



**Politechnika
Śląska**

PROJEKT INŻYNIERSKI

Zdecentralizowana aplikacja w sieci Blockchain

Paweł GOŁĄBEK

Nr albumu: 300450

Kierunek: Informatyka

Specjalność: Grafika i Oprogramowanie

PROWADZĄCA PRACĘ

Dr inż. Anna Gorawska

Katedra Informatyki Stosowanej

Wydział Automatyki, Elektroniki i Informatyki

Gliwice 2025

Tytuł pracy

Zdecentralizowana aplikacja w sieci Blockchain

Streszczenie

Celem pracy inżynierskiej było stworzenie zdecentralizowanej aplikacji działającej w sieci blockchain. W ramach realizacji pracy zaprojektowano i zaimplementowano zdecentralizowaną aplikację służącą do usprawnienia obsługi punktów ładowania samochodów elektrycznych. Zdefiniowano wymagania funkcjonalne i нефункционалне aplikacji. Wyróżniono trzy warstwy działania aplikacji. Warstwa logiki zdecentralizowanej przechowuje dane w inteligentnym kontrakcie umieszczonym w sieci *Ethereum*. Warstwa logiki lokalnej przechowuje dane w lokalnej bazie danych i zarządza przepływem danych. Warstwa prezentacji pozwala na komunikację użytkownika końcowego z resztą systemu. Aplikacja została przetestowana pod względem funkcjonalności i bezpieczeństwa. W ramach weryfikacji warstwy działającej w sieci zdecentralizowanej wykonano testy jednostkowe oraz audyt bezpieczeństwa inteligentnego kontraktu.

Słowa kluczowe

Blockchain, Inteligentny Kontrakt, Web3, DApp, Digitalizacja

Thesis title

Decentralized application on the Blockchain network

Abstract

The aim of the engineering thesis was to create a decentralized application operating on the blockchain network. As part of the project, a decentralized application was designed and implemented to improve the handling of electric vehicle charging points. The functional and non-functional requirements of the application were defined. Three layers of the application were distinguished. The decentralized logic layer stores data in a smart contract placed on the *Ethereum* network. The local logic layer stores data in a local database and manages the flow of data. The presentation layer allows the end user to communicate with the rest of the system. The application has been tested for functionality and security. As part of the verification of the decentralized network layer, unit tests and a security audit of smart contract were performed.

Key words

Blockchain, Smart Contract, Web3, DApp, Digitization

Spis treści

1	Wstęp	1
1.1	Cel pracy	1
1.2	Osadzenie problemu w dziedzinie	1
1.3	Zawartość pracy	2
2	Analiza tematu	3
2.1	Struktura łańcucha bloków	3
2.2	Transakcje w sieci blockchain	4
2.3	Mechanizm konsensusu	4
2.4	Zdecentralizowane aplikacje	5
2.5	Zastosowanie sieci blockchain w elektromobilności	6
3	Wymagania i narzędzia	7
3.1	Wymagania funkcjonalne	7
3.2	Wymagania нефункционалне	8
3.3	Przypadki użycia	9
3.3.1	Scenariusz ładowania samochodu elektrycznego przez klienta	10
3.3.2	Scenariusz dodawania ładowarki do systemu przez zalogowanego sprzedawcę	11
3.4	Narzędzia	12
4	Specyfikacja zewnętrzna	13
4.1	Wymagania sprzętowe	13
4.2	Logowanie	13
4.3	Widok administratora	14
4.4	Widok sprzedawcy	16
4.5	Widok klienta	17
5	Specyfikacja wewnętrzna	21
5.1	Architektura systemu	21
5.2	Przechowywanie danych	22
5.3	Warstwa logiki zdecentralizowanej	24

5.4	Warstwa logiki lokalnej	27
5.5	Warstwa prezentacji	31
5.6	Wymagania kompilacji	34
6	Weryfikacja i walidacja	35
6.1	Przykład testu manualnego	35
6.2	Testy automatyczne	37
6.3	Testowanie pod kątem bezpieczeństwa	40
6.4	Wykryte i usunięte błędy	41
6.5	Testowanie z użyciem realnych danych wejściowych	43
7	Podsumowanie i wnioski	45
7.1	Kierunki ewentualnych dalszych prac	45
7.2	Problemy napotkane w trakcie pracy	46
	Bibliografia	48
	Spis skrótów i symboli	51
	Spis rysunków	53
	Spis tabel	55

Rozdział 1

Wstęp

1.1 Cel pracy

Celem pracy było opracowanie zdecentralizowanej aplikacji działającej w sieci **blockchain**, wspierającej zarządzanie systemem sprzedaży energii elektrycznej w systemach ładowania samochodów elektrycznych. Aplikacja ma umożliwiać użytkownikom pełnienie różnych ról: administratora, sprzedawcy lub klienta, z odpowiednimi uprawnieniami i funkcjami. Administrator zarządza użytkownikami i kontroluje system, sprzedawcy udostępniają ładowarki klientom. Klienci korzystają z ładowarek i opłacają sesje ładowania. Wszystkie transakcje związane z użytkownikami i przepływem płatności przechowywane będą w sieci zdecentralizowanej.

1.2 Osadzenie problemu w dziedzinie

Zdecentralizowane rejestry danych oraz oparte o ich działanie zdecentralizowane aplikacje (DApps) otwierają nowe możliwości w zakresie bezpiecznego przechowywania i przesyłania danych bez potrzeby scentralizowanego pośrednika. Ogranicza to ryzyko manipulacji danymi przez osoby trzecie i zwiększa zaufanie systemów opracowanych ze wsparciem tej technologii. Trend mający na celu oddanie użytkownikom więcej kontroli nad swoimi danymi i zasobami cyfrowymi nazywany jest **Web3**. Specyficznym przykładem sieci zdecentralizowanej jest blockchain. Projektowanie zdecentralizowanych aplikacji wymaga znajomości sposobu działania inteligentnych kontraktów i metod połączenia ich z klasycznymi aplikacjami internetowymi.

Wykorzystywanie zdecentralizowanych rejestrów danych wpisuje się w globalne trendy elektromobilności i wdrażania zdecentralizowanych technologii w sektorze energetycznym i transporcie. Może też stanowić bazę dla dalszych badań nad automatyzacją i optymalizacją procesów ładowania pojazdów elektrycznych.

1.3 Zawartość pracy

W rozdziale drugim (Analiza tematu) omówiono zastosowanie technologii zdecentralizowanych. W rozdziale trzecim (Wymagania i narzędzia) omówiono wykaz wymagań funkcjonalności i wymagań niefunkcjonalnych aplikacji. Zawiera on także jej standardowe przypadki użycia oraz spis najważniejszych narzędzi wykorzystanych w realizacji projektu. W rozdziale czwartym (Specyfikacja zewnętrzna) omówiono widoczną dla użytkownika końcowego warstwę prezentacji. Opisano też każdy z dostępnych dla użytkowników widoków i sposób ich użycia. W rozdziale piątym (Specyfikacja wewnętrzna) przedstawiono technologie i metodykę zastosowaną przy implementacji warstw logiki i danych aplikacji w tym sposób działania inteligentnego kontraktu działającego w sieci zdecentralizowanej. W rozdziale szóstym (Weryfikacja i walidacja) opisano metodykę testów zastosowaną przy sprawdzeniu poprawności działania aplikacji. W rozdziale siódmym (Podsumowanie i wnioski) zawarto podsumowanie przeprowadzonych prac, uzyskane wyniki oraz sugerowane kierunki dalszego rozwoju projektu.

Rozdział 2

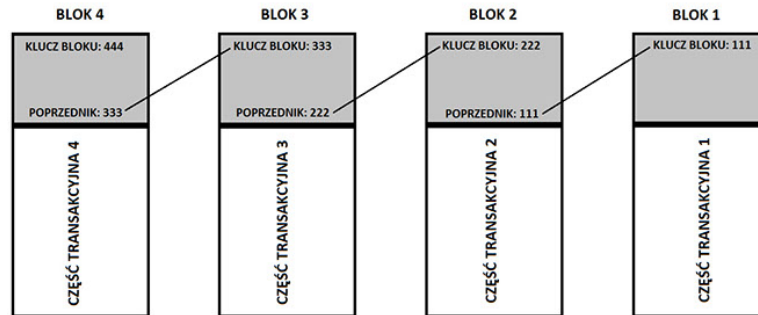
Analiza tematu

Samochody elektryczne cieszą się w krajach europejskich coraz większym zainteresowaniem. Między rokiem 2020 a 2022 liczba samochodów elektrycznych na polskich drogach wzrosła ponad trzykrotnie[9]. Trendy na rynku wskazują, że tendencja może się utrzymać, a nawet wzrosnąć.

W dobie rosnącej popularności pojazdów elektrycznych kluczowym wyzwaniem staje się efektywne i bezpieczne zarządzanie procesem ładowania. W scenariuszu masowej elektryfikacji transportu pojawia się problem szybkiego wdrażania prywatnych stacji ładowania w różnych lokalizacjach miejskich. Częścią problemu związaną z przedsięwzięciem jest stworzenie aplikacji pozwalającej użytkownikom proste wprowadzenie danych ładowarek do sieci blockchain, zachowując przy tym bezpieczeństwo i integralność systemu.

2.1 Struktura łańcucha bloków

Technologia **blockchain** opiera się na łańcuchu bloków, gdzie każda informacja jest zapisana w oddzielnym bloku. Bloki są połączone za pomocą unikalnych identyfikatorów nazywanych **hash**, co sprawia, że zmiana jednego bloku automatycznie wpływa na całą strukturę. Uproszczony schemat działania ciągu bloków przedstawiono na Rys.2.1. Znacząco utrudnia to modyfikację danych bez wykrycia i zwiększa bezpieczeństwo systemu [1]. Sposób działania takiego ciągu znacząco utrudnia to modyfikację danych bez wykrycia i zwiększa bezpieczeństwo systemu.



Rysunek 2.1: Diagram ciągu bloków, w którym klucz każdego bloku zależy od klucza bloku poprzedzającego

2.2 Transakcje w sieci blockchain

Transakcja to zapis jakiegokolwiek wymiany wartości lub informacji między uczestnikami sieci blockchain. Może to być na przykład przesłanie kryptowaluty z jednego portfela do drugiego, zapisanie danych w sieci **blockchain** czy zawarcie inteligentnego kontraktu. Transakcje w sieci **blockchain** są grupowane w bloki i dodawane do łańcucha zgodnie z określonym protokołem konsensusu opisanego w sekcji 2.3. Węzły, czyli komputery uczestniczące w sieci blockchain, weryfikują i zatwierdzają transakcje, zapewniając spójność i integralność danych[5].

2.3 Mechanizm konsensusu

Mechanizm konsensusu to proces, dzięki któremu uczestnicy sieci **blockchain** dochodzą do porozumienia co do stanu łańcucha bloków. W sieci zdecentralizowanej, takiej jak blockchain, nie ma centralnego organu, który weryfikowałby i zatwierdzał transakcje. Zamiast tego, mechanizmy konsensusu pozwalają węzłom w sieci **blockchain** współpracować, aby osiągnąć zgodność.

Najpopularniejszymi mechanizmami konsensusu są algorytmy takie jak Dowód Pracy (ang. *Proof of Work*, *PoW*) oraz Dowód Stawki (ang. *Proof of Stake*, *PoS*)[8]. Algorytm *Proof of Work* używany jest między innymi przez sieć *Bitcoin*, wymaga od węzłów rozwiązania skomplikowanego problemu matematycznego. Ten proces wymaga znacznych zasobów obliczeniowych i energii. Gdy jeden z węzłów rozwiąże problem, inne węzły w sieci zdecentralizowanej łatwo weryfikują poprawność rozwiązania i dodają nowy blok do łańcucha. Algorytm *Proof of Stake* stosowany między innymi przez sieć *Ethereum*, opiera się na posiadaniu kryptowaluty przez węzły. Węzły z większym udziałem mają większą szansę na walidację nowego bloku. Algorytm *Proof of Stake* jest bardziej energooszczędny niż algorytm *Proof of Work*, ponieważ nie wymaga intensywnych obliczeń. Dzięki mecha-

nizmom konsensusu sieć **blockchain** może być bezpieczna, transparentna i odporna na ataki, nawet w przypadku braku centralnej jednostki kontrolującej[3].

2.4 Zdecentralizowane aplikacje

Jednym z kluczowych problemów scentralizowanych aplikacji jest istnienie centralnego punktu awarii, który może zatrzymać cały system. W ostatnich latach rośnie znaczenie zdecentralizowanych sieci w logistyce, gdzie technologia **blockchain** zwiększa bezpieczeństwo i transparentność operacji [7]. Zdecentralizowane podejście, oparte na technologii **blockchain**, zapewnia wyższy poziom bezpieczeństwa i transparentności. Technologie takie jak język programowania **Solidity**, używane do tworzenia inteligentnych kontraktów w sieci **Ethereum**, mogą odegrać kluczową rolę w rozwijaniu nowoczesnych aplikacji do zarządzania ładowaniem pojazdów elektrycznych w dynamicznie rozwijającej się branży. Obecnie jednym z większych problemów związanych z wprowadzaniem płatności w sieciach zdecentralizowanych jest niewielkie zaufanie potencjalnych klientów do kryptowalut w porównaniu z innymi formami waluty [6]. Projektowanie mniejszych, prostszych do wytłumaczenia systemów pracujących w symbiozie z istniejącymi rozwiązaniami może okazać się kluczowe do przekonania potencjalnych inwestorów w przyjęcie nowych technologii. Zdecentralizowane aplikacje są już stosowane na rynkach handlu energią [2]. Innym wykorzystaniem aplikacji zdecentralizowanych jest projektowanie ich pod kątem wspierania zarządzania w branży budowlanej. W artykule *Automation in Construction*[12] autorzy wymieniają między innymi zarządzanie łańcuchem dostaw, przechowywanie dokumentów oraz zarządzanie fizycznymi aktywami jako pola, w których zastosowanie sieci zdecentralizowanych mogłoby przynieść potencjalne korzyści. Artykuł wskazuje też, że wykorzystanie aplikacji zdecentralizowanych może zwiększyć poziom zaufania stron przy przeprowadzaniu zamówień projektowych. Innym zastosowaniem aplikacji zdecentralizowanych zaproponowanym przez J. D. Preece oraz J.M. Easton w artykule *Blockchain Technology as a Mechanism for Digital Railway Ticketing* jest zarządzanie systemach biletowania przejazdów kolejowych [10] Użycie zdecentralizowanych aplikacji zostało też zaproponowane w sektorze farmaceutycznym [4] w ramach pomocy w śledzeniu i potwierdzaniu autentyczności leków.

2.5 Zastosowanie sieci blockchain w elektromobilności

Obecnie dostępne rozwiązania mogące pomóc w zakresie zarządzania ładowaniem samochodów elektrycznych (ang. *electric vehicle, EV*) obejmują zarówno scentralizowane, jak i zdecentralizowane systemy. Aplikacje do zarządzania ładowaniem samochodów elektrycznych muszą sprostać wyzwaniom związanym z integracją wielu technologii, aby uniknąć problemów z przechowywaniem danych potrzebnych do długoterminowej analizy rynkowej. Scentralizowane systemy, chociaż łatwiejsze w implementacji, niosą za sobą ryzyko związane z pojedynczym punktem awarii oraz podatnością na ataki hakerskie. Wiele publikacji takich jak artykuł *Blockchain technology in electromobility and electrification of transport*[13] zauważa potencjał zastosowania sieci **blockchain** w elektromobilności. Aplikacje zdecentralizowane mogą zostać użyte m.in. do przechowywania informacji o naturalnych surowcach użytych do produkcji samochodów elektrycznych. Innym ciekawym zastosowaniem wymienionym w artykule jest śledzenie akumulatorów stosowanych w samochodach elektrycznych w celu uproszczenia ich recyklingu. Jak zaznaczono w powyższych przykładach wdrożenie aplikacji zdecentralizowanych w różnych regionach działalności gospodarczej może przynieść korzyści dla osób zarządzających jak również dla samych użytkowników. Przedstawienie propozycji działania aplikacji zdecentralizowanej w sektorze elektromobilności może być kluczowe dla dalszego rozwoju tej branży, zapewniając większe bezpieczeństwo, transparentność oraz efektywność zarządzania zasobami energetycznymi i infrastrukturą ładowania pojazdów elektrycznych w przyszłości.

Rozdział 3

Wymagania i narzędzia

W ramach pracy zrealizowano zdecentralizowaną aplikację o nazwie *EVCharge*, zajmującą się zarządzaniem siecią ładowarek do samochodów elektrycznych. W ramach projektowania aplikacji zdefiniowano wymagania funkcjonalne i нефункционаłne i przypadki użycia. W rozdziale opisano również użyte w realizacji aplikacji narzędzia i biblioteki programistyczne. W systemie *EVCharge* zdefiniowano trzy typy użytkowników: administrator, sprzedawca oraz klient. Dla każdej z tych ról zdefiniowano wymagania funkcjonalne.

3.1 Wymagania funkcjonalne

Dla roli administratora zdefiniowano następujące wymagania funkcjonalne:

- **Dodawanie nowych użytkowników:** Administrator ma możliwość dodawania nowych użytkowników do systemu, przypisując im odpowiednią rolę (admin, sprzedawca, klient).
- **Rejestrowanie transakcji:** Administrator ma możliwość rejestrowania wypłat sprzedawcom za sesje ładowania, w tym zapisania szczegółów transakcji (kwota, hash transakcji).
- **Wypłata środków:** Administrator ma możliwość wypłaty Etheru z kontraktu do swojego portfela.
- **Przeglądanie podsumowań transakcji:** Administrator może przeglądać podsumowanie wszystkich zarejestrowanych podsumowań transakcji.

Dla roli sprzedawcy zdefiniowano następujące wymagania funkcjonalne:

- **Dodawanie ładowarek:** Sprzedawca może dodać nowe ładowarki do systemu, ustalając ich lokalizację i cenę za kWh energii.
- **Aktualizacja cen ładowarek:** Sprzedawcy mogą modyfikować ceny za kWh dla swoich ładowarek.

- **Monitorowanie dochodów:** Sprzedawcy mogą śledzić swój udział w dochodach związanych z ładowaniem, tzn. mają wgląd w swoje zarobki z sesji ładowania.
- **Przeglądanie historii sesji ładowania:** Sprzedawcy mogą przeglądać historię sesji ładowania związanych z ich ładowarkami.

Dla roli klienta zdefiniowano następujące wymagania funkcjonalne:

- **Ładowanie pojazdu:** Klienci mogą rejestrować się na dostępnych ładowarkach, wskazując zapotrzebowanie na energię (w kWh) oraz akceptując cenę za ładowanie. Sesja jest następnie autoryzowana, a koszt jest odejmowany z jego salda.
- **Zwiększenie salda:** Klient może wpłacać środki w ETH, aby zasilić swoje konto, co umożliwia mu korzystanie z usług ładowania.
- **Przeglądanie swojego salda:** Klient ma możliwość sprawdzenia swojego salda oraz historii sesji ładowania.
- **Przeglądanie historii sesji ładowania:** Użytkownicy mogą przeglądać historię swoich sesji ładowania.

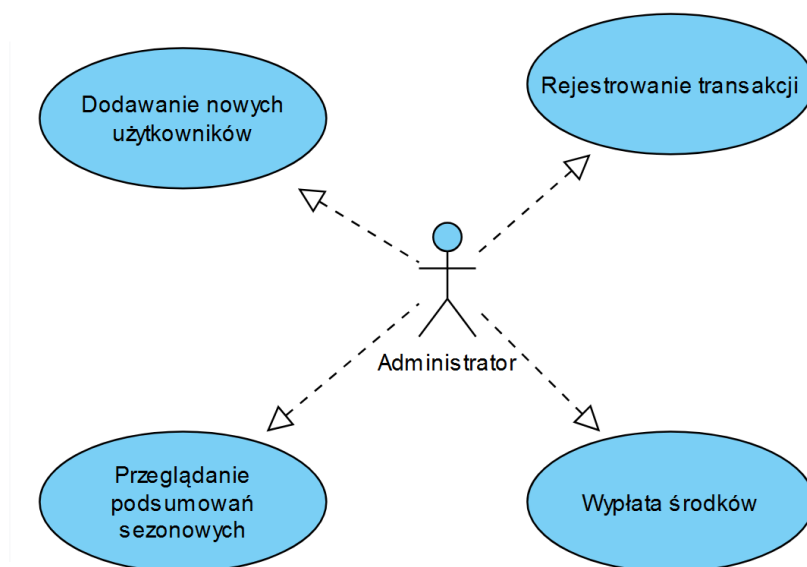
3.2 Wymagania niefunkcjonalne

Jako podstawę bezpieczeństwa aplikacji zdefiniowano wymaganie niefunkcjonalne systemu. System zapewnia bezpieczne mechanizmy autoryzacji przy każdej operacji, w tym podpisy cyfrowe dla transakcji ładowania samochodu elektrycznego. W przypadku logowania się do warstwy logiki aplikacji użytkownik jest proszony o podanie hasła. Hasła są bezpiecznie przechowywane w zaszyfrowanej formie w lokalnej bazie danych. Interfejs aplikacji (warstwa prezentacji) musi być łatwy w obsłudze, zarówno dla użytkowników administrowania, jak i dla klientów. Aplikacja powinna być dostępna przez przeglądarki internetowe dla uzyskania większej dostępności dla użytkowników końcowych. Aplikacja powinna działać w przeglądarce *Mozilla Firefox* oraz przeglądarkach opartych na projekcie *chromium* takich jak *Google Chrome* lub *Brave*.

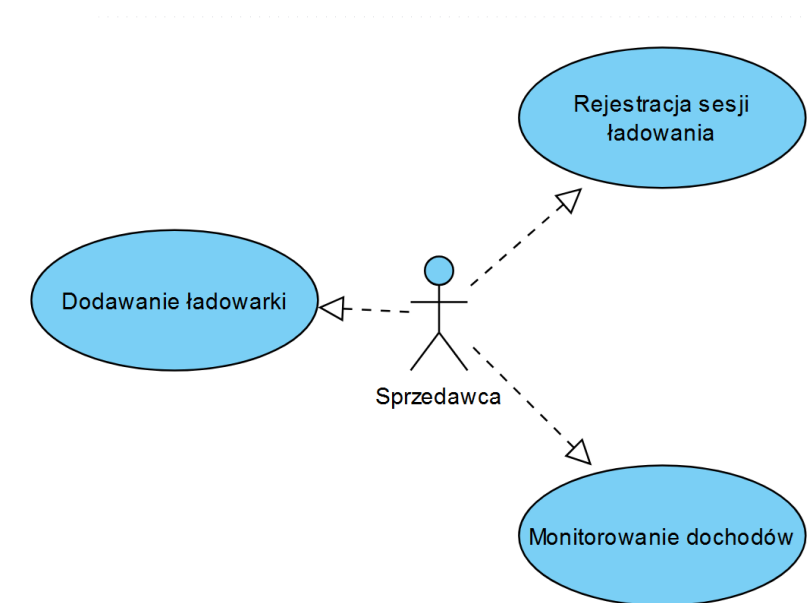
W ramach wymagań dotyczących transparentności zdefiniowano następujące wymaganie niefunkcjonalne: Wszystkie transakcje związane z płatnościami oraz sesjami ładowania muszą być zapisywane w sieci **blockchain**, by zapewnić ich integralność i transparentność dla wszystkich uczestników. Dla lepszej dostępności i usprawnienia walidacji transakcji część nowych danych może być przechowywana w lokalnej bazie danych.

3.3 Przypadki użycia

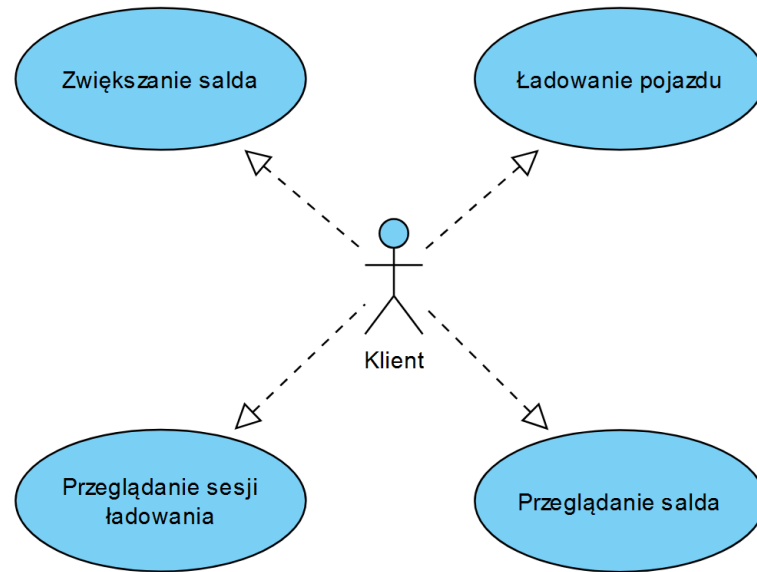
W celu szczegółowego przedstawienia przypadków użycia w systemie *EVCharge* wykonano po jednym diagramie przypadków użycia dla każdego typu użytkownika. Rys.3.1 przedstawia diagram przypadków użycia użytkownika o roli administratora. Rysunki 3.2 oraz 3.3 przedstawiają diagramy przypadków użycia użytkowników o rolach sprzedawcy i klienta.



Rysunek 3.1: Diagram przypadków użycia dla administratora



Rysunek 3.2: Diagram przypadków użycia dla sprzedawcy



Rysunek 3.3: Diagram przypadków użycia dla klienta

Wśród zaprezentowanych wcześniej przypadków użycia wybrano dwa które zaprezentowano szczegółowo w scenariuszach w punktach 3.3.1 oraz 3.3.2

3.3.1 Scenariusz ładowania samochodu elektrycznego przez klienta

Aktor główny: Klient

Warunki wstępne:

- Klient jest zarejestrowany w systemie.
- Klient jest zalogowany na swoje konto.

Warunki końcowe:

- Sesja ładowania jest zainicjowana i śledzona w systemie.
- Klient otrzymuje potwierdzenie sesji ładowania.

Scenariusz główny:

1. Klient wpisuje adres ładowarki w polu *Adres Ładowarki* w sekcji *Ładuj EV* w systemie.
2. Klient wpisuje liczbę kilowatogodzin, za które chce zapłacić w polu *Zapotrzebowanie (kWh)* w sekcji *Ładuj EV*.
3. Klient wybiera opcję *Wyślij transakcję* w systemie.
4. System weryfikuje, czy klient istnieje w systemie.

5. System sprawdza, czy ładowarka istnieje w systemie.
6. System sprawdza, czy klient posiada wystarczającą liczbę ETH w przypisanym koncie.
7. System przesyła informacje o sesji ładowania do sieci zdecentralizowanej.
8. System zapisuje informacje o sesji ładowania w lokalnej bazie danych.
9. Klient otrzymuje potwierdzenie sesji ładowania.

Scenariusze alternatywne:

5a. Ładowarka nie istnieje:

1. System informuje klienta o i wyświetla błąd.
2. Klient może zmodyfikować żądanie ładowania lub zakończyć proces.

7a. Saldo klienta jest niewystarczające:

1. System informuje klienta o niewystarczającym saldzie.
2. Klient dodaje środki do salda lub kończy proces sesji ładowania.

3.3.2 Scenariusz dodawania ładowarki do systemu przez zalogowanego sprzedawcę

Aktor główny: Sprzedawca

Warunki wstępne:

- Sprzedawca jest zarejestrowany w systemie.
- Sprzedawca jest zalogowany na swoje konto,

Warunki końcowe:

- Ładowarka jest dodana/zaktualizowana w systemie.
- Sprzedawca otrzymuje potwierdzenie akcji.

Scenariusz główny:

1. Sprzedawca wpisuje adres ładowarki w polu *Adres Ładowarki*,
2. Sprzedawca wprowadza adres ładowarki w polu *Adres Ładowarki* i preferowaną cenę za kilowatogodzinę w polu *Zapotrzebowanie (kWh)*,
3. Sprzedawca wybiera opcję *Wyślij transakcję*,

4. System weryfikuje wprowadzone dane,
5. System wprowadza dane do sieci zdecentralizowanej,
6. System wprowadza dane ładowarki do lokalnej bazy danych,
7. Sprzedawca otrzymuje potwierdzenie akcji.

Scenariusze alternatywne:

4a. Ładowarka istnieje już w sieci zdecentralizowanej:

1. System informuje sprzedawcę o problemie.
2. Sprzedawca aktualizuje dane lub kończy proces dodawania ładowarki.

3.4 Narzędzia

Do napisania i testowania aplikacji użyto następujących narzędzi:

- **Angular** - platforma użyta do napisania warstwy prezentacji programu,
- **Django** - platforma wykorzystana do napisania warstwy logiki aplikacji,
- **Sqlite3** - technologia użyta do obsługi lokalnej bazy danych programu,
- **Hardhat** - narzędzie do tworzenia, testowania i wdrażania inteligentnych kontraktów – wspiera warstwę zdecentralizowaną,
- **Solidity** - język użyty do opisanie logiki zdecentralizowanej w sieci *Ethereum*,
- **Solidity Coverage** - oprogramowanie do testowania zakresu pokrycia testów logiki zdecentralizowanej,
- **Slither** - narzędzie służące do analizy bezpieczeństwa, optymalizacji i jakości kodu Solidity,
- **Synpress** - platforma do testowania aplikacji od początku do końca i dokumentowania testów,
- **Metamask** - narzędzie do przechowywania adresów prywatnych użytkowników, ułatwia identyfikację użytkowników w sieci zdecentralizowanej.

Rozdział 4

Specyfikacja zewnętrzna

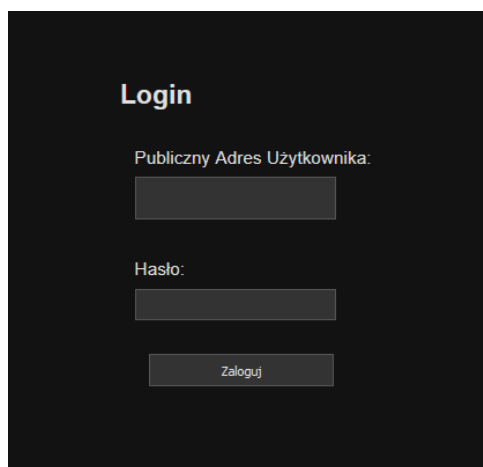
Ten rozdział omawia warstwę prezentacji działającej w przeglądarce internetowej.

4.1 Wymagania sprzętowe

Aby korzystać z aplikacji, użytkownik końcowy potrzebuje komputera ze stabilnym łączem internetowym. Stacja powinna mieć zainstalowaną dowolną przeglądarkę internetową obsługującą rozszerzenie Metamask. Niezbędne jest też posiadanie konta w rozszerzeniu *Metamask* oraz przypisanego do niego portfela kryptograficznego.

4.2 Logowanie

Stroną początkową w aplikacji jest strona logowania (Rys. 4.1). Użytkownik wprowadza adres publiczny konta powiązanego z jego adresem w sieci **blockchain**. Po wprowadzeniu hasła i zatwierdzenia danych uwierzytelniania zostaje przekierowany na widok powiązany z jego typem użytkownika.



Rysunek 4.1: Strona logowania aplikacji.

4.3 Widok administratora

Po zalogowaniu się przez administratora zostaje on przekierowany na podstronę widoczną na zrzutach ekranu widocznych na Rys. 4.2 oraz Rys. 4.3. W pierwszej części interfejsu (Rys. 4.3 może przeglądać wygenerowane wcześniej sezonowe podsumowania wpływów sprzedawców, a także opłacać je. Sezonowe podsumowania opisują adres sprzedawcy będący jego adresem publicznym w sieci **blockchain**, jego obecne *saldo* i *wkład*. *Wkład* reprezentuje liczbę ETH, którą sprzedawca zarobił w ramach oferowania klientom usług ładowania samochodu elektrycznego. Podsumowanie zawiera też całkowite zapotrzebowanie reprezentujące liczbę kilowatogodzin, która została załadowana przez jego klientów. *Całkowita należność* oznacza sumę w WEI jaka powinna zostać wypłacona przez administratora w ramach finalizacji podsumowania. Podsumowanie zawiera też pola *data rozpoczęcia* oraz *data zakończenia* wyznaczające zakres czasowy, w którym objęto wszystkie usługi ładowania, których udzielił on swoim klientom. Pole *Utworzono* zawiera dokładną datę, o której administrator utworzył to podsumowanie. Pole *Zapłacono* zawiera wartość *tak* lub wartość *nie*, zależną od tego czy administrator opłacił już sprzedawcę za oferowanie swoich usług w tym zakresie czasowym. W przypadku gdy w polu *zapłacono* znajduje się opcja *nie*, w dolnej części panelu podsumowania znajduje się przycisk *Zapłać*. Przycisk wywołuje funkcję, która łączy się z rozszerzeniem *Metamask* i oferuje użytkownikowi propozycję zapłaty. W przypadku gdy administrator wykona płatność pole *zapłacono* będzie zawierało tekst *tak* przy następnym odświeżeniu strony.

Drugą część interfejsu stanowi ekran generacji podsumowań oraz dodawania użytkowników przedstawiony na Rys. 4.3. Administrator może wybrać zakres dat, z których chce wygenerować podsumowanie lub nacisnąć przycisk *wygeneruj podsumowanie* w celu automatycznego wypełnienia pól *data początku* oraz *data końca*. W tym przypadku pole *data początku* będzie datą najstarszej nierozliczonej transakcji ładowania w systemie. Pole *data końca* będzie datą dnia poprzedzającego obecny dzień. Jeśli generowane podsumowanie obejmuje zakres czasowy pokrywający się z istniejącym podsumowaniem, wyświetlony zostanie błąd. Zalecane jest każdorazowe pozyskanie automatycznej daty, która pokryje obszar pomiędzy ostatnim nieuwzględnionym ładowaniem samochodu elektrycznego w systemie.

Podsumowania Sprzedawców

Załaduj ponownie

Adres sprzedawcy: 0x90F79bf6EB2c4f870365E785982E1f101E93b906

Rola: Seller

Saldo: 0

Wkład: 0

Całkowite zapotrzebowanie: 2615

Całkowita należność: 60145

Data rozpoczęcia: 2023-01-01

Data zakończenia: 2025-01-06

Utworzono: 2025-01-07 15:37:02

Zapłacono: tak

Rysunek 4.2: Pierwsza część interfejsu administratora

Generacja Podsumowań

Automatyczna data

Data Początku

dd . mm . rrrr

Data Końca

dd . mm . rrrr

Wygeneruj podsumowanie

Automatyczne dobranie daty dobiera datę ostatniego podsumowania i wczorajszą datę. Zalecane w większości przypadków użycia

Dodaj Użytkownika

Adres Użytkownika

Rola

Klient

Dodaj użytkownika

Rysunek 4.3: Druga część interfejsu administratora

15

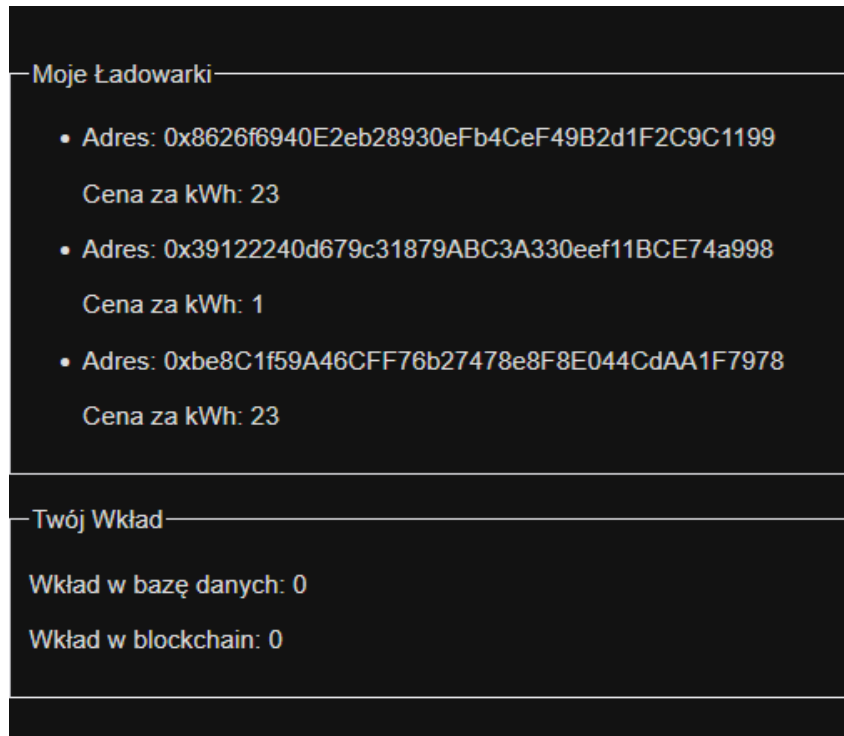
4.4 Widok sprzedawcy

Rys. 4.4 oraz Rys. 4.5 przedstawiają interfejs sprzedawcy. Sprzedawca może dodać ładowarkę wprowadzając jej adres i preferowaną cenę w ETH jaką musi zapłacić klient za każdą kilowatogodzinę, którą chce załadować swój pojazd elektryczny. Następnie potwierdza wysłanie wiadomości przyciskiem *Dodaj Ładowarkę*. Sprzedawca może aktualizować cenę swoich ładowarek wpisując ich adres, wprowadzając nową cenę i klikając przycisk *Zaktualizuj Cenę*. W przypadku gdy zostanie napotkany błąd pod interfejsem zostanie wypisany tekst informujący o nieprawidłowości. Błędem może być wpisanie niepoprawnego adresu jak adres niepasujący do żadnej ładowarki w systemie.

The image shows a dark-themed user interface for a seller. It contains two main sections. The first section, titled 'Dodaj Ładowarkę', has a label 'Adres Ładowarki:' followed by a text input field containing '3612653cD682D926'. Below this is a label 'Cena za kWh:' followed by a numeric input field containing '20,1' and a small up/down arrow icon. At the bottom of this section is a button labeled 'Dodaj Ładowarkę'. The second section, titled 'Aktualizuj Cenę Ładowarki', has a label 'Adres Ładowarki' followed by a text input field containing '330eef11BCE74a998'. Below this is a label 'Nowa Cena za kWh' followed by a numeric input field containing '15' and a small up/down arrow icon. At the bottom of this section is a button labeled 'Zaktualizuj Cenę'.

Rysunek 4.4: Pierwsza część interfejsu sprzedawcy

W drugiej części interfejsu (Rys. 4.5) sprzedawca może przejrzeć obecny stan swoich ładowarek, ich adresy publiczne oraz wyznaczone ceny za ładowanie samochodu elektrycznego w przeliczeniu na kilowatogodzinę. Widoczny jest też jego wkład, jaki zostanie mu wypłacony przez administratora przy okazji następnego podsumowania przy polu *wkład w bazę danych*, a także pełny wkład od czasu założenia jego konta w sieci **blockchain**: *Wkład w blockchain*. Wkład reprezentowany jest przez liczbę naturalną i oznacza liczbę ETH.



Rysunek 4.5: Druga część interfejsu sprzedawcy

4.5 Widok klienta

Rys. 4.6 oraz Rys. 4.7 przedstawiają interfejs klienta. Pierwsza część interfejsu pozwala na walidację swojej sesji ładowania. Użytkownik podaje adres ładowarki, którą chce aktywować oraz ilość kWh, jaką chce doładować. Adres podawany jest w polu typu przechowującym tekst. Następnie przyciskiem *Wyślij transakcję* przesyła polecenie do warstwy logiki aplikacji. Użytkownik posiada też możliwość doładowania konta dowolną kwotą w ETH. Doładowanie konta wiąże się z wykonaniem transakcji w sieci zdecentralizowanej. Przed wykonaniem takiej transakcji użytkownik musi posiadać odpowiednią kwotę na swoim koncie. Wykonanie transakcji klient finalizuje potwierdzając jej wykonanie w rozszerzeniu *Metamask* 4.8.

Zalogowano jako: klient

Połącz

Ładuj EV

Adres Ładowarki:

0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199

Zapotrzebowanie (kWh):

120

Wyślij transakcję

Doładuj konto

Kwota w wei:

10

Doładuj

Rysunek 4.6: Pierwsza część interfejsu klienta

Pokaż Moje Transakcje

Ukryj Moje Transakcje

• ID Transakcji: 4

Adres Klienta:
0xd2FD4581271e230360230F9337D5c0430Bf44C0

Adres Ładowarki:
0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199

Koszt całkowity: 10419

Ilość kWh: 453

Adres Właściciela:
0x90F79bf6EB2c4f870365E785982E1f101E93b906

Data: 2023-01-01 13:45

• ID Transakcji: 91

Adres Klienta:
0xd2FD4581271e230360230F9337D5c0430Bf44C0

Adres Ładowarki:
0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199

Koszt całkowity: 2829

• ID Transakcji: 90

Adres Klienta:
0xd2FD4581271e230360230F9337D5c0430Bf44C0

Adres Ładowarki:
0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199

Koszt całkowity: 24380

Ilość kWh: 1060

Adres Właściciela:
0x90F79bf6EB2c4f870365E785982E1f101E93b906

Data: 2025-01-02 13:22

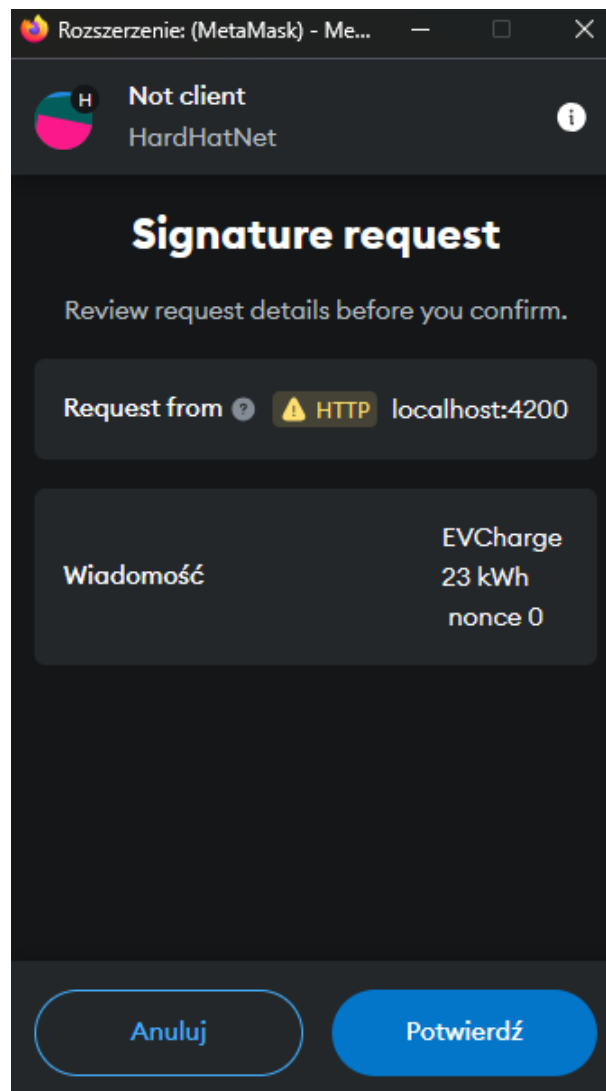
• ID Transakcji: 92

Adres Klienta:
0xd2FD4581271e230360230F9337D5c0430Bf44C0

Adres Ładowarki:
0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199

Koszt całkowity: 2829

Rysunek 4.7: Druga część interfejsu klienta



Rysunek 4.8: Potwierdzenie transakcji w rozszerzeniu *Metamask*

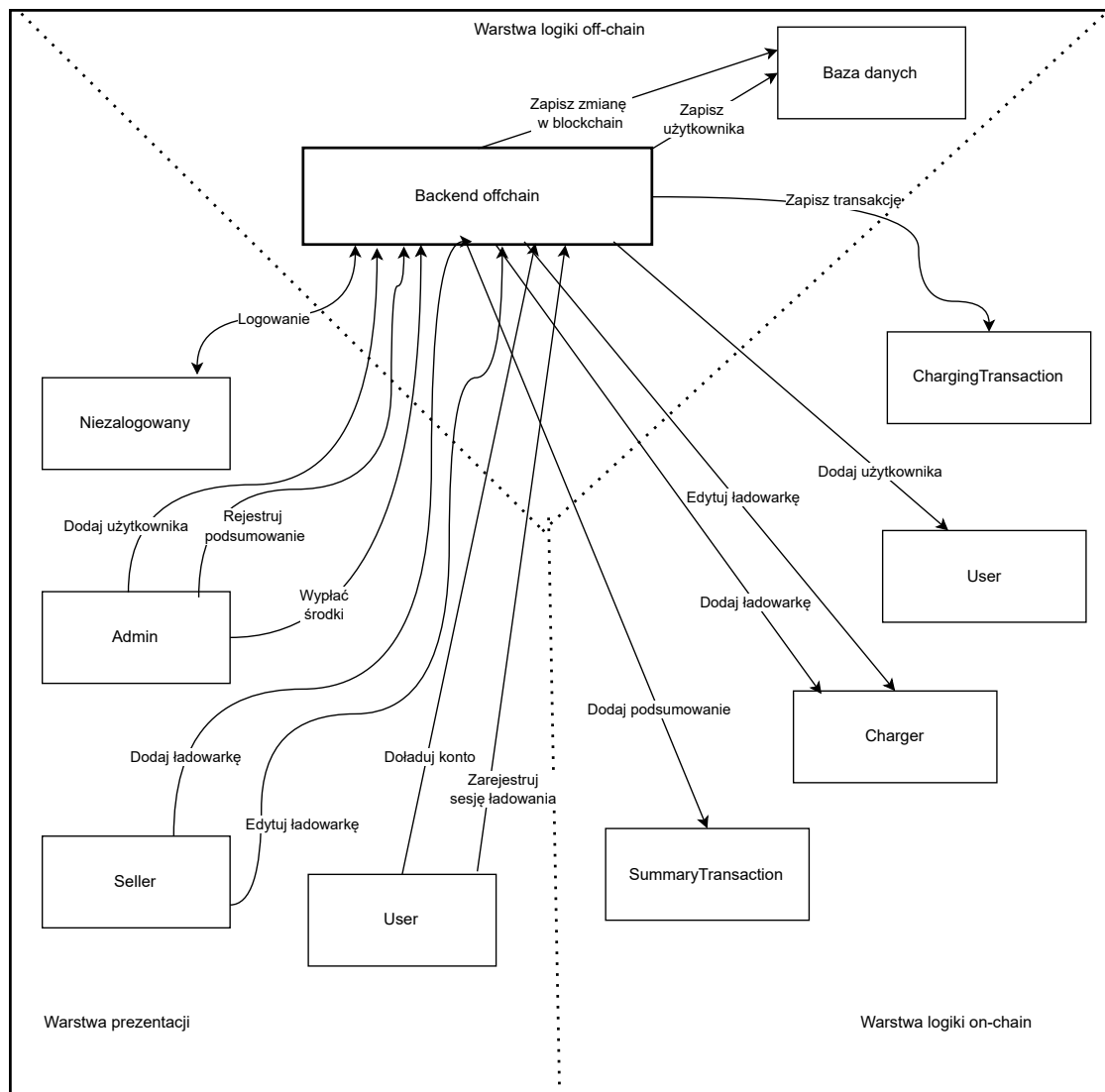
Rozdział 5

Specyfikacja wewnętrzna

5.1 Architektura systemu

Trzonem aplikacji jest kod inteligentnego kontraktu napisany w języku **solidity**. Jest on reprezentowany na Rys. 5.1 pod nazwą *on-chain*. Stanowi on logikę działania zdecentralizowanej bazy danych programu na podstawie której działa reszta aplikacji. Inteligentny kontrakt odpowiada za zbieranie informacji o użytkownikach systemu, uwierzytelnianie ich oraz wykonywanie działań na danych zbieranych w sieci **blockchain**. We fragmencie kodu 5.1 zawarto typy danych przechowywane przez inteligentny kontrakt. Aplikacja warstwy logiki stanowi sposób na szybką komunikację z systemem. Warstwa logiki lokalnej komunikuje się ze inteligentnym kontraktem za pomocą konta pierwszego administratora. Pozwala to na weryfikację żądań do sieci **blockchain** kierowanych zarówno przez klienta jak i administratora jednocześnie pozwalając na lepszą kontrolę nad przepływem danych.

Oznacza to, że do sieci **blockchain** zostaną przesłane tylko niezbędne dane w czasie gdy opcjonalne informacje będą przechowywane w lokalnej bazie danych. Baza danych została zaimplementowana przy użyciu technologii **sqlite3** ze względu na prostotę użycia i modyfikacji. Na Rys. 5.1 warstwa logiki lokalnej jest reprezentowana przez sekcję *off-chain*. Kontakt z warstwą logiki użytkownika końcowego odbywa się za pomocą warstwy prezentacji. Została ona napisana w platformie **Angular 19** ze względu na jego popularność i wszechstronność. Warstwa prezentacji w założeniu pozwala na łatwą komunikację między użytkownikiem końcowym a warstwą logiki bez potrzeby posiadania wiedzy technicznej. Warstwa prezentacji łączy się z rozszerzeniem **Metamask** w celu podpisywania wiadomości z zachowaniem bezpieczeństwa prywatnych kluczy użytkowników.

Rysunek 5.1: Podział aplikacji na sekcje *on-chain*, *off-chain* oraz warstwę prezentacji

5.2 Przechowywanie danych

W ramach warstwy logiki zdecentralizowanej zaprojektowano jeden inteligentny kontrakt. Inteligentny kontrakt reprezentuje logikę biznesową aplikacji. Inteligentny kontrakt został zaimplementowany z myślą o przechowywaniu danych z pojedynczymi transakcjami ładowania samochodów elektrycznych, informacje o użytkownikach systemu, ładowarkach i wypłatach administratorów. Poniżej opisano fragment kodu odpowiedzialny za zdefiniowanie struktur służących do przechowywania danych (Patrz fragment kodu 5.1). Do przechowywania danych transakcyjnych zaimplementowano strukturę *ChargingSession*. Zawiera ona dane o pojedynczym ładowaniu samochodu elektrycznego, w tym adresy publiczne użytkowników oraz szczegóły ładowarek zaangażowanych w transakcję. Adresy te mogą zostać użyte do zidentyfikowania struktur *User* odpowiedzialnych za przechowywa-

nie danych o pojedynczych użytkownikach i ładowarkach. Poza tym zaimplementowano też strukturę *SummaryTransaction* odpowiedzialną za przechowywanie unikalnych kluczy (*ang. hash*) transakcji sezonowych wypłat, przechowującą też liczbę wypłaconego sprzedawcy ETH, jego publiczny adres w sieci **blockchain** oraz adres publiczny obecnego administratora.

```
1 pragma solidity ^0.8.0;
2 enum Role { None, Admin, Seller, Client }
3 struct User {
4     address userAddress;
5     Role role;
6     int balance;
7     uint32 contribution;
8 }
9 struct Charger {
10    address chargerAddress;
11    uint pricePerKWh;
12    address owner;
13 }
14 struct ChargingSession {
15    address clientAddress;
16    address chargerAddress;
17    uint32 totalCost;
18    uint32 demand;
19    address chargerOwner;
20 }
21 struct SummaryTransaction {
22    address admin;
23    address seller;
24    uint256 amountPaid;
25    bytes32 transactionHash;
26 }
27 SummaryTransaction[] public summaryTransactions;
28 mapping(address => User) public users;
29 mapping(address => Charger) public chargers;
30 address public immutable initialAdmin;
31 address[] public userAddresses;
32 ChargingSession[] public chargingSessions;
```

Kod źródłowy 5.1: Fragment kodu w języku *Solidity* ilustrującego dane przechowywane w inteligentnym kontrakcie.

5.3 Warstwa logiki zdecentralizowanej

Warstwa logiki zdecentralizowanej jest głównym trzonem aplikacji. Inteligentny kontrakt przy inicjalizacji automatycznie wdraża predefiniowany adres pierwszego administratora. Po wdrożeniu do sieci zdecentralizowanej inteligentny kontrakt działa niezależnie od innych warstw systemu *EVCharge*. Poniżej wylistowano wszystkie funkcje, które wchodzi w skład inteligentnego kontraktu.

- **addUser**: Dodawanie nowych użytkowników
- **addCharger**: Dodawanie nowej ładowarki
- **updateChargerPrice**: Aktualizacja ceny ładowarki
- **getAllChargingSessions**: Sprawdzenie wszystkich sesji ładowania
- **getAllUsers**: Sprawdzenie wszystkich użytkowników
- **checkMyBalance**: Sprawdzenie salda klienta
- **checkMyContribution**: Sprawdzenie zarobków sprzedawcy
- **validateClient**: Weryfikacja klienta i autoryzacja sesji ładowania
- **adminWithdraw**: Wypłata środków przez administratora
- **increaseBalance**: Zwiększenie salda klienta
- **recordTransaction**: Rejestrowanie transakcji podsumowujących
- **getSigner**: Pobranie adresu podpisującego wiadomość
- **splitSignature**: Podział podpisu na składniki

Poniżej opisano zasadę działania kodu inteligentnego kontraktu na podstawie wybranych funkcji w języku *Solidity*:

```

1  function validateClient(address clientAddress ,
2      address chargerAddress ,
3      uint32 demand ,
4      bytes32 messageHash ,
5      bytes memory signature)
6      public returns (bool) {
7      require(users[clientAddress].userAddress != address(0), "
        User_does_not_exist");
8      require(users[clientAddress].role == Role.Client, "User_
        is_not_a_client");
9      require(chargers[chargerAddress].chargerAddress !=
        address(0), "Charger_does_not_exist");
10     require(getSigner(messageHash, signature) ==
        clientAddress, "Bad_signature");
11     uint32 totalCost = uint32(chargers[chargerAddress].
        pricePerKWh * uint256(demand));
12     require(totalCost >= chargers[chargerAddress].pricePerKWh
        , "Cost_overflow");
13     require(uint256(users[clientAddress].balance) >=
        totalCost, "Client_has_insufficient_balance");
14     chargingSessions.push(ChargingSession({
15         clientAddress: clientAddress ,
16         chargerAddress: chargerAddress ,
17         totalCost: totalCost ,
18         demand: demand ,
19         chargerOwner: chargers[chargerAddress].owner
20     }));
21     users[clientAddress].balance -= int256(uint256(totalCost)
        );
22     address chargerOwner = chargers[chargerAddress].owner;
23     users[chargerOwner].contribution += totalCost;
24
25     bool isAuthorized = true;
26     emit ChargingAuthorized(clientAddress, chargerAddress ,
        demand, isAuthorized);    // for IoT charger
27     return isAuthorized;
28 }

```

Kod źródłowy 5.2: Kod funkcji *validateClient* w języku *Solidity* odpowiadający za weryfikację użytkownika i dodawanie danych jego sesji ładowania do sieci zdecentralizowanej.

Funkcja **validateClient** przedstawiona na Rys 5.2 po otrzymaniu parametrów rozpoczyna działanie od sprawdzenia, czy podany klient istnieje w systemie i czy ma przypisaną rolę *Client*. Kolejno sprawdza, czy podana ładowarka istnieje. Następnie inteligentny kontrakt analizuje podpis klienta, sprawdzając jego zgodność z adresem użytkownika. Jeśli otrzymane parametry nie spełniają wymagań weryfikacyjnych, funkcja zwraca odpowiedni komunikat o błędzie. Jeśli podane parametry są poprawne, inteligentny kontrakt oblicza całkowity koszt ładowania na podstawie ceny za kilowatogodzinę i zgłoszonego zapotrzebowania. Następnie weryfikuje, czy saldo klienta jest wyższe niż koszt transakcji, aby mieć pewność, że może on pokryć koszty ładowania. Następnie inteligentny kontrakt upewnia się, że obliczona wartość nie przekracza maksymalnych dopuszczalnych wartości liczbowych, zapobiegając przepełnieniu liczbowemu — błędowi występującemu przy skrajnie wysokich wartościach liczbowych. Ten konkretny test nie powinien nigdy odrzucić żadnego z realnych żądań klienta, ma on na celu ochronę systemu przed błędami wywołanymi przez nieuczciwych użytkowników. Jeśli warunki są spełnione, funkcja zapisuje nową sesję ładowania w systemie. Koszt sesji zostaje odjęty od salda klienta, a równocześnie dodany do wkładu właściciela ładowarki. Na końcu funkcja wysyła zdarzenie (*ang. event*) informujące o autoryzacji ładowania jako *event* do sieci zdecentralizowanej. Informuje to teoretyczną podłączoną do sieci zdecentralizowanej aplikację obsługującą ładowarkę o tym, że klient został zautoryzowany i można rozpocząć ładowanie jego samochodu elektrycznego.

5.4 Warstwa logiki lokalnej

Lokalna baza danych przechowuje te same typy danych, które występują w inteligentnym kontrakcie. W tym przypadku dane są przechowywane w relacyjnej bazie danych. W bazie występują także dodatkowe tabele służące do rejestracji, logowania i przechowywania danych o sesji użytkownika wygenerowanych przez bibliotekę **django**.

Aplikacja warstwy logiki lokalnej korzysta ze standardowych bibliotek **django rest** służących do modyfikacji zawartości bazy danych. Korzysta także z bibliotek **jwt** do obsługi logowania. Do kreacji sygnatur w celu autentykacji w inteligentnym kontrakcie oraz komunikacji z nim użyto biblioteki **eth utils** oraz **py web3**. Struktura plików w folderze warstwy logiki lokalnej wygląda następująco:

```
EVCHARGE/  
|-- __pycache__/  
|-- migrations/  
|-- __init__.py  
|-- admin.py  
|-- asgi.py  
|-- EVCharge.code-workspace  
|-- models.py  
|-- permissions.py  
|-- serializers.py  
|-- settings.py  
|-- signals.py  
|-- tokens.py  
|-- urls.py  
|-- views.py  
|-- wsgi.py
```

Główna funkcjonalność kodu warstwy logiki lokalnej znajduje się w plikach *models.py*, *urls.py* oraz *views.py*. Plik *models.py* definiuje, w jaki sposób dane aplikacji będą przechowywane w tabelach w bazie danych. Dla przykładu fragment kodu obsługujący tabelę przechowującą informacje o zarejestrowanych ładowarkach wygląda następująco:

```
1 class Charger(models.Model):
2     charger_address = models.CharField(max_length=42, unique=True
3     )
4     price_per_kwh = models.IntegerField()
5     owner = models.ForeignKey(User, on_delete=models.CASCADE,
6     to_field='user_address', related_name='chargers')
7
8     class Meta:
9         db_table = 'charger'
```

Kod źródłowy 5.3: Fragment kodu w języku *Python* w pliku *models.py* odpowiadający za reprezentację klasy *Charger*.

Klasa *Charger* posiada pola *charger_address* *price_per_kwh* obsługujące kolejno adres ładowarki w sieci zdecentralizowanej, cenę, jaką klient musi zapłacić za doładowanie samochodu elektrycznego w przeliczeniu na pojedynczą kilowatogodzinę oraz pole *owner* będące odwołaniem do pola *user_address* w tabeli przechowującej dane użytkowników. Plik *urls.py* przechowuje adresy sieciowe punktów końcowych programu, które będą dostępne dla warstwy prezentacji programu. Przykładowo linia 24 pliku *urls.py*

```
path("api/pay-seller/", views.paySellerView.as_view(), name='pay-seller'),
```

odpowiada za przekierowanie żądania opłacenia sezonowego podsumowania przekazanego przez administratora poprzez warstwę prezentacji do odpowiedniej funkcji zdefiniowanej w pliku *views.py*.

Poniżej opisano zasadę działania kodu inteligentnego kontraktu na podstawie funkcji służącej do wypłacania sezonowych należności sprzedawcy przez administratora w pliku *views.py* (Patrz kod źródłowy 5.4).

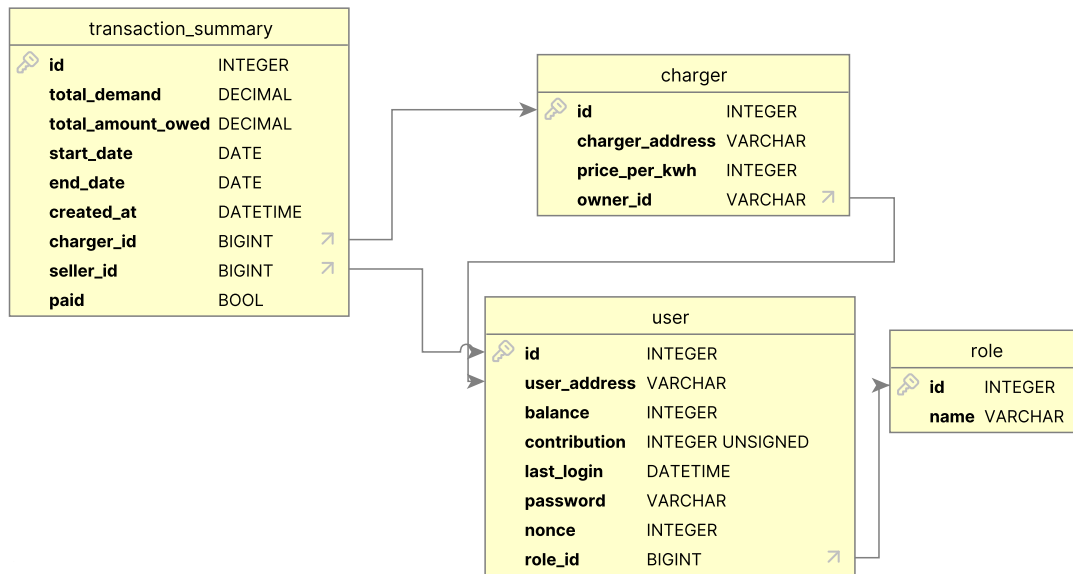
Funkcja *paySellerView* w **Django** zaczyna się od sprawdzenia, czy użytkownik wykonujący żądanie jest administratorem. Z nagłówka żądania *Authorization* wyciągany jest token JWT, który jest dekodowany za pomocą klucza sekretne (*SECRET_KEY*). Następnie sprawdzane jest, czy rola użytkownika to *Admin*. Jeśli rola użytkownika nie jest zgodna z *Admin*, zwracany jest błąd *unauthorized (401)*. W przypadku, gdy użytkownik jest administratorem, funkcja pobiera dane transakcji z bazy danych na podstawie identyfikatora przekazanego w żądaniu z warstwy prezentacji (*request.data*) i wyszukuje odpowiednią transakcję w modelu *TransactionSummary*. Z tej transakcji pobierany jest adres odbiorcy (sprzedawcy) oraz całkowita kwota do zapłaty (*total_amount_owed*). Następnie przygotowywana jest transakcja w sieci *Ethereum* z informacjami o nadawcy (adres serwera), odbiorcy (adres sprzedawcy) oraz kwocie do zapłaty, a także ustalane są wartości *gas* i *gasPrice* - koszt wykonania transakcji w sieci zdecentralizowanej. Transakcja jest

podpisywana prywatnym kluczem serwera (*server_private_key*) i wysyłana do sieci *Ethereum*. Funkcja czeka na potwierdzenie transakcji (*receipt*), a po otrzymaniu szczegółów zapisuje *hash* transakcji w bazie danych. Na koniec status transakcji w bazie danych jest zmieniany na *paid*, a funkcja zwraca odpowiedź do warstwy prezentacji z potwierdzeniem sukcesu oraz hashem transakcji.

```
1 class paySellerView(APIView):
2     def post(self, request):
3         #auth check if admin
4         auth_header = request.headers.get('Authorization')
5         token = auth_header.split(' ')[1]
6         decoded_token = jwt.decode(token, settings.SECRET_KEY,
7                                     algorithms=['HS256'])
8         user_role = decoded_token.get('role')
9         if(user_role != 'Admin'):
10             return Response({"status": "unauthorized"}, status=
11                               status.HTTP_401_UNAUTHORIZED)
12         summary = TransactionSummary.objects.get(id=request.data)
13         sender_address = server_public_key
14         recipient_address = summary.seller
15         total_amount_owed = int(summary.total_amount_owed)
16         tx = {
17             "nonce": w3.eth.get_transaction_count(sender_address)
18             ,
19             "to": to_bytes(hexstr=recipient_address.user_address)
20             ,
21             "value": total_amount_owed,
22             "gas": 21000,
23             "gasPrice": w3.to_wei(50, "gwei"),
24         }
25         signed_tx = w3.eth.account.sign_transaction(tx,
26                                                       server_private_key)
27         tx_hash = w3.eth.send_raw_transaction(signed_tx.
28                                                 raw_transaction)
29         receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
30         receipt_hash = receipt['transactionHash'].hex()
31         summary.paid = True
32         summary.save()
33         return Response({"status": "success", "data":
34                           receipt_hash}, status=status.HTTP_200_OK)
```

Kod źródłowy 5.4: Fragment kodu w języku *Python* z pliku *views.py* odpowiadający za obsługę funkcjonalności przekazywania sezonowych wypłat sprzedawcom przez administratora.

W aplikacji warstwy logiki użyto relacyjnej bazy danych. Struktura bazy danych została przedstawiona na diagramie 5.2



Rysunek 5.2: Diagram bazy danych.

5.5 Warstwa prezentacji

Warstwa prezentacji aplikacji została napisana przy użyciu platformy *Angular 19*. Platforma ta opiera się na architekturze komponentowej. Oznacza to, że przy projektowaniu strony zostaje ona podzielona na komponenty. Komponenty to mniejsze moduły wielokrotnego użytku zawierające informacje o tym, jak powinny wyglądać fragmenty strony oraz jak powinny się zachować w ich interakcjach z użytkownikiem. Ich struktura opisywana jest w języku znaczników *HTML*. Strona wizualna komponentów opisywana jest językiem *CSS*. Zachowanie komponentów oraz sposób ich zastosowania w kontekście do reszty kodu reprezentowany jest przez pliki z rozszerzeniami *spec.ts* oraz *.ts* opisane w języku programowania *Typescript*. Przykład kodu w języku *Typescript* określającego zachowanie komponentu *dashboard* wykorzystywanego do zarządzania widokiem klienta przedstawiono we fragmencie kodu 5.5. Widać na nim w jaki sposób importowane są zależności między tym komponentem a innymi, a także sposób, w który pobiera dane transakcyjne, które później wykorzystywane są do wyświetlania historii transakcji klienta.

```
1 import { Component } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3 import { ShowTransactionsComponent } from '../show-transactions/
  show-transactions.component';
4 import { ResultComponent } from '../result/result.component';
5 import { SendTransactionComponent } from '../send-transaction/
  send-transaction.component';
6 import { IncreaseBalanceComponent } from '../increase-balance/
  increase-balance.component';
7 import { TopbarComponent } from '../topbar/topbar.component';
8
9 @Component({
10   standalone: true,
11   imports: [CommonModule, ShowTransactionsComponent,
    ResultComponent,
12     SendTransactionComponent, IncreaseBalanceComponent,
    TopbarComponent],
13   selector: 'app-dashboard',
14   templateUrl: './dashboard.component.html',
15   styleUrls: ['./dashboard.component.css']
16 })
17 export class DashboardComponent {
18   resultData: any;
19   transactionData: any[] = [];
20   updateResultData(data: any) {
21     this.resultData = data;
22   }
23   handleDataFetched(data: any[]): void { this.transactionData =
    data; }
24 }
```

Kod źródłowy 5.5: Fragment kodu w języku *TypeScript* z pliku *dashboard.component.ts*, odpowiadający za obsługę funkcjonalności widoku klienta.

Struktura aplikacji *EVCharge* składa się z czterech komponentów głównych: *login* przechowujący podstronę pozwalającą na zalogowanie się osoby niezalogowanej, *dashboard*, będącym reprezentacją strony klienta, *seller-dashboard* będący podstroną sprzedawcy oraz *admin-dashboard* będący podstroną administratora. W skład każdej z tych podstron wchodzi mniejsze komponenty służące do wykonywania pojedynczych akcji, takich jak wysłanie zapytania do punktu końcowego warstwy logiki lokalnej.

Podstrona administratora używa komponentu *seller-summaries* do zarządzania sezo-

nowymi podsumowaniami, komponentu *add-user* do dodawania użytkowników, komponentu *admin-summaries* służącego do generowania nowych podsumowań sezonowych.

Podstrona sprzedawcy używa komponentu *add-charger* służącego do dodawania nowej ładowarki do systemu, komponentu *update-price* służącego do aktualizowania ceny płaconej przez użytkownika za kilowatogodzinę oraz komponentu *my-chargers* służącego do prezentowania jego zarejestrowanych ładowarek oraz ich szczegółów oraz komponentu *contribution* służącego do wyświetlania jego wkładu w sieci zdecentralizowanej.

Podstrona klienta korzysta z komponentów *send-transaction* służącego do wysłania do warstwy logiki lokalnej żądania ładowania samochodu elektrycznego, komponentu *increase-balance* służącego do doładowania konta w systemie *EVCharge* oraz komponentu *show-transactions* pozwalającego na wylistowanie poprzednich transakcji ładowania samochodu elektrycznego w systemie. W każdej podstronie zastosowano też komponent *topbar* wyświetlający nazwę aplikacji.

W warstwie prezentacji zaimplementowano też usługę *connect-service* umożliwiającą wysyłanie przez internet poleceń formułowanych przez komponenty. Komunikacja warstwy prezentacji bez tej usługi nie byłaby możliwa. Dla uproszczenia nawigowania pomiędzy poszczególnymi podstronami i ewentualnej rozbudowy strony wykorzystano funkcjonalność platformy *Angular* nazywaną *Angular Router*. Pozwala ona na przypisywanie odpowiednich komponentów podstron do odpowiednich adresów sieciowych. Została ona użyta m.in. w celu prostego przekierowywania użytkownika na odpowiednią podstronę przy logowaniu w zależności od jego roli w systemie.

5.6 Wymagania kompilacji

W celu pierwszego uruchomienia systemu należy wdrożyć inteligentny kontrakt do wybranej sieci *Ethereum* lub sieci z nią kompatybilnej. W celu edycji warstwy prezentacji użytkownik musi uprzednio zainstalować platformę **Angular 19**. W celu wprowadzenia zmian implementacyjnych na warstwie logiki lokalnej użytkownik musi zainstalować na stacji roboczej bibliotekę **Django Rest**. W celu wprowadzenia zmian w logice warstwy logiki zdecentralizowanej użytkownik może edytować kod w języku **Solidity** w dowolnym zgodnym środowisku programistycznym.

W celu uruchomienia warstwy aplikacji należy nawigować do odpowiedniego folderu, a następnie wpisać komendę *ng serve*. W celu uruchomienia warstwy logiki lokalnej należy uruchomić bibliotekę **Django Rest**. Aby to zrobić należy nawigować do odpowiedniego folderu, a następnie wpisać komendę *py manage.py runserver*. Zalecane jest uruchomienie serwera przy wykorzystaniu środowiska wirtualnego tworzonego za pomocą polecenia:

```
python -m venv nazwa_środowiska
```

a uruchamianego komendą:

```
nazwa_środowiska\Scripts\activate
```

W celu uruchomienia warstwy aplikacji zdecentralizowanej należy wdrożyć inteligentny kontrakt w wybranej testowej sieci zdecentralizowanej. W czasie testowania aplikacji używano środowiska **Hardhat**, gdzie uruchamiano lokalny węzeł sieci **blockchain** poleceniem:

```
npx hardhat node
```

a następnie wdrażano inteligentny kontrakt do sieci **blockchain** poleceniem:

```
npx hardhat run deploy.js --network localhost
```


Rozdział 6

Weryfikacja i walidacja

Testowanie aplikacji podzielono na trzy etapy. Pierwszym z nich były testy manualne. Drugą część testów obejmowała napisanie testów automatycznych oraz audytu bezpieczeństwa. Ostatnią częścią było testowanie aplikacji z użyciem realnych danych wejściowych.

Aplikację testowano manualnie w przeglądarkach Google Chrome oraz Mozilla Firefox w systemach operacyjnych Windows 11 oraz Ubuntu Desktop 24.10. Po stwierdzeniu że każdy przewidywany standardowy przykład użycia został manualnie przetestowany, kod sprawdzono za pomocą zewnętrznych narzędzi. Do tej fazy testów użyto oprogramowania **Solidity Coverage** oraz **Slither**. Po zakończeniu testów manualnych aplikację przetestowano pod kątem osiągnięcia pełnego pokrycia testami automatycznymi kodu inteligentnego kontraktu oraz pod kątem zdolności do obsługi realnych danych wejściowych. Dla każdego przypadku użycia i każdego możliwego rozgałęzienia napisano przynajmniej jeden odpowiadający test. Następnie sprawdzono czy osiągnięto pełnię pokrycia za pomocą narzędzia **Solidity Coverage**.

6.1 Przykład testu manualnego

W tej sekcji opisano pojedynczy przykład testu manualnego. Test ma na celu sprawdzenie, aplikacja działa w pełnym zakresie zarówno ze strony użytkownika jak i w warstwach logiki. Scenariusz polega na przeprowadzeniu przez użytkownika pojedynczej sesji ładowania samochodu elektrycznego. W tym celu użytkownik loguje się do aplikacji, a następnie wprowadza w pola *Adres Ładowarki* oraz *Zapotrzebowanie* odpowiednie wartości. Następnie użytkownik wybiera opcję *Wyślij transakcję* (Rys. 6.1b). Użytkownik otrzymuje prośbę weryfikacji żądania przez rozszerzenie *Metamask* (Rys. 6.2). Zakładając poprawność danych po weryfikacji żądania, powinno ono zostać wysłane do bazy danych, a także powinno zostać zarejestrowane w następnym bloku w sieci zdecentralizowanej.

```

eth_sendTransaction
  Contract call:      EVCharge#validateClient
  Transaction:        0x897294fe1d4af0f300cfbef5ff922b989ed7203a82e8f95b9069
ed8d5f6f1bb4
  From:               0xf39fd6e51aad88f6f4ce6ab8827279cfff92266
  To:                 0x5fbdb2315678afecb367f032d93f642f64180aa3
  Value:              0 ETH
  Gas used:           159987 of 259987
  Block #4:           0xe23c4a4b0a05119966e21b79653ecfc31a65869a0ca24e550f77
9b17d3e7ab38

```

(a) Transakcja widoczna w sieci zdecentralizowanej zarejestrowana w środowisku *Hardhat*.

id	total_cost	demand	is_complete	date	charger_ad	charger_ow	client_address_id
1	4	10419	453	0 2023-01-01 ...	1	3	18
2	90	24380	1060	0 2025-01-02 ...	1	3	18
3	91	2829	123	0 2025-01-02 ...	1	3	18
4	92	2829	123	0 2025-01-02 ...	1	3	18
5	93	2829	123	0 2025-01-02 ...	1	3	18
6	94	2829	123	0 2025-01-02 ...	1	3	18
7	95	782	34	0 2025-01-03 ...	1	3	18
8	96	10419	453	0 2025-01-03 ...	1	3	18
9	97	2829	123	0 2025-01-04 ...	1	3	18
10	98	28106	1222	0 2025-01-07 ...	1	3	18
11	99	2829	123	0 2025-01-07 ...	1	3	18
12	100	28428	1236	0 2025-01-07 ...	1	3	18
13	101	15111	657	0 2025-01-07 ...	1	3	18
14	102	529	23	0 2025-02-06 ...	1	3	18

(b) Fragment interfejsu użytkownika w warstwie prezentacji odpowiadający za wysyłanie żądania ładowania samochodu elektrycznego.

Ładuj EV

Adres Ładowarki:

0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199

Zapotrzebowanie (kWh):

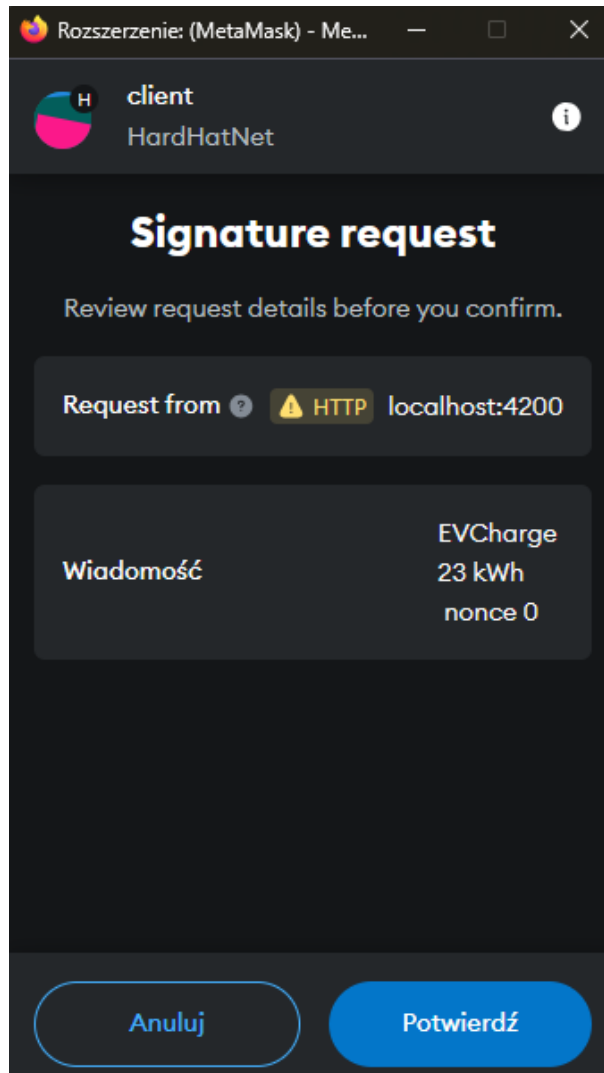
23

Wyślij transakcję

(c) Stan tabeli *charging_session* w bazie danych. Ostatnia linijka zawiera dane transakcji przeprowadzonej w ramach testu manualnego.

Rysunek 6.1: Zrzuty ekranu przedstawiające przebieg testu manualnego.

Jak widać na Rys. 6.1c. żądanie zostało pomyślnie zarejestrowane przez lokalną bazę danych. Rys. 6.1a to zrzut ekranu pokazujący, że żądanie użytkownika zostało poprawnie zapisane w sieci zdecentralizowanej i potwierdzone przez środowisko *Hardhat*.



Rysunek 6.2: Potwierdzenie transakcji przez klienta w rozszerzeniu *Metamask*.

6.2 Testy automatyczne

Do przeprowadzenia testów automatycznych wykorzystano bibliotekę *chai* potrzebną do symulowania wykonywania fragmentów kodu bez konieczności przeprowadzania testów manualnie. Dla całego systemu opracowano łącznie 37 standardowych automatycznych testów kolektywnie sprawdzających każde rozgałęzienie kodu w każdej z funkcji inteligentnego kontraktu. Jak zaprezentowano na zrzucie ekranu 6.3 testy zakończyły się pomyślnie. Jak widać na załączonym zrzucie ekranu Rys. 6.4 udało się pokryć w pełni wszystkie ścieżki w programie.

We fragmencie kodu 6.1 przedstawiono przykład dwóch testów sprawdzających poprawność działania funkcji *updateChargerPrice*. Służy ona do zmiany kwoty w ETH jaką ładowarka będzie obciążać konto klienta w przeliczeniu na kilowatogodzinę.

```
● PS D:\inż tmp\GRID\blockchain2> npx hardhat test

EVCharge
  ✓ should initialize with the correct admin
  ✓ should allow admin to add a user
  ✓ should allow a client to increase balance
  ✓ should revert when no ETH is sent with increaseBalance
  ✓ should revert if user is not client
  ...
splitSignature
  ✓ Should fail if signature length is invalid
  ✓ Should split a valid signature correctly

37 passing (39s)
```

Rysunek 6.3: Wynik przeprowadzonych testów automatycznych.

all files / contracts/ EVCharge.sol

100% Statements 53/53 100% Branches 48/48 100% Functions 14/14 100% Lines 67/67

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.28;
3 contract EVCharge {
4     enum Role { None, Admin, Seller, Client }
5 }
```

Rysunek 6.4: Fragment raportu wygenerowanego przez narzędzie **Solidity Coverage**.

```

1 it("should allow seller to update charger price", async function
  () {
2   await evCharge.connect(seller).addCharger(testAddressCharger,
    20);
3   await evCharge.connect(seller).updateChargerPrice(
    testAddressCharger, 30);
4   const charger = await evCharge.chargers(testAddressCharger);
5   expect(charger.pricePerKWh).toEqual(30);
6 });
7 it("Should fail if caller is not a seller", async function () {
8   await expect(
9     evCharge.connect(client).updateChargerPrice(
      chargerAddress, 50)
10    ).to.be.revertedWith("Only seller can update charger price");
11 });

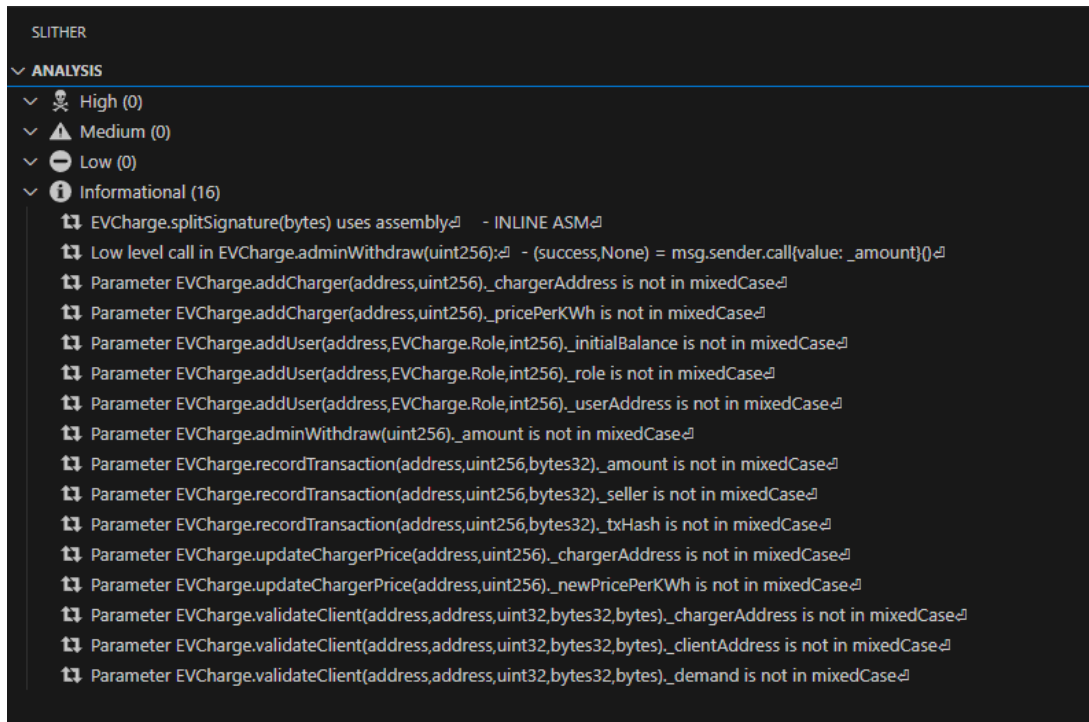
```

Kod źródłowy 6.1: Przykład testu sprawdzającego poprawność funkcji *updateChargerPrice*.

Pierwszy test sprawdza, czy funkcja poprawnie przyjmuje żądanie użytkownika w przypadku gdy autoryzowany sprzedawca próbuje ją wykonać na swojej ładowarce. W celu zwiększenia uniwersalności testu ładowarka zostaje dodana przez konto sprzedawcy na początku trwania testu. Następnie konto sprzedawcy wysyła żądanie zmiany ceny przypisanej do nowo utworzonej ładowarki. Test weryfikuje, czy cena ładowania została poprawnie zaktualizowana do nowej wartości określonej przez sprzedawcę. Drugi test sprawdza, czy funkcja poprawnie odrzuca żądanie użytkownika jeśli nie posiada on roli sprzedawcy. W tym celu użyto testowego konta o roli klienta. Zgodnie z oczekiwaniami próba wykonania funkcji w tym przypadku powinna skończyć się niepowodzeniem. Dla samej funkcji *updateChargerPrice* napisano jeszcze dwa testy. Pierwszy z nich sprawdza, czy funkcja poprawnie nie dopuszcza do zmiany ceny ładowania ładowarki nienależącej do użytkownika nawet jeśli posiada on rolę sprzedawcy, a także test sprawdzający, czy funkcja zwraca poprawny błąd w przypadku, gdy podany adres ładowarki nie pasuje do żadnej z ładowarek w systemie.

6.3 Testowanie pod kątem bezpieczeństwa

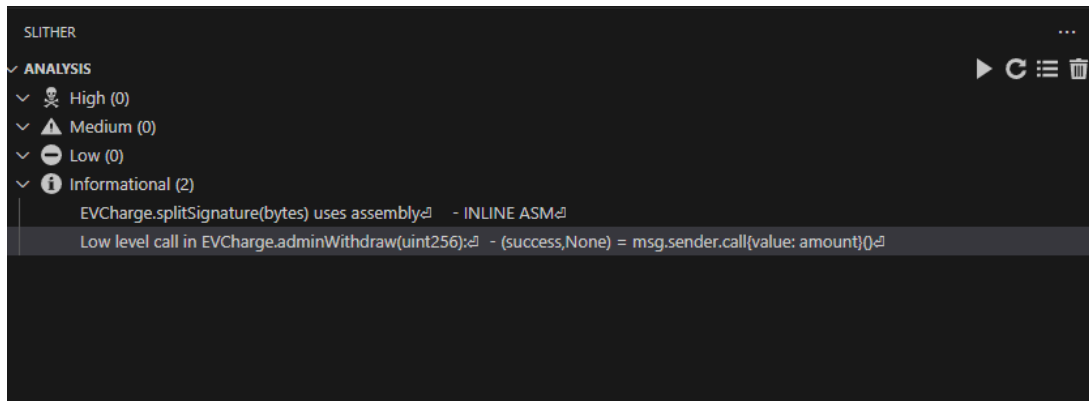
Po zakończeniu testów manualnych oraz automatycznych testów jednostkowych przeprowadzono audyt bezpieczeństwa za pomocą narzędzia *Slither*. Testy miały na celu znalezienie podatności kodu na wykorzystanie jego funkcjonalności przez nieuczciwych użytkowników. *Slither* jest narzędziem pozwalającym na skanowanie inteligentnych kontraktów pod kątem zagrożeń dla bezpieczeństwa aplikacji. Jak widać na rysunku 6.5 nie używano w kodzie zalecanego typu zapisu.



Rysunek 6.5: Raport Slither przed zastosowaniem poprawek.

Parametry funkcji były opisane z charakterystycznym dla języka Python podkreśleniem `__` przed nazwą zmiennej. Po usunięciu przedrostka Slither nie wypisywał już więcej powiadomień tego typu.

Innymi informacjami podanymi przez Slither były powiadomienia o niskopoziomych odwołaniach w kodzie. Pierwsze z nich dotyczyło użycia fragmentu kodu w języku **Assembly** w funkcji `splitSignature`. Jest on używany w programie do prostego podziału sygnatury na jej części składowe w celu zweryfikowania jej poprawności. O ile użycie niskopoziomowych odwołań w języku **Solidity** może być niebezpieczne, w tym przypadku jest ono uzasadnione.



Rysunek 6.6: Raport Slither po wprowadzeniu poprawek.

6.4 Wykryte i usunięte błędy

W czasie pisania testów wykryto kilka błędów w kodzie. Przy testowaniu fragmentu kodu odpowiadającego za sprawdzenie, czy potwierdzenie transakcji zostało wysłane przez administratora systemu wykryto, że sprawdzanie roli wysyłającego wiadomość nie ma sensu w tym kontekście (Fragment kodu 6.2). Kod w żadnym przypadku użycia nie mógł już odrzucić żądania. Jeśli użytkownik nie byłby administratorem systemu, to jego żądanie zostanie odrzucone wcześniej, niezależnie od tego, jaka rola została przypisana do jego konta. Kod został zmodyfikowany, by wyglądać w załączonym fragmencie kodu 6.3.

```

1      function recordTransaction(address seller , uint256 amount ,
2          bytes32 txHash) public {
3          require(msg.sender == initialAdmin , "Only admin can
4              record transactions");
5          require(users[seller].role == Role.Seller , "Recipient
6              must be a seller");
7          require(amount > 0 , "Amount must be greater than zero");
8          summaryTransactions.push(SummaryTransaction({
9              admin: msg.sender ,
10             seller: seller ,
11             amountPaid: amount ,
12             transactionHash: txHash
13         }));
14     }

```

Kod źródłowy 6.2: Funkcja *recordTransaction* w języku **Solidity** przed edycją.

```
1    function recordTransaction(address seller , uint256 amount ,
2        bytes32 txHash) public {
3        require(msg.sender == initialAdmin , "Only_admin_can_
4            record_transactions");
5        require(amount > 0 , "Amount_must_be_greater_than_zero");
6        summaryTransactions.push(SummaryTransaction({
7            admin: msg.sender ,
8            seller: seller ,
9            amountPaid: amount ,
10           transactionHash: txHash
11        }));
12    }
```

Kod źródłowy 6.3: Funkcja *recordTransaction* w języku **Solidity** po edycji.

Kolejny błąd występował we fragmencie kodu 6.4. W czasie testowania fragmentu kodu odpowiadającego za sprawdzenie, czy potwierdzenie transakcji zostało wysłane przez sprzedawcę, wykryto, że sprawdzanie, czy użytkownik istnieje w systemie, nie ma sensu w tym kontekście. Kod w żadnym przypadku użycia nie mógł już odrzucić żądania. Jeśli użytkownik nie istnieje, to jego żądanie zostanie odrzucone wcześniej, niezależnie od tego, jaka rola została przypisana do jego konta. Kod został zmodyfikowany (Patrz kod źródłowy 6.5)

```
1    function checkMyContribution() public view returns (int) {
2        require(users[msg.sender].role == Role.Seller , "User_is_
3            not_a_seller");
4        require(users[msg.sender].userAddress != address(0) , "
5            User_does_not_exist");
6        return int32(users[msg.sender].contribution);
7    }
```

Kod źródłowy 6.4: Funkcja *checkMyContribution* w języku **Solidity** przed edycją.

```
1    function checkMyContribution() public view returns (int) {
2        require(users[msg.sender].role == Role.Seller , "User_is_
3            not_a_seller");
4        return int32(users[msg.sender].contribution);
5    }
```

Kod źródłowy 6.5: Funkcja *checkMyContribution* w języku **Solidity** po edycji.

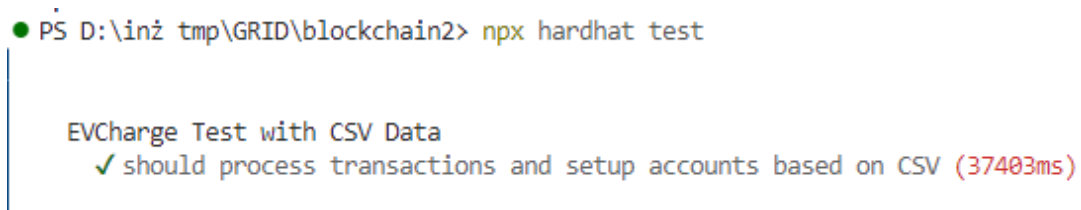
6.5 Testowanie z użyciem realnych danych wejściowych

Aplikację testowano też za pomocą danych pozyskanych z kaggle.com[11]. Są to dane realnych sesji ładowania samochodów elektrycznych z Norwegii stanowiących podstawę do przetestowania aplikacji na podstawie realnych danych wejściowych. Oryginalny zbiór danych zawierał 72857 sesji ładowania. Duża część z nich zawierała wartość 0 w polu *UserID* co prawdopodobnie było domyślną wartością wyznaczoną dla brakujących pól w zbiorze danych. Na potrzeby testów usunięto ten fragment zbioru. Zbiór wciąż pozostawał większy, niż było to potrzebne na potrzeby sprawdzenia poprawności działania systemu, ograniczono więc jego zakres do pierwszych 29 użytkowników wyznaczonych na podstawie pola *UserID*. Zbiór nie zawierał satysfakcjonującego podziału na sprzedawców. Pole *ChargerCompany* zawierało zaledwie dwie unikalne wartości: 0 oraz 1. Zamiast tego zdecydowano, że unikalni sprzedawcy zostaną wyznaczeni na podstawie pola *Location* zawierającego 12 unikalnych wartości. Symuluje to istnienie wielu sieci ładowarek operowanych przez 12 różnych sprzedawców.

UserID	ChargerID	ChargerCompany	Duration	Demand
14	511	public institution	56	28.47
15	37	company	169	10

Tabela 6.1: Przykładowe wiersze danych wejściowych

Ostatecznie dane trafiały do kodu w postaci takiej jak przedstawiono w tabeli 6.1. Dla każdego klienta tworzono losowo wygenerowane przez bibliotekę **ethers.js** konto powiązane z polem *UserID*, a następnie wprowadzano je do systemu za pomocą konta administratora. Następnie na podstawie pola *ChargerCompany* tworzono konta sprzedawców, używając tej samej metodyki. Sprzedawcom przypisywano ładowarki na podstawie pola *ChargerID*. Następnie symulowano sesję ładowania samochodu użytkownika i wysyłano dane sesji zawierające dane z pól *Duration* oraz *Demand* odpowiadające długości sesji, oraz zapotrzebowania użytkownika w kilowatogodzinach. W rezultacie zapisano 2115 osobnych sesji ładowania zarejestrowanych przez 12 sprzedawców na 315 zarejestrowanych ładowarkach. Test zakończył się pomyślnie, bez wartych zanotowania problemów, jak widać na Rys. 6.7.



```
PS D:\inż tmp\GRID\blockchain2> npx hardhat test

EVCharge Test with CSV Data
  ✓ should process transactions and setup accounts based on CSV (37403ms)
```

Rysunek 6.7: Wynik przeprowadzonych testów automatycznych z wykorzystaniem zbioru realnych danych wejściowych.

Rozdział 7

Podsumowanie i wnioski

Celem pracy było stworzenie zdecentralizowanej aplikacji działającej w sieci blockchain. Na potrzeby realizacji pracy wybrany został problem zarządzania siecią ładowarek do pojazdów elektrycznych. Stworzona została aplikacja *EVCharge* umożliwiająca zarządzanie ładowarkami, realizację płatności w ETH oraz monitorowanie transakcji i dochodów przez użytkowników systemu. Następnie aplikacja została przetestowana manualnie i automatycznie, w tym przy użyciu danych rzeczywistych. Wykonano też audyt bezpieczeństwa inteligentnego kontraktu i poprawiono błędy zauważone w całym procesie testowania. Pomyślnie zrealizowano założenia projektowe. Cel pracy został w pełni osiągnięty.

7.1 Kierunki ewentualnych dalszych prac

Praca spełnia wymagania, które zostały postawione przy jej rozpoczęciu, jednak warto zaznaczyć kilka regionów, w których może zostać rozszerzona. Funkcjonalność inteligentnego kontraktu może zostać poszerzona przez dodanie obsługi bardziej złożonych przypadków transakcji czy generowania raportów. Możliwe jest też pokrycie inteligentnego kontraktu systemem sygnatur w celu weryfikacji każdej akcji użytkowników w systemie. Integracja z zewnętrznymi systemami płatności lub innymi usługami opartymi na sieci **blockchain** zwiększyłaby uniwersalność systemu. Możliwe jest też udoskonalenie warstwy prezentacji, aby lepiej dostosować ją do potrzeb użytkowników. Obecna warstwa aplikacji spełnia swoją rolę, ale jej estetyka jest prosta i surowa. Kolejnym krokiem byłaby poprawa efektywności warstwy logiki, szczególnie w zakresie obsługi dużych ilości danych i integracji z bardziej zaawansowanymi bazami danych. Aplikacja powinna też zostać przetestowana poza siecią lokalną, na przykład w realnej testowej sieci **blockchain** takiej jak **Sapphire testnet**, co pozwoliłoby na dokładne przebadanie jej działania, biorąc pod zmienne warunki takie jak opóźnienia sieci i przerwania w łączu internetowym.

7.2 Problemy napotkane w trakcie pracy

Podczas realizacji projektu napotkano kilka wyzwań, takich jak trudności związane z zapewnieniem odpowiedniego poziomu bezpieczeństwa w inteligentnym kontrakcie, w tym ochrona przed potencjalnymi atakami i manipulacją danymi. Zarządzanie kilkoma, a czasami kilkunastoma środowiskami programistycznymi w celu połączenia i testowania każdej z warstw wymagało dobrej organizacji stanowiska. Koszty realnych transakcji w sieci **blockchain** uniemożliwiają przetestowanie aplikacji w realnej sieci bez ponoszenia wysokich kosztów finansowych. Projekt, mimo wspomnianych wyzwań, został zrealizowany w sposób zadowalający, co stanowi solidną podstawę dla jego dalszego rozwoju i ewentualnego zastosowania.

Bibliografia

- [1] Bennet Maria Putri Ayu Sanjaya Rahmania Az Zahra 2024. „Blockchain Technology: Revolutionizing Transactions in the Digital Age”. W: *Engineering Science & Technology Journal* 5 (mar. 2024), 192–199. DOI: 10.34306/ajri.v5i2.1065. URL: <https://www.adi-journal.org/index.php/ajri/article/view/1065>.
- [2] Ayman Esmat, Martijn de Vos, Yashar Ghiassi-Farrokhfal, Peter Palensky i Dick Epema. „A novel decentralized platform for peer-to-peer energy trading market with blockchain technology”. W: *Applied Energy* 282 (sty. 2021), s. 116123. DOI: 10.1016/j.apenergy.2020.116123.
- [3] Bashir Imran. *Blockchain. Zaawansowane zastosowania łańcucha bloków*. Helion, 2019.
- [4] Robert Muliawan Jaya, Valentino Dhamma Rakkhitta, Pranata Sembiring, Ivan Sebastian Edbert i Derwin Suhartono. „Blockchain applications in drug data records”. W: *Procedia Computer Science* 216 (2023). 7th International Conference on Computer Science and Computational Intelligence 2022, s. 739–748. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2022.12.191>. URL: <https://www.sciencedirect.com/science/article/pii/S1877050922022694>.
- [5] Daniel Cawrey Lorne Lantz. *Blockchain. Przewodnik po technologii łańcucha bloków. Kryptowaluty, inteligentne kontrakty i aplikacje rozproszone*. Helion, 2022.
- [6] Marta Maciejasz i Robert Poskart. *Percepcja współczesnego pieniądza i kryptowalut w świetle badań ankietowych na przykładzie młodych użytkowników pieniądza z Czech i Polski*. Red. Bogusława Drelich-Skulska i Magdalena Sobocińska. Wrocław: Wydawnictwo Uniwersytetu Ekonomicznego we Wrocławiu, 2023, s. 288–302. DOI: 10.15611/2023.67.1.16. URL: <http://hdl.handle.net/11089/52526>.
- [7] Rizwan Manzoor, B. Sahay i Sujeet Singh. „Blockchain technology in supply chain management: an organizational theoretic overview and research agenda”. W: *Annals of Operations Research* (list. 2022). DOI: 10.1007/s10479-022-05069-5.
- [8] Marcin Niedopytalski. *Blockchain i sztuczna inteligencja w ochronie danych: bezpieczne przechowywanie i analiza*. 2024.

- [9] Aldona Podgórnai-Krzykacz i Zuzanna Karaś. *Rozwój elektromobilności w Polsce: analiza społecznego postrzegania i akceptacji samochodów elektrycznych przez Polaków*. Łódź: Wydawnictwo Uniwersytetu Łódzkiego, 2024. ISBN: 978-83-8331-459-4. DOI: 10.18778/8331-459-4.8. URL: <http://hdl.handle.net/11089/52526>.
- [10] Joe Preece i John Easton. „Blockchain Technology as a Mechanism for Digital Railway Ticketing”. W: *Blockchain Technology as a Mechanism for Digital Railway Ticketing*. Lut. 2020, s. 3599–3606. DOI: 10.1109/BigData47090.2019.9006293.
- [11] Ansh Tanwar. *Zestaw danych "Residential EV Charging Data"*, www.kaggle.com. 2024. URL: <https://www.kaggle.com/datasets/anshtanwar/residential-ev-chargingfrom-apartment-buildings/data> (term. wiz. 10.12.2024).
- [12] Yuqing Xu, Xingyu Tao, Moumita Das, Helen Kwok, Hao Liu, Guangbin Wang i Jack Cheng. „Suitability analysis of consensus protocols for blockchain-based applications in the construction industry”. W: *Automation in Construction* 145 (sty. 2023), s. 104638. DOI: 10.1016/j.autcon.2022.104638.
- [13] Anna ZIELIŃSKA. „Blockchain technology in electromobility and electrification of transport”. W: *PRZEGLĄD ELEKTROTECHNICZNY* (sty. 2024).

Dodatki

Spis skrótów i symboli

- **EV** - samochód elektryczny (ang. *electric vehicle*, *EV*).
- **EVCharge** - ogólna nazwa całości zaprojektowanego systemu obejmującego każdą z warstw aplikacji.
- **Inteligentny Kontrakt** - ang. *Smart Contract* to kod zaprojektowany w celu automatyzacji przeprowadzania transakcji i modyfikowania danych przechowywanych w sieci zdecentralizowanej.
- **Blockchain** - zdecentralizowana baza danych, przechowująca informacje w blokach połączonych w łańcuch. Każda zmiana danych jest widoczna i niemożliwa do sfałszowania, co zapewnia bezpieczeństwo i autentyczność informacji. W niniejszej pracy nazywany dla uproszczenia siecią zdecentralizowaną.
- **ETH, Ethereum** - Kryptowaluta, w której płacą użytkownicy *EVCharge*.
- **WEI** - Najmniejsza jednostka *Ethereum* ($1 \text{ ETH} = 10^{18} \text{ Wei}$).
- **PoS** - Mechanizm konsensusu *Proof of Stake*.
- **PoW** - Mechanizm konsensusu *Proof of Work*.

Spis rysunków

2.1	Diagram ciągu bloków, w którym klucz każdego bloku zależy od klucza bloku poprzedzającego	4
3.1	Diagram przypadków użycia dla administratora	9
3.2	Diagram przypadków użycia dla sprzedawcy	9
3.3	Diagram przypadków użycia dla klienta	10
4.1	Strona logowania aplikacji.	13
4.2	Pierwsza część interfejsu administratora	15
4.3	Druga część interfejsu administratora	15
4.4	Pierwsza część interfejsu sprzedawcy	16
4.5	Druga część interfejsu sprzedawcy	17
4.6	Pierwsza część interfejsu klienta	18
4.7	Druga część interfejsu klienta	18
4.8	Potwierdzenie transakcji w rozszerzeniu <i>Metamask</i>	19
5.1	Podział aplikacji na sekcje <i>on-chain</i> , <i>off-chain</i> oraz warstwę prezentacji . .	22
5.2	Diagram bazy danych.	31
6.1	Zrzuty ekranu przedstawiające przebieg testu manualnego.	36
6.2	Potwierdzenie transakcji przez klienta w rozszerzeniu <i>Metamask</i>	37
6.3	Wynik przeprowadzonych testów automatycznych.	38
6.4	Fragment raportu wygenerowanego przez narzędzie Solidity Coverage . .	38
6.5	Raport Slither przed zastosowaniem poprawek.	40
6.6	Raport Slither po wprowadzeniu poprawek.	41
6.7	Wynik przeprowadzonych testów automatycznych z wykorzystaniem zbioru realnych danych wejściowych.	44

Spis tabel

6.1	Przykładowe wiersze danych wejściowych	43
-----	--	----

Lista kodów źródłowych

5.1	Fragment kodu w języku <i>Solidity</i> ilustrującego dane przechowywane w inteligentnym kontrakcie.	23
5.2	Kod funkcji <i>validateClient</i> w języku <i>Solidity</i> odpowiadający za weryfikację użytkownika i dodawanie danych jego sesji ładowania do sieci zdecentralizowanej.	25
5.3	Fragment kodu w języku <i>Python</i> w pliku <i>models.py</i> odpowiadający za reprezentację klasy <i>Charger</i>	28
5.4	Fragment kodu w języku <i>Python</i> z pliku <i>views.py</i> odpowiadający za obsługę funkcjonalności przekazywania sezonowych wypłat sprzedawcom przez administratora.	30
5.5	Fragment kodu w języku <i>TypeScript</i> z pliku <i>dashboard.component.ts</i> , odpowiadający za obsługę funkcjonalności widoku klienta.	32
6.1	Przykład testu sprawdzającego poprawność funkcji <i>updateChargerPrice</i> . . .	39
6.2	Funkcja <i>recordTransaction</i> w języku Solidity przed edycją.	41
6.3	Funkcja <i>recordTransaction</i> w języku Solidity po edycji.	42
6.4	Funkcja <i>checkMyContribution</i> w języku Solidity przed edycją.	42
6.5	Funkcja <i>checkMyContribution</i> w języku Solidity po edycji.	42