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

Кафедра інформатики факультету інформатики

Курсова робота

Система управління клієнтською базою як SaaS на прикладі компанії страхового брокера

Керівник курсової роботи
д.т.н., доц. Глибовець А. М.

_____
(підпис)

“____” _____ 2020 р.

Виконав
Василенко Андрій Миколайович

Київ 2020

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

ЗАТВЕРДЖУЮ
Зав. кафедри інформатики
к.ф-м.н., доц.
_____ С. С. Гороховський
(підпис)
“____” _____ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

Студенту: Василенко А. М. факультету інформатики 1 курсу МП
ТЕМА: Система управління клієнтською базою як SaaS на прикладі компанії страхового брокера

Вихідні дані:

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
Вступ
1. Дослідження та аналіз CRM-систем
2. Проектування системи
3. Розробка
Висновки
Список літератури

Дата видачі “____” _____ 2020 р. Керівник _____
(підпис)
Завдання отримав _____
(підпис)

Тема: Система управління клієнтською базою як SaaS на прикладі компанії страхового брокера

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу
1.	Отримання завдання на курсову роботу.	02.11.2019
2.	Огляд літератури за темою роботи.	15.11.2019
3.	Дослідження ринку готових рішень систем управління клієнтською базою.	25.11.2019
3.	Аналіз потреб та вимог до системи	05.02.2020
4.	Проектування системи на базі вимог	05.04.2020
6.	Розробка системи	15.04.2020
7.	Аналіз результатів та формування висновків	25.04.2020
8.	Коригування роботи згідно з коментарями наукового керівника	01.05.2020
9.	Оформлення слайдів	03.05.2020
10.	Захист курсової роботи	15.05.2020

Студент Василенко А. М.

Керівник Глибовець А. М.

“_____”_____ 2020

ЗМІСТ

Анотація

У даній курсовій роботі розглянуто приклад проектування та реалізації системи, яка створена для роботи з базою клієнтів. Реалізована система дає інструмент швидкого і зручного перегляду та управління базою клієнтів компанії, котра спеціалізується на брокерських послугах у сфері страхування.

Система спроектована та побудована з використанням MERN-стеку: база даних - нереляційна база **M**ongoDB, фреймворк для роботи з базою даних та створення API - **E**xpressJS, клієнтська частина - бібліотека **R**eact, серверна частина - платформа **N**ode.js. Також, в системі використовується багато інших допоміжних бібліотек та модулів.

Можливість комерційного використання застосування підтвердилось реальним прикладом такого використання. У зв’язку з цим було проаналізовано подальший напрямок розвитку застосування у його функціональній частині.

ВСТУП

Актуальність теми. Комерційні і некомерційні організації, котрі мають велику базу клієнтів потребують систему, котра б давала змогу ефективної роботи з такою базою: швидко переглядати перелік існуючих клієнтів, додавати нових клієнтів, редагувати та видаляти дані в системі та інше. Для компаній зі сфери страхування така потреба є надзвичайно актуальною, оскільки кількість клієнтів у них часто вимірюється сотнями та тисячами. Брокерська компанія у сфері страхування є одним із прикладів такої компанії. На відміну від європейської практики, брокерські компанії в сфері страхування в Україні почали розвиватися досить недавно. На сьогоднішній день досить складно знайти систему управління базою клієнтів, яка задовольняє всі потреби саме брокерської компанії. Фактично, такі компанії змушені або використовувати універсальні системи управління базою клієнтів, що часто означає відсутність специфічного функціоналу, або використовувати програми з офісного пакету (MS Excel, MS Access). Очевидно, обидва варіанти мають суттєві недоліки, основні з них - незручність користування, втрати часу, необхідність навичок використання (особливо для MS Access), багато ручної роботи та інше. Є приклад, коли брокерська компанія створювала систему управління для внутрішнього використання силами свого інформаційного відділу, але оскільки така система при проектуванні не була орієнтована на зовнішний ринок, а також була розроблена з урахуванням потреб саме однієї компанії, то вона так і не отримала комерційного розвитку.

Таким чином, стає зрозуміло, що саме брокерські компанії сьогодні гостро відчувають потребу в системі, котра б давала змогу працювати з клієнтами і враховувала б усі особливості та потреби страхового бізнесу, мала би усі переваги сучасних систем і допомагала б генерувати більшу прибутковість.

Мета дослідження: Метою даної курсової роботи є дослідження та реалізація системи управління клієнтською базою за моделлю Software-as-a-Service (SaaS), а також створення висновків та перспектив розвитку такої системи. Модель SaaS дозволяє будь-якій компанії почати користуватись готовою системою швидко та без необхідності великих інвестицій в розробку.

Для досягнення мети заплановані наступні кроки:

1. Дослідження інформаційних джерел
2. Аналіз вимог до системи
3. Вибір технологій
4. Проектування системи
5. Реалізація системи
6. Аналіз результатів роботи
7. Прогнозування подальшого вектору розвитку

Об’єкт дослідження: створення системи управління базою клієнтів для брокерських компаній

Предмет дослідження: практичне використання хмарних технологій для системи управління клієнтською базою.

РОЗДІЛ 1

1.1. CRM-системи

Для розуміння того, які саме функції має виконувати будь-яка CRM-система, треба спочатку розібратись з поняттям “Управління відносинами з клієнтами”, або Customer Relationship Management (CRM). Точне визначення цього терміну можна знайти на Вікіпедії, а саме:

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]

Отже, виходячи з вищезазначеного тлумачення, приходимо до висновку, що будь-яка CRM-система повинна мати дві основні функції:

1. збір та зберігання інформації;
2. аналітика зібраної інформації.

Така система буде складатися з окремих компонентів, котрі забезпечують виконання цих функцій:

1. Інтерфейс користувача, котрий забезпечує взаємодію користувачів безпосередньо з системою.
2. Серверна (або операційна) частина - частина, де відбувається запит, отримання, обробка даних, а також знаходяться допоміжні модулі,

котрі забезпечують повне, стабільне та безпечне функціонування системи. Серверна частина взаємодіє з клієнтським інтерфейсом шляхом отримання даних, введених користувачем в формах та шляхом передачі даних на форми та поля інтерфейсу для відображення користувачу. Також, серверна частина взаємодіє з базою даних шляхом запитів на отримання чи редагування інформації в базі(ах) даних.

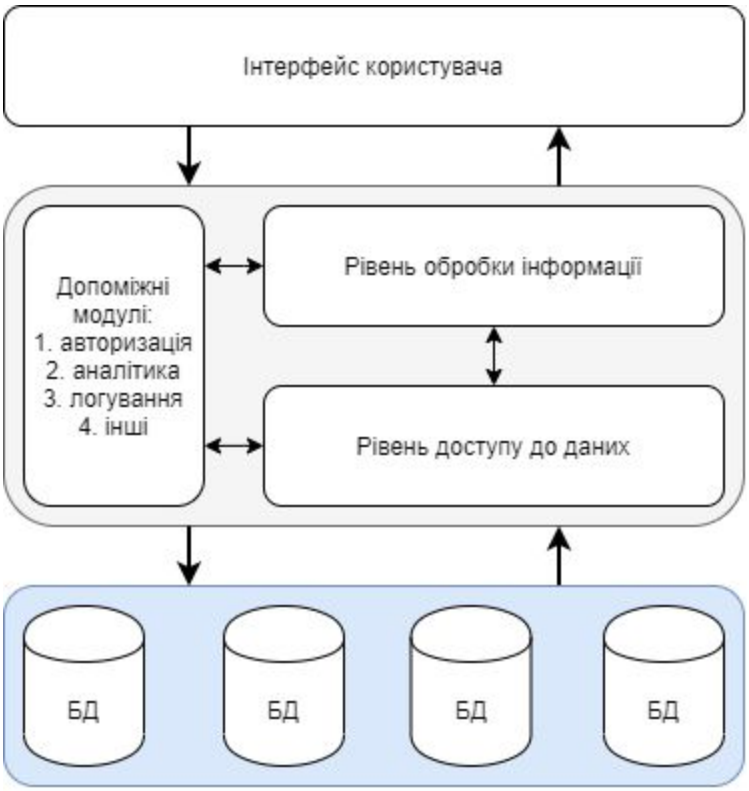


Рисунок 1. Компоненти CPM-системи

3. База даних - на цьому рівні може бути одна база даних або декілька, в залежності від потреб, масштабів та типу системи.

Сучасні CRM-системи поділяються на три основні напрями:

1. **Операційні**: більшість існуючих систем відноситься саме до цього типу. Такі системи значно спрощують взаємодію з клієнтами компанії, вони вміють виставляти рахунки, а також нагадують про необхідність контакту з клієнтом, можуть самі

відправляти sms-повідомлення, робити записи телефонних дзвінків та інше.

Операційні CRM-системи вміють:

1. Вести реєстр трафіку (дзвінки, листи, заявки на сайті);
2. зберігати в базі даних інформацію про клієнтів, заявках, угодах, завданнях тощо;
3. автоматизовувати документообіг компанії;
4. відображати поточний статус угод;
5. нагадувати про заплановані події (дзвінкі, листи, зустрічі);

2. **Аналітичні**: на відміну від операційних систем, аналітичні не тільки фіксують історію взаємодії з клієнтом, але і допомагають простежити закономірності в продажах: клієнти з яких джерел купують найчастіше, на якому етапі зривається більшість угод, як розподілені клієнти по воронці продажів - всі ці дані оновлюються в онлайн-режимі, в розрізі кожного параметра.

Мета аналітичних CRM - аналіз інформації по клієнтах та результатах продажів з метою створення більш ефективної стратегії роботи.

Аналітичні CRM вміють:

1. сегментувати клієнтську базу;
2. визначати цінність клієнта;
3. аналізувати рентабельність клієнтів;
4. моніторити поведінку клієнтів на кожному етапі операції;

5. аналізувати динаміку продажів;
6. аналізувати ефективність маркетингових інструментів;
7. спрогнозувати обсяг продажів.

3. **Спільні (колаборацiйні)**: такі CRM системи налагоджують комунікації з клієнтами для збору зворотного зв'язку. Інформація, отримана з їх допомогою, допомагає скорегувати асортимент товарів, цінову політику, а також процес обслуговування покупців. Наприклад, співробітники call-центру автосалону обдзвонюють клієнтів, задаючи питання про якість сервісу і фіксуючи відповіді. За результатами опитування закуповуються відсутні комплектуючі та додаються нові сервісні послуги.

   Як правило, такі CRM-систем розробляються як індивідуальні рішення або використовуються існуючі канали зв'язку, і інформація з них фіксується в основний CRM-системі (інтернет-форуми, соцмережі, телефонія, e-mail-листування).

Слід зауважити, що останнім часом набуває популярності комбіновані CRM-системи, тобто такі, котрі поєднують в собі елементи різних типів CRM-систем.

Згідно з даними міжнародної консалтингової компанії, близько 40% компаній впроваджують CRM-систему впродовж перших 2 років від свого заснування, а впродовж перших 5 років існування вже 65% компаній будуть мати CRM-систему. [2]

Серед найпоширеніших CRM-систем сьогодні є: bpm’online, Salesforce, SAP, Microsoft Dynamics, Zoho CRM, SugarCRM, Oracle CRM.

1.2. Особливості вимог до CRM-системи у страховому бізнесі

CRM-системи для страхових компаній повинні мати специфічний інтерфейс, котрий задовольняв би вимоги таких компаній. Специфіка інтерфейсу полягає, в основному, в певному наборі розділів системи, а також в наборі непритаманних для інших сфер полів у формах системи. Розглянемо нижче більш детально вимоги до CRM-системи з точку зору компаній зі сфери страхування.

Почнемо з розгляду елементів CRM-системи, котрі є необхідними, але водночас є спільними для всіх компаній, а не тільки для страховий компаній. Такими елементами є:

1. Профіль клієнта - розділ з даними клієнтів, де створюються, редагуються та видаляються особисті дані клієнта, а також відображаються посилання на договори чи інші документи клієнта в системі.
2. Форма договору (або контракту) - розділ для створення, редагування та видалення договорів клієнтів. Форма договорів має підтримувати можливість створення контракту як з фізичною так і з юридичною особою, а також містити обов’язкові атрибути договору, такі як “номер договору”, “дата початку” та “дата закінчення” договору, адреса та інші дані по клієнту, банківські реквізити та інше.
3. Розділ управління користувачами системи - в даному розділі створюються та видаляються користувачі, редагуються дані користувачів, змінюються права доступу користувачів.

4. Допоміжні елементи - набір елементів, котрі не є абсолютно необхідними для нормальної роботи з системою, а радше несуть допоміжну функцію. Прикладом таких елементів є поточний курс обміну валют, поточна дата, ім’я поточного користувача, мова.

Що стосується елементів CRM-системи, притаманних для компаній зі сфери страхування, то тут треба брати до уваги наступну специфіку:

1. Особливості меню - оскільки послуга страхування надається по відношенню до якогось об’єкта, то логічно мати окремий розділ для об’єктів страхування, в якому буде відображено перелік усіх застрахованих об’єктів. Додатково необхідно мати окремий розділ для відображення врегулювань по страховим подіям.
2. Специфічна термінологія - форми договорів повинні містити поля для обов’язкових атрибутів саме страхових договорів, наприклад “франшиза” (сума, яка не відшкодовується страхувальнику у разі настання страхової події), “страхова премія” (розмір платежу за послугу страхування), “бонус-малус” (коефіцієнт для врахування при калькуляції страхової премії, залежить від історії страхових випадків), “страхувальник та застрахований” (часто буває що той, хто оплачує послугу страхування не є тим, хто є застрахований).
3. Договори нефіксованого строку - система має підтримувати договори на декілька днів (туристичне страхування), квартал, півроку або рік чи більше.

4. Договір може включати багато об’єктів - один договір з одним клієнтом може мати безліч об’єктів страхування, як наприклад страхування автомобілів компанії таксі або страхування співробітників великої корпорації.
5. Необхідність роботи з фотографіями - оскільки страхові події завжди супроводжуються фотографіями місця пригоди, а також фотографіями отриманих збитків, система має містити функціонал по додаванню та видаленню фотографій по окремим страховим подіям. Також, розділ для роботи з фото має бути присутнім в розділі договорів і, бажано, в усіх інших розділах.

До специфіки CRM-системи для компаній зі сфери страхування також можна віднести тісний взаємозв’язок логічних елементів: договору, клієнта, об’єкта, врегулювання.

Договір завжди буде містити одного клієнта, і мати один або більше об’єктів страхування, та мати один або більше врегулювань.

Клієнт буде мати нуль або більше договорів, а також нуль або більше об’єктів та врегулювань.

Об’єкт буде прив’язаний завжди до одного клієнта (але клієнт з часом може змінитись), може бути прив’язаний до нуля або більше договорів страхування і мати нуль або більше врегулювань.

Врегулювання завжди будуть прив’язані до одного договору, одного об’єкту та одного клієнту.

Схематично це показано на рисунку 2 нижче.

	Договір	Клієнт	Об'єкт	Врегулювання
Договір		1	0+	0+
Клієнт	0+		0+	0+
Об'єкт	0+	1		0+
Врегулювання	1	1	1	

Рисунок 2. Матриця кількості можливих верхніх елементів для елементів зліва

Варто також зауважити, що часто доступ до CRM-системи надається представнику компанії клієнта. Це означає, що окремі поля мають бути недоступні для перегляду чи взаємодії для такого представника, оскільки можуть містити конфіденційну інформацію. Це також стосується і окремого функціоналу, наприклад додавання нових користувачів до системи або видалення контенту з системи.

1.3. SaaS як модель для розробки CRM-системи

Для початку, розглянемо що стоїть за терміном SaaS. Вікіпедія дає наступне визначення:

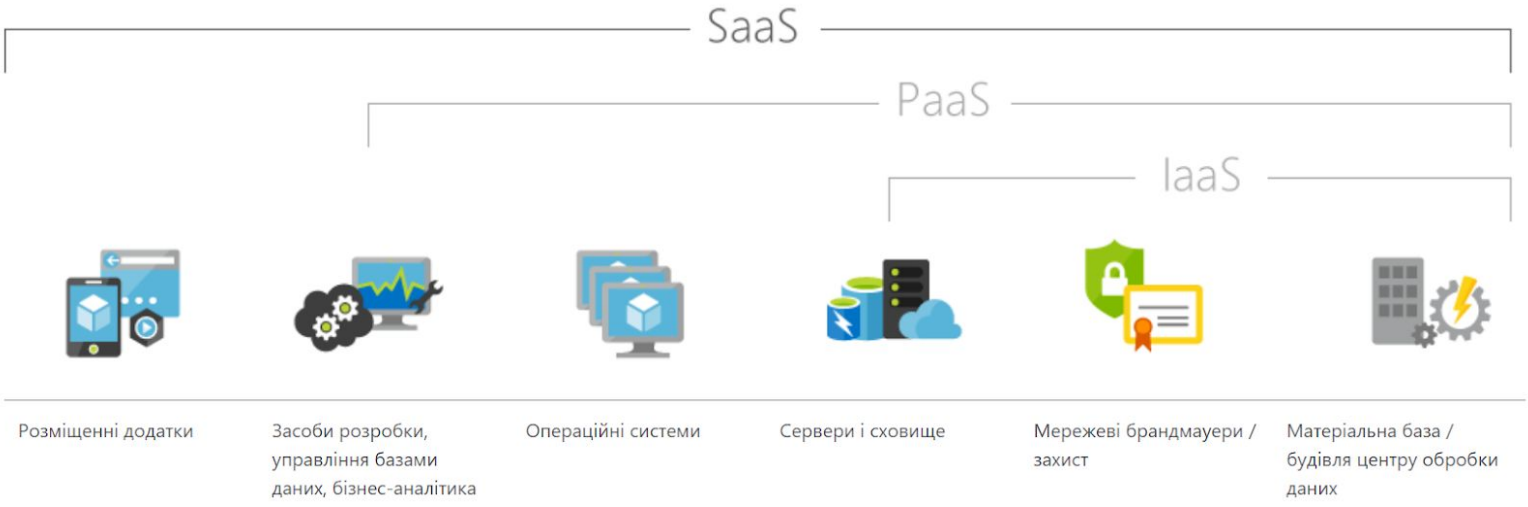
SaaS (англ. Software-as-a-Service - програмне забезпечення як послуга) - модель поширення програм споживачам, при якій постачальник розробляє веб-програму, розміщує її й керує нею (самостійно або через третіх осіб) з метою використання її замовниками через інтернет. Замовники платять не за володіння програмами як такими, а за їх використання (через API, що доступне через веб і яке часто використовують веб-служби). [4]

За моделлю SaaS, користувач програмного сервісу отримує повноцінний, повністю функціональний набір програмного забезпечення, який оплачується в міру його використання. Необхідна інфраструктура, допоміжні програми і дані додатків знаходяться в центрі обробки даних постачальника. Постачальник сервісу керує обладнанням та програмним забезпеченням на основі угоди про обслуговування, бере на себе відповідальність стосовно доступності та безпеки даних.

Така модель швидко набула широкої популярності через низку факторів. З точки зору бізнесу, безумовними перевагами є швидкість отримання продукту, а також відсутність необхідності мати персонал на підтримку та усунення помилок програми. До того ж, компанія оплачує тільки задіяні ресурси системи, а її співробітники та клієнти отримують доступ до неї фактично з будь-якого місця, де є доступ до всесвітньої мережі.

Користувачі майже не обмежені в пристроях, через які можна користуватись системою, а в разі поломки одного пристрою, можна використати інший.

Модель SaaS є однією з трьох і найширшою з точки зору повноти надання сервісу моделей хмарних обрахунків. Дві інші моделі, які є складовою частиною SaaS, є ПааС (з англ. Platform-as-a-Service - платформа як сервіс) та IaaC (з англ. Infrastructure-as-a-Service - інфраструктура як сервіс).



Однак модель SaaS не позбавлена недоліків. Як зазначає Директор Ради директорів компанії Infor Чарльз Філіпс, основними недоліками є наступні:

1. Ризики безпеки. При переході від використання локального серверу до хмарних обчислень виникає необхідність обміну важливими даними по мережі. Це генерує додаткові ризики.
2. Залежність від швидкості передачі даних інтернету. Оскільки взаємодія з програмним забезпеченням відбувається через веб-браузер по

інтернету, то швидкість інтернету безпосередньо впливає на зручність та ефективність користування програмою.

3. Значно менший контроль за системою. Компанія повністю залежить від провайдера послуг.

РОЗДІЛ 2. Проектування системи

2.1 Аналіз вимог

CRM-система для брокерської компанії у сфері страхування має, з однієї сторони, бути повноцінним інструментом роботи з базою даних такої компанії, а з другої сторони, надавати доступ для клієнтів компанії в режимі перегляду обмежених даних. Іншими словами, така система потребує реалізації набору обмежень різних типів користувачів. Також система повинна мати набір допоміжного функціоналу, котрий би полегшив процес використання системи.

Вимоги до системи можна поділити на наступні категорії:

1. Ролі та права користувачів
2. Розділ управління користувачами
3. Вимоги відображення даних в таблицях
4. Вимоги управління записами в таблицях
5. Розділ короткої аналітики
6. Додатковий функціонал:

Розглянемо детально кожний з розділів:

1. Ролі та права користувачів: користувачі компанії повинні поділятись на наступні ролі з відповідними правами:

Функціонал / Роль	Адміністратор	Співробітник
Внесення нових даних	Так	Так

(договори, клієнти, об’єкти, врегулювання)		
Видалення існуючих даних (договори, клієнти, об’єкти, врегулювання)	Всі	Тільки ті, що були внесені співробітником
Редагування існуючих даних (договори, клієнти, об’єкти, врегулювання)	Всі	Тільки ті, що були внесені співробітником
Створення нових користувачів	Так	Ні
Зміна прав нових користувачів	Так	Ні
Перегляд власних даних (договори, клієнти, об’єкти, врегулювання)	Так	Так
Перегляд всіх даних (договори, клієнти, об’єкти, врегулювання)	Так	Ні
Перегляд результатів роботи всіх співробітників	Так	Так

Серед користувачів системи будуть також клієнти компанії. Такі користувачі повинні мати виключно права перегляду даних по своїм об’єктам та врегулюванням. Весь функціонал по редагуванню та видалення даних має бути недоступним.

2. Розділ управління користувачами: система повинна мати форму для додавання нових користувачів, в якій повинна вноситись наступна інформація: логін, електронна адреса користувача, номер мобільного телефону для зв’язку, юридична назва (для

юридичним осіб), прізвище, ім’я, ідентифікатор компанії (використовуються для надання доступу клієнтам компанії), пароль та роль (клієнт, співробітник, адміністратор).

3. Вимоги відображення даних в таблицях: кожний запис в таблицях з пунктів меню “Договори”, “Клієнти”, “Об’єкти”, “Врегулювання” повинний містити зв’язану з цим записом інформацію. Тобто, користувач, знаходячись в меню “Договори” та переглядаючи дані по кожному з договорів, повинен бачити інформацію про:
    a. клієнта, котрий пов’язаний з цим договором
    b. об’ктам, які внесені в цей договір
    c. врегулюванням, які виникли з об’єктами, що внесені в договір
4. Вимоги управління записами в таблицях: кожний запис в таблицях повинен мати функціонал клонування, редагування та видалення запису. Пункти меню “Договори”, “Клієнти”, “Об’єкти”, “Врегулювання” повинні містити функціонал додавання фотографій, відображення доданих файлів, видалення файлів та скачування таких файлів.
5. Розділ короткої аналітики: користувачі (окрім користувачів-клієнтів) повинні бачити в окремому пункті меню наступну інформацію:
    a. Перелік договорів, срок закінчення котрих наступить через 2 тижні або раніше.

b. Щомісячні результати продажів співробітника (сума страхових комісій за кожен місяць поточного року)
c. Загальна сума продажів всіх співробітників з початку року (сума страхових комісій кожного співробітника з початку року).

6. Додатковий функціонал:
    a. поточний курс валют для USD та EUR
    b. вибір мови відображення: UA чи EN
    c. поточна дата
    d. ім’я користувача

Варто також зауважити, що оскільки CRM-система покликана максимально автоматизувати мануальну роботу, тому до вимог системи також було віднесено автоматичну калькуляцію розрахунків страхової премії на основі внесених даних у поля страхової суми та тарифу (за формулою Премія = Страхова Сума * Тариф), автоматичне розбиття премії на рівні частини в залежності від вказаної періодичності, а також калькуляція комісії.

2.2. Технології для реалізації

Мовою програмування було обрано JavaScript. На сьогоднішній день ця мова є однією з найпопулярніших та щороку завойовує все більше прихильників завдяки своєму динамічному розвитку.

За результатами опитувань від ресурсу Stackoverflow, у 2019 році JavaScript зайняв перше місце серед технологій за популярністю у розробників [5]

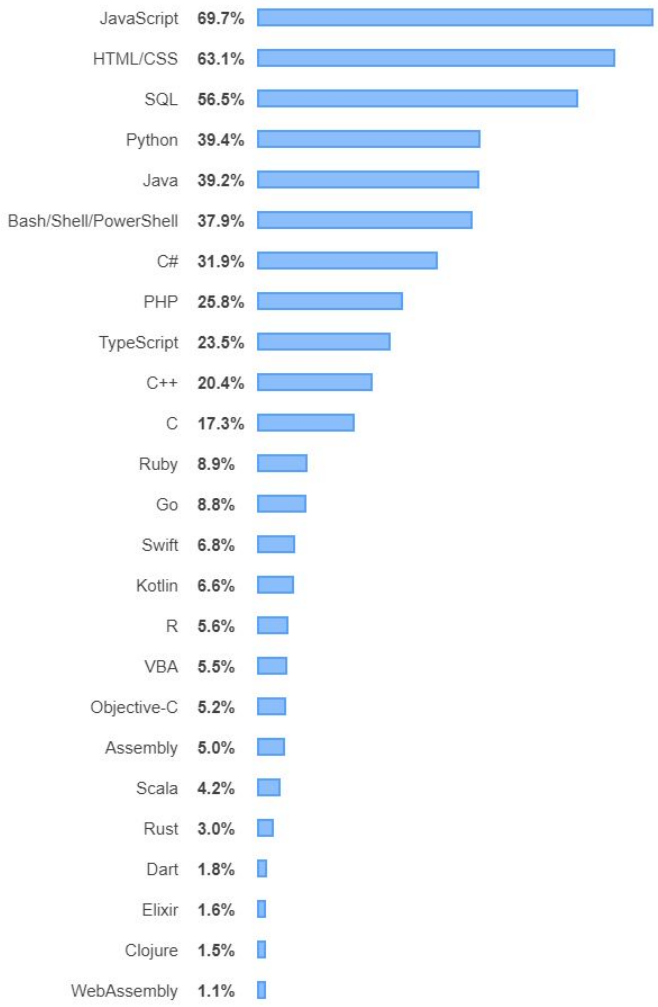


Рисунок 2.1. Найпопулярніші технології: мови програмування, скриптові мови та мови для розмітки.

Дійсно, від дати свого створення, мова JavaScript еволюціонувала від інструменту для анімації сайтів та веб-застосунків до повноцінної мови програмування, яка використовуються як на клієнтському так і на серверному рівнях.

Для вибору технології програмування веб-інтерфейсу знову звернемося до опитувальника ресурсу Stackoverflow[6]. Згідно з останніми даними, найбільш улюбленим інструментом для створення веб-інтерфейсів серед розробників з усього світу є React:

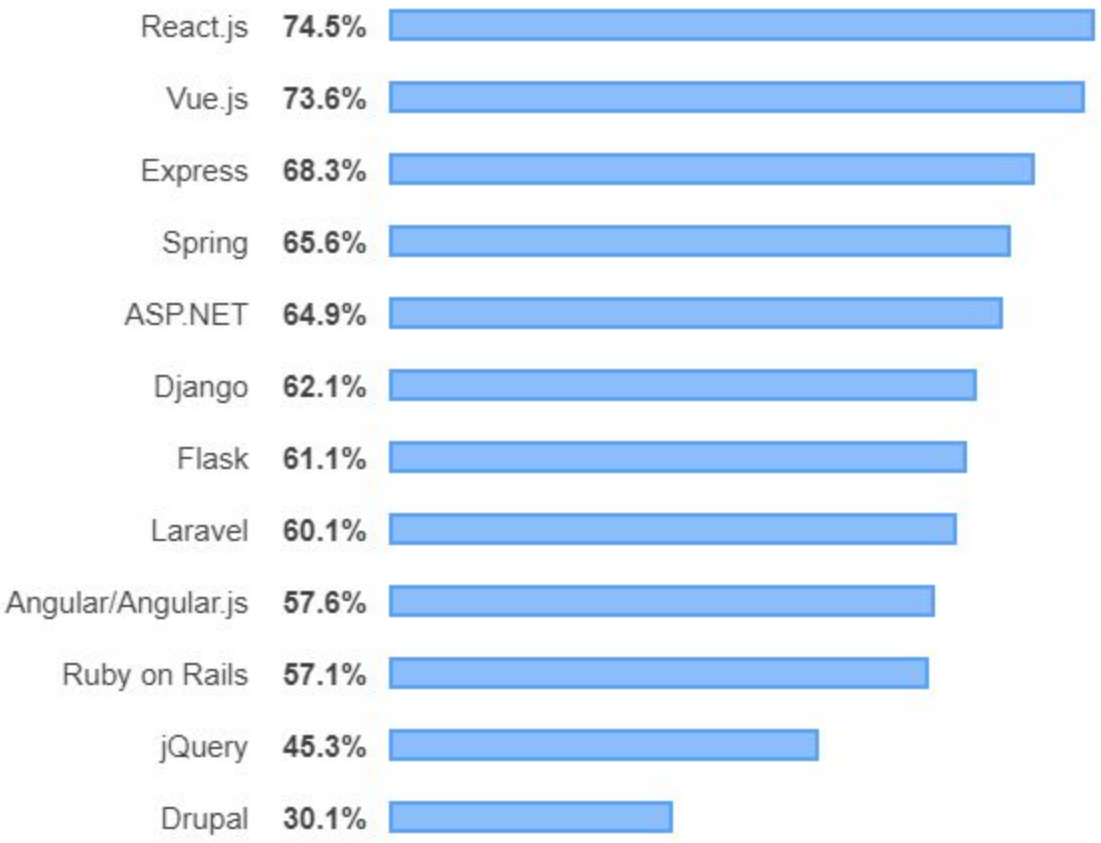


Рисунок 2.2. Найпопулярніші веб фреймворки за 2019р.

Підтвердженням популярності React є кількість скачувань бібліотеки у **Node Package Manager** у порівнянні з іншими популярними фреймворками:

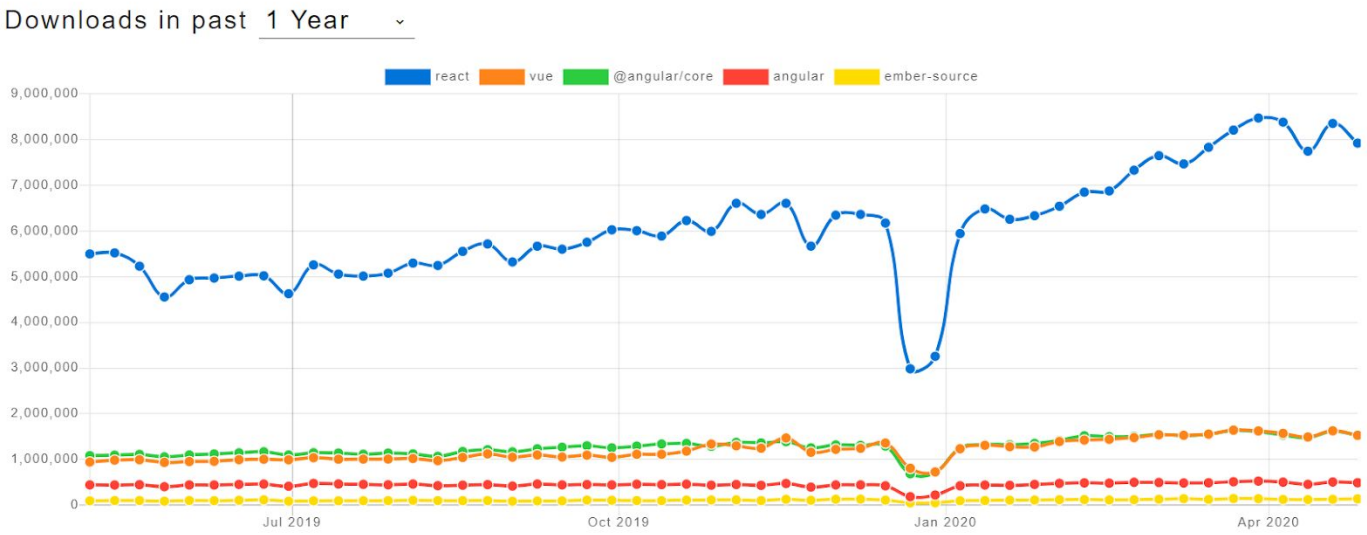


Рисунок 2.3. Кількість скачувать пакетів за рік у NPM: React, Vue, Angular, Ember

React - це бібліотека, котра була розроблена спеціалістами з Фейсбук у 2013 році. В основному використовується на веб-сайтах з високим рівнем трафіку. Вона була розроблена, коли рекламні оголошення Facebook почали набирати трафік і компанія зіткнулася з проблемами в їх кодуванні та обслуговуванні. Вони були вирішені з виходом цієї бібліотеки JavaScript. **Whatsapp, Instagram Paypal, Glassdoor, BBC** - одні з популярних компаній, що використовують React. Він надзвичайно динамічний і пропонує велику підтримку у створенні інтерактивних інтерфейсів користувача. Остання версія React, 16.8.6, була випущена 6 травня 2019 року.

Оскільки веб інтерфейс буде розроблятись за допомогою бібліотеки React на мові JavaScript, то для серверної частини було обрана платформа Node.js. Єдина мова програмування клієнтської та серверної частини значно

полегшує процес. До того ж, Node.js також має високі рейтинги серед розробників і займає перше місце з бібліотек, що не стосуються веб-інтерфейсів:

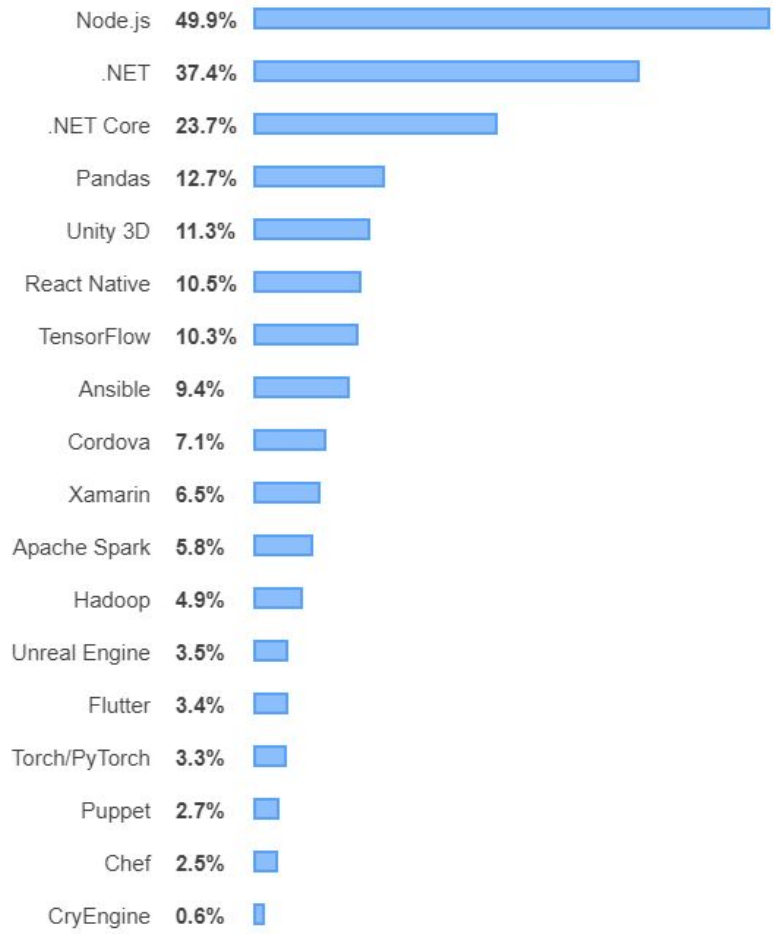


Рисунок 2.4. Найпопулярніші фреймворки, бібліотеки та інструменти, що не стосуються веб-інтерфейсів у 2019 р.

У парі з Node.js використовується фреймворк Express, котрий дозволяє легко та швидко створювати API для веб застосунків.

Для зберігання та обробки даних була обрана база даних MongoDB. Основними критеріями вибору слугували: ціна використання, швидкість

роботи, можливість ефективного збереження великих файлів (фото). MongoDB якнайкраще відповідає заданим вимогам. Ниже порівняння MongoDB та MySQL як двох популярних представників NoSQL та SQL баз даних.

	
MongoDB - це база даних з відкритим кодом, розроблена компанією MongoDB, Inc. MongoDB зберігає дані в JSON-подібних документах, які можуть відрізнятися за структурою. Це популярна база даних NoSQL.	MySQL - популярна система управління реляційними базами даних з відкритим кодом (RDBMS), яка розробляється, розповсюджується та підтримується корпорацією Oracle.
У MongoDB кожен окремий запис зберігається як "документ" .	У MySQL кожен окремий запис зберігається у вигляді рядків у таблиці.
MongoDB називається базою даних NoSQL. Це означає, що заздалегідь визначена структура вхідних даних може бути визначена та дотримана, але, якщо потрібно, різні документи в колекції можуть мати різні структури. Він має динамічну схему.	MySQL, як випливає з назви, використовує структуровану мову запитів (SQL) для доступу до бази даних. Схему неможливо змінити. Тільки вводяться входи, що відповідають зазначеній схемі.
MongoDB був розроблений з урахуванням високої доступності та масштабованості. Він включає нестандартне тиражування та шардування .	Концепція MySQL не допускає ефективної реплікації та шардування, але в MySQL можна отримати доступ до пов'язаних даних за допомогою приєднання, що мінімізує дублювання.
Динамічна схема	Фіксована схема

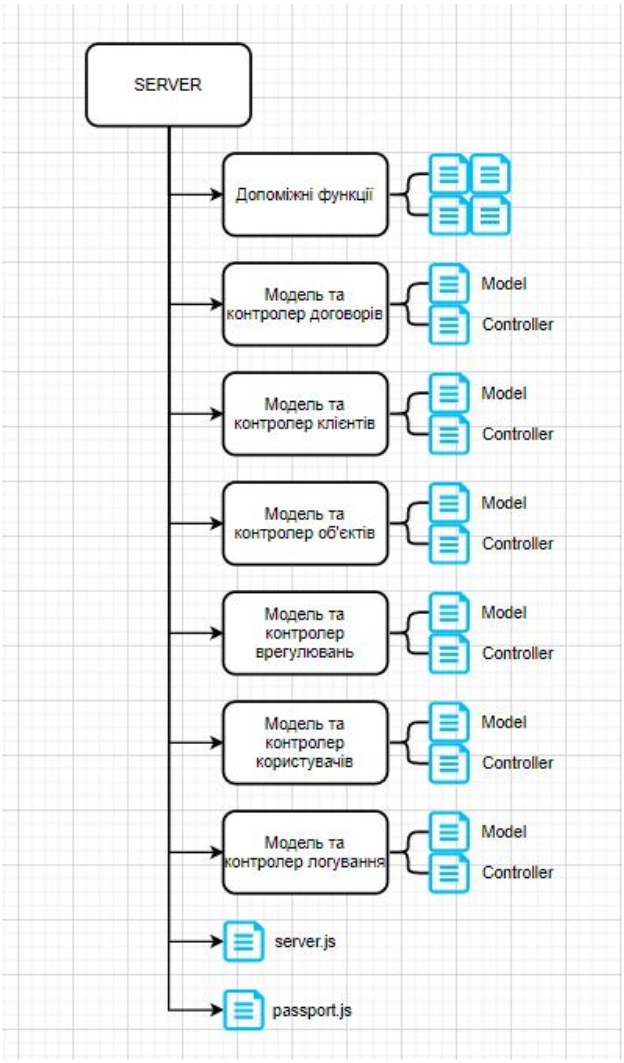
Легка у використанні для розробників	Важка для розробників. Потребує специфічні знання (для адміністраторів БД)

РОЗДІЛ 3

3.1 Розробка серверної частини

Для розробки серверної частини створимо структуру папок, де головна папка буде мати назву “server” та містити в собі підпапки для опису всіх схем (“договорів”, “об’єктів”, “клієнтів” та інші), а також для допоміжних функцій.

Візуально структура папок буде виглядати наступним чином:



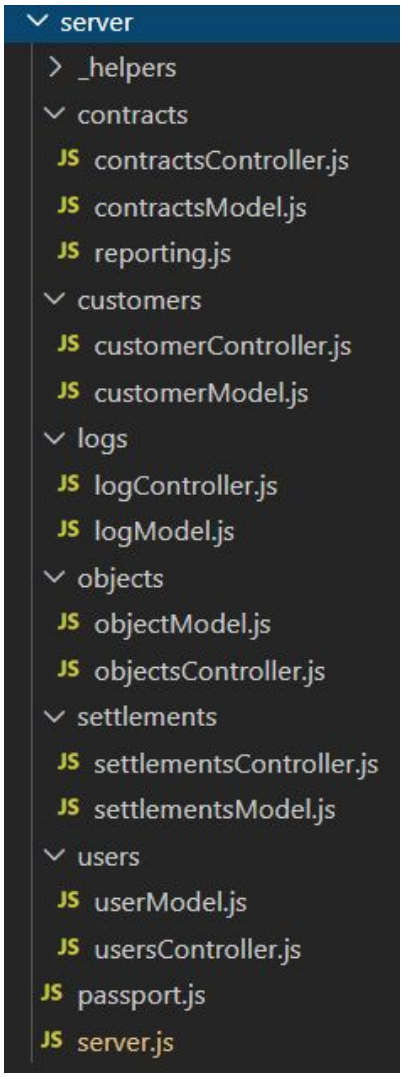


Рисунок 3.1. Схематична та реалізована структура папок та файлів

Для початку, треба приєднати базу даних та запустити сервер на обраному порті (нехай буде 5000). Для цього імпортуємо необхідні залежності в файл server.js та пропишемо налаштування:

Рисунок 3.2. Приклад коду файла server.js

server > JS server.js > ...

```
1   const express = require('express');
2   const cors = require('cors');
3   const bodyParser = require('body-parser');
4   const app = express();
5   const mongoose = require('mongoose');
6   require('dotenv').config()
7
8   mongoose.Promise = global.Promise;
9   mongoose.connect(process.env.MONGODB_URI,
10      {
11          useCreateIndex: true, useUnifiedTopology: true, useNewUrlParser: true
12      }).then(
13      () => { console.warn('DB connected') },
14      err => { console.log('Error in DB connection', err) }
15  );
16
17  app.use(cors());
18
19  app.use(bodyParser.urlencoded({ extended: false }));
20  app.use(bodyParser.json());
21
22  // start server
23  const PORT = process.env.NODE_ENV === 'production' ? (process.env.PORT || 80) : 5000;
24  app.listen(PORT, () => {
25      console.log(`Server listening on port ${PORT}`);
26  });
```

Перевіримо чи все зробили правильно та запустимо сервер:

Рисунок 3.3. Запуск серверної частини застосунку через консоль

Все працює, сервер слухає порт 5000, база даних приєднана.

Тут треба зауважити, що попередньо до цих дій був створений обліковий запис на сайті mongodb та обраний безкоштовний варіант розміщення бази у хмарі. Також був створений файл .env, котрий містить посилання на базу даних, разом з іншими параметрами. Імпорт цих даних з файла можна побачити на скріншоті вище - **require('dotenv').config()**

Опис схем розглянемо на прикладі схеми для клієнтів:

Рисунок 3.4. Приклад опису схеми клієнту

server > customers > JS customerModel.js > ...

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const schema = new Schema({
  name: { type: String, default: "" },
  type: { type: String, default: "" },
  address: { type: String, default: "" },
  phone: { type: String, default: "" },
  email: { type: String, default: "" },
  dateOfBirth: { type: Date },
  inn: { type: Number, default: 0 },
  documentType: { type: String },
  documentNumber: { type: String },
  documentIssuer: { type: String },
  documentIssueDate: { type: Number },
  documentExpiryDate: { type: Number },
  gender: { type: String },
  comments: { type: String, default: "" },
  citizenship: { type: String, default: "" },
  bankName: { type: String, default: "" },
  edrpou: { type: Number, default: "" },
  accountNumber: { type: String, default: "" },
  mfo: { type: Number },
  director: { type: String, default: "" },
  manager: { type: String, default: "" },
  customerPhotos: { type: Array, default: [] },
  objects: { type: Array, default: [] },
  contracts: { type: Array, default: [] },
  settlements: { type: Array, default: [] },
  createdDate: { type: Number, default: Date.now() },
});
schema.set('toJSON', { virtuals: true });
module.exports = mongoose.model('Customers', schema);
```

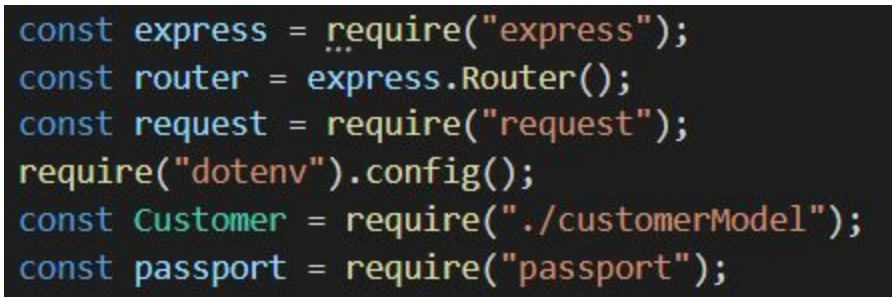
Як бачимо, схема описує всі ймовірні поля, котрі необхідно зберігати для клієнтів як фізичних так і юридичних осіб. Саме за такою схемою дані будуть збережені в MongoDB в документі під назвою Customers.

Тепер створимо контролер для клієнтів. Основні функції, котрі має виконувати контролер:

1. Додавання нового клієнта
2. Отримання всіх клієнтів
3. Редагування існуючих клієнтів
4. Видалення клієнта

Всі ці операції будуть відбуватись через виклики REST API. Створимо файл customerController.js та імпортуємо в нього необхідні залежності:

Рисунок 3.5. Приклад імпорту залежностей в модулі Node.js



Для реалізації функціоналу додавання нових клієнтів створимо функцію за посиланням **addcustomer**:

Рисунок 3.6. Приклад реалізації API для додавання нового клієнта

```
router.post("/addcustomer", passport.authenticate("jwt", { session: false }), (req, res) => {
  Customer.findOne({ inn: req.body.inn, dateOfBirth: req.body.dateOfBirth })
    .then((customer) => {
      if (customer) {
        console.warn("tried to add customer but found existing one", customer);
        return res.json(customer);
      } else {
        var newCustomer = new Customer({ ...req.body });
        newCustomer.save().then((cust) => {
          return res.json(cust);
        });
      }
    })
    .catch((err) => res.status(500).send("Whooops, customer add operation failed"));
});
```

Як випливає з коду вище, алгоритм додавання клієнта наступний:

1. Спочатку перевіримо чи вже не існує такий клієнт в базі. Для цього, спробуємо знайти серед всіх записів клієнта, з таким самим ІНН та датою народження (одного ІНН буде недостатньо, оскільки є ймовірність що клієнт відмовився від отримання ІНН). Якщо такий клієнт був знайдений, то повернемо його дані та додамо попередження в консоль.
2. Якщо клієнтів з таким ІНН та датою народження не існує, то візьмемо дані, котрі прийшли та створимо нового клієнта в базі і одразу повернемо його дані.
3. Додамо функцію обробки помилок на випадок, якщо щось піде не так.

Створимо функціонал отримання всіх клієнтів.

Рисунок 3.7 Приклад реалізації API для отримання з бази даних переліку клієнтів

```
router.get("/getcustomers", passport.authenticate("jwt", { session: false }), (req, res) => {
  Customer.find()
    .then((customers) => {
      if (!customers) {
        return res.status(401).json("Not authorised!");
      }
      if (req.user.role === "admin") {
        return res.json(customers);
      }
      if (req.user.role === "employee") {
        let filteredCustomers = customers.filter((cust) => cust.manager === req.user.lastName + " " + req.user.firstName);
        return res.json(filteredCustomers);
      }
      if (req.user.role === "client") {
        let filteredCustomers = customers.filter((cust) => cust.id === req.user.customerId);
        return res.json(filteredCustomers);
      }
    })
    .catch((err) => res.status(500).send("Whooops, customers get operation failed"));
});
```

Як бачимо, в цій функції ми додатково будемо фільтрувати увесь перелік клієнтів в залежності від того, хто саме намагається отримати дані. Користувач з роллю “Адміністратор” отримає весь перелік, “Співробітник” тільки тих клієнтів, котрі створив сам, а “Клієнт” тільки себе.

Редагування та видалення клієнтів відбувається трошки простіше, а саме у два етапи - знайти запис за унікальним ідентифікатором та внести зміни чи видалити:

Рисунок 3.8 Приклад реалізації АПІ для видалення клієнта

```
router.delete("/deleteCustomer", passport.authenticate("jwt", { session: false }), (req, res) => {
  Customer.findByIdAndDelete(req.query.id, function (err, ok) {
    if (err) return res.send(err);
    res.json({ message: "Customer deleted" });
  });
});
```

Рисунок 3.9 Приклад реалізації АПІ для редагування клієнта

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

За такою ж аналогією створимо всі інші необхідні схеми та контролери.

Додатково, створюємо папку _helpers, в яку будемо розміщувати функціонал, котрий буде допоміжним. Для початку додамо туди функцію запиту курсу валют. Для цього будемо використовувати сервіс Національного Банку України, котрий дозволяє отримувати актуальний курс валют на будь-яку дату.

Рисунок 3.10 Приклад реалізації АПІ для отримання курсу валют від НБУ

server > _helpers > JS exRate.js > ...

```
1   const express = require("express");
2   const router = express.Router();
3   const request = require("request");
4   require("dotenv").config();
5
6   router.get("/getexrate", (req, res) => {
7     request({
8       uri: "https://bank.gov.ua/NBUStatService/v1/statdirectory/exchangenew?json",
9     })
10      .pipe(res)
11      .on("error", (err) => {
12        const msg = "Error on connecting to the webservice.";
13        console.error(msg, err);
14        res.status(500).send(msg);
15      });
16  });
17
18  module.exports = router;
19
```

Далі, додамо модуль passport.js, який буде відповідати за аутентифікацію користувачів та передавати створений jwt токен.

Рисунок 3.11 Приклад використання бібліотеки для авторизації

```
server > JS passport.js > <unknown> > module.exports
1   const JWTStrategy = require('passport-jwt').Strategy;
2   const ExtractJWT = require('passport-jwt').ExtractJwt;
3   const mongoose = require('mongoose');
4   const User = mongoose.model('User');
5   require('dotenv').config();
6
7   const opts = {};
8
9   opts.jwtFromRequest = ExtractJWT.fromAuthHeaderAsBearerToken();
10  opts.secretOrKey = process.env.SECRET;
11
12  module.exports = passport => {
13      passport.use(new JWTStrategy(opts, (jwt_payload, done) => {
14          User.findById(jwt_payload.id)
15              .then(user => {
16                  if(user) {
17                      return done(null, user);
18                  }
19                  return done(null, false);
20              })
21              .catch(err => console.error(err));
22      }));
23  }
```

Імпортуємо в файл server.js всі контролери. Тепер частина файлу імпортів вигладає ось так:

Рисунок 3.12 Імпорт всіх необхідних залежностей в файл server.js

```
JS server.js > ...
const express = require('express');
const cors = require('cors');
const bodyParser = require('body-parser');
const path = require("path");
const passport = require('passport');
const userRoutes = require('./users/usersController');
const contractRoutes = require('./contracts/contractsController');
const customerRoutes = require('./customers/customerController');
const objectsRoutes = require('./objects/objectsController');
const settlementRoutes = require('./settlements/settlementsController');
const helperRoutes = require('./_helpers/fileUploader');
const LogsRoutes = require('./logs/logController');
const app = express();
const mongoose = require('mongoose');
require('dotenv').config()
```

Додаємо в файл шляхи АПІ:

Рисунок 3.13 Створення всіх шляхів АПІ в файлі server.js

```
// api routes
app.use('/api', userRoutes);
app.use('/api', contractRoutes);
app.use('/api', customerRoutes);
app.use('/api', objectsRoutes);
app.use('/api', settlementRoutes);
app.use('/api', helperRoutes);
app.use('/api', LogsRoutes);
```

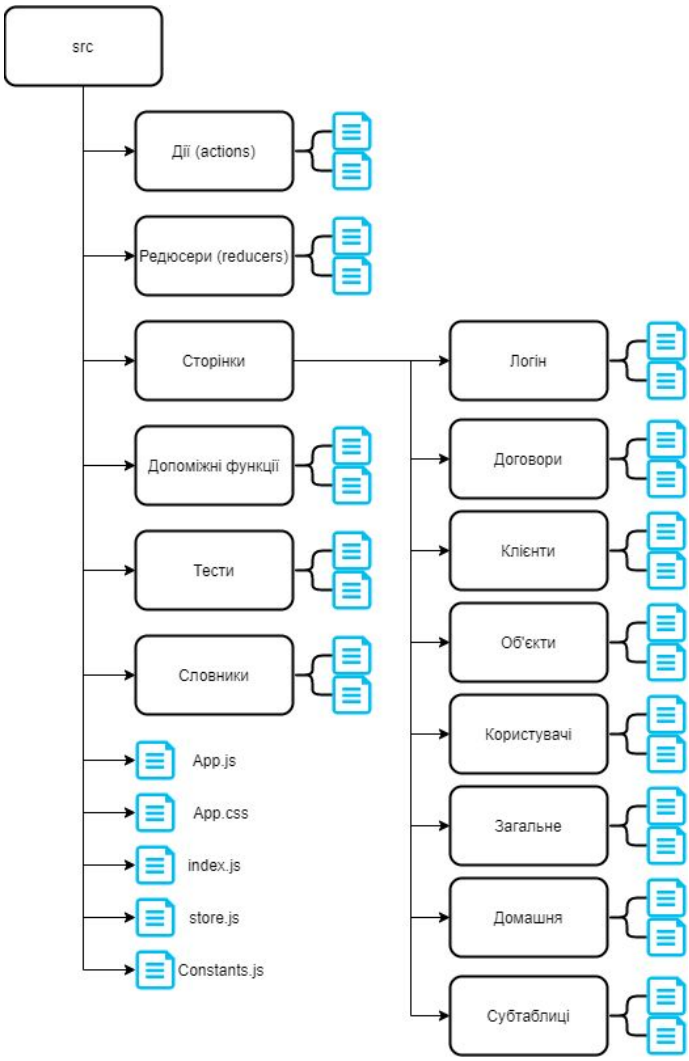
На цьому створення серверної частини закінчено. Далі будемо розширювати серверний функціонал по мірі необхідності.

3.2. Розробка клієнтської частини

Створення клієнтської частини включає в себе декілька етапів. Оскільки ми використовуємо бібліотеку React, додамо до неї допоміжну бібліотеку Redux, яка дозволяє керувати станами системи.

Використання Redux бібліотеки буде означати для нас наступну структуру папок:

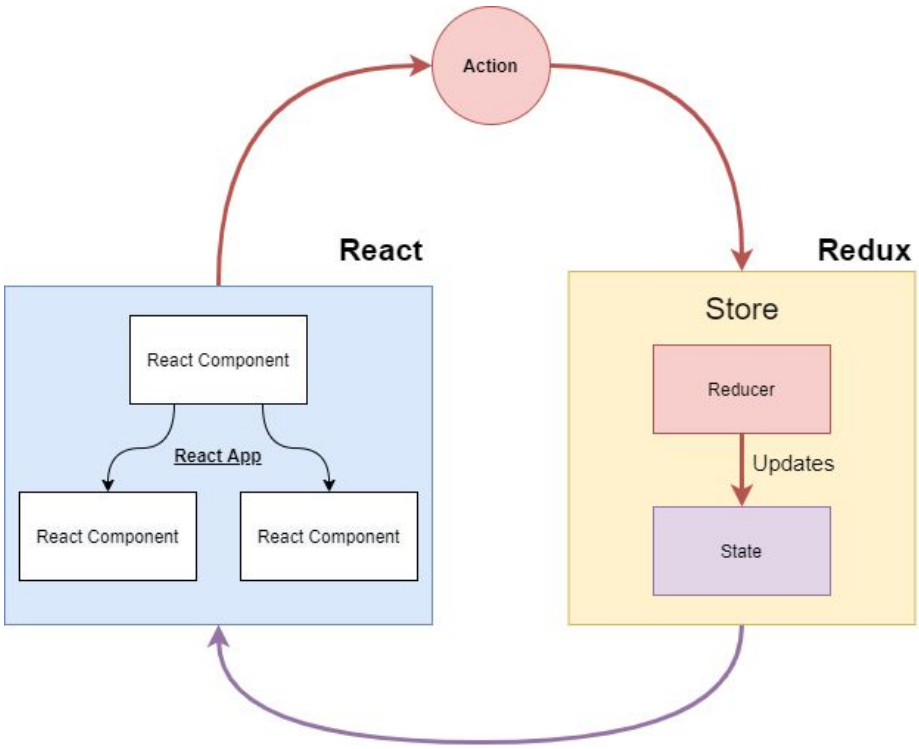
Рисунок 3.14. Приклад структура папок для створення веб-інтерфейсу



Редакс - це контейнер станів застосування. Його ***store*** зберігає все дерево станів і єдиним способом зміни стану є використання функції дії (***action***). Дія

зумовлює оновлення стану через функцію ***reducer***, в котрій фактично відбувається робота з даними (місце логіки). Оновлений ***store*** (або оновлений стан) далі використовується в застосунку. Схематично це можна зобразити так:

Рисунок 3.15 Схема роботи бібліотеки Redux



Тому для того, щоб виконувати якісь дії в нашій системі, ми маємо створити різні ***reducers*** та ***actions.*** Для початку, зробимо це для процесу логіна. Створимо дію логіна, в якій будемо передавати на шлях **/api/login** юзернейм та пароль користувача, а в разі успіху передавати на функцію **dispatch** токен.

Рисунок 3.16 Створення функції дії (action) для процесу логіну

Після створення функції ***action***, треба відповідно створити ***reducer:***

Рисунок 3.17 Створення функції reduce для процесу логіну

JS authentication.js JS authReducer.js ×

src > reducers > JS authReducer.js > ...

```
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;
    }
}
```

Тепер ми можемо створити сторінку для логіна. Але для початку приєднаємо React в файлі index.js:

Рисунок 3.18 Додавання бібліотеки React в застосунок

```
src > JS index.js
1   import React from 'react';
2   import ReactDOM from 'react-dom';
3   import './index.css';
4   import App from './App';
5   import * as serviceWorker from './helpers/serviceWorker';
6
7   ReactDOM.render(<App />, document.getElementById('root'));
8
9   // If you want your app to work offline and load faster, you can change
10  // unregister() to register() below. Note this comes with some pitfalls.
11  // Learn more about service workers: https://bit.ly/CRA-PWA
12  serviceWorker.unregister();
13
```

Та створимо основний файл застосування App.js де приєднаємо Redux:

Рисунок 3.19 Додавання бібліотеки Redux в основний файл App.js

```
src > JS App.js > ...
1   import React from "react";
2   import { Provider } from "react-redux";
3   import store from "./store";
4   import jwt_decode from "jwt-decode";
5   import setAuthToken from "./helpers/setAuthToken";
6   import { setCurrentUser, logoutUser } from "./actions/authentication";
7   import "./App.css";
8
9
10  if (localStorage.jwtToken) {
11    setAuthToken(localStorage.jwtToken);
12    const decoded = jwt_decode(localStorage.jwtToken);
13    store.dispatch(setCurrentUser(decoded));
14
15    const currentTime = Date.now() / 1000;
16    if (decoded.exp < currentTime) {
17      store.dispatch(logoutUser());
18      window.location.href = "/";
19    }
20  }
21
22  function App() {
23    return (
24      <Provider store={store}>
25      </Provider>
26    );
27  }
28
29  export default App;
30
```

Поки що в нашому застосунку ще немає жодного компонента, але все готово для їх створення. Тому створимо папку з ім’ям **Login** з двома файлами: Login.jsx (React компонент) та styles.js (стилі сторінки). Це буде наш перший компонент, на який можна буде подивитись в браузері. Стилізувати React компоненти можна декількома шляхами. Я обрав шлях створення окремого JavaScript файлу, де стилі прописані як об’єкти. Такий підхід дозволяє тримати стилі в окремому місці та імпортувати їх безпосередньо в компоненти.

Почнемо з компоненту. Імпортуємо всі залежності та створимо класс:

Рисунок 3.20 Приклад імпорту залежностей в файли React компонентів

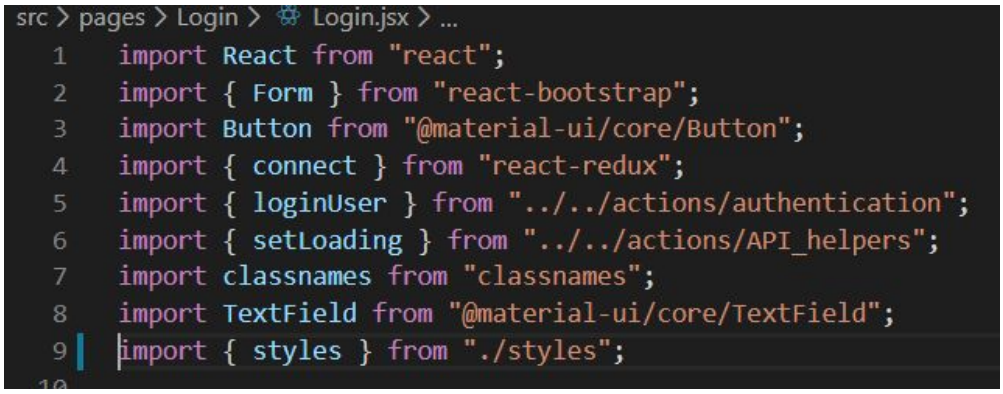


Рисунок 3.21 Приклад реалізації React компоненту

```
export class Login extends React.Component {
  constructor() {
    super();
    this.state = {
      username: "",
      password: "",
      errors: {},
    };
    this.handleInputChange = this.handleInputChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleInputChange(e) {
    this.setState({
      [e.target.name]: e.target.value,
    });
  }

  handleSubmit(e) {
    e.preventDefault();
    const user = {
      username: this.state.username,
      password: this.state.password,
    };
    this.props.loginUser(user);
  }
```

Додаємо форму для логіна з двома полями. Оскільки ми пам’ятаємо про вимогу двох мов, одразу використаємо бібліотеку **i18n**. Ця бібліотека дозволяє швидко та зручно роботи інтерфейси з декількома мовами.

Рисунок 3.22 Приклад реалізації React компоненту (продовження)

```
render() {
  const { errors } = this.state;
  return (
    <div style={styles.loginPage}>
      <div className="loginForm" style={styles.loginForm}>
        <img src="https://bisukraine.com/templates/bis/images/logo_t.svg" style={styles.logo} alt="logo" />
        <Form onSubmit={this.handleSubmit} noValidate>
          <TextField
            required
            id="username"
            className={classnames("", {
              "is-invalid": errors.username,
            })}
            label={i18n.t("username")}
            defaultValue=""
            margin="normal"
            variant="outlined"
            name="username"
            fullWidth
            onChange={this.handleInputChange}
          />
```

```
          <TextField
            required
            className={classnames("", {
              "is-invalid": errors.password,
            })}
            id="password"
            label={i18n.t("password")}
            defaultValue=""
            margin="normal"
            variant="outlined"
            name="password"
            type="password"
            fullWidth
            onChange={this.handleInputChange}
          />

          {this.state.errors.password && <p>{this.state.errors.password}</p>}
          {this.state.errors === "User not found" && <p>{this.state.errors}</p>}

          <Button variant="contained" style={styles.loginButton} type="submit">
            {i18n.t("signin")}
          </Button>
        </Form>
      </div>
    </div>
  );
}
```

І нарешті додамо посилання на **store** та **actions**

Рисунок 3.23 Приклад посилання на Redux елементи в реалізації React компонента

```
const mapStateToProps = (state) => ({
  auth: state.auth,
  errors: state.errors,
});

export default connect(mapStateToProps, { loginUser })(Login);
```

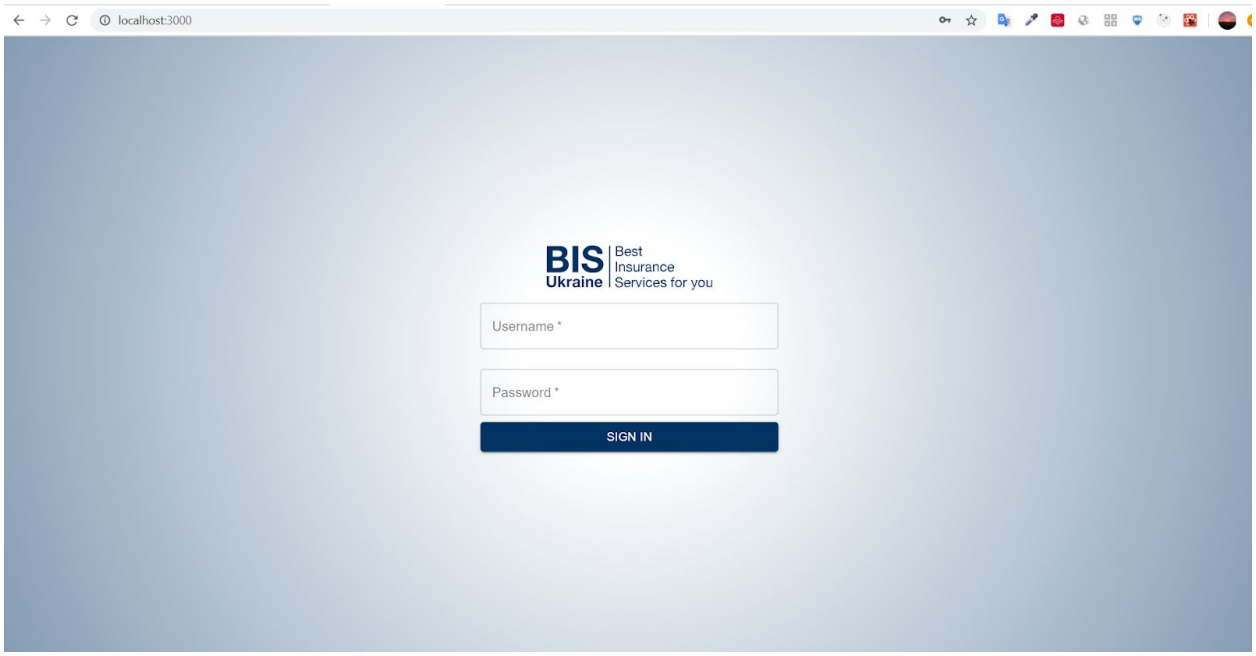
В styles.js додоємо стилі, котрі будуть використані в компоненті:

Рисунок 3.24 Приклад файлу стилей для React компонента

```
const loginPage = {
  width: "100%",
  height: "100%",
  background: "radial-gradient(circle at center, #fff 15%, #033365 189%)",
};

const logo = {
  width: 200,
};

const loginButton = {
  width: "100%",
  backgroundColor: "#033362",
  color: "white",
};

export const styles = {
  loginButton: loginButton,
  logo: logo,
  loginPage: loginPage,
  loginForm: loginForm
};
```

Перевіримо результат в браузері:

Рисунок 3.25 Сторінка логіну



Вийшло непогано. ТОВ “BIS Ukraine” - це реальна компанія, для якої створюється застосунок.

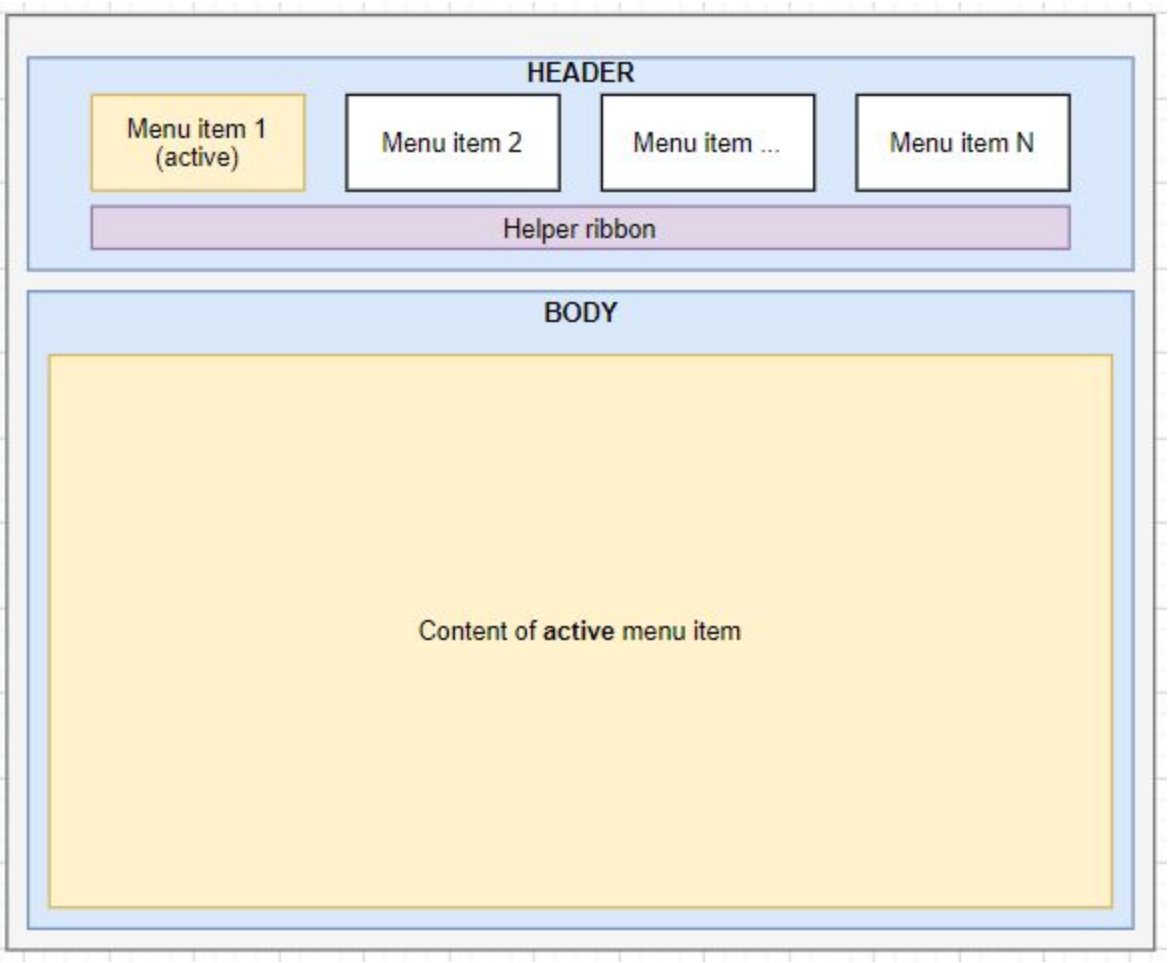
Тепер необхідно додати функціонал, який буде авторизувати користувачів і показувати контент, котрий буде недоступний для всіх інших користувачів поки вони не авторизуються в системі. Для цього будемо використовувати приватні та публічні шляхи застосунку, а для реалізації цього підходу візьмемо бібліотеку **react-router-dom.**

Створимо додаткові компонент, котрі будуть показувати нам чи відображається контент приватного шляху для неавторизованих користувачів. Тут треба зауважити, що в нашій системі має бути лише один публічний шлях - шлях, за яким відкривається сторінка логіну (це може бути шлях схожий на http://site_name/login або просто http://site_name). Всі інші

сторінки мають розміщуватися за приватними шляхами. На цьому етапі вже можемо продумати як буде складатись структура веб-інтерфейсу, які елементи мають бути в ньому присутні, які з них будуть унікалі, а які використовуватись один раз. Нижче на малюнку зображена схема веб-інтерфейсу. Він складається з наступних елементів:

1. Заголовок (header) - це компонент, котрий є єдиним для всіх сторінок і містить в собі елементи меню, а також місце для відображення допоміжної інформації (helper ribbon).
2. Елементи меню (Menu item N) - кнопки, котрі активують відображення специфічного контенту в основній частині інтерфейсу
3. Допоміжна стрічка (helper ribbon) - як вже було зазначено вище, це місце де буде відображатись допоміжна інформація
4. Основна частина інтерфейсу (body) - місце, де відображається контент поточного активного елементу меню.

Рисунок 3.26 Схема основних елементів сайту



Давайте створимо **Header** та перший елемент меню, котрий назвемо **Home**. Ці два компоненти мають бути розміщені в приватних шляхах. Поки що нам не важливо наповнення компоненту, лише його наявність.

Рисунок 3.27 Приклад реалізації компоненту React

```jsx
export class Home extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return (
      <>
        <div style={homePage}>
          TEST HOME PAGE
        </div>
      </>
    );
  }
}

const mapStateToProps = (state) => ({
  auth: state.auth
});

export default connect(mapStateToProps, {})(Home);
```

Рисунок 3.28 Приклад реалізації компоненту загального використання в React

```jsx
class Header extends Component {
  constructor(props) {
    super();
    this.state = {
      current: "home",
    };
  }

  render() {
    return (
      <Navbar style={navBarStyle}>
        <Navbar.Brand style={{ width: 145, padding: 0 }}>
          <Link to="/home">
            <img src="FINALE.svg" className="App-logo" alt="logo" />
          </Link>
        </Navbar.Brand>
        <Nav className="mr-auto">
          <Link className={"nav-link " + (current === "/home" ? "active" : "")} to="/home">
            <Trans i18nKey="home">Home</Trans>
          </Link>
        </Nav>
      </Navbar>
    );
  }
}
```

Час перейти до реалізації приватності. Створимо файл PrivateRoute.jsx, в якому будемо перевіряти чи користувач має “живу” сесію (валідний токен):

Рисунок 3.29 Приклад реалізації перевірки поточної сесії

```
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;
}
```

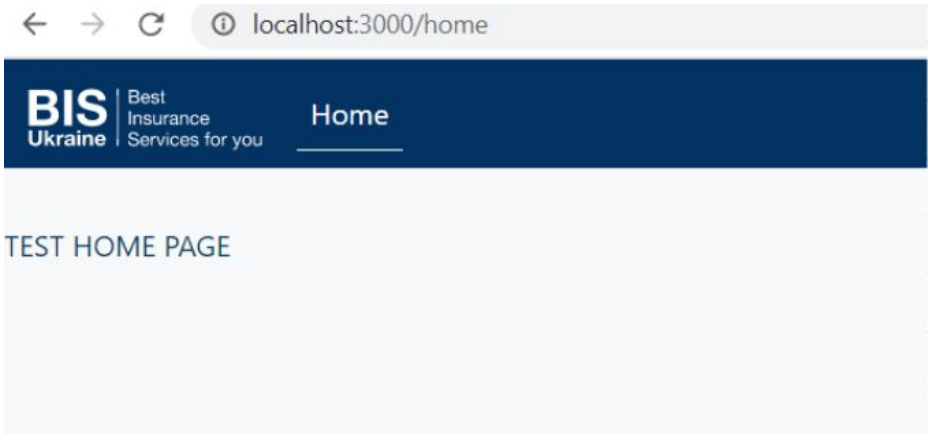
І за результатами перевірки дозволяти користувачу бачити приватний контент, або в іншому випадку перенаправляти його на сторінку логіну (в нашому випадку посилання на сторінку логіна має вигляд http://site_name/, або просто “/”):

Рисунок 3.30 Приклад реалізації перенаправлення користувача в залежності від результатів перевірки сесії

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

Для тестування функціоналу вручну створимо тестового користувача (додавання запису в базі даних) та з тестовим користувачем спробуємо залогінитись:

Рисунок 3.31 Перевірка реалізації приватного контенту застосунку



Також спробуємо перевірити як працює функціонал з неіснуючим юзером та неправильним паролем:

Рисунок 3.32 Спроба входу з користувачем, якого не існує

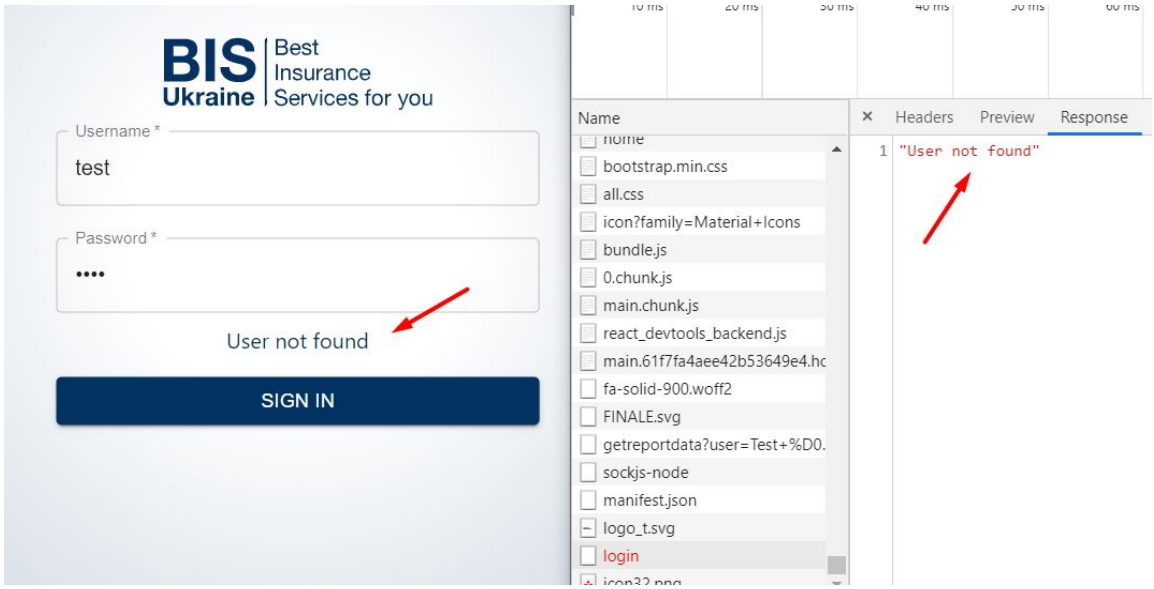
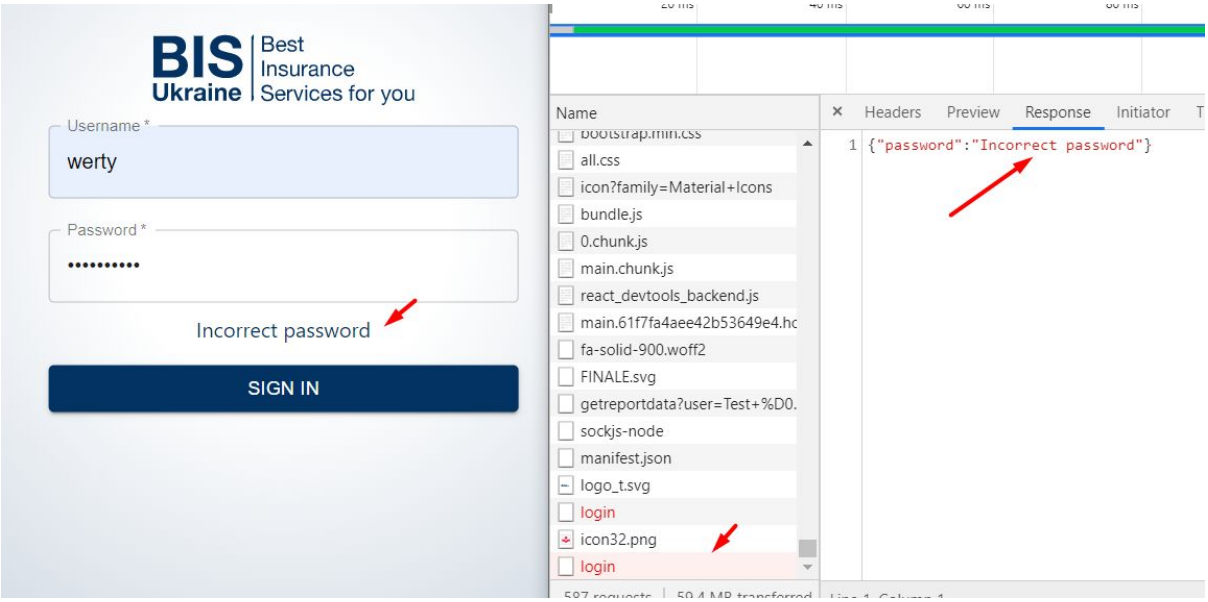


Рисунок 3.33 Спроба входу з неправильним паролем



Додамо інші елементи до меню за аналогією з **Home**.

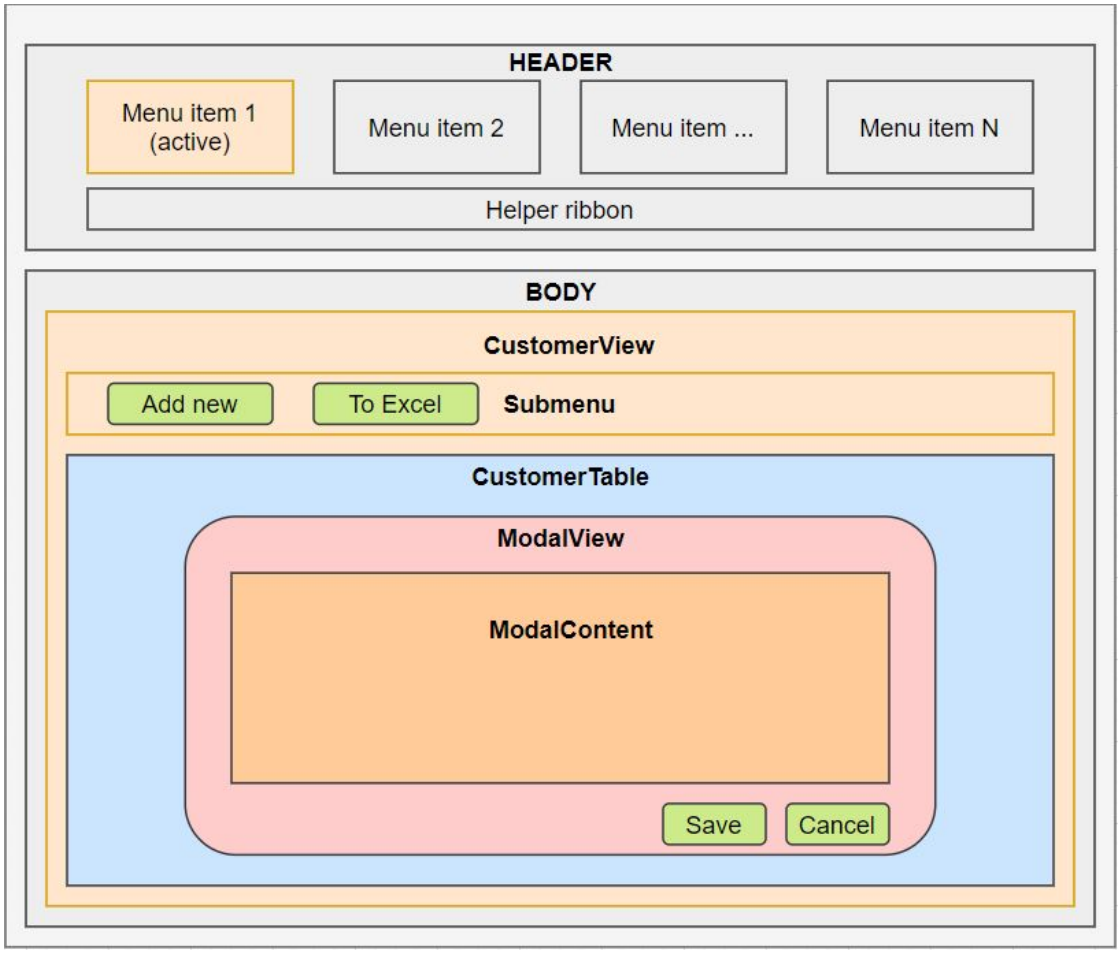
Настав час створити сторінку для клієнтів. Вище ми розглядали високорівневу структуру веб-інтерфейсу. Зараз розглянемо з яких елементів має складатись частина відображення контенту основних розділів меню:

1. Загальний контейнер (**CustomerView**) - місце, де розміщуються всі інші компоненти. В загальному контейнері також розміщуються кнопки взаємодії з таблицею клієнтів: виклик функції додавання нового клієнта та експорт таблиці в Ексель.
2. Компонент таблиці (**CustomerTable**) - основний компонент; в ньому відображається перелік всіх клієнтів з їх даними.
3. Компонент модального вікна (**ModalView**) - той компонент, котрий бачить користувач коли хоче додати нового клієнта, або внести зміни в дані існуючого клієнта. Даний компонент має безпосередньо кнопки

управління (збереження / відміни), а також містить компонент контенту модального вікна.

4. Компонент контенту модального вікна (**ModalContent**) - поля для введення даних по клієнту

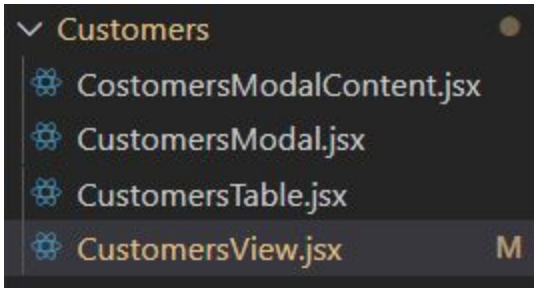
Рисунок 3.34 Схема елементів відображення контенту меню



Така структура буде використовуватись для всіх інших елементів меню (крім головної сторінки).

Створимо відповідні файли в папці.

Рисунок 3.35 Приклад файлів для реалізації сторінки



Згідно схеми, в загальний контейнер (**CustomerView**) додамо компонент таблиці та компонент модального вікна, а також кнопки управління:

Рисунок 3.36 Приклад розмітки компонента в файлі CustomerView.jsx

```
render() {
  return (
    <>
      <CustomersModal
        isShow={this.state.showModal}
        saveContent={this.saveCustomer}
        closeModal={this.closeModal}
        openModal={this.openModal}
      />

      <div style={containerStyle}>
        {this.props.auth.user.role !== "client" && (
          <div style={actionButtonsContainer}>
            <Button variant="contained" style={buttonStyle} onClick={this.openModal}>
              <i className="fas fa-plus-square"></i>
              {this.props.t("customersView.addCustomer")}
            </Button>
            <Button variant="contained" style={buttonStyle} onClick={this.exportToExcel}>
              <i className="fas fa-file-excel"></i> {this.props.t("exportToExcel")}
            </Button>
          </div>
        )}
        <CustomersTable />
      </div>
    </>
  );
}
```

Компонент контенту модального вікна додаємо в компонент модального вікна. Також, додаємо всі необхідні поля по клієнту в компонент контенту модального вікна. Нагадаю, необхідні поля для клієнта ми описували у схемі в серверній частині.

Для компоненту таблиці використаємо бібліотеку “react-table”. Це досить потужна бібліотека, котра дозволяє створювати таблиці з фільтрами різної складності, і навіть з суб-таблицями як частинами рядків таблиці. Власне суб-таблиці нам і потрібні, адже у нас є вимога відображати пов’язані з записом в таблиці клієнтів дані.

Перевіримо що вийшло:

Рисунок 3.37 Веб-інтерфейс пункта меню “Клієнти”

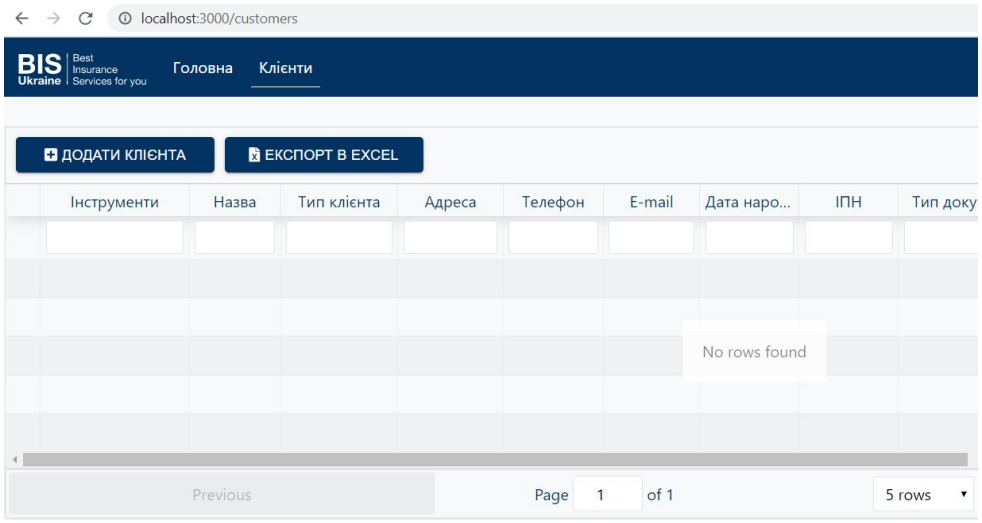
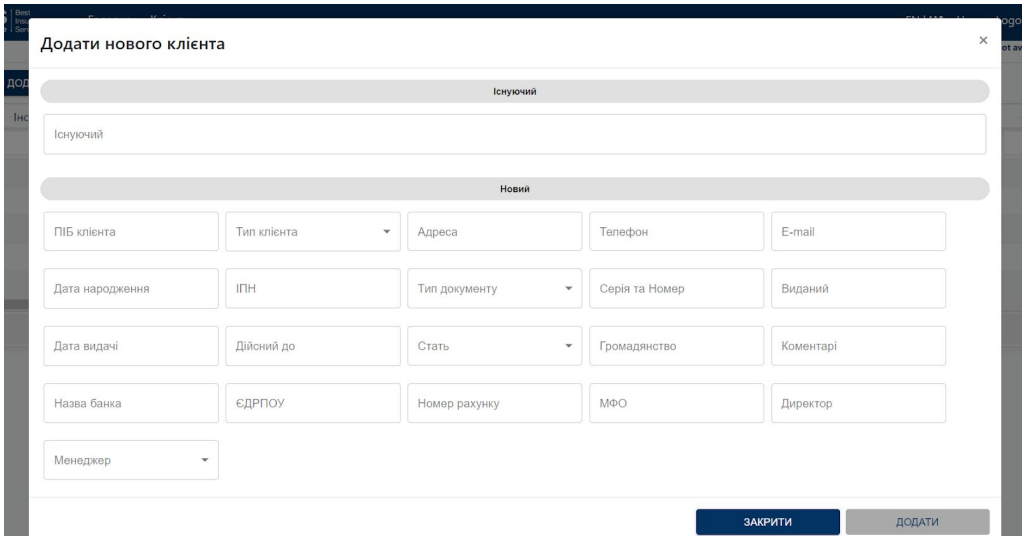
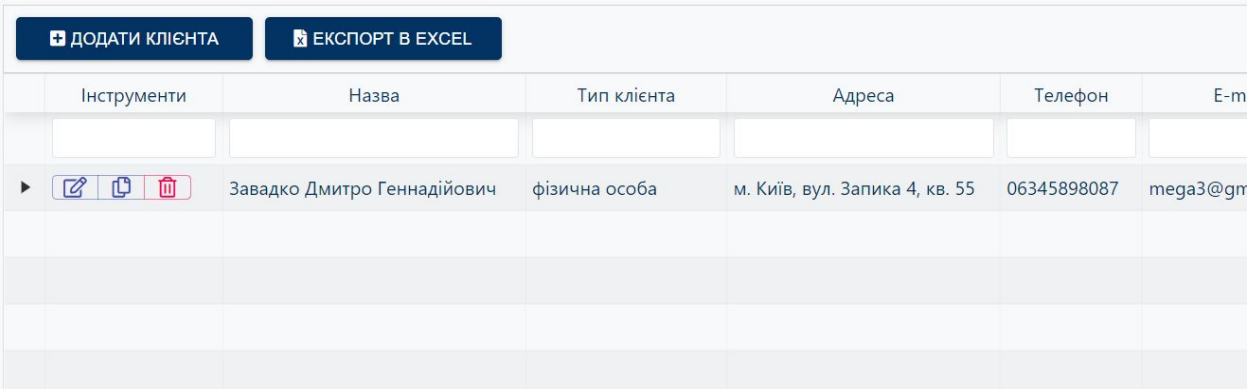


Рисунок 3.38 Веб-інтерфейс модального вікна



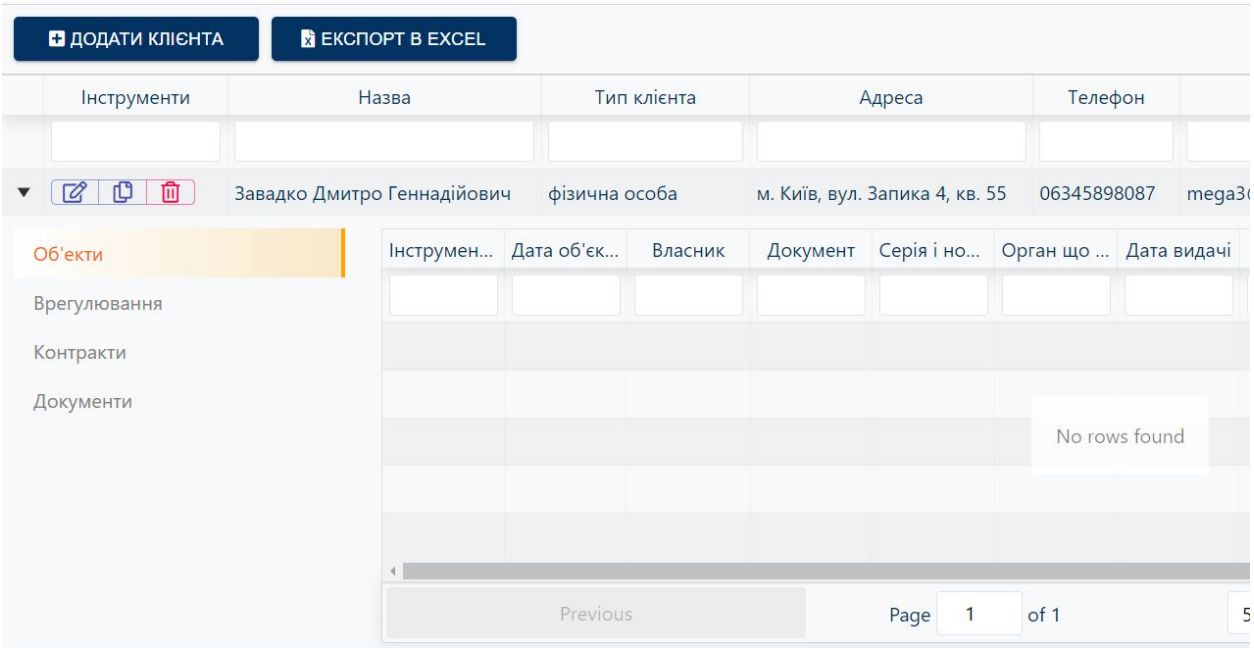
Додаємо тестового клієнта та пересвідчимось що дані відображаються правильно:

Рисунок 3.39 Таблиця з тестовими даними



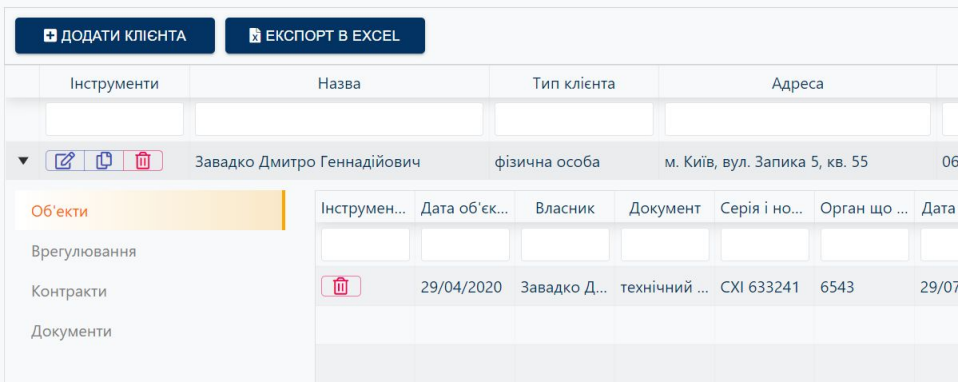
Також перевіряємо чи є суб-таблиці з пов’язаною інформацією:

Рисунок 3.40 Приклад суб-таблиці на веб-інтерфейсі



Оскільки жодного об’єкта ще не було створено, то пов’язаної інформації ще немає. Тому за аналогією з сторінкою “Клієнт” створимо інші сторінки, зокрема і сторінку “Об’єкти”, та додамо перший об’єкт до клієнта. Тепер у клієнта є один пов’язаний з ним об’єкт, а отже має з'явитись інформація по ньому:

Рисунок 3.40 Приклад суб-таблиці з тестовими даними

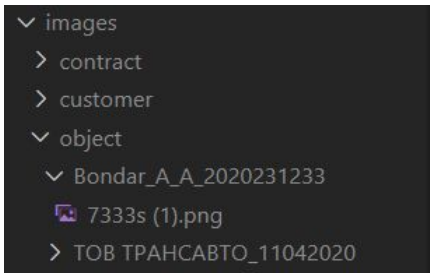


Залишилось додати функціонал по документам. Оскільки загрузка документів буде відбуватись в усіх елементах меню, тому створимо окремий компонент, котрий можна буде використовувати де потрібно. Компонент буде контейнером для стилів місця додавання документів та біліотеки **“react-dropzone”.** Ця бібліотека вже вміє приймати файли і передавати їх для зберігання. Тут треба зауважити, що безкоштовна версія монгодб має обмеження у 500 мб, а тому зберігати файли в базі ми не будемо, натомість всі файли будуть зберігатись на хмарному сервері. Функціонал зберігання файлів на сервері реалізуємо в окремому файлі fileUploader.js (в серверній частині застосунку). Логіка створення папок для зберігання файлу(ів) буде такою:

1. Спочатку визначимо до якої категорії належить файл (клієнти, об’єкти, договори, врегулювання) та оберемо відповідну папку
2. Далі в цій папці створимо ще одну папку, назву якої сформуємо за принципом *ім’я_інн* клієнта
3. Збережемо файл за отриманим шляхом (*клієнт*/*ім’я_інн/назваФайлу)*

Таким чином, файли будуть зберігатись у відповідних категоріях та у відповідних папках клієнтів.

Рисунок 3.41 Фактична структура папок для збереження файлів



3.3. Аналіз результатів роботи

За результатами роботи, ми отримали першу версію CRM системи, котра відповідає базовим вимогам користувачів. Ця система ще має великий потенціал для поліпшення та розвитку, оскільки метою роботи було створити саме мінімально робочу версію продукту.

Вибір технологій розробки задовольнив поточні потреби у повному обсязі. Більш того, такий набір технологій дозволяє з упевненістю стверджувати, що система має можливість масштабуватись та оперувати значним обсягом даних. Бібліотека React чудово справляється з побудовою веб-інтерфейсу, а також швидким відображення змін на ньому. А Node.js послугував гарною платформою для серверної частини, до того ж він також використовує мову програмування JavaScript, на якій створено веб-інтерфейс. Таке поєднання дало змогу швидше створити систему, оскільки не довелось вивчати інші мови програмування.

Обрана база даних MongoDB також справляється з покладеними на неї функціями, хоча важко виділити якусь суттєву перевагу у порівнянні з реляційними базами. У майбутньому можна ще раз проаналізувати чи використовувати дану базу далі, чи перейти на одну з реляційних баз.

Використання набору бібліотек значно пришвидшило роботу як над серверною так і над клієнтською частинами. До того ж, як правило, такі бібліотеки вже оптимізовані та протестовані, тому працюють швидко та стабільно.

Втім, бібліотеки, а точніше їх велика кількість в системі, можна віднести до недоліків системи, оскільки функціональність системи залежить від коректної роботи кожної з бібліотек. Це створює додатковий ризик, який можна було б зменшити у разі написання власного коду замість використання вже існуючого рішення.

Додатково варто відмітити, що код деяких компонентів можна оптимізувати, що буде додано як технічний борг для реалізації в наступних версіях продукту.

3.4. Перспективи подальшого розвитку

На даному етапі система готова до використання, але представляє з себе мінімальний набір функцій. Таку систему треба доповняти додатковим функціоналом.

Найперше що треба зробити - це розширений аналітичний розділ, в якому більш детально буде представлена:

1. аналітика роботи співробітників
2. аналітика даних в системі та можливість формування різних аналітичних запитів.

Окрім того, така система може бути доопрацьована для створення публічного API на випадок, якщо буде необхідна інтеграція з іншими системами.

Також, система розроблялась для використання на персональних комп'ютерах, але сучасні тренди зумовлюють необхідність доопрацювання веб-інтерфейсу для використання на мобільних пристроях.

Обов’язковим, на мою думку, буде також додавання компонентного тестування до системи, оскільки подальший розвиток буде означати збільшення бази коду, а значить і збільшення ризиків того, що новий функціонал негативним чином вплине на існуючий і призведе до помилок чи некоректної роботи.

Не зайвим буде і реалізація логування дій в системі. Оскільки така система містить дуже багато персональних даних, то відслідковування автора та суті дії може виявитись необхідною вимогою для багатьох компаній.

Висновки

Необхідність в спеціалізованих CRM системах для брокерських компаній сьогодні гостро відчувається на ринку України. Специфіка діяльності компаній, а також особливі вимоги до системи не дають змогу повноцінно використовувати універсальні CRM системи. Для задоволення попиту необхідно створювати систему, котра була б орієнтована виключно на діяльність в страховій сфері.

Різноманітні моделі побудови програмного забезпечення дозволяють створювати системи, використання яких буде мати переваги для одних компаній і одночасно цілий перелік недоліків для інших компаній. Згідно з результатами аналізу потреб компаній у сфері страхування щодо CRM системи та її функціональному наповненню, SaaS - це саме та модель, котра буде влаштовувати переважну більшість користувачів. Така модель дозволяє отримувати робочу версію продукту з мінімальними витратами, а також дає можливість використовувати її різними типами користувачів з різних пристроїв. До того ж, використання SaaS позбавляє в необхідності наймати спеціалістів для підтримки стабільності та розвитку системи. Проте такі переваги мають свою ціну, а саме підвищені ризики безпеки, залежність від швидкості інтернету та майже повна відсутність контролю за системою.

Створена версія CRM системи для брокерської компанії у сфері страхування враховує основні вимоги та може бути використана як база для подальшого розвитку системи і її повномасштабного комерційного використання.

Список використаної літератури

1. Управління_відносинами_з_клієнтами [Електронний ресурс]. - 2020. - Режим доступу: https://uk.wikipedia.org/wiki/Управління_відносинами_з_клієнтами

2. Capterra. CRM User Research Infographic [Електронний ресурс]. - 2020. - Режим доступу: https://www.capterra.com/customer-relationship-management-software/user-research-infographic

3. Види CRM-систем [Електронний ресурс]. - 2020. - Режим доступу: https://salesap.ru/vidy-crm-sistem/

4. Програмне забезпечення як послуга [Електронний ресурс]. - 2020. - Режим доступу: https://uk.wikipedia.org/wiki/Програмне_забезпечення_як_послуга

5. A Look at the Pros and Cons of SaaS [Електронний ресурс]. - 2018. - Режим доступу: https://charlesphillips.me/a-look-at-the-pros-and-cons-of-saas/

6. Developer Survey Results [Електронний ресурс]. - 2020. - Режим доступу: https://insights.stackoverflow.com/survey/2019#most-popular-technologies