div className="messageBox backgroundLight">
52                 <p className="messageText colorDark">{text}</p>
53                 <p className="currentDate enemy">{currentDate}</p>
54             </div>
55             <p className="sentText pl-10">{user}</p>
56         </div>
57     )}

```

Listing 5.5: Implementacja komponentu z pojedynczą wiadomością

Do interfejsu komunikatora może dołączyć wiele osób, które mogą wysyłać różne komendy. Pojawienie się wiadomości w komunikatorze zależne jest od tego czy została ona wysłana przez bieżącego użytkownika i w zależności od tego wyświetla ją w odpowiednim miejscu na ekranie (po lewej lub po prawej stronie). Wiadomość wysłana przez bieżącego użytkownika wyróżniona jest kolorem niebieskim, zaś wiadomości otrzymane od innych użytkowników wyróżnione są kolorem szarym. Przy wpisanej komendzie widać nazwę użytkownika, która tą wiadomość wysłała. Komponent Czat korzysta z biblioteki Socket.IO do komunikacji z serwerem za pomocą WebSockets. W komponencie zadeklarowane jest kilka stanów takich jak: name, room, users, message i messages, które zarządzane są przez funkcje useState. Za pomocą metody useEffect parsowane jest zapytanie z adresem URL przy użyciu biblioteki 'query-string'. Następnie zapytanie to jest

przekazywane do komponentu Czat poprzez parametr 'location'. Po parsowaniu zapytania, wartości name i room są używane do nawiązania połączenia z serwerem. Metoda „Join” pozwala na dołączenie do pokoju czatu. Drugim parametrem dla funkcji useEffect jest tablica zależności. Zawiera ona zmienną ENDPOINT oraz location.search. Jeżeli któryś z tych parametrów ulegnie zmianie, następuje ponowne wywołanie funkcji useEffect.

```
1 useEffect(() => {
2     const { name, room } = queryString.parse(location.search)
3     socket = io(ENDPOINT)
4     setRoom(room)
5     setName(name)
6     socket.emit('join', { name, room }, (error) => {
7         if (error) {
8             alert(error)
9         }
    }, [ENDPOINT, location.search])
}
```

Listing 5.6: Implementacja funkcji dołączenia do komunikatora po stronie klienta

W celu nasłuchiwanie wiadomości i danych pokoju przychodzących od serwera zaimplementowano kolejną funkcję useEffect. Funkcja socket.on przyjmująca argument 'message' nasłuchuje na wiadomości przychodzące od serwera. Używa parametru setMessages do aktualizowania stanu elementu messages przy wprowadzeniu nowej wiadomości. Funkcja socket.on przyjmująca argument 'roomData' nasłuchuje na dane pokoju przychodzące od serwera i dokonuje aktualizacji stanu parametru users.

```
1 useEffect(() => {
2     socket.on('message', (message) => {
3         setMessages((messages) => [...messages, message])
4     })
5     socket.on('roomData', ({ users }) => {
6         setUsers(users)
7     })
8 }, [])
```

Listing 5.7: Implementacja funkcji nasłuchiwanie wiadomości i danych przychodzących od serwera

W celu możliwości wysyłania wiadomości do serwera zaimplementowano funkcję sendMessage. Wywoływana jest ona po wypełnieniu pola tekstowego wpisaną wiadomością. Zastosowano również operację przeciw automatycznemu renderowaniu się strony po każdej przesłanej komendzie. Wiadomość zostanie wysłana, za pomocą metody emit, gdzie następnie pole tekstowe zostanie wyczyszczone.

```
1 const sendMessage = (event) => {
2     event.preventDefault()
```

```

3         if (message) {
4             socket.emit('sendMessage', message, () => setMessage(''))
5         }}

```

Listing 5.8: Implementacja funkcji do wysyłania wiadomości

Wszystkie te komponenty wchodzi w skład tworzenia widoku aplikacji, który pełni funkcję komunikatora z urządzeniem Raspberry Pi. Interfejs ten umożliwi nam zdalną komunikację z urządzeniami podłączonymi do mikrokontrolera. Aplikacja pozwala na wysyłanie poleceń i odbieranie informacji z urządzenia za pośrednictwem połączenia sieciowego. Użytkownik może wysłać wiadomości napisane w języku naturalnym do urządzeń za pomocą interfejsu komunikatora. Poprzez zdalne skonfigurowanie Raspberry Pi, urządzenie będzie odbierało polecenia z aplikacji komunikatora i wykonywało odpowiednie akcje, takie jak włączenie lub wyłączenie urządzeń podłączonych do określonych portów GPIO.

```

1 const Chat = ({ location }) => {
2     const [name, setName] = useState('')
3     const [room, setRoom] = useState('')
4     const [users, setUsers] = useState('')
5     const [message, setMessage] = useState('')
6     const [messages, setMessages] = useState([])
7
8     ....
9
10    return (
11        <div className="outerContainer">
12            <div className="container">
13                <InfoBar room={room} />
14                <Messages messages={messages} name={name} />
15                <Input
16                    message={message}
17                    setMessage={setMessage}
18                    sendMessage={sendMessage}
19                />
20            </div>
21        </div>
22    )
23 }

```

Listing 5.9: Implementacja komponentu czatu

5.8. Zarządzanie ścieżką nawigacji

Ważnym aspektem w aplikacji jest również możliwość tworzenia trasy między komponentami i wędrowanie pomiędzy nimi. Każdy komponent zawiera adres

URL, który odpowiada określonemu widokowi w aplikacji. Za pomocą biblioteki 'react-router-dom' możemy tworzyć ścieżkę nawigacji wskazującą, który interfejs ma zostać wyświetlony na stronie w odpowiedzi na zmianę adresu URL bez konieczności przeładowywania całej strony.

```
1 const App = () => {
2   return (
3     <Router>
4       <Route path="/" exact component={Join} />
5       <Route path="/chat" component={Chat} />
6     </Router>
7   );
8 }
```

Listing 5.10: Implementacja ścieżki nawigacji pomiędzy interfejsami

Aby móc wędrować pomiędzy komponentami Logowania i Czat u konieczne było stworzenie trasy z wyznaczoną ścieżką docelową. Komponent Nawigacji umożliwia aplikacji reagowanie na zmianę adresu URL w przeglądarce internetowej. Każdy z tych interfejsów zawiera swoją wyznaczoną ścieżkę, która jest potrzebna przy zmianie widoku aplikacji przez użytkownika. Zaimplementowana nawigacja umożliwia nam przechodzenie do interfejsu komunikatora po wprowadzeniu nazwy użytkownika i nazwy pokoju w komponencie Logowania. Znajdując się w interfejsie komunikatora również możemy przejść z powrotem do elementu z formularzem logowania poprzez kliknięcie przycisku zamknięcia. Znajduje się on w prawym górnym rogu okienku czatu.

5.9. Implementacja strony serwera

Strona serwera została zaimplementowana przy użyciu platformy Node.js. Umożliwia ona tworzenie szybkich i skalowalnych aplikacji sieciowych za pomocą języka JavaScript. Node.js używa modelu jednowątkowego do obsługi wielu żądań jednocześnie co oznacza, że jest on idealny do tworzenia aplikacji sieciowych takich jak serwery czy komunikatory tekstowe. Ważnym aspektem przy stosowaniu platformy Node.js jest możliwość tworzenia aplikacji w czasie rzeczywistym. Po stronie serwera zaimplementowany został cały mechanizm komunikacji z naszą maszyną, która umożliwia przede wszystkim przesyłanie oraz odbieranie danych. Biblioteka Express.js jest narzędziem sieciowym dla Node.js, która umożliwia tworzenie aplikacji opartych na serwerze. Express.js umożliwia obsługę żądań oraz mapowanie adresów URL na odpowiednie funkcje obsługujące te żądania. Głównym zastosowaniem biblioteki express.js jest możliwość tworzenia serwera HTTP, który będzie odpowiadał na żądania aplikacji klienckich. Biblioteka

express.js znajduje swoje szerokie zastosowanie przy tworzeniu aplikacji serwerowych. Poza możliwością tworzenia serwera http, express.js umożliwia również obsługę różnych żądań czy parsowanie przesyłanych żądań. Możliwa jest również obsługa sesji użytkowników, autoryzację, a także przechowywanie i udostępnianie danych. Node.js oferuje nam biblioteki http i express, które umożliwiają stworzenie serwera. Przy pomocy biblioteki Socket.IO tworzymy instancję serwera WebSocket. Następnie serwer nasłuchuje na wybranym porcie i loguje informację o uruchomieniu. Port to numer identyfikujący aplikację. Przy tworzeniu serwera konieczne jest ustalenie numeru portu, na którym możliwe będzie nasłuchiwanie i odbieranie połączeń od klientów. Użytkownicy mogą wysyłać zapytania do serwera poprzez podanie adresu IP serwera i numeru portu. Serwer odpowiada na żądania klientów poprzez wysłanie danych z powrotem do użytkownika poprzez ten sam port. Przy pomocy biblioteki express, serwer będzie nasłuchiwał żądań i odpowiadać na nie.

```
1 const http = require('http')
2 const express = require('express')
3 const socketio = require('socket.io')
4 const cors = require('cors')
5 const PORT = process.env.PORT || 5000
6 const app = express()
7 const server = http.createServer(app)
8 const io = socketio(server)
9 ....
10 server.listen(PORT, () => console.log('Server has started on port ${PORT}'))
```

Listing 5.11: Implementacja serwera

Poprzez zastosowanie biblioteki express.js możliwe jest utworzenie modułu ścieżki nawigacji przy wysyłaniu żądań. Obiekt ten służy do mapowania adresów URL z odpowiednimi funkcjami obsługi żądań. Moduł ten zawiera definicję trasy, która odpowiada żądaniom typu GET. Metoda ta stosowana jest w celu pobierania danych z serwera kiedy z nim się połączymy.

```
1 const express = require("express");
2 const router = express.Router();
3 router.get("/", (req, res) => {
4   res.send("Server is up and running");
5 });
6 module.exports = router;
```

Listing 5.12: Implementacja ścieżki serwera

5.10. Obsługa zdarzeń

Obsługa zdarzeń po stronie serwera to proces reagowania na zdarzenia, które występują w aplikacji. Przykłady takich zdarzeń obejmują żądania HTTP od użytkowników czy otrzymanie danych za pośrednictwem protokołu sieciowego. Aby obsłużyć zdarzenie po stronie serwera należy utworzyć zdarzenia nasłuchiwanie, które w odpowiedzi wykonują określone zadania.

5.10.1. Zdarzenia nasłuchiwanie

W celu odbierania zapytań od klientów i wysyłania odpowiedzi zwrotnych implementuje się zdarzenie nasłuchiwanie po stronie serwera. Gdy serwer jest uruchomiony i nasłuchuje na określonym porcie, może otrzymywać zapytania od klientów poprzez wysyłanie danych na ten port. Serwer może obsługiwać wiele różnych żądań od klientów jednocześnie, dlatego implementuje się zdarzenie nasłuchiwanie, które jest wywoływane za każdym razem gdy serwer otrzyma nowe żądanie od klienta. Zdarzenie to umożliwia serwerowi reagowanie na żądania klienta poprzez wysyłanie odpowiedzi zwrotnej. Przy pomocy biblioteki Socket.IO rejestrowane jest zdarzenie dołączenia do komunikatora poprzez zastosowanie metody 'socket.on'. Funkcja ta przyjmuje obiekt z danymi użytkownika i jest ona wywoływana za każdym razem kiedy wyrejestruje zdarzenie dołączenia do czatu. Następnie emitowane jest zdarzenie zawierające obiekt z danymi o pokoju i zalogowanych użytkownikach.

```
1 io.on('connection', (socket) => {
2     socket.on('join', ({ name, room }, callback) => {
3         const { error, user } = addUser({ id: socket.id, name, room })
4         if (error) return callback(error)
5         socket.join(user.room)
6         io.to(user.room).emit('roomData', {
7             room: user.room,
8             users: getUsersInRoom(user.room),
9         })
10        ....
11
12        callback()
13    })
14    ....
15 })
```

Listing 5.13: Implementacja zdarzenia dołączenia i pobierania danych

Żeby zdarzenie dołączenia mogło być realizowane potrzebne jest zaimplementowanie operacji dodawania nowego użytkownika. Jest to potrzebne, żeby serwer mógł aktualizować listę użytkowników zarejestrowanych w aplikacji oraz spraw-

dział wysyłane żądania. Przy pomocy funkcji dodawany jest nowy klient do listy użytkowników. Następnie pobierana jest jego nazwa, nazwa pokoju, w którym się znajduje, a także unikalny klucz identyfikatora.

```
1 const addUser = ({ id, name, room }) => {
2     name = name.trim().toLowerCase()
3     room = room.trim().toLowerCase()
4     const existingUser = users.find(
5         (user) => user.room === room && user.name === name
6     )
7     const user = { id, name, room }
8     users.push(user)
9     return { user }
10 }
```

Listing 5.14: Implementacja funkcji dodawania użytkownika

W celu umożliwienia wysyłania wiadomości przez użytkowników, implementuje się po stronie serwera zdarzenie wysyłania wiadomości. Funkcja ta jest wywoływana, gdy zdarzenie zostanie zarejestrowane. Kiedy klient wysła wiadomość do serwera, ten odbiera to zdarzenie i uruchamiana jest funkcja. Następnie serwer pobiera informacje o użytkowniku, który przesłał polecenie i emituje zdarzenie wiadomości do wszystkich użytkowników znajdujących się w tym samym pokoju. Funkcja ta działa jako most między klientem, a serwerem, umożliwiając im wymianę danych.

```
1 socket.on('sendMessage', async (message, callback) => {
2     const user = getUser(socket.id)
3     ....
4     io.to(user.room).emit('message', {
5         user: user.name,
6         text: message,
7         createTime,
8     })
9
10    callback()
11 })
12 }
```

Listing 5.15: Implementacja zdarzenia wysyłania wiadomości

Do realizacji zdarzenia wysyłania wiadomości, konieczne jest zidentyfikowanie, który użytkownik wykonał tę operację. W tym celu pobierane są dane użytkownika po przypisanym unikalnym kluczu identyfikatora. Sprawdzane jest czy identyfikator użytkownika z tablicy jest równy identyfikatorowi przekazanemu jako argument użytkownika. Jeżeli warunek zostanie spełniony to następuje zwrócenie obiektu użytkownika, który wysłał wiadomość.

```
1 const getUser = (id) => users.find((user) => user.id === id)
```

Listing 5.16: Implementacja funkcji pobierania informacji o użytkowniku

Również zaimplementowano zdarzenie rozłączania, które jest potrzebne do aktualizowania liczby uczestników korzystających z aplikacji. Funkcja ta jest wywoływana kiedy użytkownik rozłączy się z serwerem. Usuwane są informacje o użytkowniku z listy użytkowników i następnie wysyłany jest komunikat do pozostałych uczestników, informujący o jego opuszczeniu. Po rozłączeniu się klienta, aktualizowana jest lista użytkowników w pokoju.

```
1 socket.on('disconnect', () => {
2     const user = removeUser(socket.id)
3     if (user) {
4         io.to(user.room).emit('message', {
5             user: 'Admin',
6             text: `${user.name} has left.`,
7         })
8         io.to(user.room).emit('roomData', {
9             room: user.room,
10            users: getUsersInRoom(user.room),
11        })
12    })
13 })
```

Listing 5.17: Implementacja zdarzenia rozłączania

W celu aktualizacji listy użytkowników zaimplementowano metodę, która pozwoli na wykonanie tej operacji. Funkcja iteruje po tablicy obecnych użytkowników w pokoju, sprawdzając czy nazwa pokoju użytkownika jest zgodna z nazwą pokoju przekazanej do funkcji jako argument. Jeśli tak to użytkownik dodawany jest do nowej tablicy.

```
1 const getUsersInRoom = (room) => users.filter((user) => user.room === room)
```

Listing 5.18: Implementacja funkcji aktualizująca liste użytkowników

Żeby zdarzenie rozłączenia mogło być realizowane, potrzebne jest zaimplementowanie operacji usuwania użytkownika. Funkcja w celu sprawdzenia, który użytkownik opuścił czat, przyjmuje parametr identyfikatora, który jest przypisywany każdemu użytkownikowi. Warunkiem jest sprawdzenie czy identyfikator przekazanego argumentu jako argument użytkownika jest równy identyfikatorowi użytkownika z tablicy. Jeśli tak, to zwracany jest indeks tego elementu i dochodzi do usunięcia użytkownika z listy.

```
1 const removeUser = (id) => {
2     const index = users.findIndex((user) => user.id === id)
```

```
3     if (index !== -1) return users.splice(index, 1)[0]
4 }
```

Listing 5.19: Implementacja funkcji usuwania użytkownika

5.11. Translacja komend do komunikacji z mikrokontrolerem

W celu umożliwienia komunikacji tekstowej z maszyną za pomocą aplikacji, potrzebne jest nasłuchiwanie zdarzeń wysyłania wiadomości przez serwer. Kiedy klient wysyła żądanie do serwera, ten pobiera informacje o wysłanej wiadomości i emituje ją w celu komunikacji z mikrokontrolerem. Potrzebna jest lista komend napisanych w języku naturalnym, które będą odpowiadać wpisanym w aplikacji wiadomościom. Odpowiedzialne one będą za zadziałanie określonej funkcji uruchamiając dany element podłączony do maszyny.

```
1 socket.on('sendMessage', async (message, callback) => {
2     const user = getUser(socket.id)
3     switch (message) {
4         case 'blink red': {
5             blinkRed()
6             break
7         }
8         case 'turn on all': {
9             turnOnAll()
10            break
11        }
12        case 'turn off all': {
13            turnOffAll()
14            break
15        }
16        case 'sequential': {
17            sequential()
18            break
19        }
20        case 'sensor': {
21            sensor()
22            break
23        }
24        case 'run servo': {
25            servo()
26            break
27        }
28        case 'flowing leds': {
29            flowingLeds()
```

```

30         break
31     }
32     case 'stop yellow': {
33         endBlinkYellow()
34         break
35     }
36     case 'yellow on': {
37         tryYellow()
38         break
39     }
40     case 'blink yellow': {
41         blinkYellow()
42         break
43     }
44     case 'stop green': {
45         endBlinkGreen()
46         break
47     }
48     case 'green on': {
49         tryGreen()
50         break
51     }
52     case 'blink green': {
53         blinkGreen()
54         break
55     }
56     case 'red on': {
57         tryRed()
58         break
59     }
60     case 'stop red': {
61         endBlinkRed()
62         break
63     }}
64     ...
65     callback()
66 })

```

Listing 5.20: Translacja komend na odpowiednie funkcje

Gdy serwer otrzyma polecenie od klienta, sprawdzi co ta wiadomość zawiera i wywoła odpowiednią funkcję. Do implementacji określonych akcji w zależności od wprowadzonego wyrażenia zastosowano instrukcję wielokrotnego wyboru switch-case. Jest to konstrukcja decyzyjna, która umożliwia dokonanie wyboru spośród przypisanych opcji. Jeśli znaleziono pasującą wartość, kod zostanie wykonany dla tego przypadku. Instrukcja „switch” sprawdza przypisaną zmienną i porównuje ją z poszczególnymi przypadkami. Gdy zostanie odnaleziona pasująca

opcja to kod jest realizowany dla tego przypadku. Instrukcja ta ma nieogраниczony wybór opcji i zależna jest od projektanta. Dla zmiennej `message`, która jest odpowiedzialna za przechowywanie wartości wysyłanej wiadomości zostały przypisane komendy napisane w języku naturalnym. Przy pomocy zastosowanej instrukcji sprawdzana jest pasująca wiadomość, która odpowiedzialna jest za wywołanie danej funkcji. Następnie serwer nasłuchuje zdarzenie wysyłania wiadomości i przy komunikacji z urządzeniem Raspberry Pi, uruchamiany jest dany element elektroniczny. Jeżeli zostanie wpisana komenda, która nie znajduje się ujęta w instrukcji to jest ona zwrócona bez jakichkolwiek działań.

5.12. Implementacja działania elementów

W celu sprawdzenia działania komunikacji tekstowej poprzez wpisywanie w języku naturalnym odpowiednich komend, zaprojektowano stanowisko układu. W tym celu z zastosowaniem płytki prototypowej stworzono układ zawierający trzy diody LED wspomagane przez rezystor oraz serwomechanizm. Poprzez zastosowanie Raspberry Pi, płytka prototypowa umożliwia połączenie elementów elektronicznych z użyciem portów GPIO na płycie mikrokontrolera. Przy konstrukcji obwodu wybrano diody LED o kolorze żółtym, czerwonym oraz zielonym.

UKŁAD

5.12.1. Włączenie diody LED

By umożliwić komunikację elementów elektronicznych z mikrokontrolerem należy je podłączyć do odpowiednich wyprowadzeń pinów GPIO. Poprzez zastosowanie biblioteki `onOff` tworzony jest obiekt LED. Dioda o kolorze czerwonym została podłączona do pinu 4 jako wyjście. Sprawdzane jest kolejno czy stworzony obiekt ma wartość 0 co oznacza to, że element nie jest aktywny. Jeśli tak, to zmieniany jest stan na wyjściu na wartość równą 1 co powoduje zaświecenie diody LED. Identyczna funkcja zaimplementowana jest również dla dwóch pozostałych diod.

```
1 function tryRed () {
2     var LED = new onOff.Gpio(4, 'out')
3     return new Promise((resolve) => {
4         setTimeout(() => {
5             if (LED.readSync() === 0) {
6                 LED.writeSync(1)
7             }, 10)
8         }, resolve())
9     })}
```

Listing 5.21: Implementacja funkcji włączającej diode

5.12.2. Wyłączenie diody LED

W celu kontrolowania diody LED, zaimplementowano również funkcję, która pozwoli na jej wyłączenie. Jej sposób działania jest niemal identyczny co do włączenia elementu, tylko w przypadku wyłączenia, stan jest zmieniany z wartości 1 na 0. Element usuwany jest z listy eksportowanych pinów i po odłączeniu obiektu LED, zostaje on wyłączony.

```
1 function endBlinkRed () {
2     var LED = new onOff.Gpio(4, 'out')
3     return new Promise((resolve) => {
4         setTimeout(() => {
5             LED.writeSync(0)
6             LED.unexport()
7             resolve()
8         }, 3000)
9     })}
```

Listing 5.22: Implementacja funkcji wyłączającej diode

5.12.3. Włączenie wszystkich diod

W celu umożliwienia włączenia trzech diod LED stworzono nowe obiekty odpowiadające elementom. Zostały one podłączone do określonych pinów GPIO na wyjście układu. Sprawdzane jest kolejno czy stworzone obiekty mają wartość 0 co oznacza to, że elementy są nieaktywne. Jeśli tak, to zmieniany jest stan na wyjściu na wartość równą 1 co powoduje zaświecenie wszystkich trzech diod.

```
1 function turnOnAll () {
2     var LED04 = new onOff.Gpio(4, 'out'),
3         LED17 = new onOff.Gpio(17, 'out'),
4         LED27 = new onOff.Gpio(27, 'out')
5     return new Promise((resolve) => {
6         setTimeout(() => {
7             if (
8                 LED04.readSync() === 0 &&
9                 LED17.readSync() === 0 &&
10                LED27.readSync() === 0
11            ) {
12                LED04.writeSync(1)
13                LED17.writeSync(1)
14                LED27.writeSync(1)
15            }, 1)
16            resolve()
17        })
18    })}
```

Listing 5.23: Implementacja funkcji włączającej wszystkie diody

5.12.4. Wyłączenie wszystkich diod

W celu umożliwienia wyłączenia wszystkich trzech diod LED, sprawdzane jest czy zaimplementowane obiekty mają wartość 1 na wyjściu co oznacza to, że elementy są włączone i się świecą. Jeśli tak, to zmieniany jest stan na wyjściu na niski co powoduje wyłączenie wszystkich trzech diod.

```
1 function turnOffAll() {
2     var LED04 = new onOff.Gpio(4, 'out'),
3       LED17 = new onOff.Gpio(17, 'out'),
4       LED27 = new onOff.Gpio(27, 'out')
5     return new Promise((resolve) => {
6       setTimeout(() => {
7         LED04.writeSync(0)
8         LED17.writeSync(0)
9         LED27.writeSync(0)
10        LED04.unexport()
11        LED17.unexport()
12        LED27.unexport()
13        resolve()
14      }, 3000)
15    })}
```

Listing 5.24: Implementacja funkcji wyłączająca wszystkie diody

5.12.5. Naprzemienne włączanie diod

Aby umożliwić obsługę 3 diod LED, należy je podłączyć do trzech osobnych pinów GPIO. Czynność ta będzie potrzebna do osiągnięcia efektu płynącej diody. Wykonanie tej operacji ma na celu stworzenie wizualnego efektu ruchu. Możliwe jest to do osiągnięcia przy zastosowaniu szeregu diod LED, które są sekwencyjnie zapalane. W celu realizacji takiego efektu tworzone są trzy obiekty LED, które są odpowiedzialne za zarządzanie pinami GPIO o numerach, 4, 17 i 27. Obiekty te zapisywane są do tablicy i następnie zerowane. W zależności od aktualnego indeksu utworzonej tablicy, zmieniana jest wartość stanu z 0 na 1 co powoduje włączenie diody. W momencie osiągnięcia pozycji ostatniego elementu tablicy, zmieniany jest kierunek włączania diod. Przy zastosowaniu funkcji interwałowej, możliwe jest powtarzanie się czynności włączania i wyłączania elementów o określony czas co daje efekt wizualnego ruchu diod. Po upływie 5 sekund, elementy usuwane są z listy eksportowanych pinów co powoduje odłączenie obiektu LED i zakończenie procesu.

6. Obsługa

6.1. Obsługa aplikacji

Do zaimplementowania całego działania aplikacji skorzystano z edytora kodu źródłowego Visual Studio Code. Za pomocą tego oprogramowania możliwe jest przeprowadzanie testów bądź debugowanie stworzonego projektu. Visual Studio Code jest edytorem dostępnym na wielu systemach operacyjnych takich jak Windows czy Linux. W celu przeprowadzenia testów poprawności działania aplikacji wykonano szereg operacji, które pozwoliły na ocenę działania zaimplementowanego kodu. Visual Studio Code umożliwia uruchomienie aplikacji webowej za pomocą przeglądarki internetowej. Możliwe było przeprowadzenie badań nad funkcjonalnością aplikacji. Poprzez stworzenie kilku użytkowników, przeprowadzone zostały testy nad komunikacją tekstową z maszyną i sprawdzenie czy uzyskana zostanie odpowiedź na przesłane komendy.

6.2. Zaimplementowane polecenia

Głównym założeniem projektu była komunikacja z maszyną poprzez wpisywanie odpowiednich komend w języku naturalnym. Miały one na celu uzyskanie odpowiedzi przez mikrokontroler w postaci odpowiednich czynności. Wiadomości wpisane w komunikatorze miały spowodować uruchomienie danego elementu wykonawczego podłączonego od określonego pinu GPIO. W celu przeprowadzenia takiej komunikacji dokonano translacji funkcji na odpowiednie komendy napisane w języku naturalnym, które powodowały uzyskanie takiego wyniku. W aplikacji zaimplementowano polecenia takie jak:

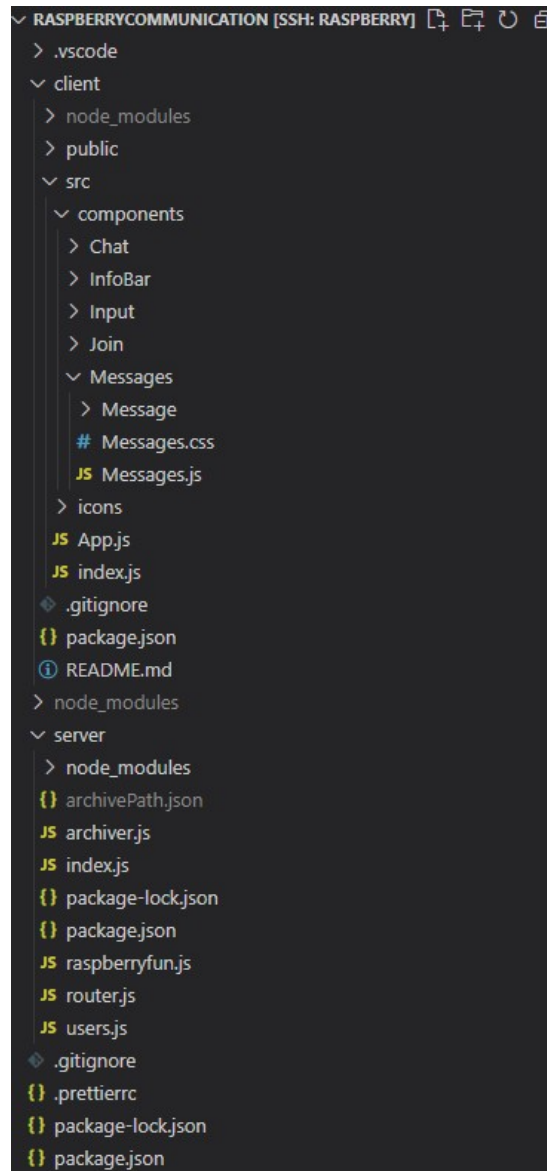
- **red on**, komenda ta powoduje włączenie się diody LED i zaświecenie kolorem czerwonym
- **green on**, komenda ta powoduje włączenie się diody LED i zaświecenie kolorem zielonym

- **yellow on**, komenda ta powoduje włączenie się diody LED i zaświecenie kolorem żółtym
- **Moduł pigpio**, biblioteka ta korzysta z frameworka Node.js. Moduł pigpio może być używana w celu kontrolowania i monitorowania sygnałów z wejść/wyjść cyfrowych oraz modulację szerokości impulsów na Raspberry Pi. Dzięki tej bibliotece jesteśmy w stanie tworzyć projekty z użyciem urządzeń elektronicznych takie jak diody LED czy serwomechanizmy, które są podłączone do portów GPIO urządzenia Raspberry Pi. Biblioteka pigpio znalazła szerokie zastosowanie w aplikacjach gdzie wymagane jest sterowanie urządzeniami elektrycznymi lub elektronicznymi.
- **stop red**, komenda ta powoduje wyłączenie diody LED o kolorze czerwonym poprzez zmienienie stanu z wartości 1 na 0
- **stop green**, komenda ta powoduje wyłączenie diody LED o kolorze zielonym poprzez zmienienie stanu z wartości 1 na 0
- **stop yellow**, komenda ta powoduje wyłączenie diody LED o kolorze żółtym poprzez zmienienie stanu z wartości 1 na 0
- **blink red**, komenda ta powoduje efekt migania diody LED o kolorze czerwonym
- **blink green**, komenda ta powoduje efekt migania diody LED o kolorze zielonym
- **blink yellow**, komenda ta powoduje efekt migania diody LED o kolorze żółtym
- **flowing leds**, komenda ta powoduje uzyskanie efektu wizualnego ruchu diod
- **run servo**, komenda ta pozwala na sterowanie serwomechanizmem
- **sequential**, komenda ta powoduje sekwencyjne wykonywanie się poleceń, które spowodują uruchomienie zainstalowanych elementów wykonawczych, jeden po drugim
- **turn on all**, komenda powoduje włączenie wszystkich trzech diod LED.

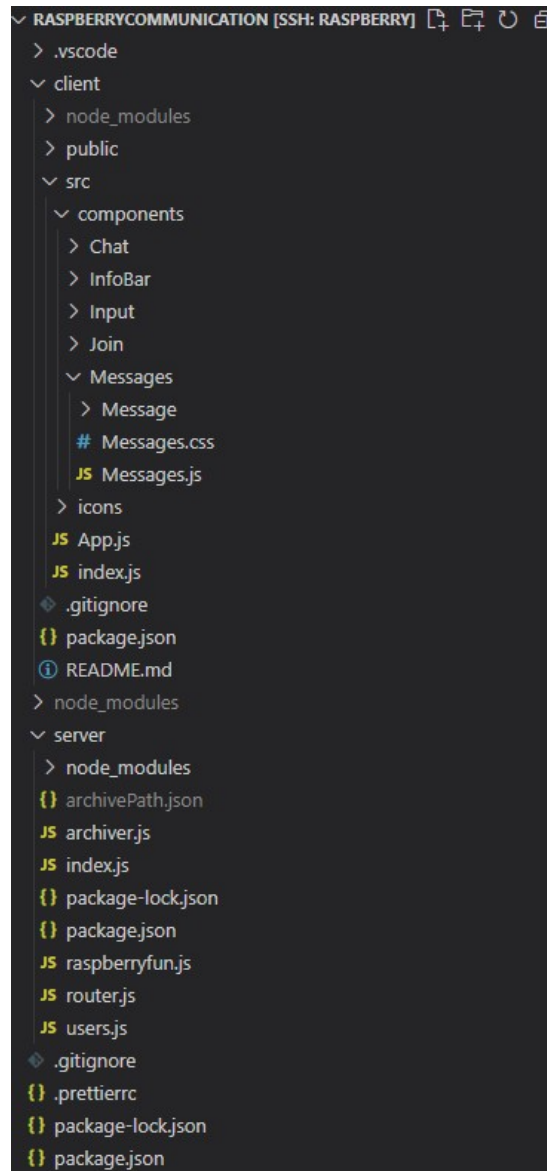
— **turn off all**, komenda powoduje wyłączenie wszystkich trzech diod LED

6.3. Zarządzanie komunikatorem

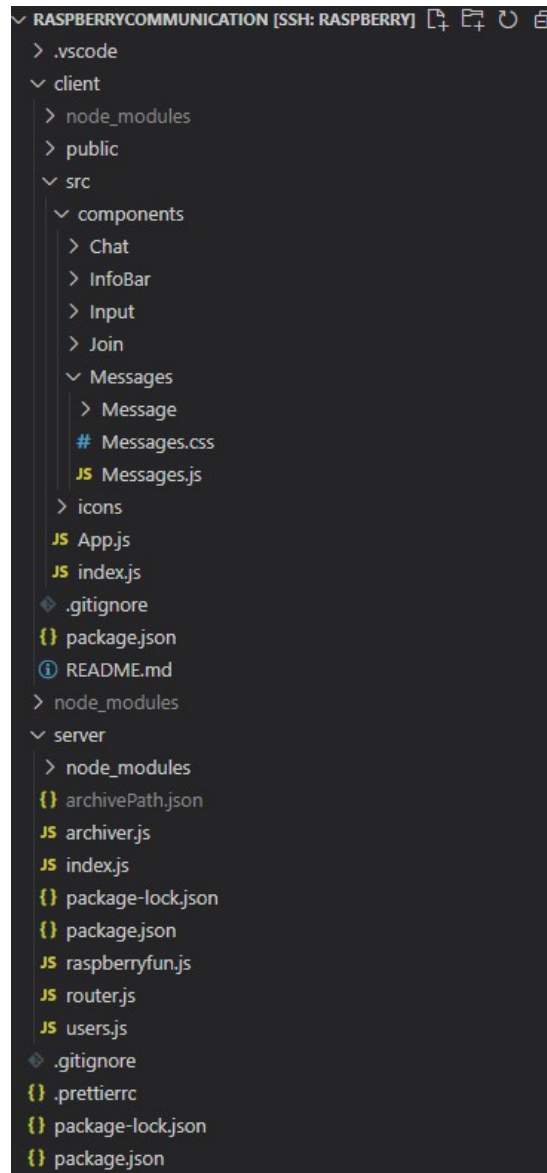
W celu możliwości dołączenia do czatu, użytkownik musi wpisać swoją nazwę pod którą będzie widoczny oraz nazwę pokoju. Dane te muszą zostać wypełnione w odpowiednich polach tekstowych. Jeżeli użytkownik zostawi puste pole, w którym musi wpisać swoją nazwę lub nazwę pokoju, nie będzie on miał dostępu do zalogowania się do komunikatora. W celach testowych wpisano jedynie nazwę pokoju do którego użytkownik chciałby dołączyć. Pole tekstowe zawierające nazwę użytkownika, pod którą chciałby być wyświetlany zostało pominięte. Przy braku wypełnienia wszystkich danych, nie został uzyskany dostęp do zalogowania się do komunikatora poprzez pojawienie się przycisku, który przekierowałby użytkownika do czatu. W momencie kiedy użytkownik wypełni wszystkie potrzebne dane, w aplikacji pojawi się przycisk umożliwiający zalogowanie się do aplikacji. Poprzez kliknięcie przycisku Zaloguj, użytkownik zostanie przekierowany do komunikatora tekstowego. Po wypełnieniu wymaganych danych i kliknięciu przycisku logowania, użytkownik zostanie przekierowany do czatu. Tutaj będzie mógł nawiązać komunikację z maszyną poprzez wpisanie odpowiedniego tekstu. W celu przeprowadzenia testów, w której zostały zastosowane dodatkowe technologie frontendu i backendu dla aplikacji webowej, stworzono drugiego użytkownika. Dołączył on do pokoju czatu z tą samą nazwą co poprzedni użytkownik. Wykonanie takiej operacji pozwoli na możliwość korzystania z komunikatora przez kilku zalogowanych klientów. W celu sprawdzenia poprawności funkcjonowania aplikacji wykonano testy, które ukazują interfejsy z dwóch perspektyw. Wiadomość bieżącego użytkownika będzie ukazana po prawej stronie z niebieskim tłem. W przeciwnym razie wysłany komunikat będzie widoczny po lewej stronie wyróżniony szarym kolorem. Na Rysunku 6.6 została przedstawiona perspektywa użytkownika o nazwie user1. Wiadomości wysłane przez klienta zarejestrowanego pod nazwą user2 wyświetlane są po lewej stronie komunikatora wyróżnione szarym tłem. Drugi użytkownik poprzez wpisanie polecenia `flowing leds` przesłał żądanie do mikrokontrolera co wynikiem było uzyskanie efektu płynącej diody. Wiadomość ta została zapisana w archiwum i użytkownik pod nazwą user1, który również zalogował się do tego samego pokoju może zobaczyć wysłane wiadomości. Na Rysunku 6.8 zostały przedstawione obie perspektywy. Ukazują one jak widoczne są wiadomości wpisane przez określonego użytkownika. Oba te polecenia wysłane przez klientów zostały zapisane w archiwum i po ponownym zalogowaniu się do czatu o tej samej nazwie pokoju, klienci czatu będą w stanie odtworzyć wiadomości, które zostały przez nich wysłane. Użytkownik o nazwie user1 wpisał



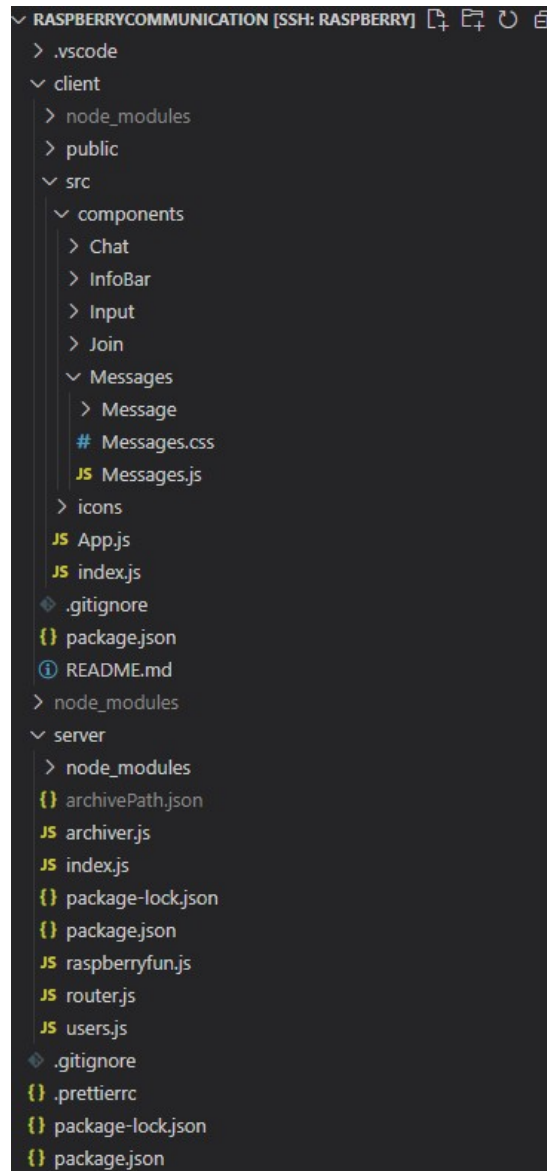
Rysunek 6.1: Formularz logowania użytkownika do aplikacji



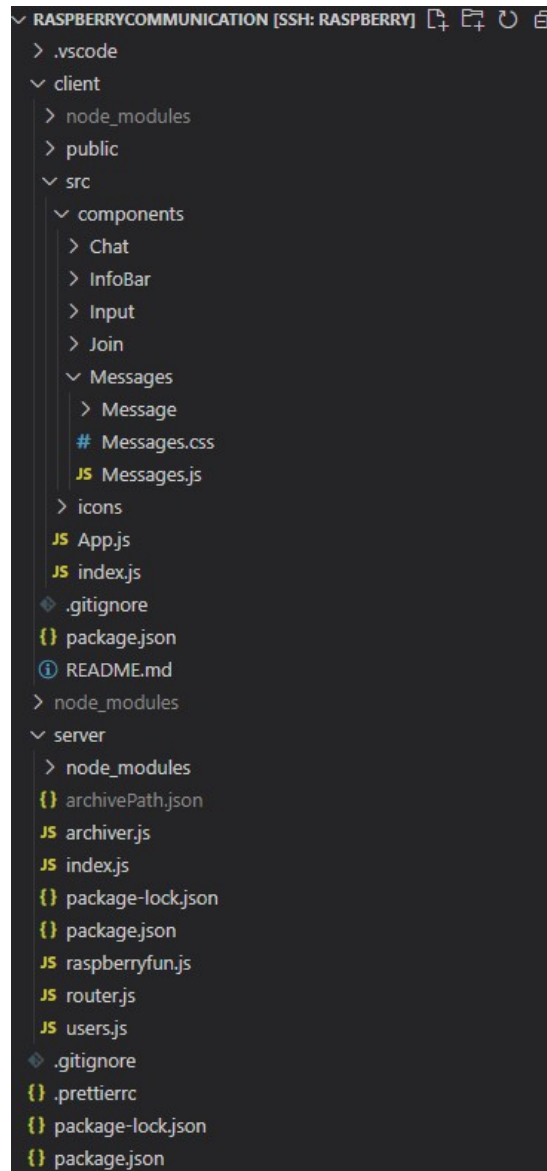
Rysunek 6.2: Próba dołączenia do komunikatora



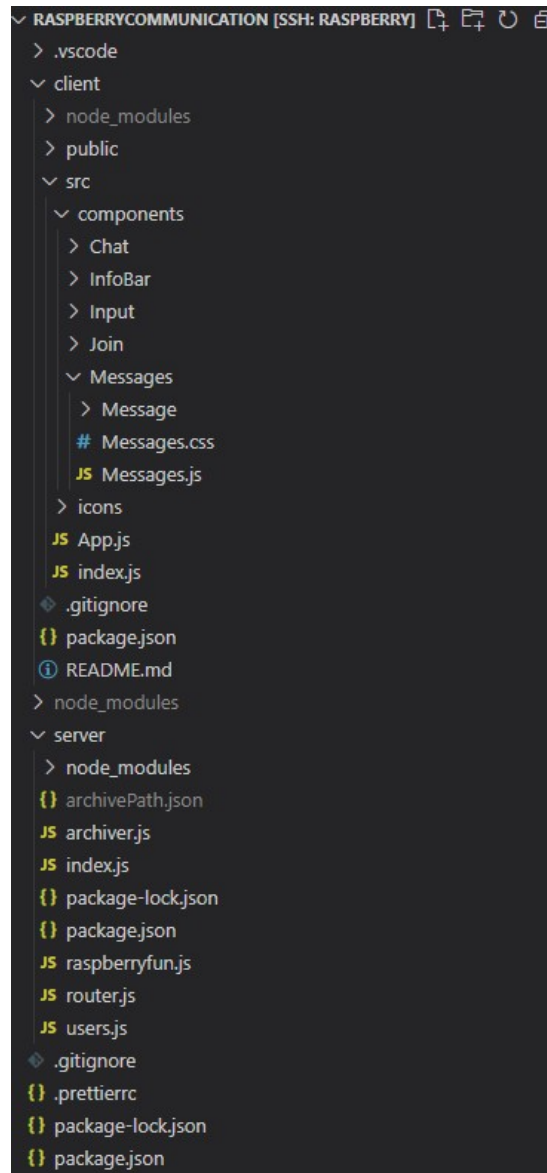
Rysunek 6.3: Logowanie do komunikatora



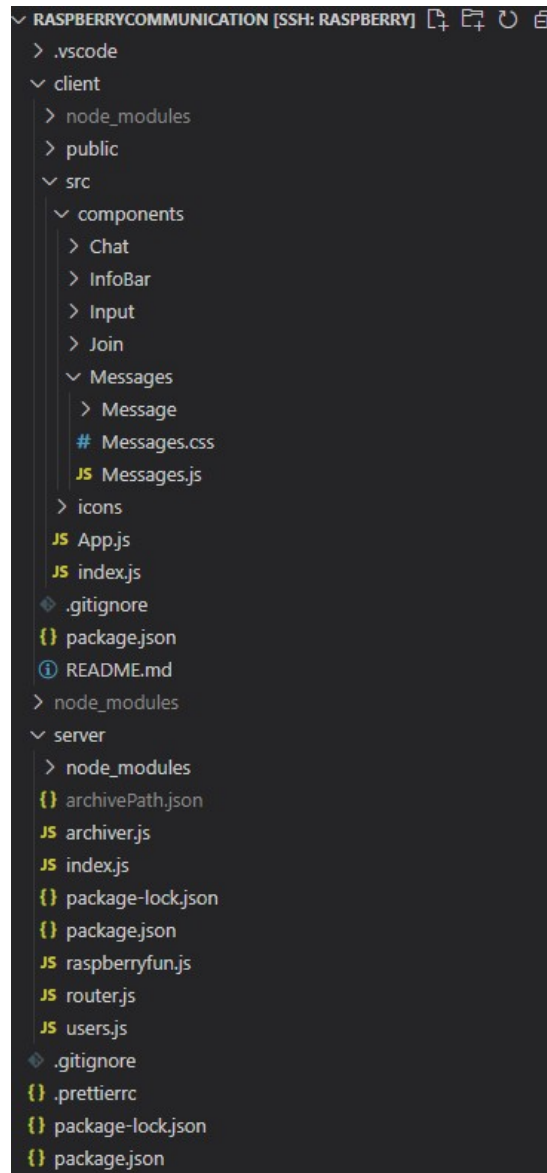
Rysunek 6.4: Interfejs użytkownika po zalogowaniu



Rysunek 6.5: Tworzenie drugiego użytkownika



Rysunek 6.6: Testowanie komunikacji tekstowej



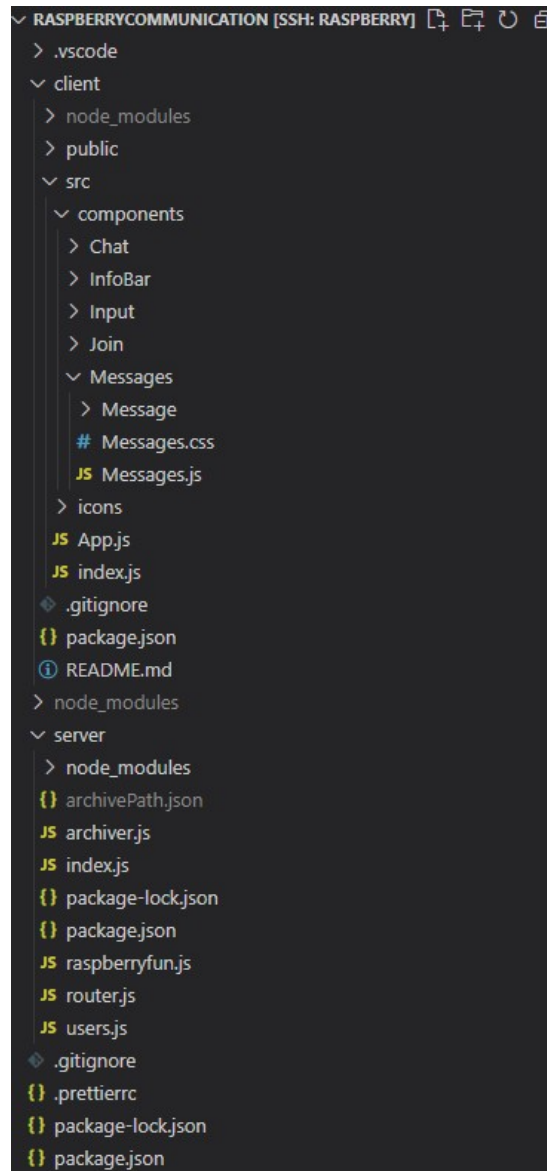
Rysunek 6.7: Wywołanie funkcji umożliwiającej sterowanie serwomechanizmem

polecenie o nazwie `run servo` i wiadomość ta pojawiła się w czacie po obu stronach. Komenda ta umożliwia sterowanie serwomechanizmem, który przesuwa się o określony kąt do momentu osiągnięcia zadanej pozycji. Do przeprowadzenia kolejnego testu funkcjonalności komunikatora, użytkownik o nazwie `user2` wpisał polecenie o treści `sequential`. Powoduje to sekwencyjne wykonywanie się zaimplementowanych metod, które odpowiedzialne są za uruchomienie zainstalowanych elementów elektronicznych przy komunikacji z mikrokontrolerem. Wiadomość ta jest widoczna w obu perspektywach. Dodatkowo zaimplementowano mechanizm, który będzie sygnalizował opuszczenie użytkownika z czatu. W tym celu stworzono kolejnego klienta, który dołączył do czatu, a następnie go opuścił. Komunikat został wysłany do użytkowników, którzy korzystają wciąż z komunikatora. Zostali oni poinformowani o tej sytuacji. Po wpisaniu określonej liczby wiadomości, czat z wiadomościami zacznie się przewijać do dołu ukazując najwcześniejsze wysłane polecenia. Żeby zobaczyć komendy, które zostały wysłane dawniej i nie mieszczą się w interfejsie użytkownika, konieczne będzie przewijanie czatu. Zastosowano mechanizm, który pozwoli na przewinięcie wiadomości z powrotem do najnowszej poprzez kliknięcie przycisku w kształcie koła znajdującego się nad przyciskiem wysyłania wiadomości. W perspektywie użytkownika o nazwie `user2` widoczna jest kropka przy pasku przewijania, która pozwoli na wykonanie takiej czynności. Poprzez kliknięcie przycisku zamykającego komunikator znajdujący się w pasku nawigacyjnym po prawej stronie zostaniemy przeniesieni ponownie do formularza. Będziemy musieli wypełnić ponownie pola tekstowego wymagające podanie nazwę użytkownika i pokoju by móc dołączyć do czatu. Po wpisaniu ponownie danych i zalogowaniu się jako użytkownik o nazwie `user1` jesteśmy ponownie przekierowani do komunikatora w raz z całym archiwum wysłanych dotychczas wiadomości.

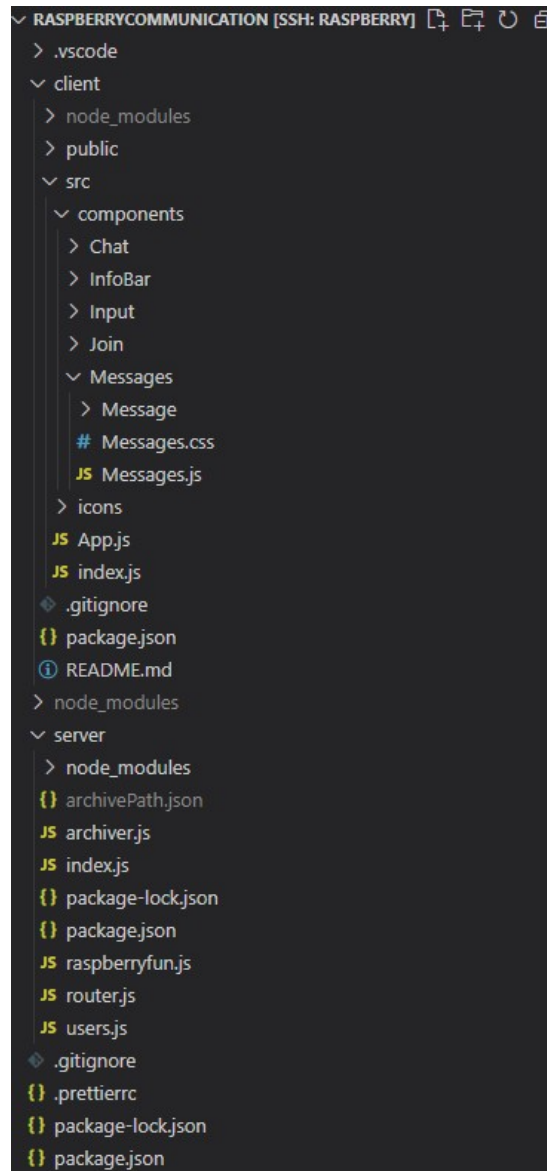
6.4. Przedstawienie uzyskanych wyników

Poprzez wpisanie określonych komend w komunikatorze, użytkownik dostawał odpowiedzi od urządzenia w postaci działania poszczególnych elementów wykonawczych. Poniżej przedstawiono uzyskane wyniki przy wpisaniu polecenia:

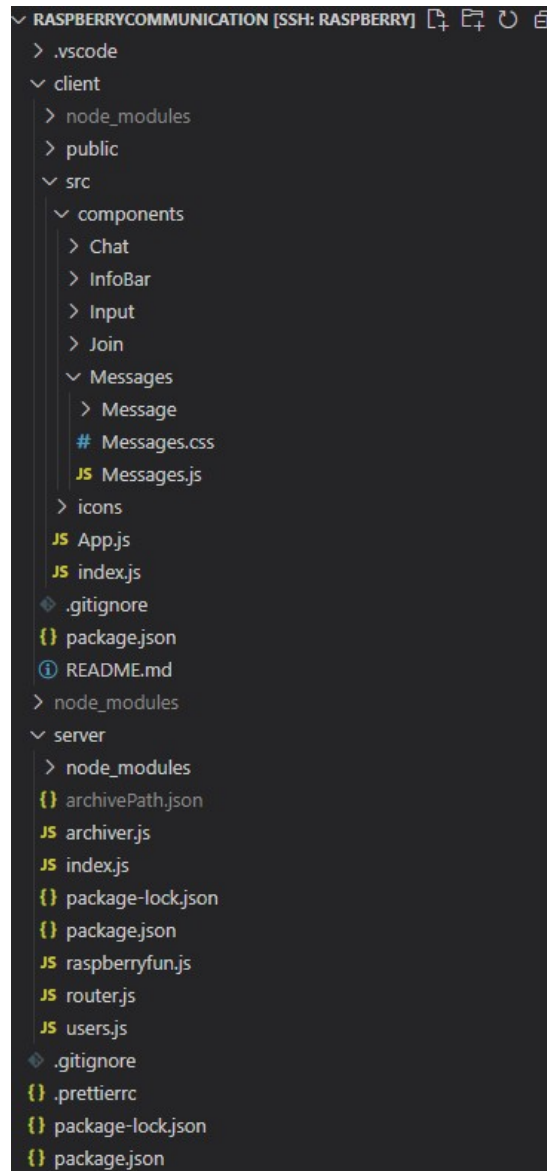
- **red on**, komenda ta powoduje włączenie się diody LED i zaświecenie kolorem czerwonym
- **green on**, komenda ta powoduje włączenie się diody LED i zaświecenie kolorem zielonym



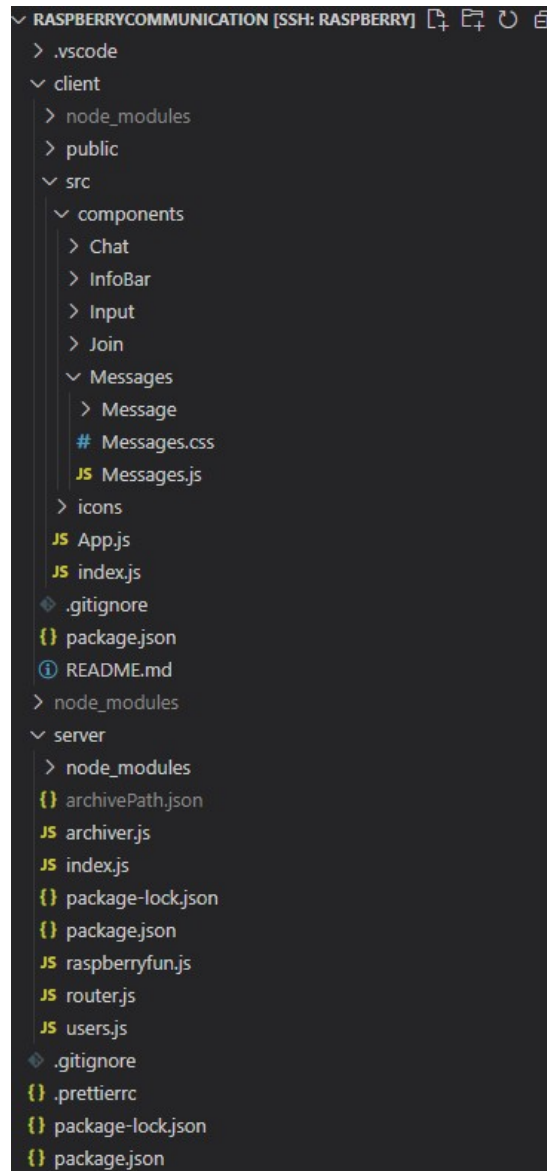
Rysunek 6.8: Sekwencyjnie wykonywanie się zaimplementowanych metod



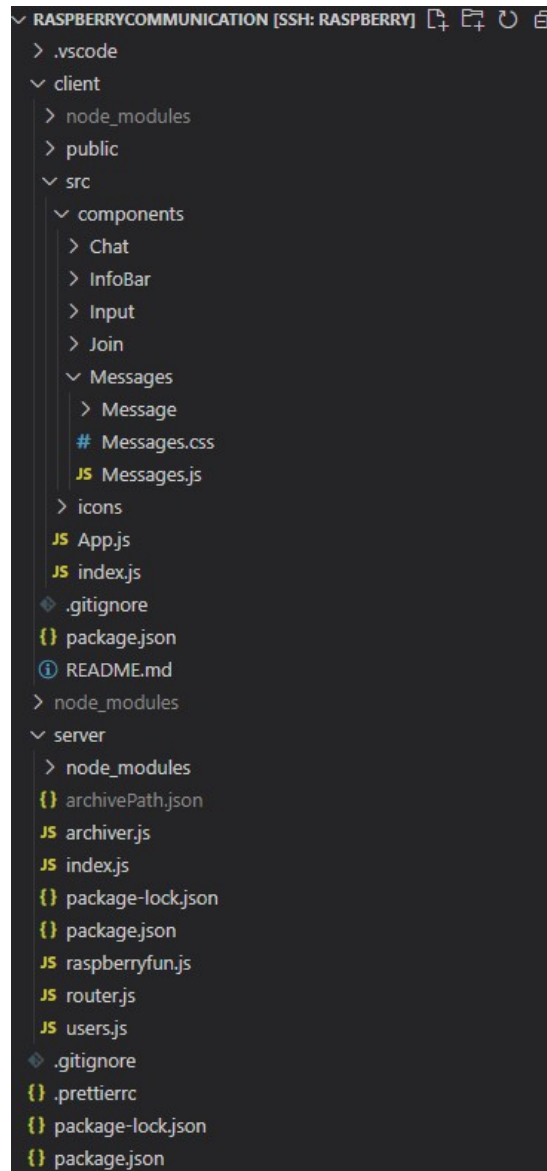
Rysunek 6.9: Widok komunikatu o opuszczeniu czatu przez użytkownika



Rysunek 6.10: Interfejs zawierający archiwum wiadomości



Rysunek 6.11: Interfejs logowania użytkownika



Rysunek 6.12: Interfejs przedstawiający zawartość konwersacji

- **yellow on**, komenda ta powoduje włączenie się diody LED i zaświecenie kolorem żółtym
- **Moduł pigpio**, biblioteka ta korzysta z frameworka Node.js. Moduł pigpio może być używana w celu kontrolowania i monitorowania sygnałów z wejść/wyjść cyfrowych oraz modulację szerokości impulsów na Raspberry Pi. Dzięki tej bibliotece jesteśmy w stanie tworzyć projekty z użyciem urządzeń elektronicznych takie jak diody LED czy serwomechanizmy, które są podłączone do portów GPIO urządzenia Raspberry Pi. Biblioteka pigpio znalazła szerokie zastosowanie w aplikacjach gdzie wymagane jest sterowanie urządzeniami elektrycznymi lub elektronicznymi.
- **turn on all**
- **stop green**
- **turn off all**
- **yellow on i green on**
- **run servo**, komenda ta powoduje efekt migania diody LED o kolorze zielonym

7. Zakończenie

7.1. Zakończenie

W ramach wykonania postawionych celów projektu zaprojektowano aplikację webową. Pełniła ona rolę czatu, która pozwalała na komunikację tekstową pomiędzy człowiekiem, a maszyną. W aplikacji zastosowano dodatkowe technologie frontendu i backendu co pozwoliło na wygodną obsługę programu przez użytkownika. Poprzez wysyłanie odpowiednich komend, możliwe było przeprowadzenie komunikacji z urządzeniem jakim było Raspberry Pi. W aplikacji zaimplementowano metody, które pozwalały na sterowanie elementami wykonawczymi podłączonymi do odpowiednich pinów GPIO mikrokontrolera. Poprzez skonfigurowanie protokołu SSH i stworzenie statycznego adresu IP możliwe było zdalne kontrolowanie urządzenia. Istotną rolę odegrała również implementacja komunikacji klienta z serwerem. Wykonanie tej operacji umożliwiło wymianę danych z mikrokontrolerem w czasie rzeczywistym. Pozwoliło to na wysyłanie poleceń i uzyskanie odpowiedzi bez żadnych zwłok czasowych. Przeprowadzone testy utwierdziły poprawność zaimplementowanego komunikatora. Wszystkie założenia zawarte w projekcie zostały spełnione i program można uznać za stabilny. Aplikację również można rozbudować o dalsze funkcjonowania. Zarówno stronę klienta jak i serwera można rozwinąć o różne czynności. Sam interfejs użytkownika może zostać rozbudowany o technologie rozwijające funkcjonalność aplikacji, a także ułatwiające obsługę przez użytkownika. Również można rozbudować strukturę układu połączonego z mikrokontrolerem. Poprzez zaimplementowanie odpowiednich funkcji po stronie serwera, możliwe będzie zastosowanie więcej operacji podczas komunikacji tekstowej. W tych aspektach, aplikacje webowe są nieograniczone i mogą zostać rozwinięte do poziomu nieprzekraczającego granic wyobraźni człowieka.