Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Побудова інформаційної системи управління взаємовідносинами з клієнтами для брокерських компаній у страховій галузі

Текстова частина

магістерської роботи

за спеціальністю „Інженерія програмного забезпечення” 121

Керівник магістерської роботи:
д.т.н., доц. А. М. Глибовець

_____
(підпис)

“____” _____ 2021 р.

Виконав:
студент Василенко А. М.
“____” _____ 2021 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики, к.ф.-м.н.
_____С. С. Гороховський
(підпис)
„____”_____2021 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу

студенту 2 р. н. магістерської програми “Інженерія програмного забезпечення” Василенко Андрію Миколайовичу
Розробити Інформаційну систему управління взаємовідносинами з клієнтами для брокерських компаній у страховій галузі

Зміст текстової частини до магістерської роботи:

Зміст
Анотація
Вступ
1. Огляд та класифікація систем управління взаємовідносинами з клієнтами
2. Розробка архітектури системи управління взаємовідносинами з клієнтами для брокерських компаній
3. Реалізація системи управління взаємовідносинами з клієнтами для брокерських компаній з інтеграцією блокчейн платформи
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2021 р.

Керівник
А. М. Глибовець,
доктор технічних наук, доцент

_____
(підпис)

Завдання отримав
А. М. Василенко

_____
(підпис)

Тема: Інформаційна система управління взаємовідносинами з клієнтами для брокерських компаній у страховій галузі

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	05.11.2020	
2.	Огляд літератури та ресурсів за темою роботи	20.11.2020	
3.	Аналізу існуючих рішень та підходів	04.12.2020	
3.	Розробка архітектури системи	27.12.2020	
4.	Створення програмного забезпечення згідно з архітектурою	22.01.2021	
5.	Інтеграція з зовнішніми системами та мережами	15.02.2021	
6.	Написання пояснювальної записки	20.04.2021	
7.	Створення слайдів для доповіді та написання доповіді	27.04.2021	
8.	Аналіз отриманих результатів з керівником, написання доповіді та попередній захист магістерської роботи	04.05.2021	
9.	Корегування роботи за результатами попереднього захисту	07.05.2021	
10.	Остаточне оформлення пояснювальної записки та слайдів	11.05.2021	
11.	Захист магістерської роботи (проекту)	18.06.2021	

Студент _____

Керівник _____

“____”_____

ЗМІСТ

Анотація

В цій магістерській роботі надається огляд систем управління взаємовідносинами з клієнтами, побудована архітектура такої системи, а також створена система управління взаємовідносинами з клієнтами для брокерських компаній у страховій сфері, з використанням блокчейн технології на базі Ethereum як елемента підтвердження достовірності даних, а також джерела не конфіденційних даних клієнтів стосовно їх страхових випадків.

Ключові слова: система управління, ведення клієнтської бази, блокчейн, Ethereum.

ВСТУП

Актуальність. Діяльність будь-якої брокерської компанії у сфері страхування неодмінно потребує налагоджений процес ведення бази клієнтів. З метою оптимізації цього процесу та найбільш ефективного використання часу при створенні та при подальшому комерційному використанні такої бази є сенс розглянути можливість придбання або створення спеціалізованої системи. Якщо раніше використання універсальної системи управління взаємовідносинами була єдиним можливим варіантом, то на сьогодні вже існують системи, створені під конкретні типи бізнесу. Така вузьконаправлена система враховує безліч деталей, необхідних для одного бізнесу і зайвих для іншого. Як результат, використання спеціалізованої на страховому бізнесі системи дозволить швидко та зручно створювати нових клієнтів, додавати договори, робити пролонгації, вносити страхові випадки, встановлювати взаємозв'язки між елементами та багато іншого. З розвитком технологій відкриваються нові можливості для функціонального наповнення таких систем. В даній роботі описано створення системи з використанням сучасних архітектурних практик та технологій, а саме застосування підходу SaaS (software as a service), хмарне зберігання даних та інтеграція блокчейн технології.

Мета дослідження. Побудувати архітектуру системи управління взаємодії з клієнтами на засадах сучасних технологій та створити програмне забезпечення на базі даної архітектури.

Завдання дослідження. Зробити огляд та аналіз сучасних архітектурних принципів та практик у побудові архітектури для систем

управління взаємовідносинами з клієнтами. Ознайомитись з існуючими бібліотеками та програмами, котрі можуть бути використані для побудови системи. Ознайомитись з блокчейн технологією та мережами, а також проаналізувати варіанти їх інтеграції з системою.

Об'єкт дослідження: створення системи управління взаємовідносинами з клієнтами для брокерських компаній з використанням блокчейн мережі.

Предмет дослідження. Компоненти та бібліотеки для системи управління взаємовідносинами з клієнтами, їх функціональне наповнення та API, блокчейн мережа та способи її інтеграції до системи.

Джерела дослідження - друкована література, електронні ресурси (в тому числі сайти компаній, котрі пропонують системи управління, форуми, сайти бібліотек тощо), довідники, матеріали конференцій, вихідні коди програм та бібліотек.

Наукова новизна передбачається у використанні блокчейн технології на базі Ethereum для запису в блокчейн мережу та подальшому отриманні з мережі актуальної інформації щодо страхових випадків клієнта брокерської компанії.

РОЗДІЛ 1. Огляд систем управління взаємовідносинами з клієнтами

1.1. Поняття систем управління взаємовідносинами з клієнтами

Існує велика кількість визначень системи управління взаємовідносинами. Досить повно цей термін описує наступне визначення - прикладне програмне забезпечення для організацій, призначене для автоматизації стратегій взаємодії з замовниками (клієнтами), зокрема для підвищення рівня продажів, оптимізації маркетингу і поліпшення обслуговування клієнтів шляхом збереження інформації про клієнтів і історію взаємовідносин з ними, встановлення і поліпшення бізнес-процесів і подальшого аналізу результатів. [23]

Поняття системи управління взаємовідносинами з клієнтами складається з двох елементів:

1. управління взаємовідносинами з клієнтом - налагоджений процес збору, зберігання, обробки, аналізу та видалення інформації по клієнтам компанії. Визначення цьому процесу також міститься у Вікіпедії: “Управління відносинами з клієнтами (англ. Customer relationship management (CRM), укр. сі-ар-ем) — поняття, що охоплює концепції, котрі використовуються компаніями для управління взаємовідносинами зі споживачами, включаючи збір, зберігання й аналіз інформації про споживачів, постачальників, партнерів та інформації про взаємовідносини з ними.” [1]
2. система або програмне забезпечення - визначення цьому поняттю також міститься у Вікіпедії: “Програмне забезпечення (програмні засоби) (ПЗ; англ. software) — сукупність програм системи оброблення інформації та програмних документів, необхідних для експлуатації цих програм.” [2]

Однак, треба зауважити, що метою цієї роботи є не просто створення програмного забезпечення, але створення програмного продукту. Для розуміння різниці слід визначити поняття програмного продукту:

Програмний продукт (англ. software product, software as a product (SaaP)) — програмне забезпечення, розроблене для вирішення задачі масового попиту та призначене для постачання користувачам. Програмний продукт відрізняється від просто програмного забезпечення максимально узагальненим набором вхідних даних, ретельним тестуванням, наявністю документації, гарантії та технічної підтримки. [3]

Таким чином, сукупність цих двох елементів дає нам розуміння, що система управління взаємовідносинами з клієнтами - це програмний продукт, котрий покликаний забезпечити виконання процесу налагодження та підтримки взаємовідносин з клієнтом різних видів компаній. Зважаючи на тему роботи, програмний продукт буде будуватись для визначеного виду компаній, а саме - брокерські компанії у сфері страхування.

1.2. Класифікація систем управління взаємовідносинами з клієнтами.

На сьогоднішній день існують багато досліджень з приводу класифікації систем управління взаємовідносинами з клієнтами. Незважаючи на постійний розвиток таких систем та їх різноманітні спрямування, всі існуючі системи можна розділити за наступними категоріями:

1.2.1 Операційні (або транзакційні) системи

Операційні системи працюють з процесами автоматизації ведення взаємовідносин з клієнтами за допомогою програмного забезпечення, котре використовується в офісах компаній, включаючи автоматизацію корпоративного маркетингу, обслуговування та

підтримки клієнтів, а також автоматизація збуту. Відповідно, впорядкування робочого процесу та автоматизація в офісах продажу здійснюється через таку операційну систему, яка включає збір даних, обробку транзакцій та контроль робочих процесів у відділах обслуговування, маркетингу та збуту. Історично склалося так, що операційні системи залишалися основною частиною видатків підприємства, оскільки організації постійно впроваджують системи автоматизації для збуту та розробляють технологічні програми. Таким чином, постачальники операційних систем в основному зосереджуються на наданні більш широкого спектру рішень. Прийнято вважати, що мета операційних систем - це інтеграція технологій, процесів та людей через очікування та перспективи споживачів. [4]

Крім того, використання операційних систем є запорукою вдосконалення якості обслуговування клієнтів та процесів. Такі системи дозволяють інтеграцію та автоматизацію функцій продаж та маркетингу. Як результат, використання операційних систем вважається ефективним підходом, який є повністю економічно вигідним.

Отож, організації отримали очевидні вигоди від використання операційних систем за рахунок зниження витрат, збільшення доходу, швидкого вирішення проблем тощо.

Прикладом операційних систем є Hubspot та Salesforce.

1.2.2 Аналітичні системи:

Аналітичні системи орієнтовані на збір, зберігання та інтерпретацію даних. Вони сортують та обробляють дані про клієнтів таким чином, щоб компанія могла зрозуміти, який підхід до продажу добре працює

з яким типом клієнтів, і як налаштувати підхід до продажів для різних сегментів бази продажів.

Прикладом аналітичних систем є Zoho Analytics та Wave.

1.2.3 Колаборaційні системи:

Колаборaційна система залучає кілька різних компаній та/або підприємств, що працюють разом, щоб створити та підтримувати велику базу клієнтів, яка створює додаткову цінність у зв'язках між бізнесом. Цього можна досягти за допомогою систем винагородження за лояльність, які присуджують бали на всіх підприємствах, і можуть бути використані за придбання асортименту товарів чи послуг від цих підприємств, або шляхом безперервного переплетення різних цілей різних підприємств та їх баз клієнтів через систему для покращення обслуговування клієнтів, а також можливість охопити більше потенційних клієнтів за допомогою реклами.

Прикладом колаборaційних систем є Pipedrive та Copper.

1.2.4 Стратегічні системи:

Ці системи прагнуть отримати довгострокову конкурентну перевагу, оптимально забезпечуючи цінність та задоволення для клієнта та отримуючи бізнес цінність від взаємодії. Таким чином, знання про клієнтів і їх уподобання є надзвичайно важливими для всієї організації. [6]

Стратегічні системи фокусуються на клієнтах. Збір інформації про клієнтів та взаємодії між ними та бізнесом може призвести до покращення взаємовідносин. Стратегічні системи не тільки дають вам важливу інформацію стосовно клієнтів, але натомість коригують

або налаштовують спосіб взаємодії з клієнтами в довгостроковій перспективі. Це стає особливо корисно коли ви займаєтеся бізнесом, де основна увага приділяється довготривалим стосункам, а не швидким продажам та коротким кампаніям. Стратегічна система повністю орієнтована на задоволення вимог клієнта товарами та/або послугами, які надає компанія. Вона спрямована на збільшення продажів через чітке розуміння того, що саме шукає замовник, та надання цієї інформації ефективно та своєчасно.

Останнім часом набуває популярності комбіновані системи, тобто такі, котрі поєднують в собі елементи різних типів систем.

Відповідно до аналітики консалтингої компанії, приблизно 40% компаній впроваджують систему управління взаємовідносинами з клієнтами впродовж перших двох років своєї діяльності. При цьому, 65% компаній впровадять таку систему протягом перших 5 років існування. [5]

РОЗДІЛ 2. Розробка архітектури системи управління взаємовідносинами з клієнтами

2.1. Архітектурно значущі вимоги

При розробці архітектури системи в першу чергу потрібно брати до уваги вимоги до системи та її функціональних характеристик, технічних обмежень та ряду інших чинників . Слід виокремити ті вимоги, котрі будуть мати значний вплив на архітектуру. Архітектурно значуща вимога, або ASR (від англ. *architecturally significant requirement*), - це будь-яка

вимога, яка сильно впливає на наш вибір структурних елементів для архітектури. Це є відповідальністю архітектора програмного забезпечення знайти архітектурно значущі вимоги. Вимоги розділяють на чотири категорії:

1. **обмеження** - незмінні дизайнерські рішення, зазвичай надаються, іноді вибираються;
2. **атрибути якості** - зовнішні видимі властивості, що характеризують те, як система працює в конкретному контексті;
3. **впливові функціональні вимоги** - особливості та функції, які вимагають особливої уваги в архітектурі;
4. **інші впливові чинники** - час, знання, досвід, навички, офісна політика, ваші власні виразні упередження та всі інші речі, які впливають на прийняття рішень. [7]

Для визначення вимог, котрі мають бути враховані у проектуванні архітектури було опитано декілька потенційних користувачів такої системи. Отримані відповіді були проаналізовані і згруповані, а результат викладений в таблиці:

Вимога	**Тип**	**Контекст**
Система має працювати з блокчейн платформою для зберігання даних	Бізнес	На сьогодні вже існують системи управління взаємовідносинами, але жодна ще не інтегрована з блокчейн. Система має зберігати дані клієнта в блокчейн мережі.
Система має бути доступна з будь-якого пристрою, підключеного до інтернет	Бізнес	Користувачі повинні мати можливість користування системою з будь-якого пристрою підключеного до мережі Інтернет, а не тільки з

		офісу компанії-користувача системою
Система повинна мати доступ по ролям з відповідними обмеженнями	Бізнес	Система буде надаватися в користування як самій компанії-брокеру, так і клієнтам такої компанії, а отже кожен користувач повинен мати відповідний та адекватний доступ до даних в системі
Має підтримувати останню версію Google Chrome	Технічна	Система має коректно працювати в останній версії браузера Google Chrome
Має зберігати дані на локальній БД та щоночі робити копію на хмарну БД	Технічна	Оскільки компанія буде зберігати свою базу клієнтів в системі, необхідно забезпечити копіювання даних в додаткову базу даних на випадок якщо локальна база перестане працювати
Має має працювати на Windows сервері	Технічна	Система буде розгорнута на Windows сервері
Блокчейн дані не мають містити персональну інформацію	Бізнес	Оскільки різні страхові компанії будуть мати доступ до даних своїх клієнтів, необхідно забезпечити забезпечити неможливість отримання персональних даних клієнтів по договорам, котрі не належать цим страховим компаніям
Статус договорів має	Бізнес	Оскільки кожен договір

автоматично змінюватись відповідно до терміну дії		має дату початку і дату закінчення, необхідно забезпечити автоматичну зміну стану договору відповідно до його термінів дії.

2.2. Проектування архітектури

Враховуючи вищезазначені вимоги, можемо зробити висновок, що для побудови системи підійде багаторівнева архітектура.

Компоненти всередині багаторівневої архітектури організовані в горизонтальні рівні, кожен рівень виконує певну роль у програмі (наприклад, логіка презентації або бізнес-логіка). Хоча шаблон багаторівневої архітектури не визначає кількість та типи рівнів, які повинні існувати в шаблоні, більшість багаторівневих архітектур складаються з чотирьох стандартних рівнів: презентації, бізнесу, рівня доступу даних та рівень баз даних. У деяких випадках бізнес-рівень та рівень доступу даних об'єднуються в один бізнес-рівень, особливо коли доступу даних вбудована в компоненти бізнес-рівня. Таким чином, менші програми можуть мати лише три рівні, тоді як більші та складніші бізнес-програми можуть містити п’ять і більше рівнів. [8]

Рис. 1.1. Рівні та потік даних у багаторівневій архітектурі

Кожен рівень комунікує тільки з одним рівнем нижче. Така архітектура чітко визначає належність компонентів та упорядковує поток даних між компонентами та у застосунку в цілому. Застосовуючи дану обрану архітектуру для нашого застосунку отримаємо наступну схему компонентів системи:

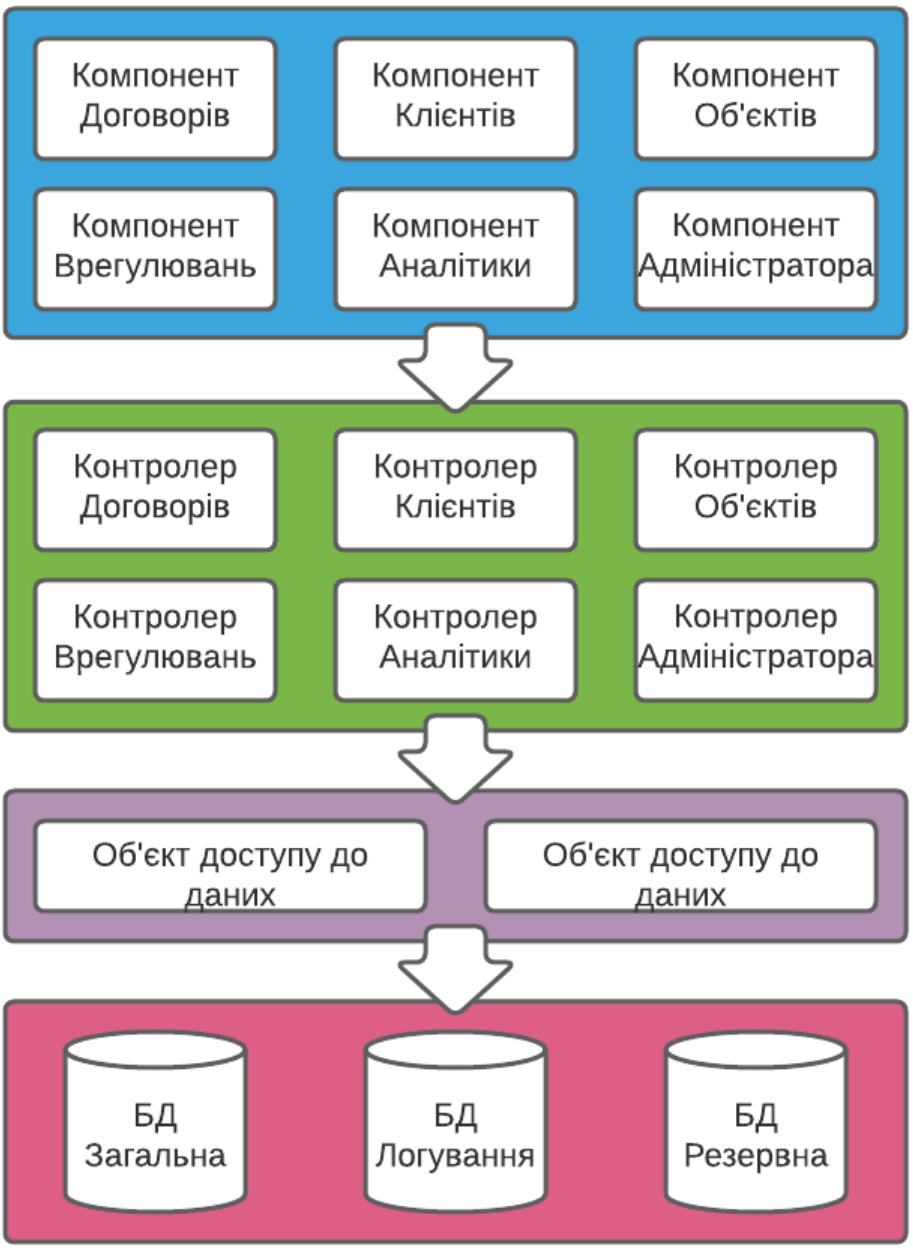

Рис. 1.2. Структурні елементи застосунку при багаторівневій архітектурі

Маючи таку архітектуру не складно зробити діаграму послідовностей застосунку для, наприклад, процесу додавання нової сутності:

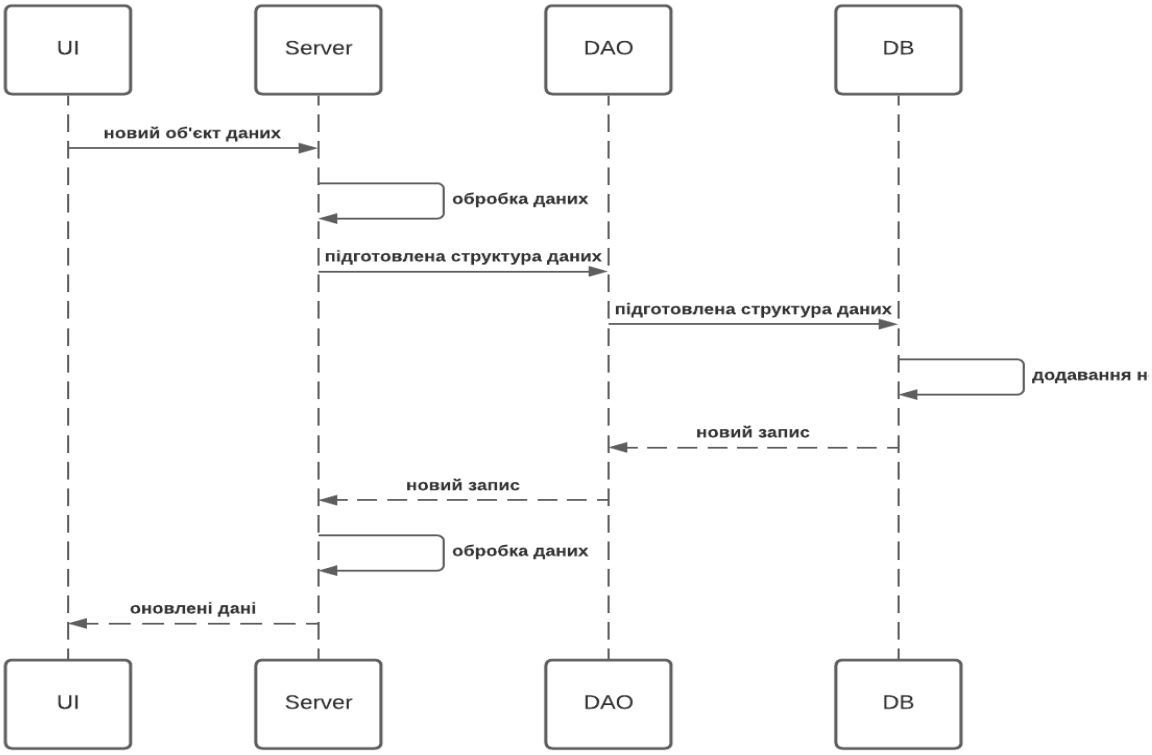

Рис. 1.3. Діаграма послідовностей для застосунку при багаторівневій архітектурі

Відповідно до вимог можемо продумати логічні елементи серверної частини. Вона має враховувати наявність локальної та хмарної баз даних. З точки зору оптимізації застосунку є сенс розглянути створення декількох баз даних. Наприклад, для того щоб зменшити навантаження та прискорити роботу застосунку, буде створена окрема база даних для логування дій користувачів. Записи в таку базу не будуть впливати на швидкодію системи у разі, якщо в цей час інший користувач буде отримувати дані з бази даних, котра зберігає дані по договорам. Отож, компонентами серверної частини будуть наступні:

Рис. 1.4. Компоненти серверної частини застосунку

Схожою за структурою логічних компонентів є клієнтська частина. На рис. 1. 5. зображені компоненти клієнтської частини.

Для комунікації з серверною частиною будемо використовувати архітектурний стиль взаємодії компонентів REST.

Для керування станом системи буде використовуватись бібліотека Redux.

Організація папок буде відбуватись за принципом сторінок (елементів меню застосунку).

Рис. 1. 5. Компоненти клієнтської частини застосунку

РОЗДІЛ 3. Реалізація системи управління взаємовідносинами з клієнтами для брокерських компаній на базі архітектури

3.1. Вибір технологій

Основою для вибору технологій слугують декілька факторів. Найбільш значущі серед них є наступні:

1. Тип або складність проекту
2. Наявність готових рішень
3. Популярність та тренд розвитку технології
4. Вимоги до безпеки та швидкодії

Враховуючи вищезазначені фактори, для реалізації системи обрано мову програмування JavaScript та набір технологій, відомий за абревіатурою MERN, тобто MongoDB, Express, React, NodeJS.

Однією з переваг такого набору є використання однієї мови програмування для серверної та клієнтської частин.

Мова JavaScript стабільно займає перше місце у світі серед найпопулярніших мов програмування [9], має дуже широку підтримку та співтовариство, демонструє гарні результати у швидкодії систем, побудованих на цій мові. До відомих сайтів, побудованих з JavaScript входять наступні:

- Google, LinkedIn, eBay, Yahoo - використовують JavaScript для клієнтської та серверної частин
- Facebook, Youtube, Amazon, Wikipedia, Twitter - використовують JavaScript для клієнтської частини [10]

Серед вимог до системи є використання блокчейн технології, а саме можливість запису інформації по клієнту та можливість отримання такої інформації з блокчейн мережі. Для вибору блокчейн платформи зробимо порівняння характеристик декількох найбільш популярних з них:

	Ethereum 	Bitcoin 	Ripple 	Cardano 
Блокчейн покоління	Друге	Перше	Перше	Перше
Механізм консенсусу	PoW (PoS planned)	PoW	RPCA	PoS
Час додавання в блок	14 сек	600 сек	4 сек	20 сек
Швидкість транзакцій (трнзц/сек)	5	4.5	1500	250
Час поповнення рахунку	5 хв	40 хв	майже миттєво	10 хв
Смарт контракти	Так	Так	Ні	Ні
Децентралізовані застосунки	Так	Ні	Ні	Ні
Витрати енергії на отримання консенсусу	Багато (половина від Bitcoin)	Багато	Мало	Мало

Виходячи з даних порівняння можемо констатувати, що платформа Ethereum цілком задовольняє вимоги до системи та може бути використана як інструмент для їх реалізації. Ethereum дає можливість:

1. створювати розумні контракти та децентралізовані застосунки (dApps)
2. швидко додавати новий функціонал застосунку

3. легко інтегрувати платформу в веб-застосунки
4. отримати підтримку за рахунок великої кількості послідовників
5. отримання достатньої швидкості роботи

Ethereum також показує динамічний тренд розвитку платформи. Серед найближчих планів - перехід від моделі Proof-of-Work до Proof-of-Stake, що значно збільшить швидкість обробки транзакцій, зменшить споживання електроенергії приблизно на 99%, а значить залучить ще більше послідовників.

3.2. Реалізація клієнтської частини

Для швидкого старту створення клієнтської частини застосунку використаємо інструмент create-react-app [17]. Інструмент налаштовує середовище для використання нових можливостей JavaScript, оптимізує додаток для фінального використання і забезпечує комфорт під час розробки [18]. Результатом виконання буде створення наступної структури проекту:

```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
```

```
├── App.css
├── App.js
├── App.test.js
├── index.css
├── index.js
├── logo.svg
└── serviceWorker.js
└── setupTests.js
```

Фактично, ми отримали готову структуру проекту, а також готові до використання деякі команди:

1. npm start - запускає застосунок у режимі розробки
2. npm test - запускає виконання тестів
3. npm run build - оптимізує код та підготовлює застосунок для використання.

Для більш зручного управління станом застосунку будемо використовувати бібліотеку Redux.

Redux - це передбачуваний контейнер стану для програм JavaScript. Він допомагає писати програми, які поводяться передбачувано, працюють в різних середовищах (клієнтських, серверних та власних) і легкі для тестування [19].

Redux оперує наступними сутностями:

1. Action - опис дії користувача
2. Reducer - функція, котра визначає як саме буде змінюватись стан системи
3. Store - об’єкт, котрий зберігає все дерево станів застосунку
4. State - поточній стан застосунку

Схематично Redux можна представити наступним чином:

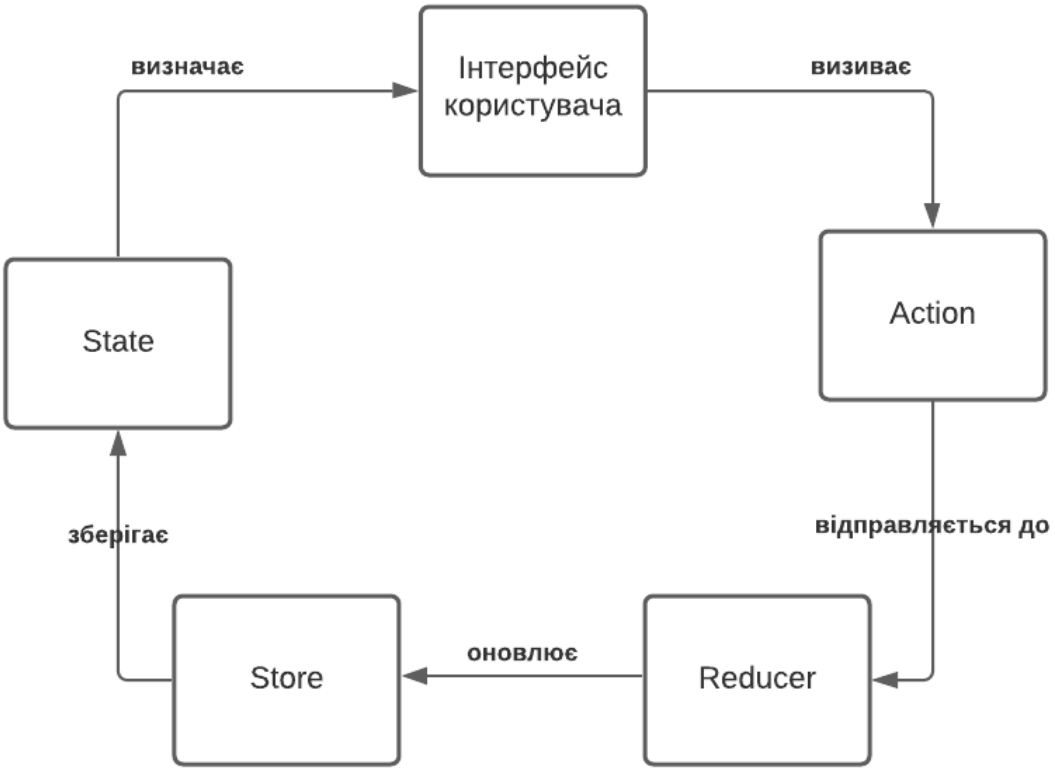

Рис. 2.1. Схема роботи бібліотеки Redux

Додатково до Redux підключимо бібліотеку react-router. React Router - це сукупність навігаційних компонентів, які декларативно складаються з додатком [20]. Вона дозволить нам побудувати навігацію по нашому застосунку, а також створити публічні та захищені адреси за допомогою яких буде реалізована авторизація. Всі сторінки застосунку, окрім сторінки для авторизації, маю бути захищені (приватні) та відображатись тільки для користувачів, котрі успішно авторизувалися.

Для цього створимо компонент сторінок Login та Home. Перший має бути публічним, а другий - приватним. У разі, якщо користувач системи не авторизувався, або строк дії його сесії закінчився і він при цьому намагається відкрити приватний компонент, такий користувач має бути

примусово перенаправлений на сторінку авторизації. Реалізація цієї логіки відбувається в окремому компоненті:

```
import React from 'react'
import { Redirect, Route } from 'react-router-dom'
import { connect } from 'react-redux'
import { logoutUser } from '../../actions/authentication';

const PrivateRoute = ({ component: Component, ...rest }) => {
    class AuthContainter extends React.Component {
        constructor(props) {
            super(props);
            this.handleRender = this.handleRender.bind(this);
            this.activeSession = this.activeSession.bind(this);
        }
        activeSession() {
            if (!localStorage.getItem('jwtToken')) { return false }
            if (
                 !this.props.auth.isAuthenticated ||
                 !this.props.auth?.user.exp
               ) { return false }
            if (this.props.auth.user.exp * 1000 < Date.now())
                 { return false }

            return true
        }

        handleRender(props) {
            if (this.activeSession()) {
                return <Component {...props} />;
            } else {
                this.props.logoutUser();
                return <Redirect to={{ pathname: "/", state: { from:
props.location } }} />;
            }
        }

        render() {
            return (
            <>
                <Route {...rest} render={this.handleRender} />
            </>
            )
```

```
        }
    }

    function mapStateToProps(state) {
        return {
            auth: state.auth,
            loading: state.helpers.loading
        }
    }

    const AuthCont = connect(mapStateToProps,
{logoutUser})(AuthContainter);
    return <AuthCont />
}

export default PrivateRoute
```

Як видно з коду і відповідно до правил використання Redux, попередньо був створений Action для дії logoutUser:

```
export const logoutUser = (history) => (dispatch) => {
  localStorage.removeItem("jwtToken");
  setAuthToken(false);
  dispatch({
    type: "SET_CURRENT_USER",
    payload: {},
  });
  if (history) {
    history.push("/");
  }
};
```

Підключення вищезазначених бібліотек та компонентів відбувається в App.js файлі, котрий вже був створений за допомогою create-react-app:

```
import React from "react";
import { BrowserRouter as Router, Route, Switch } from
"react-router-dom";
import { Provider } from "react-redux";
```

```
import store from "./store";
import "./App.css";
import PrivateRoute from "./pages/Common/PrivateRoute";

function Home() {
  return <p>This is private</p>
}
function Login() {
  return <p>This is public</p>
}
function App() {
  return (
    <Provider store={store}>
      <Router>
        <div className="App">
          <Switch>
            <PrivateRoute exact path="/home" component={Home} />
            <Route exact path="/" component={Login} />
          </Switch>
        </div>
      </Router>
    </Provider>
  );
}

export default App;
```

Тепер якщо спробувати відкрити приватний компонент, то користувач буде направлений на сторінку авторизації.

Додамо дію авторизації користувача. Для авторизації будемо використовувати логін та пароль, котрий буде відправлятись на серверну частину застосунку для генерації токена формату JSON Web Token у разі коректності введених даних.

```
export const loginUser = (user) => (dispatch) => {
  axios
    .post(URL + "/api/login", user)
    .then((res) => {
      const { token } = res.data;
      localStorage.setItem("jwtToken", token);
      setAuthToken(token);
```

```
      const decoded = jwt_decode(token);
      dispatch({
        type: “SET_CURRENT_USER”,
        payload: decoded,
      });
      return res.data.payload;
    })
    .catch((err) => {
      dispatch({
        type: GET_ERRORS,
        payload: err.response ? err.response.data : null,
      });
    });
};
```

Отриманий токен буде розкодований та доданий до стану системи відповідно до функції, описаній у Reducer:

```
import isEmpty from '../helpers/is-empty';

const initialState = {
    isAuthenticated: false,
    user: {},
}

export default function(state = initialState, action ) {
    switch(action.type) {
        case “SET_CURRENT_USER”:
            return {
                ...state,
                isAuthenticated: !isEmpty(action.payload),
                user: action.payload
            }
        default:
            return state;
    }
}
```

Структура веб-інтерфейсу та елементи, котрі бути в ньому присутні, зображені на схема нижче:

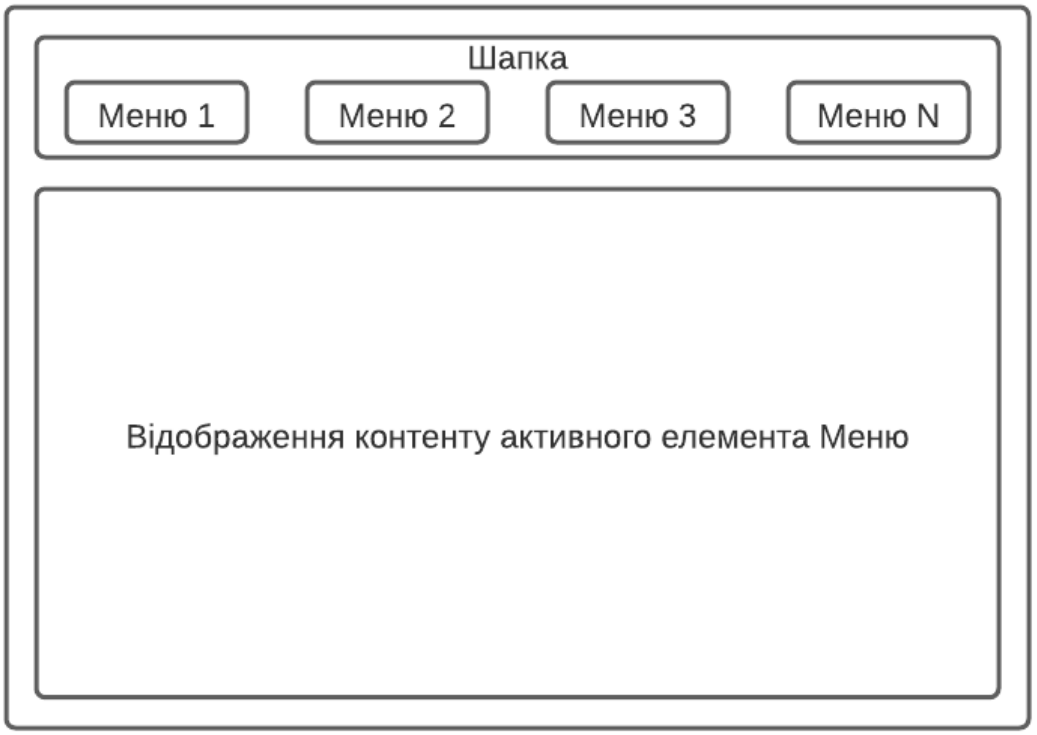

Рис. 2. 2. Схема елементів веб-інтерфейсу застосунку

За цією структурою матимемо навігацію по застосунку, котра реалізується пунктами меню, та відображення контенту активного меню нижче.

Створення компонентів, котрі будуть відображати дані по договорам, клієнтам, об’єктам страхування та врегулюванням (процес розгляду страхового випадку) розглянемо на прикладі компонента для відображення об’єктів страхування.

Такий компонент буде складатись с декількох вкладених один в одний компонентів. Структуру можна відобразити наступним чином:

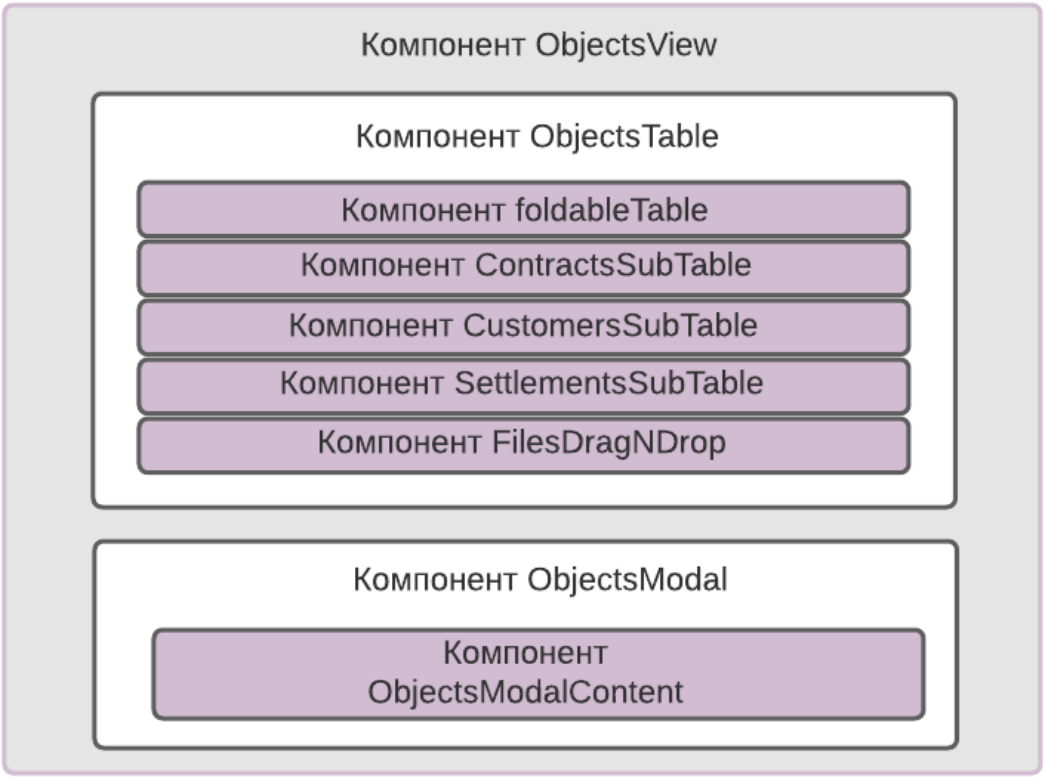

Рис. 2. 3. Структура композитного React компоненту у застосунку

Кожен компонент відповідає за окрему логіку:

1. ObjectsView - контейнер для інших компонентів, а також зберігає логіку додавання нового об’єкту. Код компоненту має наступний вигляд:

```
export class ObjectsView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      showAddNewObjectModal: false,
    };
    this.saveObject = this.saveObject.bind(this);
    this.closeModal = this.closeModal.bind(this);
    this.openAddNewObjectModal = this.openAddNewObjectModal.bind(this);
  }
```

```
  openAddNewObjectModal = function () {
    this.setState({
      showAddNewObjectModal: true,
    });
  };

  closeModal = function () {
    this.setState({
      showAddNewObjectModal: false,
    });
  };

  saveObject(data) {
    let preparedObject = prepareObject(data);

    this.props.addObject(preparedObject).then((res) => {
      this.props.getObjects();
      this.setState({
        showAddNewObjectModal: !this.state.showAddNewObjectModal,
      });
    });
  }

  render() {
    return (
      <>
        <div>
          <ObjectsModal
            isShow={this.state.showAddNewObjectModal}
            saveContent={this.saveObject}
            closeModal={this.closeModal}
            openModal={this.openAddNewObjectModal}
          />
          <div style={containerStyle}>
            {this.props.auth.user.role !== "client" && (
              <div style={actionButtonsContainer}>
                <Button variant="contained" style={buttonStyle}
id="addObjectButton" onClick={this.openAddNewObjectModal}>
                  <i className="fas fa-plus-square"></i>
                  {this.props.t("addNew")}
                </Button>
              </div>
            )}
```

```
            <ObjectsTable />
          </div>
        </div>
      </>
    );
  }
}
```

2. ObjectsTable - компонент табличного відображення клієнтів. Містить вкладені в себе додаткові компоненти, а також відповідає за логіку редагування, клонування, видалення окремого об’єкту. Програмний код компоненту наведений у *Додатку А*.
3. ObjectsModal - компонент модального вікна для додавання та редагування даних об'єкта. Містить в собі окремий компонент, котрий відповідає за наповнення модального вікна.
   Код даного компоненту наведений у *Додатку Б*:

Така сама структура застосовується для побудови інших компонентів. Після їх створення як результат роботи отримаємо наступний інтерфейс користувача:

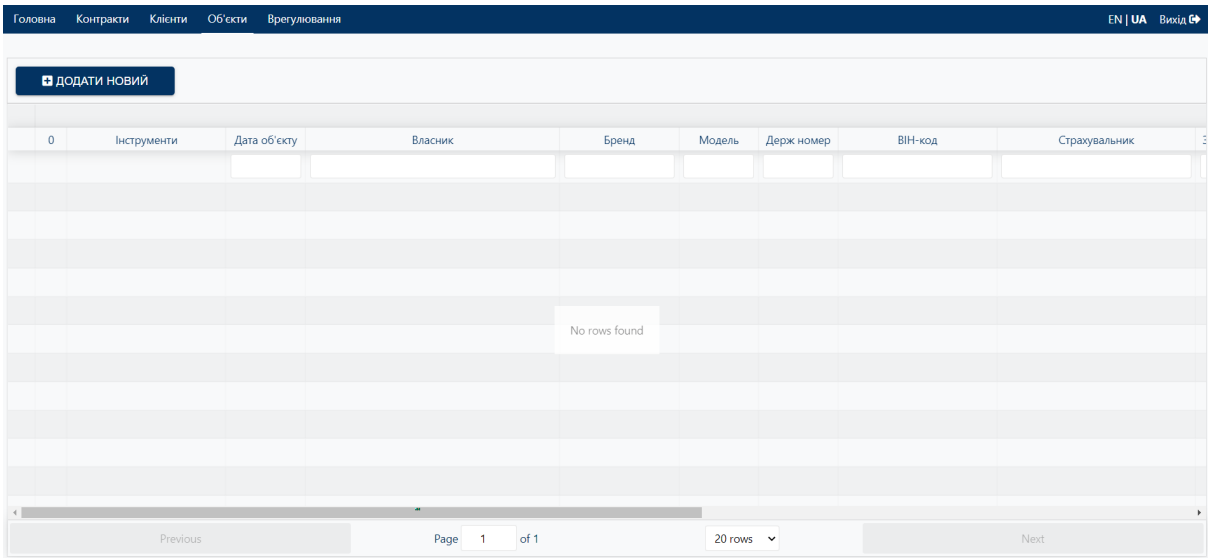

Рис. 2. 4. Меню “Об’єкти” в інтерфейсі користувача системи

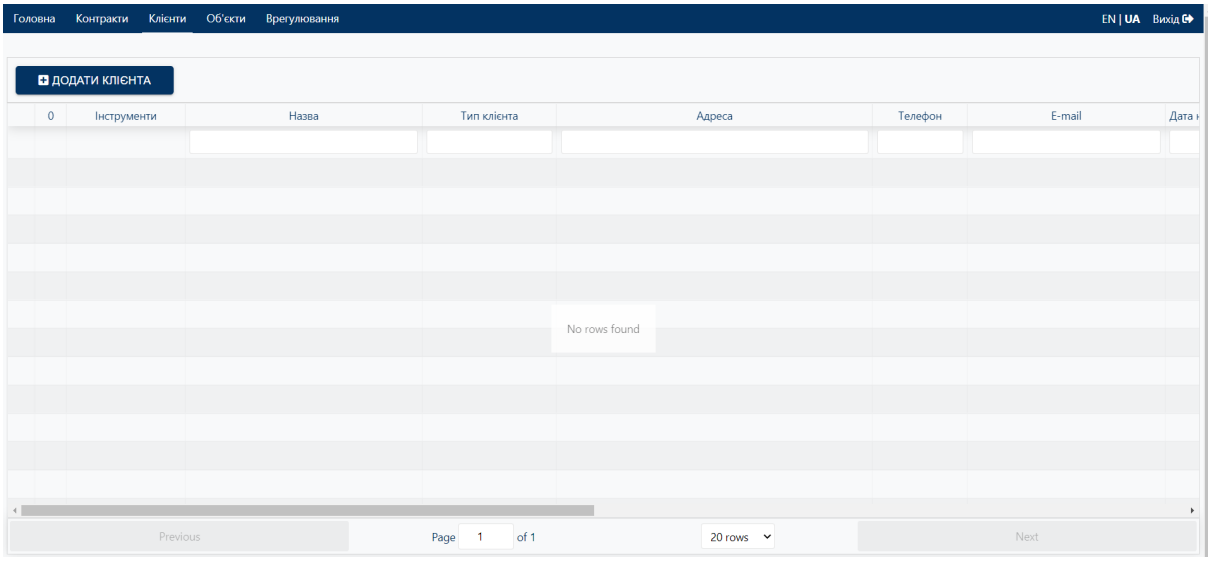

Рис. 2.5. Меню “Клієнти” в інтерфейсі користувача системи

Однією з вимог до системи є відображення пов’язаних сутностей. Додавання нового договору страхування в систему означатиме додавання трьох нових сутностей - сам договір, клієнт та об’єкт страхування. Ці сутності повинні мати посилання одне на одного. В подальшому, може бути додана ще одна сутність - врегулювання (при настанні страхового випадку(ів)). Схема посилань наведена на Рис. 2.6

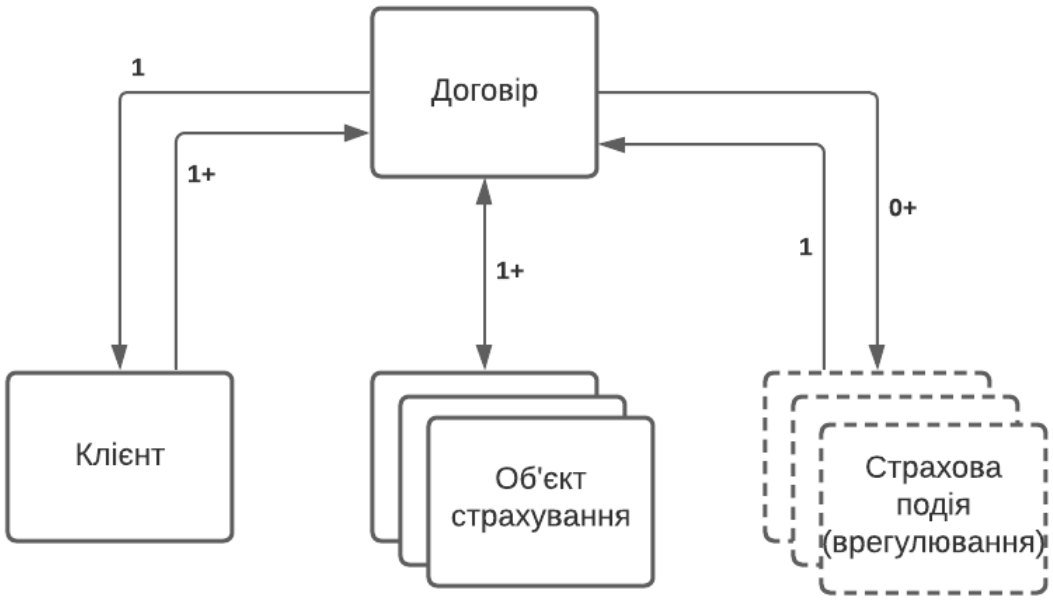

Рис. 2.6. Посилання сутностей в системі

Додаємо першого клієнта, об’єкт страхування та договір страхування аби пересвідчитись що посилання на сутності працюють коректно:

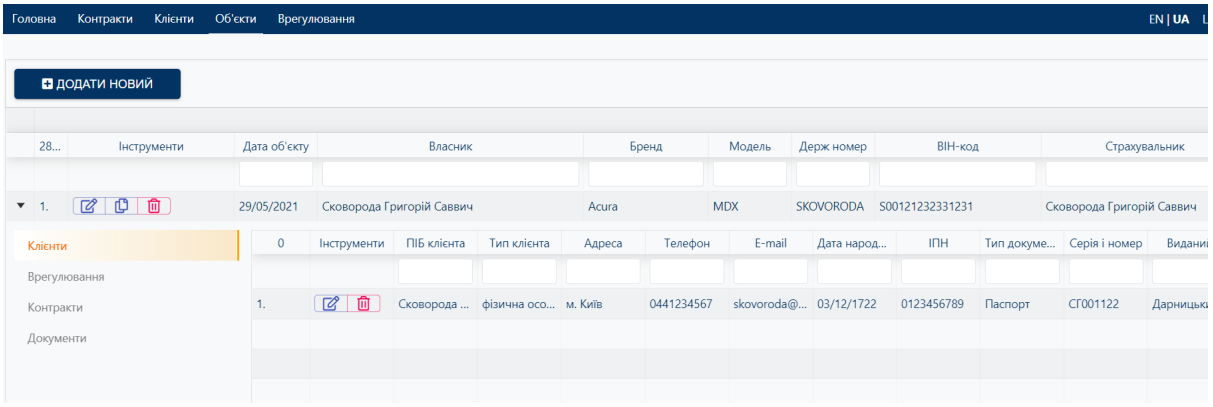

Рис. 2. 7. Меню “Об’єкти” в інтерфейсі користувача системи з відображенням вкладеного компоненту “Клієнти”

Знаходячись в пункті меню “Об’єкти” та розкривши деталі об’єкту бачимо додаткові розділи “Клієнти”, “Врегулювання”, “Контракти”, “Документи”. Інформація хто є клієнтом за цим об’єктом відображена у відповідному розділі.

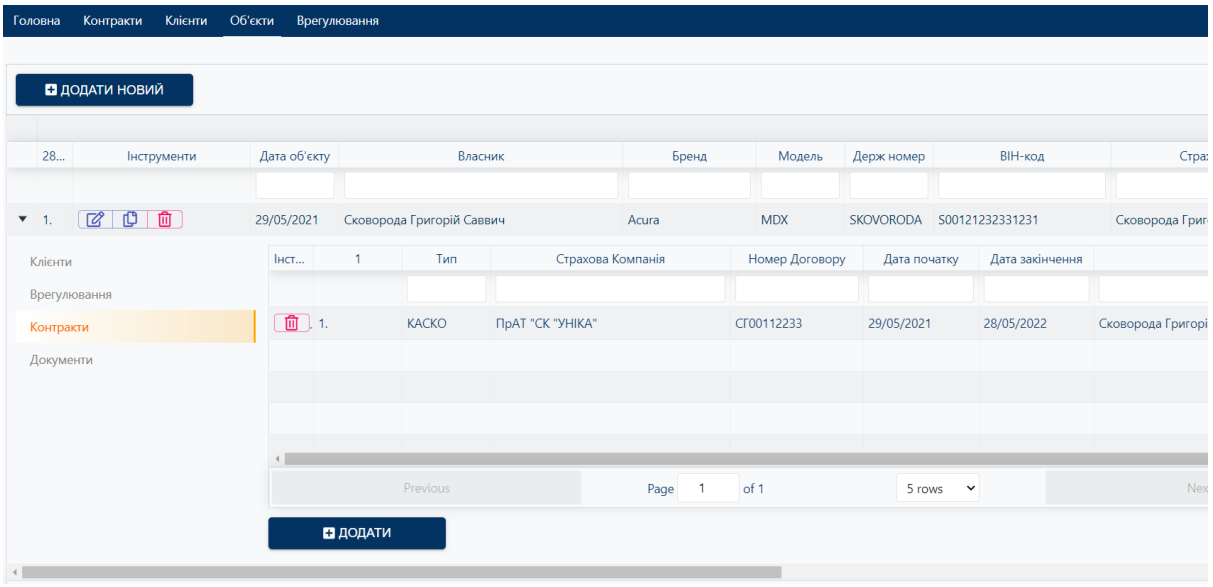

Рис. 2. 8. Меню “Об’єкти” в інтерфейсі користувача системи з відображенням вкладеного компоненту “Контракти”

Також, бачимо інформацію стосовно контрактів (договорів страхування) за цим об’єктом у відповідному розділі.

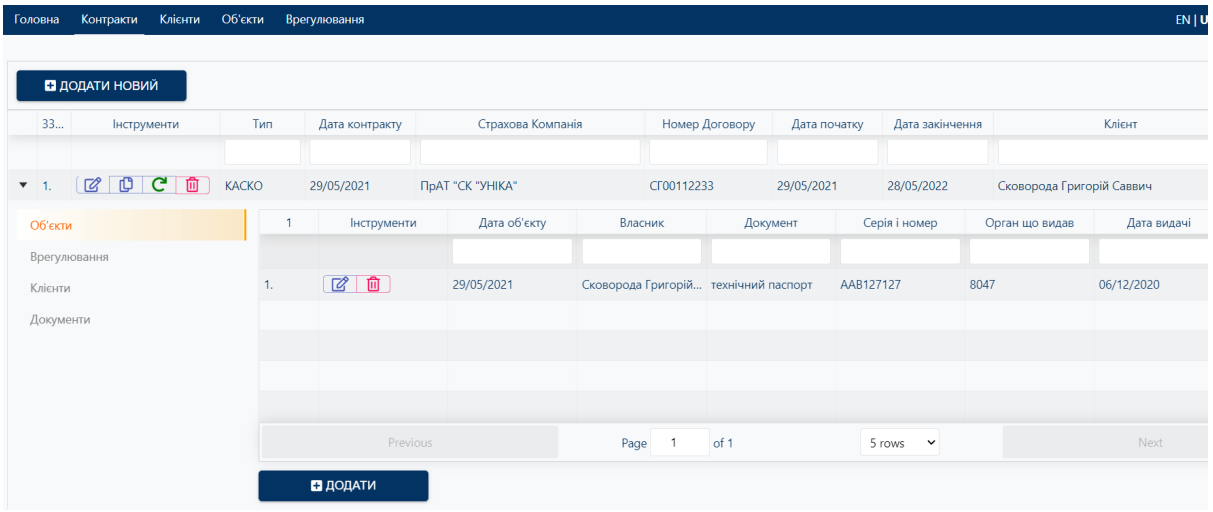

Рис. 2. 9. Меню “Контракти” в інтерфейсі користувача системи з відображенням вкладеного компоненту “Об’єкти”

На Рис. 2. 8 бачимо інформацію пункта меню “Контракти”, а також розділі конкретного контракту. Інформація про об’єкт даного контракту відображена у відповідному розділі.

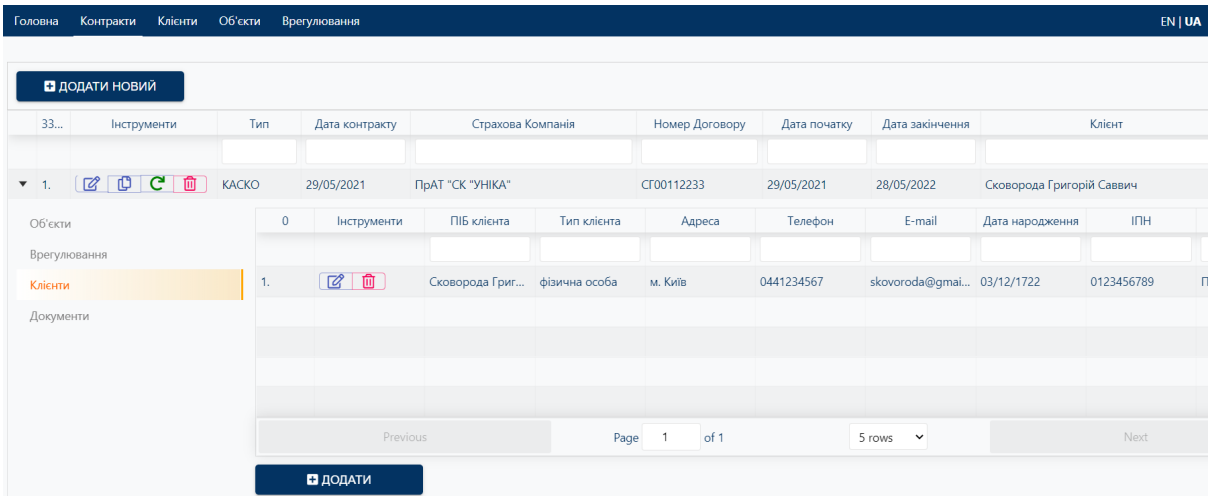

Рис. 2. 10. Меню “Контракти” в інтерфейсі користувача системи з відображенням вкладеного компоненту “Клієнти”

Інформація про клієнта, котрий підписав даний контракт, відображена у відповідному розділі.

3.3. Реалізація серверної частини

Побудова серверної частини базується на раніше зпроектовній архітектурі. При реалізації компонентів маємо враховувати наступні вимоги:

1. об'єм інформації має надаватись відповідно до ролі
2. файли картинок мають зберігатись локально
3. розробка та тестування нового функціоналу не мають впливати на систему, котра використовується

Підключення до бази даних та опис адрес API системи імплементуємо в файлі server.js. На даному етапі логіка всіх компонентів ще не реалізована. Першим кроком буде додано користувачів системи. Інші компоненти будуть реалізовані та імпортовані в файл з часом. Таким чином, код буде мати наступний вигляд:

```
const express = require("express");
const compression = require("compression");
const cors = require("cors");
const path = require("path");
const passport = require("passport");
const userRoutes = require("./users/usersController");
const mongoose = require("mongoose");
require("dotenv").config();
require("./passport")(passport);
require("events").EventEmitter.defaultMaxListeners = 30;
mongoose.Promise = global.Promise;

const app = express();
async function start() {
  try {
      const DB_URL = process.env.NODE_ENV === "production" ?
      process.env.LOCAL_SERVER_MONGODB_URI : process.env.MONGODB_URI;
      const PORT = process.env.NODE_ENV === "production" ?
      process.env.PORT || 80 : 5000;

    await mongoose.connect(DB_URL, {
      useCreateIndex: true,
      useUnifiedTopology: true,
```

```
      useNewUrlParser: true,
      socketTimeoutMS: 300000,
      connectTimeoutMS: 300000,
    });

    app.listen(PORT, () => {
      console.log(`Server listening on port ${PORT}`);
      console.log(`Connected to ${process.env.NODE_ENV === "production"
? "LOCAL SERVER MONGODB" : "CLOUD MONGODB"}`);
    });
  } catch (err) {
    console.log("Error while starting a server", err);
  }
}

app.use(passport.initialize());
app.use(compression());
app.use(cors());

app.use("/api", userRoutes);
app.use("/static", express.static(path.join(__dirname, "../images")));
start();
```

Ми реалізували різні зони роботи, тому при старті застосунку буде обиратись або тестова зона або робоча. Також, всі картинки ми будемо зберігати локально у папці `images`.

Імплементація вимоги рольової моделі буде відбуватись безпосередньо в компонентах серверної частини. Створимо компонент для обробки логіки по договорам компанії.

Компонент має виконувати наступні функції:

1. Додавання нового запису
2. Отримання записів (відповідно до ролі)
3. Редагування існуючого запису
4. Видалення запису

Основними ролями в системі та об’єм інформації відповідно до ролі є:

1. співробітник - має отримувати всі договори, котрі були створені саме цим співробітником
2. клієнт компанії - має отримувати виключно договори, підписані з даною компанією
3. страхова компанія - має отримувати договори, за якими дана страхова компанія виступає як компанія, котра надає послуги страхування

Враховуючи вищезазначене, код модулю на отримання переліку договорів має вигляд:

```
router.get("/contracts/getcontracts", passport.authenticate("jwt", {
session: false }), (req, res) => {
  var filter = {};
  if (req.user.role === "insCompany") {
    filter = { insCompany: req.user.legalName };
  } else if (req.user.role === "employee") {
    let userName = req.user.lastName + " " + req.user.firstName;
    filter = { $or: [
      { contractAgent: userName },
      { contractManager: userName }
      ] };
  } else if (req.user.role === "client") {
    filter = { customers: req.user.customerId };
  }

  Contract.find(filter)
    .lean()
    .then((contracts) => {
      if (!contracts) {
        return res.status(401).json("Not authorised!");
      }
      return res.json(contracts);
    })
    .catch((err) => res.status(500).send("Get contracts operation
failed", err));
});
```

Редагування контракту є досить простим - за унікальним ідентифікатором знаходиться контракт який треба редагувати та вносяться відповідні зміни. Метод findByIdAndUpdate приймає два аргументи - ідентифікатор за яким відбувається пошук запису в документі та об’єкт змін:

```
router.put("/contracts/update", passport.authenticate("jwt", { session: false }), function (req, res) {
  Contract.findByIdAndUpdate(
    req.body.params._id,
    req.body.params,
    {
      useFindAndModify: false,
    },
    function (err, resp) {
      res.json(resp);
    }
  );
});
```

Додавання нового договору значно складніше оскільки при додаванні договору ми одразу створюємо нового клієнта (якщо це новий клієнт) або знаходимо існуючого клієнта, також створюємо новий об’єкт (якщо новий) або знаходимо існуючий об’єкт, а також створюємо сам договір, після чого маємо зробити посилання всіх сутностей один на одного. Посилання необхідні для того, щоб при створенні договору можна було побачити хто клієнт та який об’єкт страхування за цим договором. Відповідно, в сутності клієнта має бути посилання на договір та на об’єкт, а у сутності об’єкта - на договір і клієнта. Код модулю на додавання договору буде виглядати наступним чином:

```
router.post("/contracts/add", passport.authenticate("jwt", { session: false }), async function (req, res) {
  try {
    let foundObject = await Objects.findOne({
      objectVIN: req.body.objectVIN,
      objectPlates: req.body.objectPlates,
      insured: req.body.insured,
      objectAddress: req.body.objectAddress,
```

```
            objectDescription: req.body.objectDescription,
          }).exec();
          let foundCustomer = await Customer.findOne({ inn:
req.body.inn, dateOfBirth: req.body.dateOfBirth }).exec();

          var newObject = new Objects({ ...req.body });
          var newCustomer = new Customer({ ...req.body });
          var newContract = new Contract({ ...req.body });

          var newObjectId = foundObject ? foundObject._id :
newObject._id;
          var newCustomerId = foundCustomer ? foundCustomer._id :
newCustomer._id;

          if (foundObject) {
            console.info("Add objects: found existing one",
foundObject);
            let updateBody = foundObject.toObject();
            updateBody.contracts.push(newContract._id.toString());

            Objects.findByIdAndUpdate(
              foundObject._id,
              { $set: updateBody },
              {
                useFindAndModify: false,
                new: true,
              },
              function (err, resp) {
                if (err) console.log("error in object update:", err);
                return resp;
              }
            );
          } else {
            newObject.contracts = [newContract._id.toString()];
            newObject.customers = [newCustomerId.toString()];
            var createdObj = await newObject.save();
          }

          if (foundCustomer) {
            console.info("Add customer: found existing",
foundCustomer.name);
            let updateBody = foundCustomer.toObject();
            updateBody.contracts.push(newContract._id.toString());
```

```
            updateBody.objects.push(newObjectId.toString());
            Customer.findByIdAndUpdate(
              foundCustomer._id,
              { $set: updateBody },
              {
                useFindAndModify: false,
                new: true,
              },
              function (err, resp) {
                if (err) console.log("error in customer update:", err);
                return resp;
              }
            );
          } else {
            newCustomer.contracts = [newContract._id.toString()];
            newCustomer.objects = [newObjectId.toString()];
            var createdCust = await newCustomer.save();
          }

          newContract.objects.push(newObjectId.toString());
          newContract.customers.push(newCustomerId.toString());

          newContract.save((err, contr) => {
            let ob = foundObject ? foundObject : createdObj;
            let cu = foundCustomer ? foundCustomer : createdCust;
            let resp = { contract: contr, customer: cu, object: ob };
            return res.json(resp);
          });
        } catch (err) {
          console.log(err);
        }
      });
```

Операція видалення контракту має не тільки видаляти сам контракт, але й посилання на такий контракт в інших сутностях, а саме в об’єктах страхування, клієнтах та врегулюваннях:

```
      router.delete("/contracts/delete", passport.authenticate("jwt", {
session: false }), (req, res) => {
        Contract.deleteOne(
          {
            _id: req.query.id,
          },
          function (err, ok) {
```

```
        if (err) return res.send(err);
        res.json({ _id: req.query.id, message: "Deleted" });
      }
    );

    //delete from objects
    Objects.updateMany(
      { contracts: { $elemMatch: { $eq: req.query.id } } },
      {
        $pull: { contracts: req.query.id },
      }
    ).lean();

    //delete from customers
    Customer.updateMany(
      { contracts: { $elemMatch: { $eq: req.query.id } } },
      {
        $pull: { contracts: req.query.id },
      }
    ).lean();

    //delete from settlements
    Settlement.updateMany(
      { contracts: { $elemMatch: { $eq: req.query.id } } },
      {
        $pull: { contracts: req.query.id },
      }
    ).lean();
  });
```

Інші модулі також будуть мати подібну реалізацію отримання інформації відповідно до ролі.

Код модулів для інших сутностей реалізований за аналогією до вищеописаного модулю.

Ще однією вимогою до системи є актуалізація статусу договорів відповідно до строку дії. Таку актуалізацію краще проводити вночі, коли система не використовується людьми. Зміна статусу буде відбуватись за допомогою окремого модулю. Використовуючи cron задамо час виконання функції на третю годину ночі щодня:

```
const cron = require("node-cron");
const Contract = require("../contracts/contractsModel");

var statusUpdater = function () {
  //cron job to update contract statuses
  cron.schedule("0 3 * * *", function () {
    console.log("Nightly contracts status update started");
    Contract.find()
      .then((contracts) => {
        let contractsToUpdate = [];
        function dateConvertion(stringDate) {
          let dateArr = (stringDate.substr(0, 2) + "/" +
stringDate.substr(2, 2) + "/" + stringDate.substr(4)).split("/");
          return new Date(dateArr[2], Number(dateArr[1]) - 1,
dateArr[0]).getTime();
        }

        contracts.forEach((contract) => {
          if (
            dateConvertion(contract.contractEndDate) < Date.now() &&
            contract.contractStatus !== "пролонгований" &&
            contract.contractStatus !== "недіючий" &&
            contract.contractStatus !== "прострочений"
          ) {
            contract.contractStatus = "прострочений";
            contractsToUpdate.push(contract);
          }
        });

        if (contractsToUpdate.length > 0) {
          contractsToUpdate.forEach((contract) => {
            Contract.findByIdAndUpdate(
              contract._id,
              contract,
              {
                useFindAndModify: false,
              },
              function (err, res) {
                console.log("contract updated: " + contract.contractNum
+ " : " + contract.contractEndDate);
              }
            );
          });
```

```
        } else {
          console.log("no outdated contracts");
        }
      })
      .catch((err) => console.log("Update contracts failed", err));
  });
};
```

Останньою частиною є реалізація створення копії локальної бази в хмарну. Знову ж таки, для цього створимо допоміжний модуль, котрий також за допомогою cron розкладу буде робити бекап локальної бази вночі та після цього переносити дані копії в хмарну базу:

```
const cron = require("node-cron");
const spawn = require("child_process").spawn;
require("dotenv").config();
const mongoose = require("mongoose");
mongoose.Promise = global.Promise;

var dbCopier = function () {
  cron.schedule("04 * * *", () => {
    console.log("Nightly DB backup process started");
    let backupProcess = spawn("C:/Program
Files/MongoDB/Server/4.2/bin/mongodump.exe", [
      "-o=C:/srv/BIS-APP/bis/server/_helpers/dbbackup",
      "--db=test",
    ]);

    backupProcess.on("exit", async (code, signal) => {
      if (code) {
        console.log("Backup process exited with code ", code);
      } else if (signal) {
        console.error("Backup process was killed with singal ",
signal);
      } else {
        console.log("Successfully backedup the DB");
      }
    });
  });

  cron.schedule("30 04 * * *", () => {
```

```
    console.log("Nightly DB sync process started");
    let syncProcess = spawn("C:/Program
Files/MongoDB/Server/4.2/bin/mongorestore.exe", [
      "--uri",
      "mongodb+srv://xxxxx:yyyyyy@cluster0-wdvxz.mongodb.net/test",
      "--db",
      "test",
      "--drop",
      "C:/srv/BIS-APP/bis/server/_helpers/dbbackup/test",
    ]);

    syncProcess.on("exit", (code, signal) => {
      if (code) {
        console.log("syncProcess exited with code ", code, signal);
      } else if (signal) {
        console.error("syncProcess was killed with singal ", signal);
      } else {
        console.log("Successfully synced the DB");
      }
    });
  });
};
```

3.4. Інтеграція з блокчейн мережею

Виходячи з вимог, система має забезпечувати підпис даних зі сторони страхового брокера як підтвердження коректності внесених в систему даних, а також підпис за сторони страхової компанії як підтвердження розгляду випадку та прийняття рішення. Очевидно, що страхова компанія не має бачити страховий випадок до моменту підпису зі сторони брокера.

Інформація про страховий випадок має надсилатись в блокчейн мережу тільки після останнього підпису.

Весь процес можно схематично відобразити на діаграмі послідовностей:

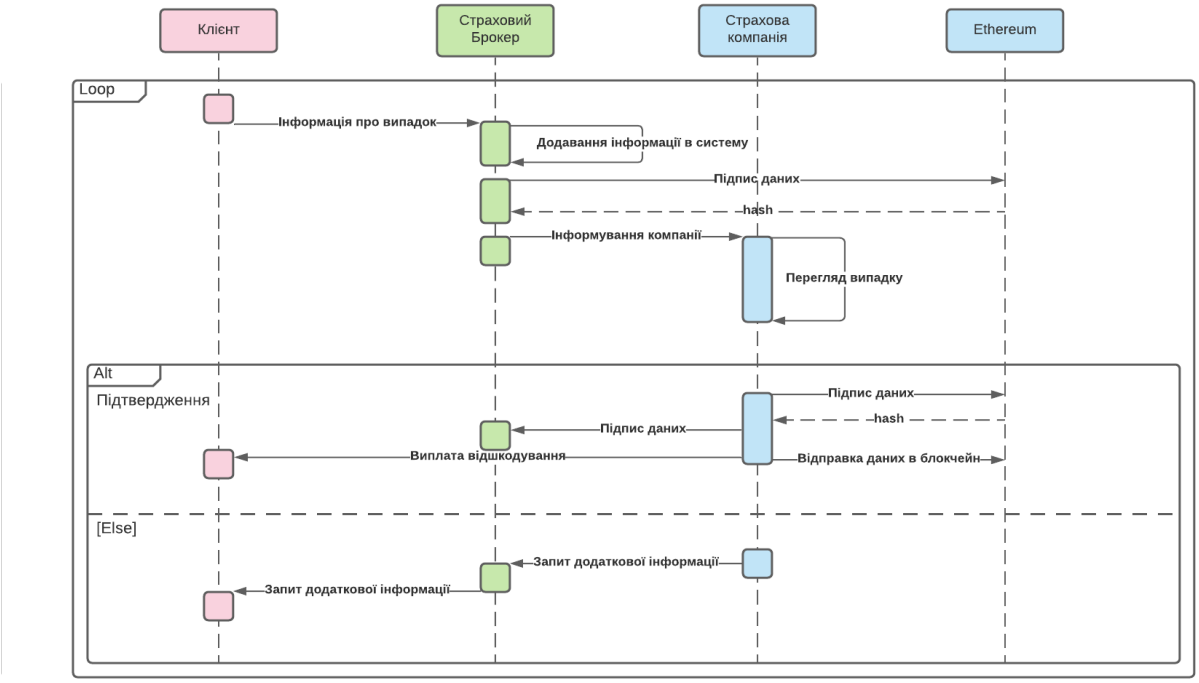

Рис. 3. 1. Діаграма послідовностей для процесу страхового врегулювання з використанням блокчейн платформи

Інтеграція з Ethereum буде відбуватись на клієнтській частині за допомогою web3.js - колекції бібліотек, що дозволяють взаємодіяти з локальним або віддаленим вузлом Ethereum за допомогою HTTP, IPC або WebSocket [11].

Для цілей створення та тестування смарт контрактів будемо використовувати застосунок Ganache. Ganache - це персональна блокчейн платформа для швидкого створення застосунків на Ethereum та Corda. Можна використовувати Ganache протягом усього циклу розробки; що дозволяє розробляти, розгортати та тестувати dApps в безпечному та детермінованому середовищі.[12]

Першим кроком буде створення проекту в Ganache, котрий буде мати блокчейн адреси гаманців з позитивним балансом токенів. Створення такого проекту відбувається в застосунку Ganache автоматично і по завершенні маємо наступний результат:

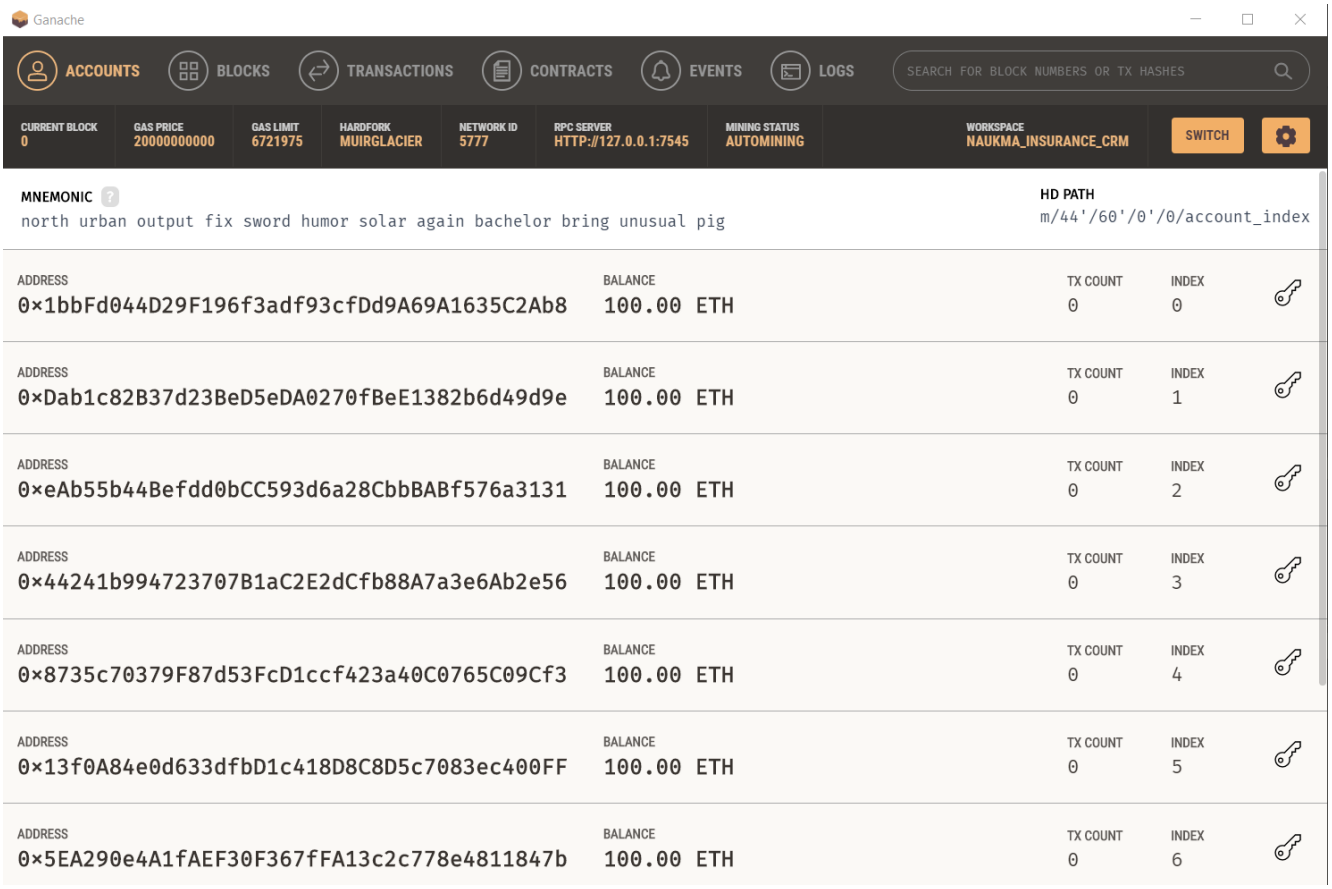

Рис. 3. 2. Інтерфейс користувача застосунку Ganache з створеною тестовою робочою зоною

Як бачимо, ми створили робочу зону в тестовому середовищі, де в нашому розпорядженні є 10 гаманців з балансом у 100 ETH на кожному, автоматичним майнінгом, яка доступна за адресою HTTP://127.0.0.1:7545.

Для підключення до платформи нам достатньо завантажити у систему бібліотеку Web3 та зробити її імпорт у необхідних компонентах. Після цього можна використовувати вбудовані методи бібліотеки. Фрагмент коду компонента, котрий демонструє використання методу підпису даних у функції:

```
async brokerSignForBlockchain(settlement) {
    const web3 = new Web3("http://127.0.0.1:7545");
    const accounts = await web3.eth.getAccounts();
    let address = accounts[0];
```

```
    let dataToSign = {
      customerId: settlement.customersId,
      insObject: settlement.objectId,
      eventType: settlement.eventType,
      damage: settlement.damage,
      incidentDate: settlement.incidentDate,
    };
    web3.eth.sign(dataToSign, address, (error, signature) => {
      if (error) {
        console.error("Failed to sign", error);
        return;
      }
      let settlementToUpdate = {
        _id: settlement._id,
        ownSignature: signature,
      };
      this.props
        .editSettlement(settlementToUpdate)
        .then((signed) => {
          window.alert("Broker signed: ", signed);
        })
        .catch((error) => console.error("Failed to update settlement
with broker signature", error));
    });
  }
```

Дана функція виконує накладання підпису на страховий випадок зі сторони брокерської компанії.

Страховий випадок, котрий ще не підписаний зі стороні брокерської компанії немає сенсу відображати для страхової компанії. А після накладання такого підпису, страхова компанія в веб-інтерфейсі повинна побачити не тільки цей страховий випадок, а й мати кнопку накладання свого підпису на цей випадок. На рисунках 3. 3. та 3. 4. зображено як виглядає веб-інтерфейс користувачів від брокерської та страхової компанії. Кожен користувач має відповідну кнопку підпису.

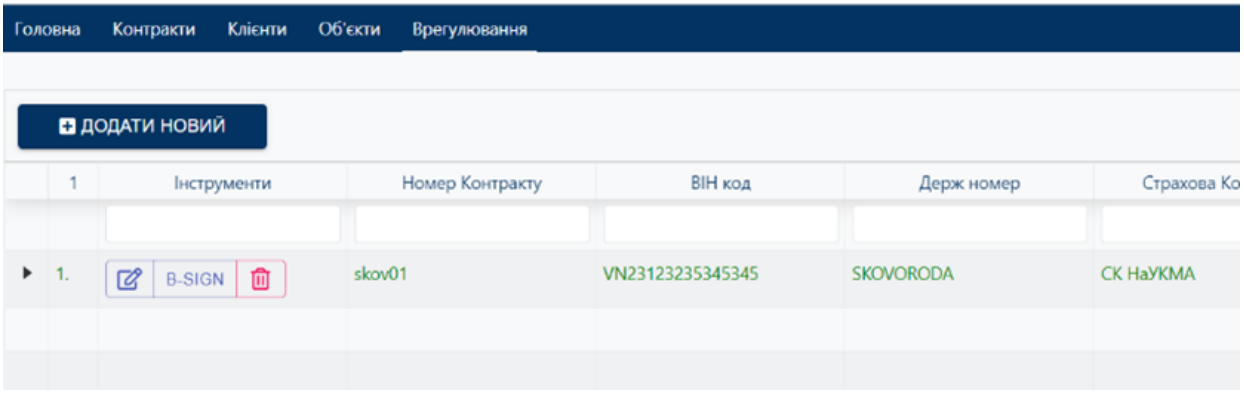

Рис. 3. 3. Веб-інтерфейс для користувача системи від брокерської компанії

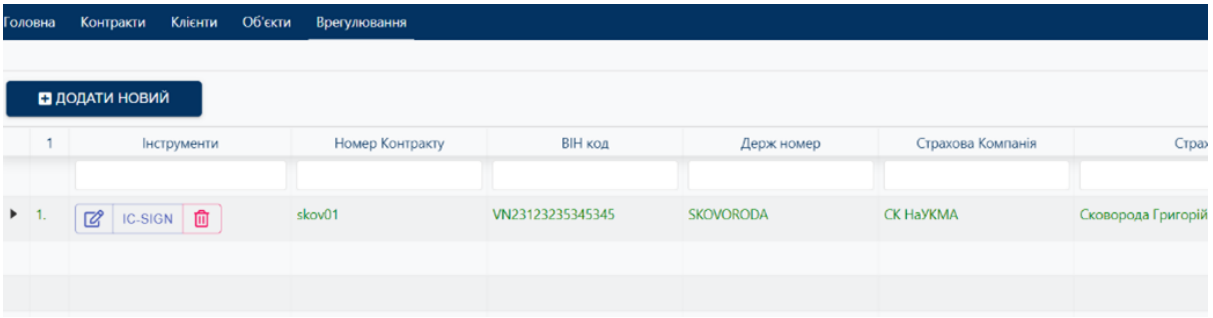

Рис. 3.4. Веб-інтерфейс для користувача системи від страхової компанії

Функція підпису страхового випадку зі сторони страхової компанії створена за повною аналогією, з однією відмінністю - після успішного підпису страховий випадок одразу надсилається в блокчейн мережу.

Для цього необхідно створити смарт-контракт та загрузити його в мережу Ethereum.

Смарт контракт має виконувати дві функції:

1. Додавання інформації по клієнту до блоку
2. Отримання інформації по клієнту за його ідентифікатором

Смарт-контракт створюється на мові Solidity, котра була спеціально розроблена для написання смарт-контрактів в Ethereum. Контракт буде мати наступний вигляд:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.0;
```

```
contract Settlements {
    mapping (string => string[]) settlementsRecords;

    function addSettlement (
        string calldata _customerId,
         string calldata _settlementDetails) public {

        string[] storage custArr;
        custArr = settlementsRecords[_customerId];
        custArr.push(_settlementDetails);
        settlementsRecords[_customerId] = custArr;
    }

    function getSettlementsByCustomerId (string calldata _custId) view
public returns(string[] memory) {
        return settlementsRecords[_custId];
    }
}
```

Тут ми створюємо змінну *settlementsRecords* типу *mapping*, котра фактично реалізує інтерфейс асоціативного масиву для зберігання інформації по клієнту, де ключом до масиву буде унікальній ідентифікатор клієнта. Далі по коду є декларація двох функцій - додавання інформації в змінну settlementsRecords та отримання інформації за ідентифікатором клієнта.

Цікавим є той факт, що смарт контракти оперують не більш ніж 16 змінними на весь контракт. Тому, замість того, щоб передавати декілька аргументів у контракт, ми передаємо один аргумент, котрий містить всю необхідну інформацію у вигляді тексту.

Після реалізації контракту та його завантаження до мережі Ethereum отримаємо адресу контракту:

```
Replacing 'Settlements'
-----------------------
> transaction hash:    0x5aea2a40d7c27ea1dffcb69834563f30f5e1d820500c1d9f0e71674cf1092a66
> Blocks: 0            Seconds: 0
> contract address:    0x50FF74861C23Fbf60afc7685205918042Df096Ed
> block number:        15
> block timestamp:     1621877259
> account:             0x754F0421B2f00f4Eb2cDe6A33cbD21aD2E029d64
> balance:             99.96865026
> gas used:            468104 (0x72488)
> gas price:           20 gwei
> value sent:          0 ETH
> total cost:          0.00936208 ETH


> Saving migration to chain.
> Saving artifacts
-------------------------------------
> Total cost:          0.00936208 ETH
```

Рис. 3. 5. Результат завантаження смарт контракту в мережу Ethereum

В нашому випадку адреса контракту відображається у рядку:

contract address: 0x50FF74861C23Fbf60afc7685205918042Df096Ed

Маючи смарт-контракт, можемо переходити до його фактичного використання в системі. За допомогою наступної функції додамо в блокчейн мережу інформацію про страховий випадок по клієнту (фрагмент коду компонента):

```
sendToEthereum(web3, settlement, address) {
    const blockchainedSettlements = new web3
      .eth
      .Contract(SMC.SETTLEMENTS_ABI, SMC.SETTLEMENTS_ADDRESS);
    let objectOfIncident = this.props
            .objects
            .find((obj) => obj._id === settlement.objectId);
    let dataToBeBlockchained = {
      eventType: settlement.eventType,
      incidentDate: settlement.incidentDate,
      eventDescription: settlement.eventDescription,
      damage: settlement.damage,
      guilty: settlement.guilty,
      objectVin: objectOfIncident.objectVIN,
      objectDescription: objectOfIncident.objectDescription,
```

```
    objectAddress: objectOfIncident.objectAddress,
  };

  blockchainedSettlements
    .methods
    .addSettlement(
          settlement.customerId,
          JSON.stringify(dataToBeBlockchained))
    .send({
      from: address,
      gas: 6721975,
    })
    .once("receipt", (receipt) => {
      window.alert("Settlement added", receipt);
    });
}
```

Метод Contract бібліотеки web3 приймає два аргументи - бінарний інтерфейс застосування контракту (ABI) та адресу смарт-контракту в мережі Ethereum.

Бінарний інтерфейс застосування контракту (ABI) - це стандартний спосіб взаємодії з контрактами в екосистемі Ethereum, як за межами блокчейну, так і для взаємодії між контрактами. Дані кодуються відповідно до їх типу, як описано в специфікації. Кодування не є самоописом, і тому для декодування потрібна схема. [13]

ABI нашого контракту можна знайти в файлі Settlements.json, котрий створюється як результат компіляції контракт та має наступний вигляд:

```
export const SETTLEMENTS_ABI = [
  {
    inputs: [
      {
        internalType: "string",
        name: "_customerId",
        type: "string",
      },
      {
        internalType: "string",
        name: "_settlementDetails",
```

```
            type: "string",
          },
        ],
        name: "addSettlement",
        outputs: [],
        stateMutability: "nonpayable",
        type: "function",
      },
      {
        inputs: [
          {
            internalType: "string",
            name: "_custId",
            type: "string",
          },
        ],
        name: "getSettlementsByCustomerId",
        outputs: [
          {
            internalType: "string[]",
            name: "",
            type: "string[]",
          },
        ],
        stateMutability: "view",
        type: "function",
        constant: true,
      },
    ];
```

Результатом виконання даної функції додасть інформацію в мережу Ethereum і поверне нам receipt.

Після того, як була реалізована частина підпису та відправлення даних в блокчейн мережу можемо приступати до реалізації фінальної частини - отримання даних з мережі. Так само за допомогою web3 визиваємо метод смарт контракту:

```
async getBlockainedSettlements(row) {
    const web3 = new Web3("http://127.0.0.1:7545");
    const blockchainedSettlements = new web3
      .eth
```

```
      .Contract(SMC.SETTLEMENTS_ABI, SMC.SETTLEMENTS_ADDRESS);

    let customersSettlements = await blockchainedSettlements
      .methods
      .getSettlementsByCustomerId(row._id).call();

    this.setState({
      blockchainedData: [].concat(JSON.parse(customersSettlements)),
      showBModal: true,
    });
  }
```

Оскільки дані в блокчейн ми зберігаємо у вигляді тексту, то відповідь треба розпарсити за допомогою функції JSON.parse()

Внесемо необхідні зміни у веб-інтерфейс для можливості отримання даних користувачем.

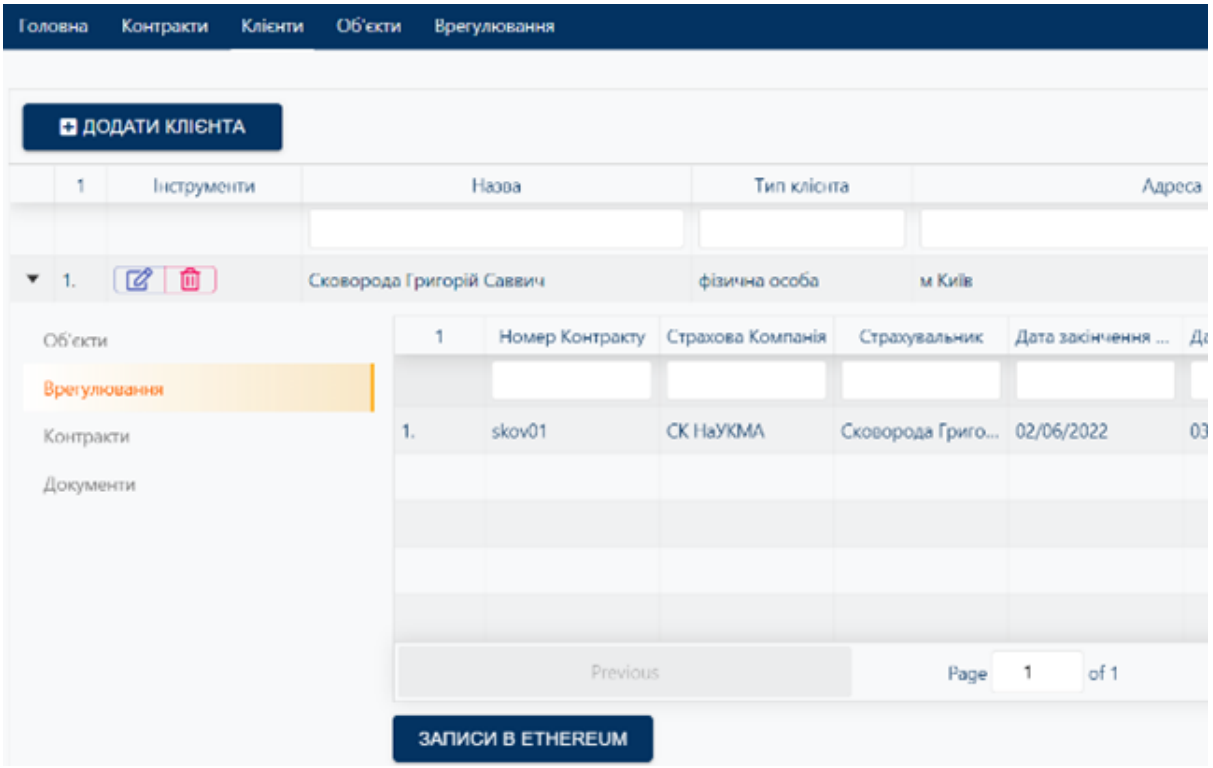

Рис. 3. 6. Кнопка отримання даних по страховим випадкам клієнта з мережі Ethereum.

Тепер, користувач системи може в будь-який час отримати дані по всім страховим випадкам клієнта натиснувши відповідну кнопку. Результат виконання зображено на рисунку 3. 7.

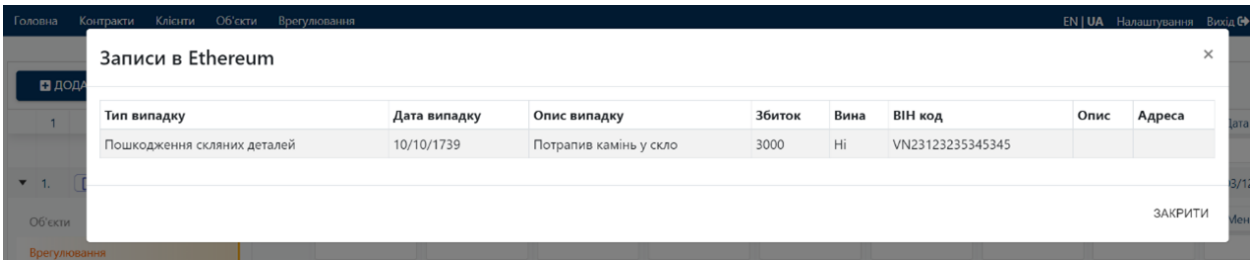

Рис. 3. 7. Дані, отримані з мережі Ethereum

Таким чином ми реалізували інтеграцію з блокчейн платформою, створили смарт контракт на додавання інформації в блокчейн та її отримання, загрузили смарт контракт в блокчейн та викликали методи створеного контракту.

Висновки

В першому розділі даної роботи було надано визначення системи управління взаємовідносинами з клієнтами, зроблено огляд та класифікація існуючих на сьогодні видів систем управління взаємовідносинами, надані приклади таких систем.

У другому розділі було надано визначення архітектурно значущим вимогам, а також були проаналізовані основні вимоги до системи управління взаємовідносинами з клієнтами. Базуючись на вимогах була розроблена архітектура системи.

В третьому розділі описано процес створення застосунку з урахування розробленої архітектури та її практична реалізація. Описано які саме технології та бібліотеки було обрано для реалізації та чому, як створюється серверна та клієнтська частини застосунку, а також відображено як відбувається інтеграція застосунку з блокчейн платформою з описом процесу роботи з блокчейн технологією.

В результаті роботи була побудована система, котра відповідає всім вимогам, котрі перед нею ставилися і у разі необхідності може слугувати базою для розширення і подальшого повноцінного комерційного використання.

Використання блокчейн платформи дає унікальні переваги перед аналогічними системами, оскільки дозволяє зберігати та отримувати необхідну інформацію таким чином, що достовірність цієї інформації не буде викликати сумнівів серед усіх учасників процесу.

Список літератури

1. Управління_відносинами_з_клієнтами. [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Управління_відносинами_з_клієнтами
2. Програмне забезпечення. [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Програмне_забезпечення
3. Програмний продукт. [Електронний ресурс]. Режим доступу: https://uk.wikipedia.org/wiki/Програмний_продукт
4. Hussein A. Al-Homery, Hasbullah Asharai, Azizah Ahmad. The Core Components and Types of CRM. *Pakistan Journal of Humanities and Social Sciences January – March 2019*, Volume 7, No. 1, pp. 121 – 145
5. Capterra. "*CRM User Research Infographic*". [Електронний ресурс]. 2020. Режим доступу: https://www.capterra.com/customer-relationship-management-software/user-research-infographic
6. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. *Pearson. Canada,* 2017.
7. Kumar V., Reinartz W. (2018) Strategic CRM. In: Customer Relationship Management. Springer Texts in Business and Economics. Springer, Berlin, Heidelberg.
8. Keeling M. *Design It! From Programmer to Software Architect.* The Pragmatic Programmers. Raleight, North Carolina, USA. 2017.
9. Richards M. *Software Architecture Patterns.* O'Reilly Media, Inc., CA, USA, 2015
10. Bass L., Clements P., Kazman R. Software Architecture in Practice, 3rd Edition. *Addison-Wesley Professional. Canada,* 2012.

11. Chan Rosalie. *"The 10 most popular programming languages, according to the Microsoft-owned GitHub"*. 2019. [Електронний ресурс]. Режим доступу: https://www.businessinsider.com/most-popular-programming-languages-github-2019-11
12. Programming languages used in most popular websites. [Електронний ресурс]. Режим доступу: https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites
13. web3.js: Ethereum JavaScript API. [Електронний ресурс]. Режим доступу: https://web3js.readthedocs.io/en/v1.3.4/
14. Ganache. *Truffle Suite.* [Електронний ресурс]. Режим доступу: https://www.trufflesuite.com/docs/ganache/overview
15. Solidity. [Електронний ресурс]. Режим доступу: https://docs.soliditylang.org/en/v0.5.3/abi-spec.html
16. MongoDB. . [Електронний ресурс]. Режим доступу: https://docs.mongodb.com/
17. Mongoose. [Електронний ресурс]. Режим доступу: https://mongoosejs.com/
18. React. A JavaScript library for building user interfaces. [Електронний ресурс]. Режим доступу: https://reactjs.org/
19. Create-react-app. [Електронний ресурс]. Режим доступу: https://github.com/facebook/create-react-app
20. Redux. A Predictable State Container for JS Apps. [Електронний ресурс]. Режим доступу: https://redux.js.org/
21. React Router. [Електронний ресурс]. Режим доступу: https://reactrouter.com
22. McCubbin Gregory. "*How to Build Ethereum Dapp with React.js. Complete Step-By-Step Guide*" 2021. [Електронний ресурс]. Режим

доступу: https://www.dappuniversity.com/articles/ethereum-dapp-react-tutorial

23. Система управления взаимоотношениями с клиентами. [Електронний ресурс]. Режим доступу: https://ru.wikipedia.org/wiki/Система_управления_взаимоотношениями_с_клиентами

Додаток А. Програмний код компоненту ObjectsTable

```
import React from "react";
import { connect } from "react-redux";
import ReactTable from "react-table-6";
import matchSorter from "match-sorter";
import "react-table-6/react-table.css";
import FoldableTableHOC from "react-table-6/lib/hoc/foldableTable";
import { Nav, Row, Col, Tab } from "react-bootstrap";
import ContractsSubTable from "../Subtables/ContractsSubTable";
import CustomersSubTable from "../Subtables/CustomersSubTable";
import SettlementsSubTable from "../Subtables/SettlementsSubTable";
import ButtonGroup from "@material-ui/core/ButtonGroup";
import FilesDragNDrop from "../Common/DragAndDrop";
import CancelIcon from "@material-ui/icons/Cancel";
import Button from "@material-ui/core/Button";
import { getObjects, editObject, deleteObject, addObject,
removeObjectReference } from "../../actions/objects";
import { deleteFile } from "../../actions/files";
import { editContract, getContracts, removeContractReference } from
"../../actions/contracts";
import { editCustomer, getCustomers, removeCustomerReference } from
"../../actions/customers";
import { getSettlements } from "../../actions/settlements";
import { setLoading } from "../../actions/API_helpers";
import { URL } from "../../CONSTANTS";
import ObjectsModal from "./ObjectsModal";
import { setSubModalContent } from "../../actions/modals";
import { withTranslation } from "react-i18next";
import { buttonStyle } from "../Common/Styles";
import { ClipLoader } from "react-spinners";
import AttachItemModal from "../Subtables/AttachItemModal";
import { addSlashesToString } from "../../helpers/dateTransformations";
const FoldableTable = FoldableTableHOC(ReactTable);

export class ObjectsTable extends React.Component {
  constructor(props) {
    super();
    this.state = {
      row: null,
      filtered: props.objects.length,
      showModal: false,
```

```
  };
  this.selectTable = React.createRef();
  this.updateContent = this.updateContent.bind(this);
  this.closeModal = this.closeModal.bind(this);
  this.openModal = this.openModal.bind(this);
  this.handleRowExpanded = this.handleRowExpanded.bind(this);
  this.closeAttachItemModal = this.closeAttachItemModal.bind(this);
  this.addItems = this.addItems.bind(this);
}

updateContent(data) {
  this.props.setLoading(true);
  if (!data._id) {
    data._id = this.state.row._id;
  }

  this.props.editObject(data).then(
    (res) => {
      console.log("SUCCESS!", res);
      this.props.getObjects();
      this.props.setLoading(false);
    },
    (err) => {
      console.log("Error: ", err);
      this.props.setLoading(false);
    }
  );

  this.setState({
    showModal: false,
  });

  this.props.setSubModalContent({});
}

handleAddNewClick = function (target, row, context) {
  this.setState({
    showAttachItemModal: !this.state.showAttachItemModal,
    attachItemModalViewType: target,
    row,
  });
};
closeAttachItemModal() {
```

```
  this.setState({
    showAttachItemModal: !this.state.showAttachItemModal,
    attachItemModalViewType: null,
  });
}

addItems(data) {
  this.setState({ row: Object.assign({}, { ...data }) });
  this.updateContent(data);
}

handleRowExpanded(newExpanded, index, event) {
  if (this.state.expanded[index]) {
    this.setState({
      expanded: { [index]: false },
    });
  } else {
    this.setState({
      expanded: { [index]: true },
    });
  }
}

handleAddedImage(row) {
  this.setState({
    expanded: { [row.index]: true },
  });
}

closeModal() {
  this.props.setSubModalContent({});
  this.setState({
    showModal: false,
  });
}

openModal(row) {
  this.setState({
    showModal: true,
    row,
  });
}
```

```
  deleteObject(data) {
    if (window.confirm("Are you sure you want to delete item?")) {
      this.props.deleteObject(data._id);
    }
  }

  downloadFile(dir) {
    window.open(URL + "/static/" + dir.folder + "/" + dir.fileName,
"Window Title", "width=500, height=450");
  }

  deleteFile(obj, ind) {
    obj.objectPhotos.splice(ind, 1);
    this.props.deleteFile(obj).then((res) => {
      console.log("file deleted: " + res);
    });
  }

  putObjectDataToStoreAndOpenModal(object) {
    this.props.setSubModalContent(object);
    this.openModal(object);
  }

  cloneObject = function (object) {
    let clonedObject = Object.assign({}, { ...object });
    this.addClonedLabelToProperties(clonedObject);
    this.deleteNotNeededProperties(clonedObject);

    this.props.addObject(clonedObject).then((res) => {
      this.addNewObjectToCustomer(clonedObject, res);
      this.addNewObjectToContract(clonedObject, res);
      this.props.getObjects();
    });
  };

  addClonedLabelToProperties(clonedObject) {
    clonedObject.owner = "cloned_" + clonedObject.owner;
    clonedObject.objectVIN = "cloned_" + clonedObject.objectVIN;
    clonedObject.createdDate = new Date().getTime();
  }

  deleteNotNeededProperties(clonedObject) {
   delete clonedObject.settlements;
```

```
    delete clonedObject.objectPhotos;
    delete clonedObject._id;
    delete clonedObject.__v;
  }

  addNewObjectToContract(clonedObject, res) {
    clonedObject.contracts.forEach((contractId) => {
      let contractToUpdate = this.props.contracts.filter((contr) =>
contr._id === contractId);
      contractToUpdate.forEach((contract) => {
        contract.objects.push(res._id);
        this.props.editContract(contract);
      });
    });
  }

  addNewObjectToCustomer(clonedObject, res) {
    clonedObject.customers.forEach((custId) => {
      let customerToUpdate = this.props.customers.filter((cust) =>
cust._id === custId);
      customerToUpdate.forEach((customer) => {
        customer.objects.push(res._id);
        this.props.editCustomer(customer);
      });
    });
  }

  removeReference = function (from, objRow, subTableRow) {
    this.props.removeObjectReference(from, objRow.original,
subTableRow);
  };

  render() {
    const sortings = {
      dateSort: (a, b, desc) => {
        if (desc) {
          return new Date(Number(a.substr(4)), Number(a.substr(2, 2)) -
1, Number(a.substr(0, 2))) <
            new Date(Number(b.substr(4)), Number(b.substr(2, 2)) - 1,
Number(b.substr(0, 2)))
            ? -1
            : 1;
        }
```

```
        return new Date(Number(a.substr(4)), Number(a.substr(2, 2)) -
1, Number(a.substr(0, 2))) >
          new Date(Number(b.substr(4)), Number(b.substr(2, 2)) - 1,
Number(b.substr(0, 2)))
          ? 1
          : -1;
      },
    };
    const filters = {
      startEnd: (filter, row) =>
row[filter.id].startsWith(filter.value) &&
row[filter.id].endsWith(filter.value),
      includes: (filter, row) => {
        return
String(row._original[filter.id]).toLowerCase().includes(filter.value.to
LowerCase());
      },
      dateIncludes: (filter, row) => {
        return
String(row._original[filter.id]).includes(filter.value.replace(/\D/gm,
""));
      },
      matchSrtr: (filter, rows) => matchSorter(rows, filter.value, {
keys: ["firstName"] }),
    };

    return (
      <>
        {this.props.loading && (
          <div className="loading">
            <ClipLoader color="#00BFFF" size={150} />
          </div>
        )}
        <AttachItemModal
          fromRow={this.state.row}
          contentType={this.state.attachItemModalViewType}
          saveContent={this.addItems}
          closeModal={this.closeAttachItemModal}
          showAttachItemModal={this.state.showAttachItemModal}
        />
        <ObjectsModal isShow={this.state.showModal}
closeModal={this.closeModal} openModal={this.openModal} />
        <FoldableTable
```

```
            data={this.props.objects.sort((a, b) => {
              return b.createdDate - a.createdDate;
            })}
            onFoldChange={(newFolded) =>
              this.setState((p) => {
                return { folded: newFolded };
              })
            }
            folded={this.state.folded}
            filterable
            collapseOnDataChange={false}
            ref={this.selectTable}
            onFilteredChange={() => {
              this.setState({ filtered:
this.selectTable.current.wrappedInstance.getResolvedState().sortedData.
length });
            }}
            defaultFilterMethod={filters.includes}
            columns={[
              {
                columns: [
                  {
                    Header: this.state.filtered,
                    width: 40,
                    filterable: false,
                    Cell: (row) => {
                      return row.index + 1 + ".";
                    },
                  },
                  {
                    Header: this.props.t("actions"),
                    width: 200,
                    filterable: false,
                    accessor: "action",
                    Cell: (row) => (
                      <ButtonGroup size="small" variant="outlined"
color="primary" aria-label="small outlined button group">
                        <Button
                          disabled={this.props.auth.user.role ===
"client"}

onClick={this.putObjectDataToStoreAndOpenModal.bind(this,
row.original)}
```

```
                        >
                          <i className="far fa-edit fa-lg"></i>
                        </Button>
                        <Button disabled={this.props.auth.user.role ===
"client"} onClick={this.cloneObject.bind(this, row.original)}>
                          <i className="far fa-copy fa-lg"></i>
                        </Button>
                        <Button
                          disabled={this.props.auth.user.role ===
"client"}
                          color="secondary"
                          onClick={this.deleteObject.bind(this,
row.original)}
                        >
                          <i className="far fa-trash-alt fa-lg"></i>
                        </Button>
                      </ButtonGroup>
                    ),
                  },
                  {
                    Header: this.props.t("objectsView.objDate"),
                    accessor: "objDate",
                    filterMethod: filters.dateIncludes,
                    sortMethod: sortings.dateSort,
                    Cell: (props) => <span>{props.original.objDate ?
addSlashesToString(props.original.objDate) : null}</span>,
                  },
                  {
                    Header: this.props.t("objectsView.owner"),
                    width: 320,
                    accessor: "owner",
                  },
                  {
                    Header: this.props.t("objectsView.objectBrand"),
                    width: 150,
                    accessor: "objectBrand",
                  },
                  {
                    Header: this.props.t("objectsView.objectModel"),
                    accessor: "objectModel",
                  },
                  {
                    Header: this.props.t("objectsView.objectPlates"),
```

```
                accessor: "objectPlates",
              },
              {
                Header: this.props.t("objectsView.objectVIN"),
                width: 200,
                accessor: "objectVIN",
              },
              {
                Header: this.props.t("objectsView.insurer"),
                width: 250,
                accessor: "insurer",
              },
              {
                Header: this.props.t("objectsView.insured"),
                accessor: "insured",
              },
              {
                Header: this.props.t("objectsView.status"),
                accessor: "status",
              },
            ],
          },
          {
            Header: this.props.t("objectsView.additionalInfo"),
            foldable: true,
            columns: [
              {
                Header: this.props.t("objectsView.objectDocType"),
                accessor: "objectDocType",
              },
              {
                Header: this.props.t("objectsView.objectDocNumber"),
                accessor: "objectDocNumber",
              },
              {
                Header: this.props.t("objectsView.objectDocIssuer"),
                accessor: "objectDocIssuer",
              },
              {
                Header:
this.props.t("objectsView.objectDocIssueDate"),
                accessor: "objectDocIssueDate",
                filterMethod: filters.dateIncludes,
```

```
                  sortMethod: sortings.dateSort,
                  Cell: (props) => (
                    <span>{props.original.objectDocIssueDate ?
addSlashesToString(props.original.objectDocIssueDate) : null}</span>
                  ),
                },
                {
                  Header:
this.props.t("objectsView.objectDescription"),
                  accessor: "objectDescription",
                },
                {
                  Header: this.props.t("objectsView.objectYom"),
                  accessor: "objectYom",
                },
                {
                  Header: this.props.t("objectsView.objectFullWeight"),
                  accessor: "objectFullWeight",
                },
                {
                  Header:
this.props.t("objectsView.objectEmptyWeight"),
                  accessor: "objectEmptyWeight",
                },
                {
                  Header:
this.props.t("objectsView.objectEngineVolume"),
                  accessor: "objectEngineVolume",
                },
              ],
            },
          ]}
          defaultPageSize={20}
          className="-striped -highlight"
          SubComponent={(row) => {
            return (
              <div className="subtable-block">
                <Tab.Container id="vertical-nav"
defaultActiveKey="first">
                  <Row style={{ height: "100%" }}>
                    <Col style={{ maxWidth: 300 }}>
                      <Nav variant="pills" className="flex-column">
                        <Nav.Item>
```

```
                          <Nav.Link
eventKey="first">{this.props.t("contract.Customers")}</Nav.Link>
                        </Nav.Item>
                        <Nav.Item>
                          <Nav.Link
eventKey="second">{this.props.t("contract.Settlements")}</Nav.Link>
                        </Nav.Item>
                        <Nav.Item>
                          <Nav.Link
eventKey="third">{this.props.t("contract.Contracts")}</Nav.Link>
                        </Nav.Item>
                        <Nav.Item>
                          <Nav.Link
eventKey="fourth">{this.props.t("contract.Documents")}</Nav.Link>
                        </Nav.Item>
                      </Nav>
                    </Col>
                    <Col className="subtable-right-panel">
                      <Tab.Content>
                        <Tab.Pane eventKey="first">
                          <CustomersSubTable
                            customers={this.props.customers.filter((el)
=> row.original.customers.some((el2) => el2 === el._id))}

removeCustomer={this.removeReference.bind(this, "customers", row)}
                            single={true}
                            row={row}
                          />
                          {this.props.auth.user.role !== "client" && (
                            <div>
                              <Button
                                disabled={this.props.auth.user.role ===
"client"}
                                variant="contained"

onClick={this.handleAddNewClick.bind(this, "customers", row)}
                                style={buttonStyle}
                              >
                                <i className="fas fa-plus-square"></i>
                                {this.props.t("Add new")}
                              </Button>
                            </div>
                          )}
```

```
                        </Tab.Pane>
                        <Tab.Pane eventKey="second">
                          <SettlementsSubTable

settlements={this.props.settlements.filter((el) =>
row.original.settlements.some((el2) => el2 === el._id))}
                            row={row}
                          ></SettlementsSubTable>
                        </Tab.Pane>
                        <Tab.Pane eventKey="third">
                          <ContractsSubTable
                            contracts={this.props.contracts.filter((el)
=> row.original.contracts.some((el2) => el2 === el._id))}
                            row={row}

removeContract={this.removeReference.bind(this, "contracts", row)}
                          ></ContractsSubTable>
                          {this.props.auth.user.role !== "client" && (
                            <div>
                              <Button
                                disabled={this.props.auth.user.role ===
"client"}
                                variant="contained"

onClick={this.handleAddNewClick.bind(this, "contracts", row)}
                                style={buttonStyle}
                              >
                                <i className="fas fa-plus-square"></i>
                                {this.props.t("Add new")}
                              </Button>
                            </div>
                          )}
                        </Tab.Pane>
                        <Tab.Pane eventKey="fourth">
                          <div style={{ position: "relative" }}>
                            <div
                              className="photos-container"
                              style={{
                                textAlign: "center",
                                display: "inline-block",
                                position: "absolute",
                                width: 400,
                                top: 0,
```

```
                                }}
                              >
                                <FilesDragNDrop
                                  objectData={{
                                    pref: row.original.owner,
                                    suf: row.original.objDate,
                                    entity: { object: row.original },
                                  }}
                                  docFolder={"object"}

handleAddedImage={this.handleAddedImage.bind(this, row)}
                                ></FilesDragNDrop>
                              </div>
                              <div
                                style={{
                                  position: "relative",
                                  left: 400,
                                }}
                              >
                                <div style={{ display: "inline-block",
position: "relative" }}>

{row.original.objectPhotos.sort().map((fileName, ind) => {
                                    let dir = {
                                      folder:
                                        "object/" +
                                        row.original.owner
                                          .split("")
                                          .filter((el) => ["'", '"', "+",
"/", ".", "*"].every((badS) => badS !== el))
                                          .join("") +
                                        "_" +
                                        row.original.objDate,
                                      fileName: fileName,
                                    };
                                    return (
                                      <div
                                        key={ind}
                                        className="filesUploaded"
                                        style={{
                                          margin: "10px 5px",
                                        }}
                                      >
```

```
                                        <div
                                          style={{
                                            width: 150,
                                            height: 150,
                                            border: "1px solid
lightgray",
                                            borderRadius: 9,
                                            textAlign: "right",
                                            position: "relative",
                                          }}
                                        >
                                          <div style={{ height: 25 }}>
                                            <p
                                              style={{
                                                overflowWrap: "anywhere",
                                                fontSize: "12px",
                                                overflow: "hidden",
                                                textOverflow: "ellipsis",
                                                whiteSpace: "nowrap",
                                                paddingLeft: 5,
                                                width: 120,
                                              }}
                                            >
                                              <a
                                                style={{
                                                  color: "#033362",
                                                  overflowWrap:
"anywhere",
                                                  fontSize: "12px",
                                                }}
                                                href={URL + "/static/" +
dir.folder + "/" + dir.fileName}
                                                target="_blank"
                                                rel="noopener noreferrer"
                                              >
                                                {dir.fileName}
                                              </a>
                                            </p>
                                            <CancelIcon
                                              style={{
                                                position: "absolute",
                                                top: 0,
                                                right: 0,
```

```
                                                color: "#dc5a5a",
                                              }}

onClick={this.deleteFile.bind(this, row.original, fileName)}
                                            />
                                          </div>

{dir.fileName.substr(dir.fileName.length - 4) !== ".pdf" && (
                                            <div style={{ height: 100,
textAlign: "center", background: "#e1e1e1", padding: "2px 0px" }}>
                                              <div
                                                style={{
                                                  backgroundImage:
'url("' + URL + "/static/" + dir.folder + "/" + dir.fileName + '")',
                                                  height: 110,
                                                  backgroundSize:
"cover",
                                                  backgroundPosition:
"center",
                                                }}
                                              ></div>
                                            </div>
                                          )}

{dir.fileName.substr(dir.fileName.length - 4) === ".pdf" && (
                                            <div style={{ position:
"absolute", height: 125, overflow: "hidden" }}>
                                              <embed
                                                name="plugin"
                                                width="100%"
                                                src={URL + "/static/" +
dir.folder + "/" + dir.fileName}
                                                type="application/pdf"
                                              ></embed>
                                            </div>
                                          )}
                                        </div>
                                      </div>
                                    );
                                  })}
                                </div>
                              </div>
                            </div>
```

```
                        </Tab.Pane>
                      </Tab.Content>
                    </Col>
                  </Row>
                </Tab.Container>
              </div>
            );
          }}
        />
      </>
    );
  }
}

const mapStateToProps = (state) => ({
  auth: state.auth,
  loading: state.helpers.loading,
  customers: state.customers.customersList,
  contracts: state.contracts.contractsList,
  objects: state.objects.objectsList,
  settlements: state.settlements.settlementsList,
  fileList: state.files.fileList,
});

export default connect(mapStateToProps, {
  getObjects,
  editObject,
  deleteObject,
  addObject,
  deleteFile,
  setSubModalContent,
  getSettlements,
  getContracts,
  editContract,
  getCustomers,
  removeContractReference,
  removeCustomerReference,
  removeObjectReference,
  editCustomer,
  setLoading,
})(withTranslation()(ObjectsTable));
```

Додаток Б. Програмний код компоненту ObjectsModal

```
import React from "react";
import { connect } from "react-redux";
import { Modal } from "react-bootstrap";
import ObjectsModalContent from "./ObjectsModalContent";
import Button from "@material-ui/core/Button";
import { emptyContract } from "../Common/emptyContract";
import { setLoading } from "../../actions/API_helpers";
import { addObject, getObjects, editObject, deleteObject } from
"../../actions/objects";
import { setSubModalContent } from "../../actions/modals";

export class ObjectsModal extends React.Component {
  constructor(props) {
    super(props);
    const content = Object.keys(this.props.modalContent).length ?
this.props.modalContent : emptyContract;
    this.state = {
      isSaveButtonDisabled: true,
      currentContract: content,
    };
    this.handleInputChanges = this.handleInputChanges.bind(this);
    this.closeModal = this.closeModal.bind(this);
  }

  handleInputChanges(updatedFields) {
    this.setState({
      isSaveButtonDisabled: false,
      currentContract: updatedFields,
    });
  }

  saveAndClose(data, isEditMode, notCloning, event) {
    if (isEditMode) {
      this.props.setLoading(true);
      if (!data._id) {
        data._id = this.props.modalContent._id;
        data.customers = this.props.modalContent.customers;
        data.contracts = this.props.modalContent.contracts;
        data.settlements = this.props.modalContent.settlements;
      }
```

```
      this.props.closeModal();
      this.props.setSubModalContent({});

      this.props.editObject(data).then(
        (res) => {
          this.props.getObjects();
          this.props.setLoading(false);
        },
        (err) => {
          console.log("Error in Objects modal", err)
          this.props.setLoading(false);
        }
      );
    } else {
      this.props.saveContent(data);
    }

    this.setState({
      isSaveButtonDisabled: true,
      currentContract: emptyContract,
    });
  }

  closeModal() {
    this.setState({
      isSaveButtonDisabled: true,
      currentContract: emptyContract,
    });
    this.props.closeModal();
  }

  updateState() {
    if (this.props.modalContent.objectAddress) {
      let tempObj = {};
      for (let key in this.state.currentContract) {
        tempObj[key] = this.props.modalContent[key];
      }

      this.setState({
        currentContract: tempObj,
      });
    }
  }
```

```
  render() {
    const isEditing = Object.keys(this.props.modalContent).length ?
true : false;
    const notCloning = true;

    return (
      <>
        <Modal
          show={this.props.isShow}
          onEnter={this.updateState.bind(this)}
          onHide={this.closeModal}
          size="lg"
          dialogClassName="modal-90w"
        >
          <Modal.Header closeButton>
            <Modal.Title>{isEditing ? "Edit object" : "Add new
object"}</Modal.Title>
          </Modal.Header>
          <Modal.Body>
            <ObjectsModalContent contract={this.state.currentContract}
setData={this.handleInputChanges} />
          </Modal.Body>
          {
            <Modal.Footer>
              <Button
                variant="contained"
                classes={{
                  root: "buttonStyle",
                  disabled: "disabledButtonStyle",
                }}
                onClick={this.closeModal}
              >
                Close
              </Button>
              <Button
                variant="contained"
                classes={{
                  root: "buttonStyle",
                  disabled: "disabledButtonStyle",
                }}
                disabled={this.state.isSaveButtonDisabled}
```

```
                onClick={this.saveAndClose.bind(this,
this.state.currentContract, isEditing, notCloning)}
              >
                {isEditing ? "Save changes" : "Add"}
              </Button>
            </Modal.Footer>
          }
        </Modal>
      </>
    );
  }
}

const mapStateToProps = (state) => ({
  modalContent: state.modals.modalContent,
});

export default connect(mapStateToProps, { setLoading, addObject,
getObjects, editObject, deleteObject, setSubModalContent
})(ObjectsModal);
```