МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

Кафедра САПР



Бакалаврська кваліфікаційна робота

на тему :

«Розроблення системи контролю безпечності фінансово-ділових відносин між різними суб'єктами»

Виконав:

ст. гр. КН-44

Лютий Н.І.

ЛЬВІВ 2019

1. **Вступ**

В сучасному світі, з його рівнем доступності технологій інтернету, широкими можливостями для розвитку людей у сфері роботи з інформаційними технологіями, і ще ширшими – для несанкціонованої роботи з тими ж системами, питання безпеки виноситься на перше місце.

З кожним днем все більше договорів заключаються у мережі, все більше людей можуть співпрацювати один з одним, жодного разу не бачивши один одного в реальному світі. Все більше даних генерується пристроями та людьми, і в рази об’ємніша кількість зберігається в існуючих дата-центрах.

У часи зародження інтернету питання безпеки передачі даних, доступу до них не відігравало настільки важливої ролі, як зараз. У світлі подій минулих років, коли з корпорацій з мільярдами користувачів по всьому світу, витягували дані, що обчислювались сотнями терабайтів, якщо не ектабайт, було отримано доступ до сотень тисяч аккаунтів, дані людей використовувались аналітичними організаціями у цілях маніпуляції суспільною думкою. Безпека не лише особистих даних чи фотографій опинилась під загрозою – неодноразові атаки хакерів на фінансові установи та на заклади, які відповідають за національну безпеку різних країн, спричинили втрати у розмірі мільйонів доларів, підвищення недовіри до стійкості фінансової системи.

Фінансові дані звичайних користувачів також знаходяться під ударом. Ніхто не може бути впевненим, що його телефон завтра не взламають, не підберуть пароль до його аккаунту в платіжній системі.

Онлайн платежі на сьогодні займають левову частку ринку, їх використовують через зручність і швидкість проведення операцій, не лише прості користувачі, а й люди, які приносять прибуток державі – приватні підприємці, великі корпорації, середні фірми. Небезпека проведення онлайн платежів, заключається не лише в можливості викрадення даних третіми особами, але й привласнення коштів недобросовісними виконавцями, що мають достатньо можливостей для імітації процесу виконання договору , а після отримання першої, кількох або повної оплати, зникають.

З наведених вище причин, вирішено на основі власного бачення, розробити систему, яка поєднуватиме такі риси як безпечність транзакцій, захищеність даних у сховищах баз даних, а також елементи контролю фінансових операцій – створення та підписання договорів між суб’єктами онлайн, використовуючи цифровий підпис, призначення автоматичного або, за необхідності, фізичного аудитора, який аналізуватиме процес виконання угоди сторонами, і прийматиме рішення про правомірність дій учасників фінансових відносин, попереджаючи можливі порушення зі сторін замовників та виконавців.

Метою даної бакалаврської роботи є розроблення системи контролю безпечності фінансово-ділових відносин між різними суб'єктами.

**1. Огляд існуючих систем та аналогів**

На даний момент існує кілька технологій, які забезпечують контроль над фінансами при виконанні певних ділових операцій. Їх умовно можна розділити на наступні категорії:

А) Програми, що забезпечують контроль над картковим рахунком фізичної особи. Основне застосування це мобільний банкінг (перерахування коштів з однієї картки на іншу, оплата комунальних послуг, поповнення мобільного рахунку).

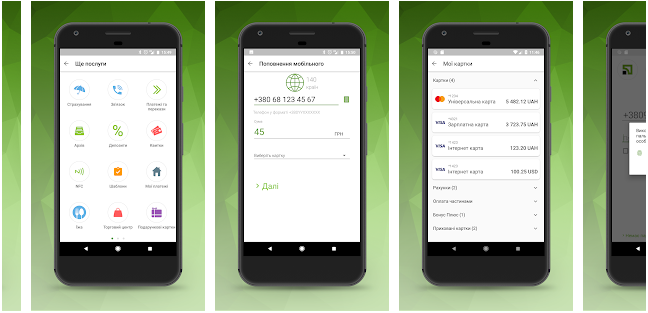
Б) Програми, які об’єднують кілька карток користувача, не спеціалізуючись конкретно на одному банку – емітенту. В основному призначені для онлайн операцій, безпечного розрахунку на світових купівельних майданчиках, оплати в магазинах з допомогою технології NFC.

В) Фріланс-системи, що утримують гроші на тимчасовому рахунку, допоки виконавець не надіслав готового продукту.

Розглянемо детальніше ці категорії на основі найбільш популярних представників.

Яскравими прикладами існуючих програмних продуктів для оперування власними картковими рахунками, на українському ринку, можна назвати Приват24, Монобанк, UkrsibbOnline.

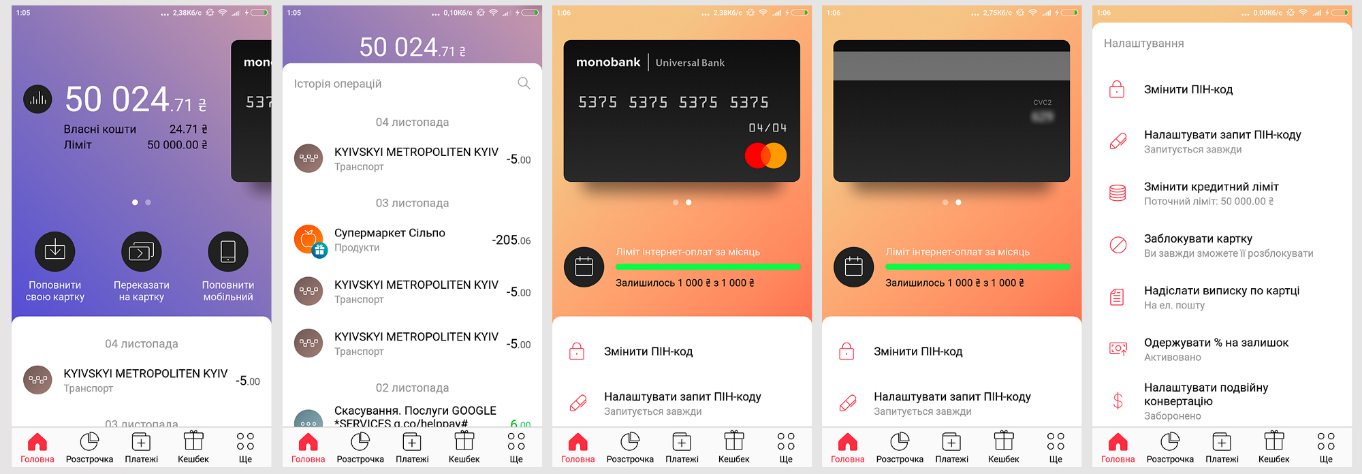
**Приват24** дозволяє оперувати власними картками, отриманими в Приватбанку, пересилати свої кошти на інші рахунки, отримувати кредит на свою картку прямо з телефону, налаштувати накопичувальну систему, яка з кожного поповнення чи зняття коштів перераховуватиме певну суму або відсоток на Скарбничку. Також, присутня можливість поповнювати мобільний рахунок будь-якого оператора, оплачувати комунальні платежі, здійснювати платежі між банками, оплачувати рахунки по реквізитах, купувати квитки на різноманітний транспорт (трамваї, потяг, літаки).(рис.1.1)



*Рис.1 – Інтерфейс Приват24*

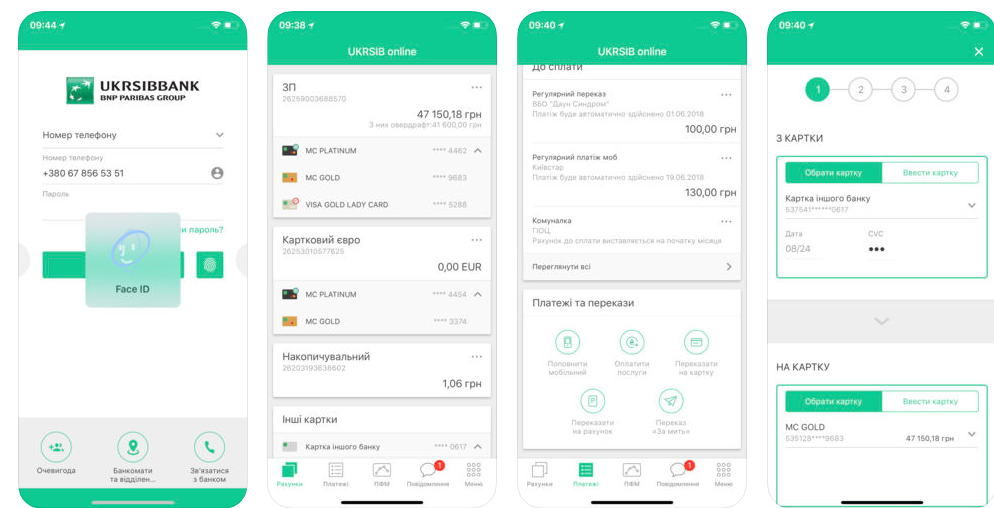
На даний момент це один із найбільш часто використовуваних додатків в українському сегменті фінансового ринку, та до його недоліків ми можемо занести його перевантаженість – у ньому надто багато функціоналу, який захаращує програму, і відволікає від найчастіше використовуваних функцій.

**Монобанк** – відносно новий гравець на ринку, який фактично знаходиться під опікою Універсал банку. Серед інших програмних продуктів вирізняється простотою, продуманим дизайном без надлишкових функцій, зручністю у використанні. Спектр послуг у нього схожий до Приват24. Через його продуманість з точки зору досвіду користувачів, протягом року він є продуктом з високою швидкістю приєднання нових клієнтів.(рис.1.2)



*Рис.1.2 – Інтерфейс Монобанку*

**UkrsibbOnline** – платформа для управління рахунками в Urksib банку. Має веб та мобільну версію.(рис. 1.3)



*Рис.1.3 – Інтерфейс додатку Ukrsibb Online*

Платформа пропонує доволі зручний користувацький інтерфейс, він не перевантажений зайвими функціями, як і Монобанк. Дизайн більш продуманий, ніж у Приват24, але гірша логіка навігації і зрозумілість віджетів, ніж у Монобанку.

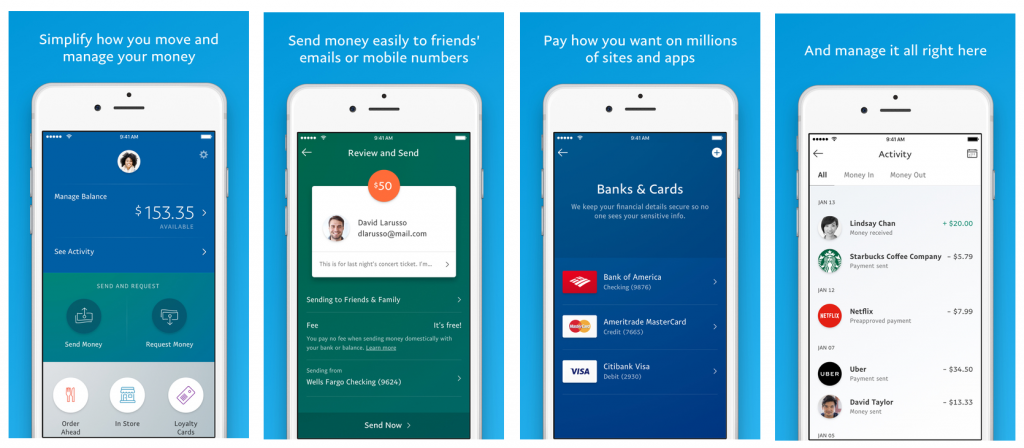
На українському ринку платформи такого плану не єдині, вони існують і в інших фінансових установах, менш поширених за клієнтською базою. До переваг таких систем можна однозначно віднести зручність у виконанні щоденних фінансових операцій, відсутність необхідності у довгому вивченні даної системи для оперування нею, швидкість обробки транзакцій, можливість здійснювати фінансові дії навіть у вихідні чи святкові дні.

Недоліки – часті повідомлення про переривання транзакцій, можливість отримати доступ стороннім особам, які мають достатньо даних про клієнта.

Серед об’єднавчих програм можна виділити такі:

* PayPal.
* LiqPay.
* Google Pay.
* Apple Pay.
* Samsung Pay.
* Mi Pay.

**PayPal** є справжнім першопрохідцем серед програмних продуктів, які об’єднують можливість користуватися продуктами (картками), або й рахунками різних фінансових установ у одному місці. (рис.1.4)



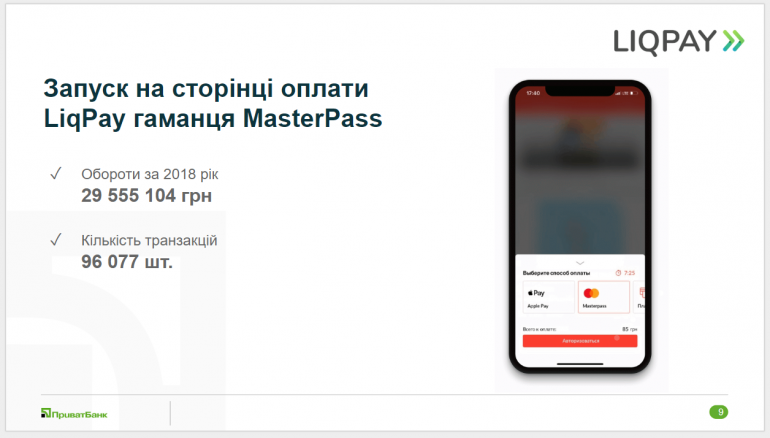
*Рис.4 – Інтерфейс мобільного додатку Pay Pal*

Це найпоширеніший у світі спосіб розрахунків в інтернеті [<https://uk.wikipedia.org/wiki/PayPal>].Серед її головних рис можна охарактеризувати:

* Покриття. Програма є поширеною у світових масштабах, і дозволяє проводити фінансові операції без жодної необхідності стояти у живих, і часто, довжелезних чергах для здійснення міжнародної оплати
* Можливість безпечно об’єднати дані карток різноманітних банків-емітентів. Для системи неважливий банк, що випустив картку – вона використовує стандартні дані карток (її номер, термін дії, CVV код) для проведення транзакцій, тим самим усуваючи необхідність тримати ці картки при собі фізично
* Можливість створити корпоративний рахунок і приймати оплату через систему PayPal. Дозволяє приймати оплату на сайтах, використовуючи кнопки або готове API, розроблене компанією, допомагає створювати персоналізовані шаблони на оплату рахунків.
* Можливість користуватися системою One Touch – здійснювати покупки на перевірених сайтах, навіть без необхідності вводити поштову адресу та пароль
* Одна з основних переваг – клієнтам системи немає необхідності вводити на купівельних платформах безпосередньо дані їхніх рахунків. Вони зберігаються в системі, і для оформлення замовлення достатньо скористатись своєю електронною поштою та паролем. Це надає один з кращих варіантів захисту покупців від фішингових атак, оскільки перевірку фінансової добропорядності покладено на саму автоматичну систему, яка здатна виявити підробку в випадках, коли людина могла б не побачити різниці між певними платформами і/або сайтами, на яких вона має намір здійснювати фінансові операції.

**LiqPay** – українська розробка, яка позиціонує себе як альтернатива PayPal.

Аккаунт користувача у цій системі є прив’язаним до номеру телефону у міжнародному форматі, рахунок є прив’язаним до банківської картки Приватбанку, що може створити зайві витрати часу для оформлення картки банку.(рис.1.5)



*Рис.1.5 – Запуск на сторінці оплати LiqPay гаманця MasterPass*

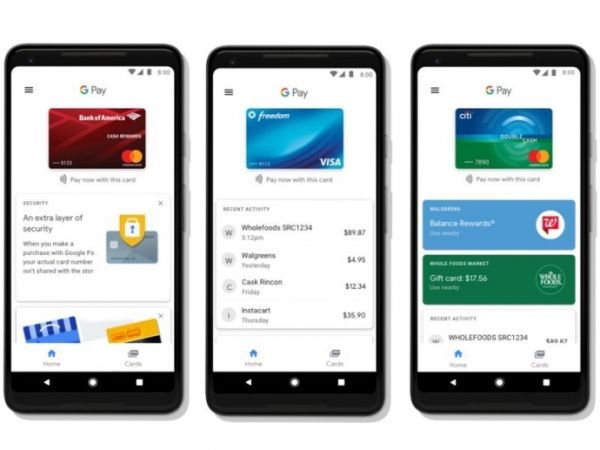
Основні можливості даної системи:

* Здійснення мікроплатежів ( від 0.02$)
* Здійснення масових платежів
* Проведення миттєвих переказів між LiqPay рахунками
* Виведення коштів на будь-яку картку з системи Visa
* Прийняття платежів на сайтах
* Здійснення обміну валют між рахунками LiqPay
* Переведення в готівку чеки Google

Безпека фінансових операцій забезпечується за допомогою використання технології OTP ( одноразового пароля ), що надсилається в SMS – повідомленні. Сервіс має сертифікати GoDaddy Secure Web Site, Verified by Visa та MasterCard SecureCode. [<https://uk.wikipedia.org/wiki/LiqPay>]

**Google Pay** ( раніше Android Pay ) – також дозволяє об’єднати в собі дані кількох карт різних емітентів, безпечно дозволяючи здійснювати фінансові операції, використовуючи базис попередньої розробки Google Wallet. Надає можливість здійснювати оплату в реальних магазинах також за допомогою смартфона, без жодної необхідності мати при собі платіжну картку.

Саме можливість здійснення оплати через безконтактні термінали одним дотиком смартфона є характеристикою, що найбільш відрізняє Google Pay серед розглянутих вище систем.(рис. 1.6)

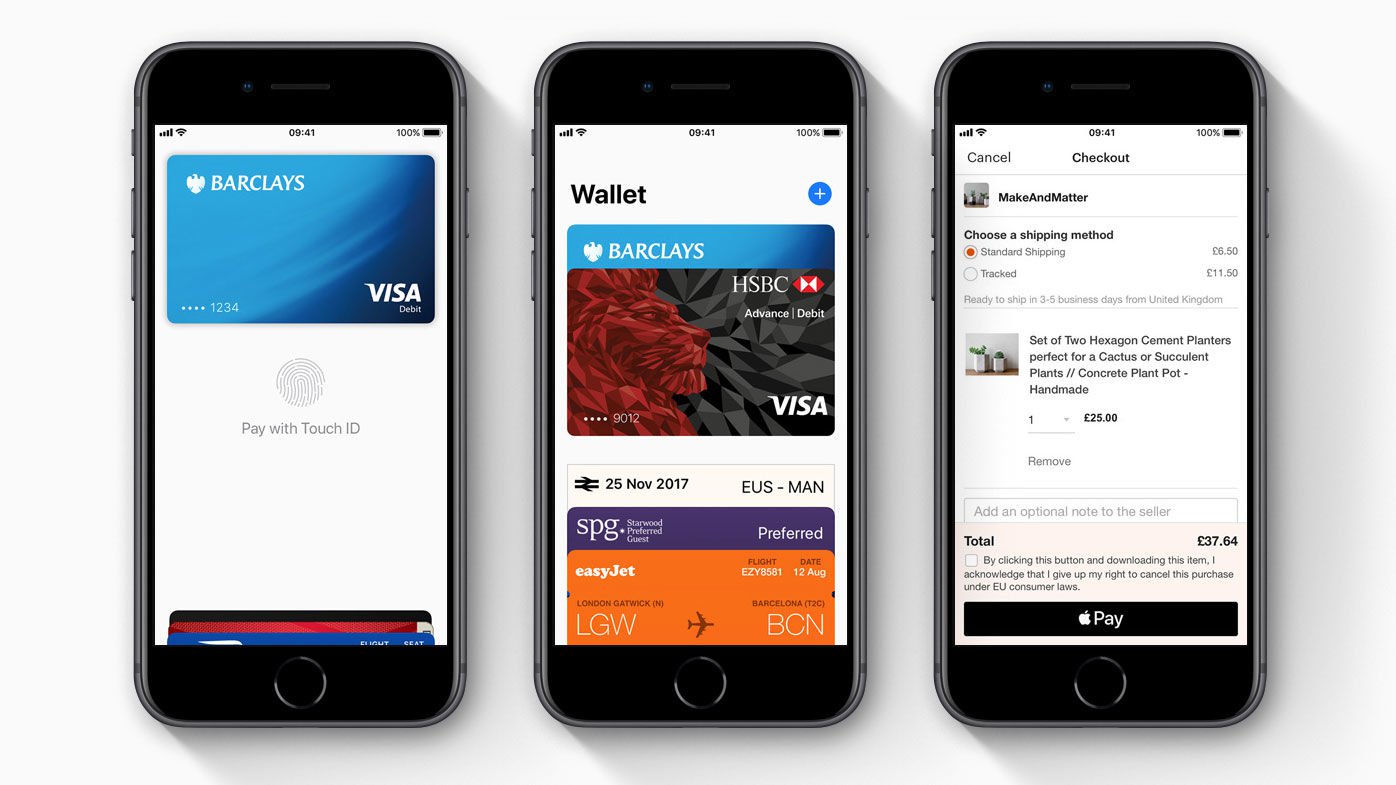


*Рис.1.6 – Інтерфейс додатку Google Pay*

Для авторизації та ідентифікації користувача використовуються дактилоскопічний датчик смартфону, якщо такий присутній, або ж активація через відповідний код доступу.

При оплаті користувачем, сервіс не надсилає на термінал номер розрахункової карти, а натомість генерує віртуальний рахунок, який містить дані рахунку користувача. Cервіс можна використовувати за наявності на пристрої операційної системи Android Wear або Android версії вище 4.4, та присутності робочого NFC

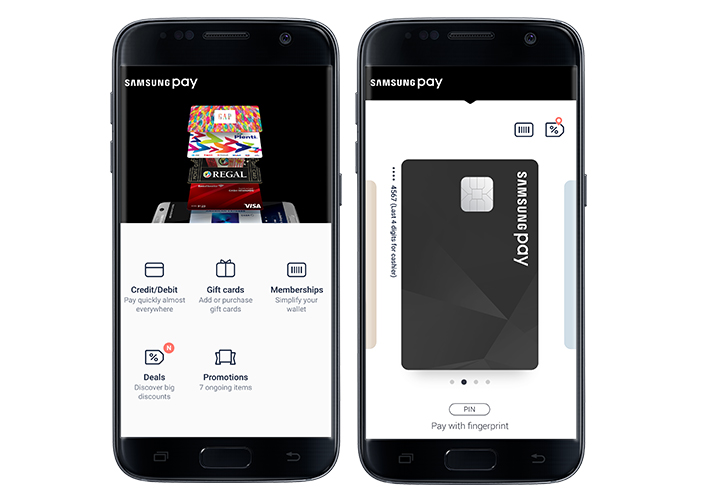
**Apple Pay** – платіжна система, конкурент Google Pay, фактично повторює її функціональні особливості, також працює лише на пристроях власної екосистеми Apple (IPhone, Apple Watch etc.). Фактично, відрізняються вони лише рівнем охоплення аудиторії та мінімально інтерфейсом.(рис1.7)



*Рис. 1.7 – Інтейфейс додатку Apple Pay*

**Samsung Pay** – аналог попередніх систем, має можливість здійснювати платежі з допомогою смартфонів та годинників. Особливістю даної платформи є можливість оплати не лише з допомогою чіпа NFC, а й генерувати імпульси схожі до тих, що виникають при зчитуванні магнітної стрічки, з технологією MST.

Система обмежена використанням лише на пристроях Samsung, через що є менш популярною за попередні, які накладають обмеження на рівні операційної системи, а не бренду. В Україні, на даний момент, сервіс не є представленим. (рис.1.8)



*Рис. 8 – Інтерйфейс додатку Samsung Pay*

**Mi Pay** – система, що повторює шлях Samsung Pay, з тим доповненням, що система обмежена не лише використанням всередині бренду, а й використанням в географічних локаціях Китаю.

Наступним типом є системи, які виступають у ролі поручителів і не дають можливості розгорнути діяльність шахраям.

Платформа **Freelance**.

Платформа популярна завдяки можливості для багатьох розробників і замовників працювати безпосередньо, уникаючи витрат часу на менеджмент, аналітику, проходження через бюрократичні кола компаній та даючи змогу банально економити, прибираючи зайві елементи з фінансово-виробничого ланцюга. Сервіс зарекомендував себе як надійний організатор відносин замовників та виконавців. В роботі сервісу траплялися такі випадки, що замовник оплатив частину коштів авансом, а проект не було виконано, або ж після показу демо-версії виконавцем замовник оплатив всю роботу, а виконавець зник, не надіславши замовнику жодних вихідних кодів програмного продукту.

Звичайно, що подібні випадки погано впливають на рівень довіри людей до такої платформи, формуючи думку про ненадійність і високу імовірність шахрайських дій.

Як розумний вихід з такої ситуації було запропоновано систему «Безпечної угоди».

В рамках погодження виконавця і користувача, що працюють за такою системою, визначаються наступні характеристики її роботи:

* Впевненість виконавця в тому, що після завершення роботи над проектом його праця буде справедливо оплаченою
* Впевненість замовника у тому, що виконавець не зникне, після отримання грошей і не завершивши проект
* Впевненість замовника в тому, що він зможе повернути свої кошти, якщо виконавець зробив проект недобросовісно, без врахування побажань замовника, або ж якість виконання проекту є неналежною, або ж невідповідною заявленому рівню знань виконавця

Для реалізації попередніх вимог, система діє наступним чином – після того, як умови виконання роботи між виконавцем і замовником було узгоджено, система резервує обумовлену частину коштів на своїх рахунках, виступаючи свого роду банківською коміркою та арбітром.

Після виконання всіх умов і робіт, виконавець надсилає запит системі про виведення коштів, і якщо проект задовольняє замовника, отримує свою обумовлену винагороду.

**Upwork** – одна з лідируючих фріланс – платформ, що утворена в результаті об’єднання сервісів oDesk i Elance. Має найбільшу кількість замовників та проектів серед усіх платформ. До безумовних переваг користування системою належать простота інтерфейсу, інтуїтивність використання, швидкість початку роботи з платформою (достатньо зареєструватись). На Upwork - платформі використовується аналогічна система безпеки угод, що забезпечує прозорість і надійність фінансових операцій.

**Висновки**:

Розглянуто типи програм, що дозволяють користувачам здійснювати, зручно контролювати та налаштовувати фінансові операції, проаналізовано наявні на ринку платформи для управління фінансовими рахунками або картками клієнтів, досліджено їхні основні особливості з точки зору користувачів та їх досвіду.

На прикладі вищенаведених систем можна стверджувати, що на ринку вже існує широкий вибір програмних продуктів для різноманітних цілей, найбільш цікавими з технологічної точки зору є системи Freelance та Google Pay, з яких можна почерпнути цікаві з нашої точки зору технології.

В розробленій системі буде використано схожий принцип безпеки угод, а також надана можливість користувачам редагувати угоди онлайн, за погодженням іншого учасника договору, та здійснювати операції мобільного банкінгу (на плафтормі сайту), використовуючи кошти, які знаходяться на рахунках, що доступні для системи.

**2. Аналіз технологій для реалізації проекту**

**2.1 Аналіз технологій для створення і контролю транзакцій**

Для реалізації системи використано технології, які нададуть можливість взаємодіяти з персональними даними клієнтів, їх фінансовими даними та рахунками.

В основі будь-якого обміну даними, які є вразливими до перехоплення, на даний час лежить використання принципу транзакцій.

Транзакція – це дія або певна послідовність дій, яка відбувається за принципом «все або нічого». Для збереження даних валідними і для уникнення колізій, транзакції діють наступним чином – при старті транзакції запам’ятовується стан систем, між якими відбуваються певні операції. Якщо на будь-якому етапі транзакції виникає помилка, неочікувана ситуація або пошкодження даних, всі попередні дії, що були здійснені в межах транзакції анулюються до попереднього запам’ятованого стану. Таким чином можна досягти впевненості в тому, що весь пакет дій виконано, а при помилці – що не пошкоджено дані жодної з сторін транзакції.

Проте, транзакції використовують не одинично, а як елементи програмної системи. Враховуючи їхні переваги, вони є модулем, що застосовується у кожній програмі, пов’язаної з передачею великої кількості приватних даних. Серед найбільш відомих соціуму систем, вони використовуються як основа технології блокчейну.

З поширенням в останні роки використання блокчейну велику популярність здобули розумні контракти.

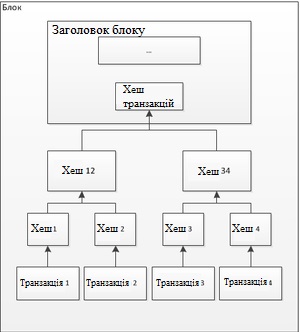
**2.1.1 Хеш-функція**

**Хеш-функція** – математичний алгоритм, в основі якого лежить рекурсивне обчислення результату, здебільшого рядкового коду попередньо обмеженої довжини. Функція отримує на вхід , до прикладу, певний текст випадкової довжини, який називається праобразом, та обчислює значення результату.

У процесі підписання файла тим же цифровим підписом виконується визначена хеш-функція, яка генерує рядок обмеженої довжини, що має назву «дайджест повідомлення». З допомогою хеш-функції ми отримуємо гарантію того, що жоден біт зі всієї інформації не було зміненно чи видалено.

Якщо в алгоритмі формування електронного цифрового підпису використовується стійка однобічна хеш-функція, то немає жодного існуючого способу перенести підпис з одного документу чи файлу на інший, або ж якимось іншим чином змінити повідомлення, що було підписане. Кожна, навіть найменша зміна у підписаному файлі буде знайдена під час проведення звіряння цифрових ключів.

**2.1.2 Технологія Блокчейн**



*Рис 2.1 – Умовне архітектурне представлення технології Блокчейн*

Блокчейн представляє собою розподілену базу даних, в якій знаходиться на збереженні впорядкований ланцюжок записів , які звуться блоками, що з кожною транзакцією в мережі стає довшим.

Весь обсяг даних, що знаходяться в базі, є захищеним від підробки, видалення або спотворення. Серед інформації кожного блоку є часовий штамп, хеш-код попереднього блоку та дані транзакцій, які відображаються у вигляді хеш-дерева.

Копія цієї бази або ж її компонент одночасно зберігається на множині комп’ютерів та проходить синхронізацію відповідно до формально

заданих правил побудови ланцюга блоків. Дані блоків не знаходяться у зашифрованому вигляді, однак вони є захищеними від змінення криптографічним способом через хеш – ланцюжок.

Розподілена база даних такого типу є закладеною в основу технології криптовалюти Біткоїн, яка стала фундаментом сьогоденного стандарту обігу та захищеності віртуальних коштів.

У Біткоїна існує два типи користувачів: звичайні особи, що користуються цифровою валютою, та люди, що виступають власниками і адміністраторами обладнання, яке обчислює хеш-функцію та формує блоки – майнери.

Для запису нових транзакцій у систему Біткоїна та схожих систем використовується блок транзакцій, як спеціальна структура для запису. Він містить у собі відомості про успішно проведені транзакції, дерево їхніх хешів, а окрім того, заголовок, в якому розташовані службові дані, де зокрема міститься хеш-код попереднього блоку. Кожен наступний блок є фактичним підтвердженням правильності попереднього і його материнських блоків.

Для того, щоб транзакція вважалась підтвередженою ( успішною ), її формат та всі підписи зобов’язані пройти перевірку і бути записаними у новий блок.

Для високої надійності транзакцій перевірка достовірності потребує наявності кількох наступних блоків. Кожен блок має посилання на попередній. За умови, що всі блоки є вибудованими в лінію, можна побачити історію проведення транзакцій за весь період існування системи, від створення, до дати обчислення хеш-коду останнього блоку.

Зазвичай, навмисна модифікація інформації в будь-якій з копій розподіленої бази або навіть на кількох копіях не буде визнана істинною, оскільки вона не відповідатиме правилам. Деякі зміни можуть бути прийнятими, якщо їх внести в абсолютно всі копії бази. До прикладу, видалення кількох неправильно або помилково сформованих блоків.

Блок побудований таким чином, що містить у собі заголовок та список транзакцій. Заголовок містить свій окремий хеш-код, хеш попереднього (материнського для нього) блоку, хеші транзакцій та додаткову службову інформацію.

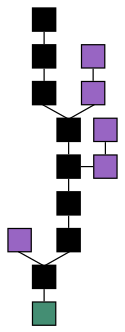
Першою транзакцією в блоці в системі Біткоїну завжди вказується отримання комісії, яка є фактичною нагородою для майнера за створений блок. Після неї записано всі або обрані попередні транзакції, які іще не були записаними у попередні блоки.

Для транзакцій у блоках використовується технологія деревовидного хешування. Транзакції, окрім першої про нарахування винагороди за створення блоку майнером, містять всередині атрибути input посилання на транзакцію, за допомогою якої на цей рахунок було отримано Біткоїни. Комісійні операції можуть фактично зберігати в атрибуті різноманітну інформацію, оскільки в них не існує батьківських транзакцій.

В системі Біткоїн існує поняття обмеження значення хеш-коду заголовку, яке має бути більшим або рівним певному числу, величина якого є постійно змінною. На такі умови платформа пішла для забезпечення стабільності системи, щоб майже завжди правильним було твердження про створення одного блоку протягом десяти хвилин, а на створення 2016 блоків – приблизно два тижні.

Через такий хід розробників, часто виникають ситуації, коли створений блок не проходить перевірку на до відповідність хеш-коду і відкидається системою, в результаті чого майнер цього блоку не отримає комісію за його створення.

Ймовірними є ситуації, коли від різних користувачів надходить на підтвердження блок, в якого той же материнський блок, що й в іншого майнера. В такі моменти відбувається галуження системи, яке надходить на розгляд арбітражем майнерів, як тільки відновлюється стабільне поширення блоків. (рис. 2.2)

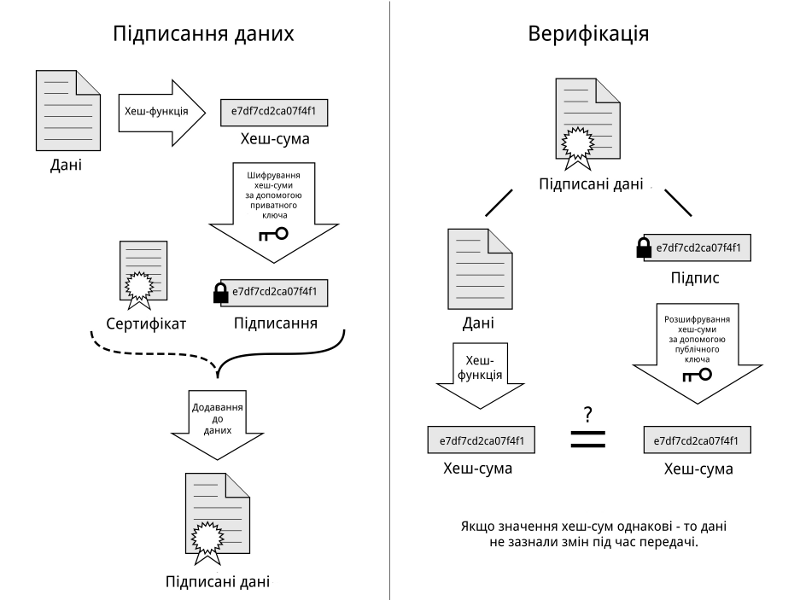


*Рис.2.2 – Галуження ланцюжка блокчейну*

Клієнти обирають гілку(ланцюг), яка має найбільшу довжину та найважчий рівень складності як єдину достовірну, а інші гілки, які йдуть паралельно обраній, втрачають свою актуальність і підтвердженість, що відповідно також виявляється умовою позбавлення майнерів їх винагороди.

**2.1.3 Електронний цифровий підпис**

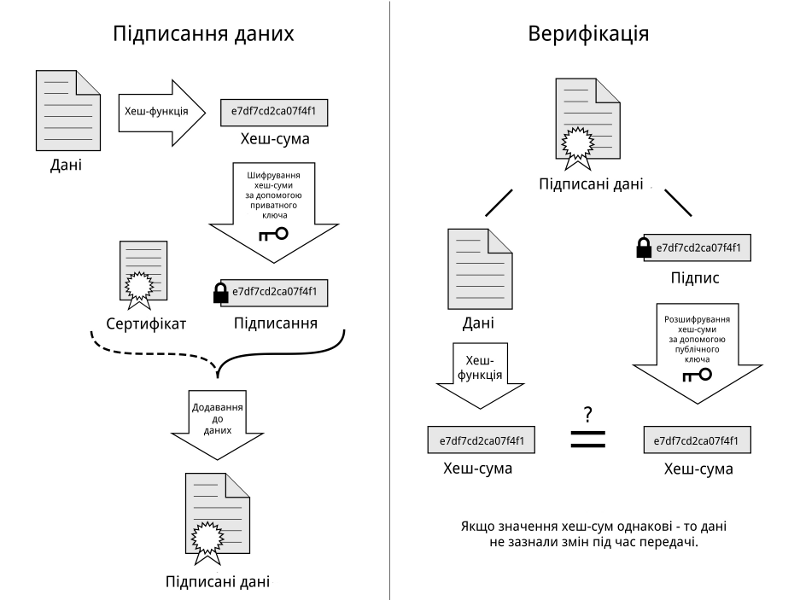
**Цифровий підпис –** наслідок еволюції криптографії до рівня, коли з допомогою відкритого і закритого ключа, а також певної хеш-функції можна бути впевненим, що певний файл створено конкретним автором, і жодних змін під час його переміщення чи надсилання не було внесено.(рис. 2.3)



*Рис 2.3 – Ілюстрація підписання даних електронним цифровим підписом*

Електронний цифровий підпис має можливість поставити будь-яка особа, яка його попередньо згенерувала, разом із копією необхідного цифрового ключа.

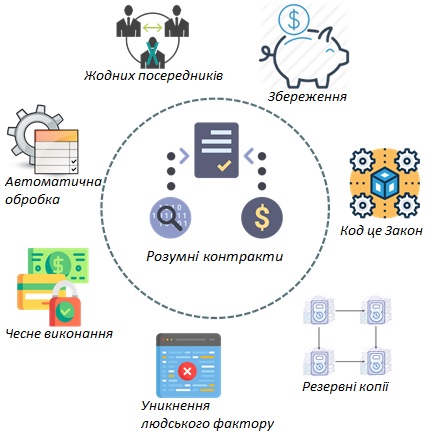
Окрім того, цифровий підпис, як і фізичний, виступає абсолютною гарантією і підтвердженням того, що відправник не має жодної можливості чи права відмовитися від свого авторства щодо підписаного ним файлу чи певної інформації. На рисунку 2.4 зображено процес верифікації даних, підписаних із використанням технології ЕЦП.



*Рис. 2.4 Верифікація даних, що підписані ЕЦП.*

**2.1.4 Розумні контракти**

Розумні контракти – це певний вид угод, який представляють собою зашифровані математичні алгоритми. Підписання, редагування, доповнення або аннуляція контрактів можлива лише з використанням комп’ютерних програм в рамках глобального Інтернету.(рис. 2.5)



*Рис 2.5 – Графічне умовне представлення розумних контрактів*

Розумні контракти це можливість врегулювання і контролю відносин сторін шляхом закріплення їх виражених побажань, домовленостей чи обов’язків у певному коді, що може бути зчитаним та розшифрованим за допомогою комп’ютера або сервера.

Блок-ланцюг транзакцій на основі розумного контракту є доступним для абсолютно всіх користувачів цього блоку, що має за мету максимально ускладнити будь-які зміни в історії і блоках, та зробити втручання ззовні неможливим.

Розумні контракти є певною мережею, в якій всі блоки так чи інакше пов’язані між собою. Всі дії користувачів із підписання, виконання, редагування чи завершення контракту записуються у реєстр журналу, який постійно розтиражовується у блоки. Оскільки існує багато копій поточного стану системи, яка, окрім кількісного фактору збереження даних, має криптографічну систему захисту, користувачі можуть бути певні, що їх дані у безпеці.

Розумні контракти є одним з найбільш надійних засобів захисту від підробок даних, вставляння шкідливого коду без перевірки і виконання на всіх вузлах системи.

**2.2 Шифрувальні методи та алгоритми**

Криптографічний захист має ключову роль у безпеці даних користувачів. Якою б складною не була інфраструктура системи, скільки б копій її сховища не існувало – якщо вона не достатньо надійно шифрує дані, вона не варта нічого.

Шифр Цезаря справедливо можна вважати одним з засновників сучасної криптографії. Хоча теперішні алгоритми у сотні разів складніші, проте ідея все та ж – щоб інформацію знали лише певні люди – її необхідно зашифрувати. Шифрований текст може прочитати лише той, хто знає ключ.

На даний момент всі шифри поділяються на симетричні та асиметричні.

До шифрів, які активно використовуються у криптографії в сучасні часи, віднесено:

* RSA (абревіатура з прізвищ Rivest, Shamir та Adlerman)
* DES (Data Encryption Standard)
* Triple DES (потрійний алгоритм DES)
* AES (Advanced Encryption Standard)

**RSA** – Шифрувальний алгоритм, який використовується за допомогою відкритого ключа, і базується на обчислювальній складності задачі факторизації великих чисел.

Цей алгоритм є придатним для шифрування звичайних даних, і для використання у генеруванні цифрового підпису. Він часто використовується у багатьох криптографічних програмних продуктах.

Алгоритм RSA складається із чотирьох основних частин:

* Генерація ключів
* Їх шифрування
* Їх розповсюдження
* Розшифрування шифротексту

RSA є алгоритмом однобічної перестановки, що на практиці означає те, що навіть перехопивши шифрограму, неможливо отримати відкритий текст, не знаючи секретного ключа.

**DES** – один із симетричних криптографічних алгоритмів шифрування даних, розроблений для уряду США, що до кінця 90-х років перебував у широкому міжнародному застосуванні.

DES є алгоритмом, що використовує шифрування блоками по 64 біти – на вхід і на вихід йде блок того ж розміру, зашифрований 56 – бітним ключем.

В основі алгоритму лежать стандартні криптографічні операції перемішування та підстановки, разом із пораундним виконанням.

На теперішній час шифр вважається ненадійним, оскільки розмір ключа є надто малим для того, щоб гарантувати захист від лобової атаки за методом послідовного перебору.

**Triple DES –** це не окремий різновид, а швидше, нестандартний режим, у якому дані шифруються з допомогою алгоритму DES.

Його особливостями вважаються втричі більша довжина ключа, яка вдвічі підвищує стійкість алгоритму порівняно із первинною реалізацією.

Однак, такі недоліки як низька швидкість шифрування і аналогічна до стандартного DES важкість програмної реалізації привела до того, що в реальності Triple DES поступається алгоритму AES.

**AES** – один із найбільш сильних представників симетричного блочного шифрування, із розміром блоку у 128 біт, і ключами, які можуть варіюватись у розмірі (128, 192 або 256 біт). Є прийнятим американським урядом як стандарт шифрування.

Для стандартного ключа у 128 біт алгоритм виконує 10 раундів, в яких послідовно запускаються такі операції:

* Заміна байтів у блоках
* Зсув рядків блоків ( розмір блока 4х4)
* Змішування колонок
* Операція XOR простого ключа ( кожен байт блоку піддається операції XOR з кожним байтом Round Key

**2.3 Інструменти реалізації**

Окрім вибору доречного алгоритму шифрування, важливе місце у розробці системи контролю безпечності займають такі пункти як вибір мови програмування, допоміжних фреймворків, вибір технології комункації модулів між собою, визначення необхідних характеристик та реалізації бази даних та системи управління нею, дослідження програмного продукту на баги, проблемні місця, вибір системи автоматичної збірки та розгортання системи.

**2.3.1 Мова програмування серверної частини.**

Для реалізації серверного модуля розробленого програмного продукту, обрано мову Java із чималою історією використання в сфері розроблення бізнес-платформ, що славляться своєю стабільністю, надійністю, портативністю та простотою у підтримці.

Java є об’єктно – орієнтованою мовою. Основу ООП, як парадигми, складають чотири принципи: інкапсуляція, успадкування, поліморфізм та абстракція. Вона втілює в життя всі ці принципи, дозволяючи розробникам без зайвого коду і надмірних налаштувань створювати потужні програми.

Java побудована на основі віртуальної машини, що на практиці надає потужну кросплатформенність, доступну не лише для виконання програми на будь – якому компютері, але й сервері, мобільному телефоні чи іншому пристрої, на якому встановлена і запущена віртуальна машина java.

Кросплатформенність реалізується таким чином: спершу програма на java компілюється у проміжний байт-код, який зчитується віртуальною машиною, і вже залежно від платформи, віртуальна машина інтерпретує байт – код у машинний код.

Основну частину синтаксису Java запозичена з С++. Однак, порівняно з ним , у Java є ряд серйозних переваг:

* Неможливість створити глобальну змінну, що не належить жодному з класів.
* Зарезервоване, але не дозволене до використання ключове слово goto, яке приносить в програму набагато більше плутанини і варіантів для помилок, ніж користі від використання.
* Автоматична збірка сміття ( Garbage collector ).

Cервіс, який слідкує за всіма створеними об’єктами у програмі, і проводить ряд перевірок над ними. З його допомогою програміста позбавили необхідності слідкувати за станом пам’яті, і вручну видаляти об’єкти. Якщо протягом одного циклу збірки сміття на об’єкт немає жодного прямого посилання, а якщо і є, та вони мають порожнє значення, тоді об’єкт видаляється із пулу об’єктів та з пам’яті загалом.

* Відсутність зайвих структур типу struct і union, всі необхідні характеристики використано в класах.
* Точність базових типів мови є сталою, і не залежить від того, скількох бітна система використовується для запуску програми.
* Неможливість звертання до видаленого об’єкта через вказівник, оскільки, допоки на об’єкт існує посилання в програмі, він не видаляється з пам’яті.
* Відсутність множинного наслідування, яка принесло більше проблем, ніж можливостей у розробці. Неможлива ситуація конфлікту між членами класу, які успадковуються від базових класів.
* Java не має підтримки вказівників, що значно спрощує взаємодію з об’єктами і позбавляє від необхідності постійно аналізувати, яким чином необхідно передавати об’єкт, і звільнює час розробника і увагу для виконання важливіших завдань.
* В Java вбудовано стандартні значення для створюваних об’єктів, для integer – 0, boolean – false, непримітивний об’єкт – null.
* Вбудована перевірка на звертання до неіснуючого об’єкту масиву – в таких випадках генерується exception.

У Java вбудовано строгу типізацію – вона дозволяє уникнути переважної більшості помилок при роботі з різними типами даних, і гарантує неможливість запуску програми із невідповідними типами.

Java, як мова програмування, також приваблива тим, що за роки її існування навколо неї зібралося надзвичайно обширне співтовариство програмістів, які працюють із нею. Сотні сайтів в мережі Інтернет покликані допомагати розробникам із різноманітними питаннями або проблемами.

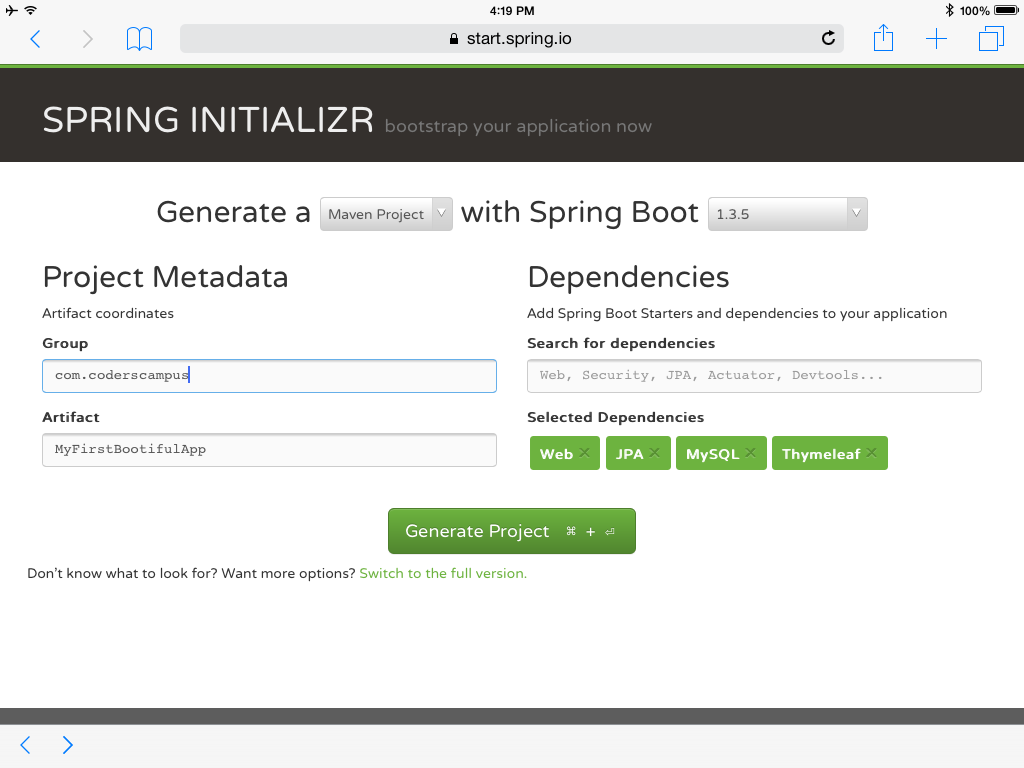
Майже всі завдання уже колись були вирішені кимось з мережі, і можливість пізнавати різні підходи програмування в межах однієї мови надає широкі можливості для саморозвитку.

**2.3.2 Фреймворки для розробки серверної частини**

Для Java існує неймовірно велика кількість різноманітних бібліотек, фреймворків, готових рішень і систем, які в рази спрощують розробку різноманітних програмних продуктів на будь-якому рівні проекту.

Фреймворки вже стали буденністю програміста, і навряд чи в теперішній час існує програміст, який не використовував би як мінімум двох фреймворків у своєму проекті.

Одним із широко застосовуваних фреймворків у Java є сімейство Spring Framework. Це цілий набір із відточених модулів, які можуть бути застосовані для вирішення різноманітних задач.(рис. 2.6)



*Рис. 2.6 – Інтерфейс Spring Framework Initializer*

Найчастіше застосовувані модулі Spring:

* **Spring Core**

Базовий модуль, на якому побудовані всі решта.

Відповідає за створення, конфігурацію та управління бінами (об’єктами, що створюються і надаються в Spring). Біни можна налаштовувати використовуючи один з трьох способів, або змішуючи їх , що не рекомендовано практикувати:

* Конфігурація через XML-файл (вважається застарілою)
* Конфігурація з допомогою аннотацій (метаданих). На теперішній час найбільш поширений спосіб.
* Конфігурація в java-файлі. Використовується, коли необхідно надати біну певних нестандартних властивостей.

Для ініціалізації оголошених бінів в тепершній час використовується аннотація *@Autowired*, яка розміщується над полем, що має бути проініціалізоване, або над конструктором класу. Якщо існує кілька варіантів реалізації одного біна, тоді використовується аннотація @Qualifier, для задання унікального імені для біна.

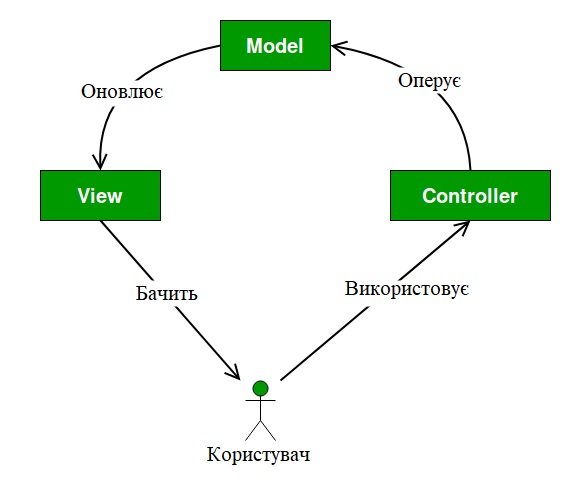
* **Spring Security.**

Модуль, який дозволяє налаштовувати дозволи доступу від різноманітних сторонніх API . В основному, прописуются URL-адреси, з яких можна надсилати запити на наш сервіс, а також, на які ендпоінти можна надсилати http-запити без автентифікації, а на які обов’язково необхідний авторизований користувач. Spring Security дозволяє фактично двома класами конфігурації налаштувати всю безпеку надсилання та приймання запитів сервлетом. Робиться це за допомогою розширення класу WebSecurityConfigurerAdapter, а також наслідуванням класу Filter.

* **Spring MVС**

Керує прийомом і надсиланням http запитів та відповідей. Він дозволяє за допомогою набору аннотацій замапити певні URL-адреси на необхідні точки входу в програму, вказати які HTTP-методи можуть бути прийнятими на конретний вхід, який медіа-тип запиту та отримувати необхідні параметри в якості URL-змінних.

В основі Spring MVC є використання паттерну програмування MVC – Model, View, Controller. Ці поняття дозволяють розбити програму на відповідні логічні рівні, що спрощує роботу з нею і надає більше гнучкості при розробці. На рисунку 2.7 зображено умовну схему роботи паттерну MVC.



*Рис 2.7 Принцип роботи шаблону MVC.*

**Model** – це сутність об’єкта, яким оперують. Це дані, що надсилаються і надходять. В основному моделями слугують класи-домени, в яких немає жодної бізнес-логіки, окрім полів та методів, що їх ініціалізують та читають. Це дозволяє їм бути максимально легкими, здебільшого вся логіка роботи з об’єктами-доменами виноситься в сервіс-рівень.

**View** – відображення, здебільшого це певна JSP( Java Servlet Page ) сторінка, яка містить http-шаблон, певні CSS-стилі та скрипти для виконання їх на стороні клієнта.

**Controller** – сервіс, що отримує запити, розподіляє їх відповідно до HTTP-методу та URL по своїх методах, та після обробки, повертає користувачу певний View, відповідно до запиту.

* **Spring Validation.**

Легкий, у сенсі об’єму даних, модуль, що дозволяє, використовуючи аннотаційний підхід, позначати поля і класи, які необхідно перевіряти на дотримання певних крайніх умов. Дозволяє із легкістю створювати власні валідаційні аннотації, які часто виручають в умовах необхідності перевірки кількох умов на різних полях об’єкту.

Для створення власної реалізації валідатора достатньо успадкувати і перевизначити метод isValid(), а також створити відповідну аннотацію, наявність якої викликатиме відповідну перевірку.

* **Spring Cloud**

Потужна платформа для управління розподіленими системами. Дозволяє віддалено піднімати сервери, сервіси, здійснювати їх налаштування, використовувати можливості інтелектуальної маршрутизації, налаштовувати перехоплення запитів у мікропроксі-сервісах, використання глобального замикання додатків, дослідження, моніторинг та управління станом кластера.

* **Spring Data JPA**

В основі проекту лежить паттерн репозиторію, який використовується як об’єкт доступу до даних ( DAO – Data Access Object ). З допомогою імплементації стандартних JPA-інтерфейсів користувач та розробник одразу ж отримує можливість використовувати потужності стандартних CRUD ( Create Red Update Delete ) операцій. Основна перевага - можливість для створення методів, які звертатимуться до БД, не пишучи власноруч реалізацію – достатньо за правилами, які пропонуються в автопідстановці середовища розробки програмного продукту, прописати назву методу, який має виконувати необхідну операцію, і на основі назви цього методу Spring спершу встановлює проксі для перехоплення виклику, і скеровує його у автоматично згенерований метод, який створює запит на основі назви кастомного методу.

Для фахівців в сфері проектування запитів в БД надано можливість самим його прописувати, – достатньо оголосити аннотацію @NamedQuery, і при зверненні до даного методу інтерфейсу Спрінг буде використовувати оригінальний запит, а не свою вбудовану реалізацію.

* **Spring Boot**.

В його основі лежить максимально можлива кількість автоматичних конфігурацій для різноманітних модулів проекту, що дозволяє фактично у межах кількох хвилин запустити проект, без необхідності налаштовувати з’єднання, конфігурації класів-бінів і інших властивостей проекту. Для кожного існуючого модуля Spring у ньому знаходяться файли автоматичної конфігурації. Він є визнаний спільнотою програмістів як один з стандартних автоматичних налаштовувачів, що дозволяє швидко розгортати проект із мінімальними затратами часу. Його головна особливість – можливість користуватися без критичної необхідності розбиратися у всіх налаштуваннях платформи. Для більшості проектів вистачає автоматичних налаштувань, а за необхідності, не складає проблем знайти і застосувати необхідну ручну конфігурацію.

Окрім самої розробки, Spring надає підтримку у ще одному важливому аспекті розробки програмних продуктів – тестуванні.

**2.3.3 Фреймворки для тестування**

Промисловим стандартом у тестуванні визнано Junit Framework, який надає широкі можливості для модульного та інтеграційного тестування системи.

Основним рушієм тестувальних можливостей фреймворку є набір із кількох потужних аннотацій або їх сімейств.

**@RunWith(Клас.class)** – визначає, із яким розширенням необхідно запускати тестовий клас. Розширення можуть містити інтеграцію з базами даними, або ж повністю ізольовувати елементи тестового класу від інших компонентів програмного продукту.

**@Test** – аннотація, яка займає перше безумовне місце за частотою використання у тестових модулях. Позначає метод як той, що необхідно запустити і протестувати. Під час виконання тесту виводиться в консоль додаткова інформація ходу програми, виводяться логи запуску певних програм, делегування завантажень необхідних методів, записи завантаження скриптів бази даних та інша профілактична інформація. Якщо тест провалюється, у більшості випадків виводиться стек-трейс виникнення помилки, а якщо такої помилки немає, а просто результат не сходиться з очікувани , то виводиться наявний результат виконання поряд із очікуваним.

**@Test (expected = ExceptionType.class)**

Спеціальний варіант тестової аннотації для перевірки роботи тих ситуацій, коли в результаті виконання певної умови програми має бути викинуто виняткову ситуацію (exception). Зручна можливість тестувати винятки без необхідності заключення виклику методу в try…catch і наступною ручною перевіркою відповідності винятку до очікуваного.

**@BeforeAny** – аннотація, що розміщується над певним методом, що має бути викликаним перед виконанням кожного тесту з певного класу.

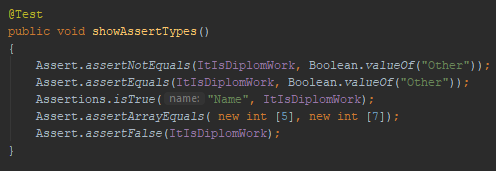
**@BeforeAll** – ініціалізаційний метод, що має бути запущений один раз перед виконанням всіх тестів (наприклад, для заповнення таблиць бази даних або створення псевдо-об’єктів).

**@AfterAny** – код, що повинен бути виконаним після завершення тесту.

**@AfterAll** – метод, який виконується перед виходом із тестового класу, дозволяє виконати завершальні операції ( відновлення попереднього стану системи, звільнення ресурсів, очищення всіх кешів ).

Всі тестові фреймворки використовують службове слово java – assert, яке перевіряє істинність переданого їй параметру, і, якщо він фальшивий, викидає AssertionError – що означає негайний вихід та завершення програми.

Зараз є можливість, з допомогою бібліотек Assert, Assertions, Assert4J, TestCase та ін. перевіряти передані параметри на рівність, нерівність, на правдивість твердження, перевіряти списки на наявність певного елементу, порівнювати цілі колекції елементів між собою, об’єкти, їх поля, параметри. Зрештою, існують і банальні перевірки на більшість чи меншість одного числа відносно іншого. На скріншоті 2.8 зображено застосування різноманітних типів ассертів.



*Рис 2.8 Використання різних типів перевірок.*

Всі ці інструменти, при застосуванні відповідно з найкращими теперішніми практиками тестування, надають можливість розробнику постійно здійнювати контроль якості системи, що неодмінно призводить до загального покращення стану проекту, і можливості бути впевненим у роботі частин проекту окремо та їх взаємодії між собою.

**2.3.4 Мова програмування клієнтської частини**

Вибір тут не настільки широкий, як для серверної частини, проте, мови сайтів важко порівнювати між собою: деякі застосовуються водночас і в серверній розробці, і в розробці клієнтських застосунків, інші виконують конкретні завдання лише на стороні користувача.

З допомогою рейтингу популярності мов програмування обрано JavaScript.

Його синтаксис схожий до Java, проте, це єдина схожість. JS – скриптова мова програмування, що дозволяє оперувати даними сайту, його видозмінювати, проводити певні обрахунки, генерувати графіки. Можливості JavaScript також дозволяють писати серверну частину з допомогою фреймворка Node.js, а також він може бути використаний для написання native-додатків – коли код, написаний на JS, інтерпретується в Kotlin або Swift з метою бути портованим на відповідну мобільну ОС.

В JS є псевдо-підтримка об’єктно-орієнтованого програмування, а також можливість використовувати модулі функціонального програмування.

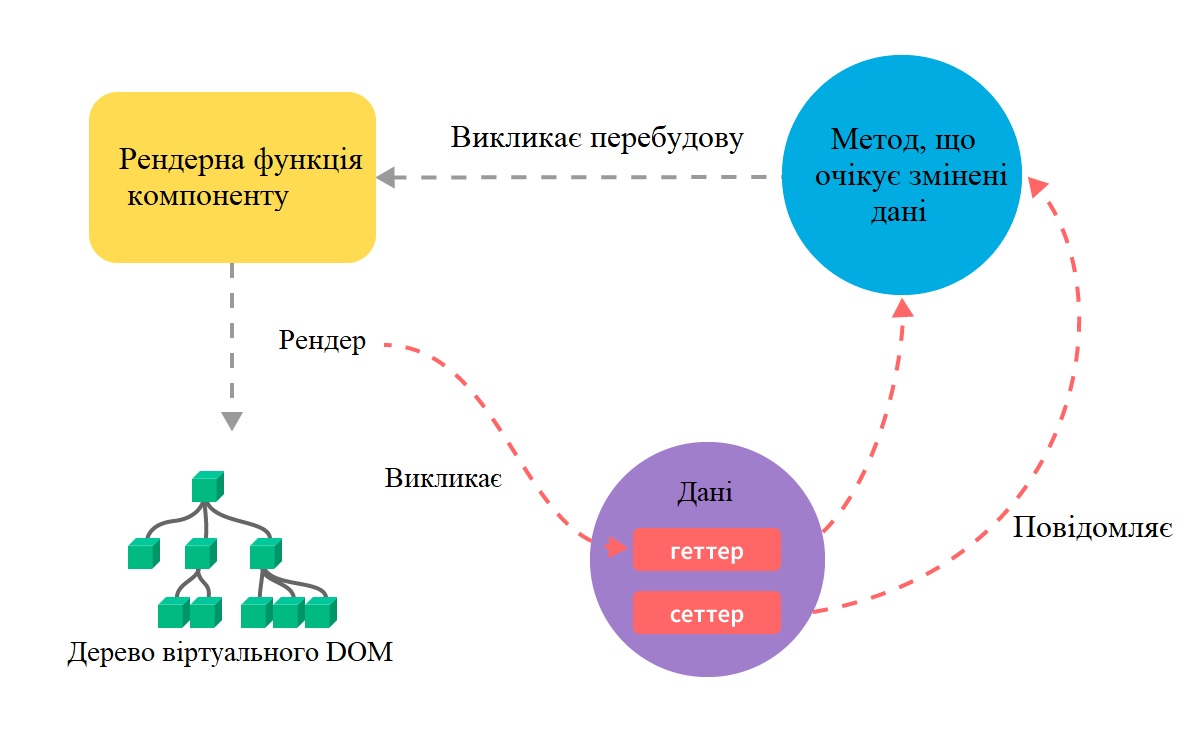
Однією з переваг JS перед іншими мовами для програмування сайтів є надзвичайно велика кодова база бібліотек та фреймворків, серед яких можна надзичайно швидко знайти і обрати необхідний для конкретного проекту.

**2.3.5 Фреймворк для клієнтської частини**

Серед десятків варіантів було обрано **Vue.js,** автором якого став один із розробників **Angular.js.** Перевагою **Vue.js** є його низький поріг входу, що дозволяє швидко розпочати розробку з його допомогою, а також надзвичайно детальна та перекладена багатьма мовами офіційна документація, яка містить всі необхідні теоретичні відомості про фреймворк, приклади роботи та найкращі варіанти застосування на реальних прикладах.

**Vue.js** є фреймворком, який працює за реактивним принципом, що означає майже миттєве оновлення даних на сторінці, після того, як вони надійшли на клієнтську частину.

Він розумно спроектований – завдяки наявності віртуального DOM, з реальним DOM проводиться мінімально необхідна кількість операцій, оновлюючи лише ті дані, що були змінені – статичні дані залишаються без змін, і на них не витрачаються додаткові ресурси комп’ютера. На рисунку 2.9 зображено схему роботи віртуального DOM.



*Рис 2.9 Схема роботи віртуального DOM*



*Рис 2.10 Компонентна архітектура Vue.js*

Завдяки компонентній архітектурі, **Vue.js** дозволяє будувати гнучні та стабільні програми, якими легко керувати, а ще легше розробляти – велика кількість вбудованих корисних бібліотек під рукою в програміста в будь-який час. Одна з найбільш перспективних розробок на даний час, через що її й було обрано.

**2.4 Найкращі підходи у проектуванні та архітектурі систем.**

Архітектура програмного продукту – це те, що може вирізнити його з-поміж десятків аналогічних рішень, і виступити останнім аргументом для замовника, який розглядає програму серед схожих.

Програма із продуманою архітектурою є гнучкою, що означає легкість внесення змін. Її код має низький рівень зв’язності, що зменшує ймовірність порушення роботи одного модуля програми, при внесенні змін у інший.

Зрештою, програмний продукт із продуманою архітектурою має використовувати та розвивати такі підходи як SOLID, DRY, YAGNI та ін.

Розглянемо їх конкретніше.

**2.4.1 SOLID**

Це абревіатура від 5 принципів, яка включає такі поняття як:

* *Single responsibility principle (Принцип єдиної відповідальності).*

Кожен модуль та елемент програми мають мати єдине призначення. Об’єкти роботи модуля контроллера можуть бути глобальнішими, ніж, до прикладу, сервісу, проте, вони мають бути об’єднані спільною предметною областю. Щодо сервісів, вони мають реалізовувати лише одну конкретну логіку для певного типу об’єктів. Не можна допускати ситуації, коли в сервісі виконуються різного типу операції, оскільки він стає прив’язаним до багатьох частин коду, використовується в різних частинах проекту, що приводить до ускладнення роботи з ним, погіршення загальної якості коду як такої, ускладнення процесу тестування сервісу, виникнення багів та нестандартних ситуацій. На першому етапі створення такого сервісу може виглядати виправданим, проте, це лише до етапу внесення будь-яких змін. Все, що можна розділити на окремі логічні частини, сервіси або модулі, має бути розділене.

* *Open/Closed principle (Принцип відкритості та закритості)*

Цей принцип несе в собі представлення однієї з парадигм об’єктно-орієнтованого програмування – інкапсуляцію.

В теперішній час дотримання цього принципу є одним із обов’язкових етапів при розробці програмних продуктів, до яких матимуть доступ треті особи.

В його основі необхідність позбавлення користувача можливості напряму звертатись до полів певних об’єктів, для уникнення їх непередбаченої видозміни або видалення. З даною метою у Java існують 4 типи модифікаторів доступу, з допомогою яких програміст задає рівень доступу до певного поля. Хорошою практикою вважається робити приватними усі поля, які можуть бути зміненими, і мати до них доступ лише через спеціальні методи – гетери та сеттери, якщо такі є передбачені логікою класу. Для фінальних змінних, особливо статичних ( які виступають у ролі констант) нормальною практикою вважається використання публічного модифікатора доступу, оскільки змінити їх неможливо.

Принцип стосується не лише полів, але й класів, говорячи про те, що вони мають бути відкритими до розширення, але закритими до змін.

Тобто наступним розробникам можна надати можливість унаслідування класу, але не дозволяти його змінювати у його первинній імплементації.

* *Liskov substitution principle (Принцип підстановки Лісков)*

Прицип, що пропагує поняття – об’єкти в програмі можуть бути заміненими їхніми нащадками без зміни програмного коду.

На практиці це означає, що якщо спершу у програмі використовувся загальний батьківський клас, який згодом стало необхідно замінити на реалізацію нащадка, то для програміста не має виникати жодних проблем при такій заміні – достатньо замінити тип об’єкта та додати імпорт на його клас. Для правильно реалізованого програмного коду цього має бути достатньо – підміна відбувається за кілька секунд.

* Interface segregation principle (Принцип розділення інтерфейсів)

Інтерфейси відіграють важливу роль у програмному продукті – вони виступають певними провайдерами необхідної поведінки або логіки. З часом розроблення продукту інтерфейси мають властивість розростатись, ставати більш загальними та вміщувати логіку різного плану – це погана ситуація для структурованості коду та його якості, яка вирішується достатньо простим чином – краще мати багато спеціалізованих по дуже вузьких критеріях інтерфейсів, ніж один збірний. Слідування такому підходу дозволяє підтримувати чистоту коду проекту та залишити після себе логічний продукт, який легко підтримувати та розширювати.

* Dependency inversion principle (Принцип інверсії залежностей)

Всі залежності всередині програмного продукту повинні будуватись на основі абстракцій, що не мають бути залежними від деталей та особливостей реалізації. Модулі рівні, що знаходяться вище по ієрархії програмного продукту, не мають залежати від тих, що знаходяться нижче.

**2.4.2 DRY**

Don`t repeat yourself – не повторяй себе самого.

Достатньо простий принцип для розуміння, але буває зрідка складним для реалізації у погано спроектованих системах. Суть його проста – не варто в різних місцях програмного продукту заново писати одну і ту ж логіку. Як вихід із такої ситуації, логіку необхідно винести в окремий метод, і замінити всі можливі місця в коді викликом цього методу. Програма таким чином стає читабельнішою, не має плутанини між частинами коду, що мають схожу функціональність.

**2.4.3 YAGNI**

You ain`t gonna need it – тобі це не знадобиться.

Принцип, який часто порушується людьми з малим досвідом у сфері програмування, точніше тими, що приступають до розробки без чітко окресленого завдання.

Суть помилки, якої припускають ці розробники, проста. Захопившись виконанням завдання, вони починають розробляти методи, або й цілі модулі такого коду, який зараз не є необхідним, але, за їх ходом думок, може точно знадобитись в майбутньому. Такий підхід забирає багато часу і ресурсів у розробника програмного продукту, та грошей – у замовника.

Для виконання принципу необхідно чітко слідувати поставленому завданню і не вносити додаткових правок у систему, не порадившись із більш досвідченими розробниками.

**2.4.4 KISS**

Keep it short and simple – збережи це коротким та простим

Під час розробки часто виникає ситуація, коли перед розробником стоїть задача, для вирішення якої існує кілька способів, один з яким може виглядати довшим за кількістю коду, але легшим для розуміння, а інший дозволяє двома-трьома рядками вирішити ваше завдання. У такі моменти хороший розробник повинен пам’ятати, що він завжди не буде на цьому проекті – будуть програмісти після нього, для яких його рішення не виглядатиму абсолютного логічними або виправданими в конкретній ситуації. Для уникнення непорозумінь, під час розробки і проектування практикувалось ставлення себе на місце іншого програміста, що дозволяло майже одразу виявити, чи достатньо певний метод зрозумілий, і чи виконує він те, що має.

**2.5.1 Клієнт-серверна архітектура**

Це одна з головних концепцій у розробці розподілених мережевих програмних продуктів. Передбачає взаємодію між клієнтом та сервером з використанням в якості комунікаційного шлюзу мережі Інтернет, та надсилання даних один одному. На рис 3.1 зображено один з варіантів реалізації клієнт-серверної архітектури.



*Рис.3.1 Реалізація клієнт-серверної архітектури з кількома клієнтами.*

В цій концепції існують такі ключові компоненти:

* Набір серверів або сервісів, що виступають провайдерами певних даних або послуг програмам, що викликають їх.
* Набір клієнтських застосунків, які використовують наявні сервери або сервіси.
* Комп’ютерна мережа, яка контролює з’єднання та надає змогу сторонам взаємодіяти між собою.

Сервери у стандартному підході не залежать один від одного. Проте, вони можуть викликати певні функції один одного, якщо один сервіс був розділений на кілька мікросервісів. Клієнтські застосунки зобов’язані працювати паралельно та водночас незалежно, не втручаючись в роботу один одного.

Фактично стандартом на даний час є ситуація, коли сервер обробляє запити від різних клієнтських програм, а клієнтська програма звертається до різних сервісів, залежно від ступені завантаженості сервера. Головне правило – клієнтські застосунки можуть і навіть мають знати про доступні системі сервери, проте вони ніяким чином не мають мати жодних даних, що відносяться до інших клієнтських застосунків.

Існує поділ на два види клієнт-серверної архітектури, в залежності від кількості логічних рівнів операцій: дво- та трьохрівневі.

Для **дворівневої** клієнт-серверної архітектури передбачено взаємодію лише двох модулів системи – клієнтського застосунку та сервера.

Залежно від того, де знаходиться функціонал обробки, формування та управління інформацією, розрізняють тонкі та товсті клієнтські застосунки.

Здебільшого, товстими клієнтами називають пристрої з обмеженою продуктивністю - планшетні комп’ютери, мобільні телефони (кнопочні) та ін.

Для **трьох-рівневої** системи виокремлено такі види функціональних частин:

* Представлення, тобто інтерфейс користувача, що відповідає за зручну комунікацію користувача із серверною частиною та формування зручного представлення відповіді сервера.
* Бізнес-логіка програмного продукту. Рівень, на якому реалізовано всі основні операції із обробки даних та їх перетворення.
* Управління збереженою інформацією, тобто певна база даних та система управління нею. Забезпечує можливість доступу до даних авторизованим користувачам, і зберігання будь-якого типу інформації.

Основною відмінністю трирівневої клієнт-серверної архітектури від дворівневої є виділення процесу обробки даних (бізнес-логіки) як окремого модуля програмної системи. Це привело до спрощення самих програмних продуктів та розробки в цілому.

**2.5.2 Підходи до проектування серверної частини системи**

Серверна частина продукту поділяється на такі логічні модулі:

* Класи простих об’єктів (POJO)
* Сервіси
* Менеджери
* Репозиторії
* Процесори
* Контроллери

Кожен із цих модулів відповідає за свою логічну частину в роботі системи.

Розпочнемо із фундаменту – класи простих об’єктів, або **POJO**( Plain Old Java Object ) класи. Використовуються як каркас об’єктів, є представленнями речей з предметної області, в даному випадку існують такі класи як User, Company, Document, Judge, Account, Bank, Employee, Employer, Authentication та ін. Дані об’єкти використовуються як всередині сервера, так і виступають представленнями документів, що зберігаються в базі даних, і використовуються для прийняття та відправлення даних із клієнтської сторони проекту.

Наступний рівень – **сервіси**. Фактично, саме в сервісах знаходиться основна бізнес-логіка програмного продукту. Для створення і використання сервісу хорошим навиком вважається написання для нього інтерфейсу, який буде надавати поведінку сервісу, і використовувати його лише в якості імплементації інтерфейсу. Використовуючи такий підхід до розробки сервісного рівня, програмному продукту надається вищий рівень гнучкості, а також зменшується зв’язність між модулями і класами одного чи різних рівнів абстракції. Програміст має можливість створювати різну імплементацію до певної загальної поведінки, що на практиці виступає втіленням поліморфізмом. А при використанні Spring Core, розробнику не треба навіть вручну створювати об’єкти певних імплементацій – достатньо оголосити в сервісі полем загальний для певної поведінки інтерфейс, а клас-імплементація, з допомогою динамічної типізації, буде підібрано системою під час виконання програмного коду.

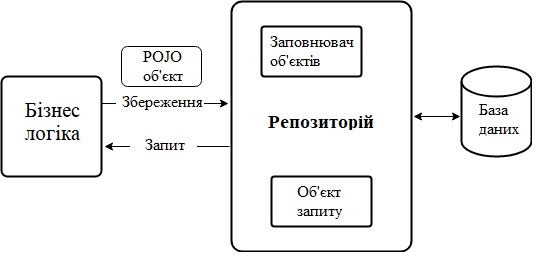
Оскільки при розробці програмного продукту, для хорошої якості коду створюється багато розділених по категоріях і особливостях сервісів, настає момент, коли їх кількість стає проблемою, що ускладнює життя програмісту. Для таких випадків передбачено підхід, за якого сервіси, які працюють з однаковими об’єктами або мають схожий функціонал, не використовують напряму, а вбудовують їх у клас-**менеджер**, який постачає нам необхідну реалізацію відповідно до необхідностей та потреб програмного продукту.

При практичній реалізації цього підходу на допомогу розробнику приходять структурні паттерни проектування проектів.

Будівельник, фасад або декоратор – відповідно до цілей розробника, реалізовується один або кілька паттернів одночасно. Переважна більшість програмістів обирає паттерн фактично в процесі реалізації рішення задачі, орієнтуючись на зрозумілість підходу та його відповідність вимогам завданням, поставленим перед розробником.

З допомогою менеджерів, основні об’єкти сервісів, якими ми оперуємо, знаходяться і створюються в єдиному місці програми, дозволяючи нам точно контролювати кількість цих об’єктів, а також здійснювати їх конфігурацію без зайвих дій.

**Репозиторій** – це концепція зі зберігання набору даних для сутностей різних типів. Простішими словами, репозиторій – це паттерн, що опікується коллекціями об’єктів, які можна фільтрувати, видозмінювати, повертати користувачу відповідно до умов, закладених у його запиті. У паттерні репозиторію для розробника неважливо, де і яким чином зберігаються об’єкти, з якими він працює – це є компонентом реалізації, і інтерфейс репозиторію не надає жодної інформації про ту ж БД, оскільки ззовні всі методи репозиторію мають виглядати і працювати одним чином, для максимальної простоти роботи з ними, а деталі реалізації цих методів залишаються всередині системи. Налаштування з’єднання із базою залишається прерогативою розробника, проте в цьому шаблоні це не несе додаткових змін у реалізований метод роботи з даними. На рисунку 3.2 зображено схематичне представлення роботи шаблону репозиторію.



*Рис 3.2 – Принцип роботи шаблону репозиторію*

На прикладі конкретно цього програмного продукту використано за основу реалізовані JPA – репозиторії з зовнішнього програмного модуля – Spring Data. В цих реалізаціях вже прописані стандартні CRUD-методи (для читання, оновлення, створення та видалення об’єктів). Усі решта необхідних методів, як, до прикладу, пошук користувача за його ідентифікатором, прописано з допомогою «магічних» методів Spring Data JPA, які, як було розглянуто вище, генерують необхідні методи доступу та повернення даних суто за назвою методу.

**Процесори** виступають допоміжними класами для сервісів. Суть процесорів у тому, що вони відповідають за дуже вузьку частину роботи з даними, здебільшого в контексті лише одного сервісу. Фактично, процесори використовують задля максимального зменшення кількості приватних методів, які використовуються в сервісах, оскільки їх наявність захаращує код та робить можливість підтримки такого продукту важчою. Особливо важливу роль процесори відіграють під час всього процесу тестування, оскільки, по-перше, публічні методи набагато простіше тестувати, аніж приватні, а по-друге, наяність додаткового рівня абстракцій надає можливість створювати юніт-тести, які покривають дуже вузькі місця програми, і надають більшу впевненість у роботоздатності та правильності коду.

Процесор має застосовуватись лише як член одного сервісу. Якщо з’являється метод, через який один процесор використано у різних сервісах, відтак розробнику необхідно обдумати можливість винесення такого методу на вищий рівень того ж сервісу, або спрощення двох сервісів до спільного знаменника.

**Контроллери** це класи – обгортки для точок входу і виходу із програми.

Вони виступають реалізацією шаблону контроллера, метою якого і є надання користувачу та системі точок входу у сервіс, з допомогою яких можна із ним комунікувати. Приймають конкретні передбачені запити із клієнтської частини, використовуючи принцип REST.

**REST** (Represantational State Transfer) - підхід, який визначає, як розробник має оперувати і створювати API, щоб він був зручним і використовував HTTP-методи за призначенням. На вхід і вихід дані подаються у форматі JSON ( JavaScript Object Notation), що є фактично загальноприйнятим стандартом на теперішній момент.

У специфікації до REST вказано основні вимоги :

* За допомогою HTTP-GET запиту можна лише отримувати певні дані, в результаті цього запиту не має відбуватись жодних інших дій, окрім як повернення клієнту запитуваного ресурсу
* PUT-метод бажано використовувати для створення певних об’єктів або розміщення певних елементів системи уперше. Окрім URL та шапки методу, має ще й тіло, в якому можна передавати дані.
* POST-метод використовується в основному для надсилання чутливих даних в його тілі, і оновлення при цьому стану певного об’єкту чи ресурсу.
* DELETE, відповідно до простої логіки, має мати лише одне функціональне призначення – видалення об’єкту або його груп за певною умовою ( чи без неї, що не рекомендується )

Зручним для використання розробником при реалізації контролерів є модуль Spring MVC, який було використано у цьому програмному продукті. З допомогою задання над класом-контролером та методами вхідними точками анотацій з URL-адресами цих точок та дозволеними до використання HTTP-методами, цей рівень архітектури програмного продукту було реалізовано достатньо швидко.

**Висновки:**

В даному розділі розглянуто основні технології, які застосовуються в схожих програмних продуктах, і вже підтвердили свою надійність. Обрано мову програмування бек-енду та фронт-енду. Визначено основні фреймворки, які допоможуть зменшити час розробки продукту та спростити його. Представлено фреймворк для тестування, який використовується у переважній більшості enterprise-продуктів для контролю якості коду.

1. **Архітектура та проектування програмного продукту**

**3.1.1 Втілення архітектури та проектувальних шаблонів.**

Для проектування програмного продукту було обрано розробку відповідно до моделі екстремального програмування.

Ця модель є найбільш поширеною серед гнучких методологій розробки програмного забезпечення. Основними її цілями є підвищення якості програмних продуктів та високий рівень сприйняття наданих замовниками змін. Одним із визначником даної методології є проведення частих випусків (релізів) розроблюваного програмного забезпечення.

Інші характеристики даної методології включають в себе: парне програмування, часте проведення код рев’ю ( огляду програмного коду на якість, відсутність зайвих змін або можливих зародків помилок), обов’язкове модульне та інтеграційне тестування програмного продукту, високий рівень покриття тестами розроблюваних застосунків, уникнення створення зайвої функціональності.

При використанні екстремального програмування розроблюваний проект є набагато гнучкішим до змін, ніж використовуючи методології, що передбачають попереднє повне проектування системи.

Одними з її небагатьох мінусів є:

* Часта необхідність в рефакторингу
* Відсутність чіткої й зрозумілої специфікації та документації

Одначе, на противагу цим мінусам існують такі переваги:

* Легкий процес внесення змін у програмний продукт
* Відсутність надлишкового проектування
* Використання в розробці найбільш простих значень, принципів та методів.

Оскільки базова функціональність програми була визначена наперед, і майбутні зміни можливі завдяки гнучкому проектуванню, екстремальна модель зарекомендувала себе як хороше рішення для розробки продукту з часто змінними вимогами і обмеженим для її виконання часом.

4 базові активності, що використовуються при розробці програмних продуктів: програмування, тестування, слухання та дизайн.

В рамках тестувальної активності створюється багато модульних тестів, покликані покривати дрібні частини програми. Перевага їх у тому, що при достатній кількості і якості, вони можуть покрити майже весь програмний продукт.

Слухання – обов’язковий момент, завжди необхідно чути відгук замовника про систему, знати, яку точно йому необхідно створити бізнес-логіку для максимального успіху його компанії.

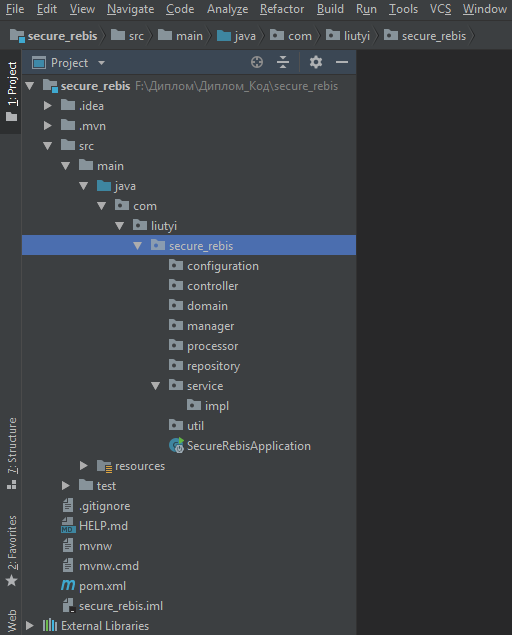
Дизайн – жодна хороша система не була побудована без мінімального проектування. Воно допомагає уникати високої зв’язності системи, спрощувати важкі методи і дозволяти легко вносити зміни в систему в майбутньому.

12 основних прийомів, що використовуються при екстремальному програмуванні:

* Розробка, яку спрямовує тестування ( Test driven development )
* Гра в планування ( planning game )
* Замовник завжди поруч з командою (Whole team, onsite customer)
* Парне програмування (Pair programing)
* Безперервна інтеграція (Continuous integration)
* Рефакторинг коду (Design improvement, refactoring)
* Часті релізи малого об’єму (Small Releases)
* Простота (Simple design)
* Метафора системи (System metaphor)
* Спільне володіння кодом (Collective code ownership)
* Стандарт програмування (Code convention)
* 40-годинний робочий тиждень (40hour week)

Глобально архітектуру проекту можна розглядати як клієнт-серверну, що є загальновизнаним стандартом у розробці таких програмних продуктів.

Загальна структура пакетів побудована наступним чином, зображеним на рисунку 3.3.



*Рис. 3.3 Структура пакетів серверної частини*

Дана структура забезпечує чітку модульність системи та надає можливість розробнику швидко переходити від розробки одного компонента до іншого.

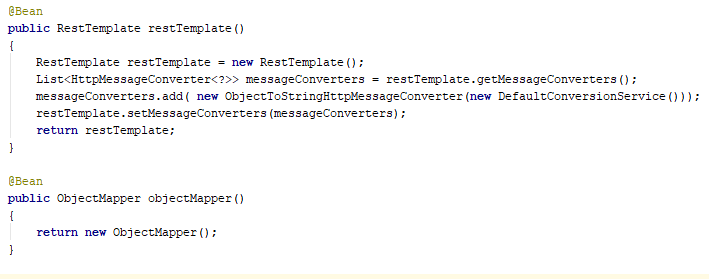
У пакеті конфігурації було розміщені такі файли як: ApplicationBeans, WebMvcConfig, SecurityConfig,

В класі ApplicationBeans використано анотацію @Configuration, яка слугує для Spring Boot міткою, що в даному класі знаходяться налаштування.

Серед власноруч налаштованих бінів: ObjectMapper, RestTemplate.

Для RestTemplate вручну додано в налаштування необхідний в роботі з складними об’єктами ObjectToStringHttpMessageConverter.

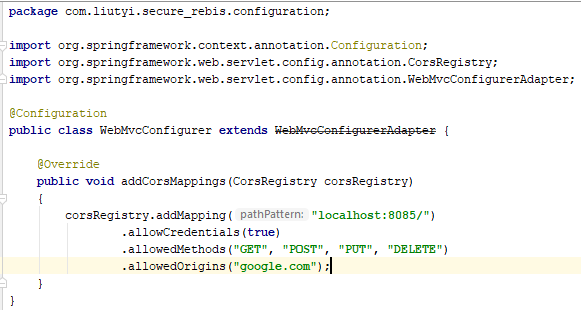
Після налаштування, об’єкт повертається як результат виконання однойменного методу. Цей метод не викликається вручну, а запускається з допомогою виконавчих сервісів фреймворка Spring під час ініціалізації полів класів і бінів у програмі. На рисунку 3.4 зображено структуру класу ApplicationBeans.



*Рис 3.4 Структура класу ApplicationBeans*

У класах WebMvcConfig та SecurityConfig знаходяться конфігурації для вказання системі, із яких доменів дозволено приймати запити, а також, використання яких HTTP-методів є дозволеним. Також, там знаходиться налаштування, до яких вхідних точок програми є безумовний зовнішній доступ, а до котрих необхідно доступатись із авторизаційним токеном.

На рисунку 3.5 зображено налаштування дозволених HTTP-методів та серверів, з яких здійснюється доступ.



*Рис 3.5 Налаштування модуля Spring Web MVC*

На наступному рисунку зображено налаштування безпеки програмного продукту, з допомогою конфігурації модуля Spring Security.



*Рис 3.6 Конфігурація модуля Spring Security*

У пакеті контроллерів є UserController, DocumentController, FinanceController,

AuthenticationController, SchedulerController.

Кожен з них є вхідною точкою до групи певних класів, об’єднаних об’єктом, над яким вони працюють.

**UserController** приймає запити на створення, видалення користувача, внесення змін у його профіль, блокування та розблокування облікового запису користувача, надсилання ним відкритих запитів до інших користувачів.

**DocumentController** приймає об’єкти доданих документів, запити на зміну наповнення цих документів, на додавання до документів інших користувачів, на активацію/деактивацію документа, його отримання або цілковите видалення із сховища програмного продукту.

**FinanceController** відповідальний за прийняття та передачу обробку над запитами, в яких містяться чутливі фінансові дані користувачів – номери їхніх рахунків, паролі, секретні фрази. Цей контроллер проводить подвійну ідентифікацію користувача, використовуючи крім токена, що генерується системою, фінансовий номер телефону користувача для підтвердження таких операцій як переказ коштів або припинення виплат за зобов’язаннями.

**AuthenticationController** є достатньо простим, одначе відповідає суто за функції логінування юзера, його реєстрації та виходу із системи. Контролює також автоматичний вихід із системи після 5 хв бездіяльності корисутвача.

**SchedulerController** приймає запити на створення планових платежів, їх редагування та видалення.

На наступних скріншотах зображено вигляд кількох контроллерів посеред процесу розробки програмного продукту:

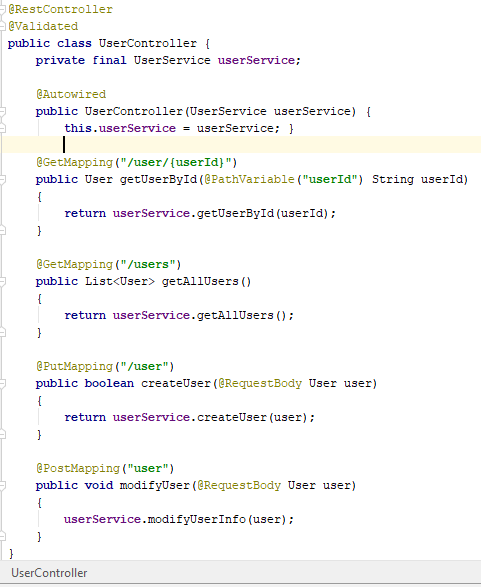


Рис 3.7 Вигляд класу UserController



*Рис 3.8 Клас AuthenticationController*

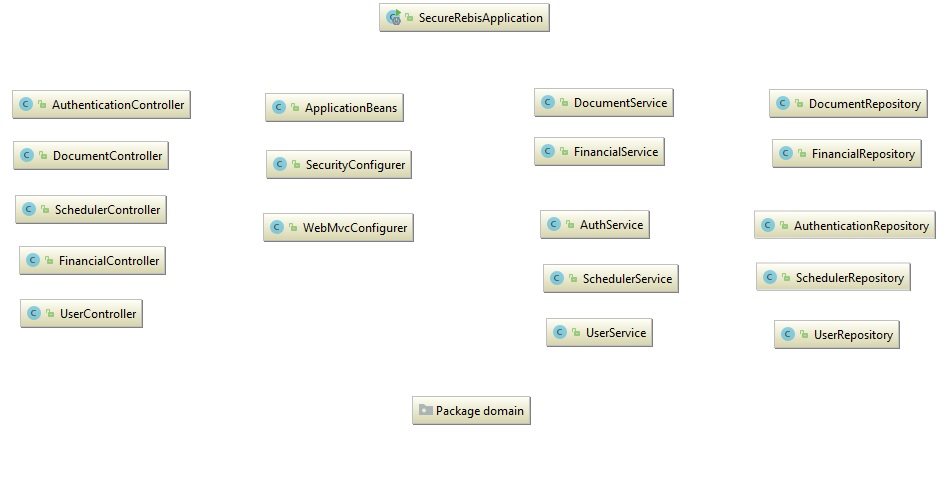
Інші контроллери схожі за своєю побудовою та методами до наведених, тому немає потреби наводити лістинги їх усіх.

Така ж ситуація з сервісами – принцип проектування усіх однаковий, достатньо розглянути як приклад лістинг сервісу DocumentService, що відповідає за обробку всіх документів, та такі дії як їх створення, модифікація та видалення.

Лістинг сервісу DocumentService:

**import** com.liutyi.secure\_rebis.domain.Document;  
**import** com.liutyi.secure\_rebis.repository.DocumentRepository;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.stereotype.Service;  
  
**import** javax.validation.constraints.NotBlank;  
**import** javax.validation.constraints.NotEmpty;  
**import** java.util.List;  
  
@Service  
**public class** DocumentService {  
  
 **private final** DocumentRepository **documentRepository**;  
  
 @Autowired  
 **public** DocumentService(DocumentRepository documentRepository) {  
 **this**.**documentRepository** = documentRepository;  
 }  
  
 **public boolean** createDocument(String userId, @NotBlank String documentName)  
 {  
 checkUserExist(userId);  
  
 **if** (isThereAnyDocumentWithName(documentName)){  
 **return false**;  
 } **else** {  
 Integer nextId = **documentRepository**.generateNextId();  
 **return documentRepository**.createDocumentWithId(documentName, nextId);  
 }  
 }  
  
 **private boolean** isThereAnyDocumentWithName(String documentName) {  
 **return documentRepository**.isThereAnyDocWithName(documentName);  
 }  
  
 **private void** checkUserExist(String userId) {}  
  
 **public** Document updateDocument(Document document)  
 {  
 checkDoesUserHaveAccessToDocument(document);  
 **return documentRepository**.updateDocument(document);  
 }  
  
 **private void** checkDoesUserHaveAccessToDocument(Document document) {  
 }  
  
 **public boolean** deleteDocument(Document document)  
 {  
 **return documentRepository**.deleteDocument(document);  
 }  
  
 **public** Document getDocumentById(String docId)  
 {  
 **return documentRepository**.findOneById(docId);  
 }  
  
 **public** List<Document> getAllDocumentsOfUser(String userId)  
 {  
 **return documentRepository**.findAllByUserId(userId);  
 }  
}

Загальна класова діаграма програмного продукту:



*Рис 3.9 Класова діаграма програмного продукту*

**3.2 Алгоритми та схеми роботи**

Окрім планування класів та їх архітектури, важливу основу системи становлять алгоритми та схеми роботи, обрані для максимальної простоти системи та зручності користувача.

Нижче подано приклади деяких з алгоритмів та схем роботи, використаних у розробленому програмному продукті:

**3.2.1 Типи реєстрації користувача:**

1. Реєстрація через логін і пароль, що зберігаються у системі

У такому разі, користувач проходить повний процес реєстрації.

Всі дані про нього зберігаються лише на серверах системи.

Логін користувача зберігається у незашифрованому вигляді, оскільки в якості логіну користувачі здебільшого використовують досить поширену інформацію – адресу їхньої електронної пошти або номер мобільного телефону.

Пароль користувача зберігається в базі даних у вигляді хеш-коду, який вираховується із зашифрованого паролю. Відновити пароль користувача, навіть отримавши прямий доступ до бази даних – неможливо.

1. Реєстрація через аккаунт Google.

Якщо в користувача є обліковий запис у системі Google, він може використати цей аккаунт для доступу в систему. При першому вході йому необхідно в віджеті входу Google надати дозвіл на використання програмним продуктом особистих даних користувача. Це спрощує доступ до системи, оскільки всі дані про користувача стають доступними їй без необхідності ручного введення, а також позбавляє користувача необхідності самому реєструвати свої банківські картки – якщо він надасть дозвіл, необхідні дані будуть використовуватись через його обліковий запис.

Більшість інформації системою в такому разі не зберігається, головне , що існує таблиця в базі даних, де зберігається ідентифікатор користувача в системі і токен, з допомогою якого відбувається авторизація у сервіси Google.

**3.2.2 Шифрування**

Для шифрування чутливих даних користувачів використано вбудовану реалізацію шифру AES – 256 в стандартному пакеті Javax.

Робота алгоритму на програмному рівні виглядає наступним чином:

1. Відбувається ініціалізація створеного об’єкту класу KeyGenerator, в конструктор якого було передано міжнародну назву шифру – AES.
2. Після того, як об’єкт було створено, він конфігуриться для роботи з 256-бітними ключами
3. З допомогою налаштованого об’єкту класу KeyGenerator генерується випадковий 256-бітний секретний ключ, що буде використовуватись для шифрування та дешифрування даних.
4. Наступний крок – отримання об’єкту класу Cipher, також попередньо налаштованого для роботи із AES – шифром.
5. Об’кт налаштовується: вказується режим роботи – Шифрування або Дешифрація. Також, для роботи із об’єктом йому передається сутність попередньо згенерованого секретного ключа.
6. З допомогою методу doFinal отримується кінцевий зашифрований ( або розшифрований) текст.

На рисунку 3.10 зображено цей же алгоритм у вигляді блок-схеми.



Рис 3.10 Блок-схема алгоритму шифрування

**3.2.3 Створення договору**

Після того, як система була повідомлена про те, що користувача бажає створити договір, система в першу чергу перевіряє, чи користувач вже вніс дані про фінансові рахунки, із якими обов'язково має бути пов'язаний договір, у систему.

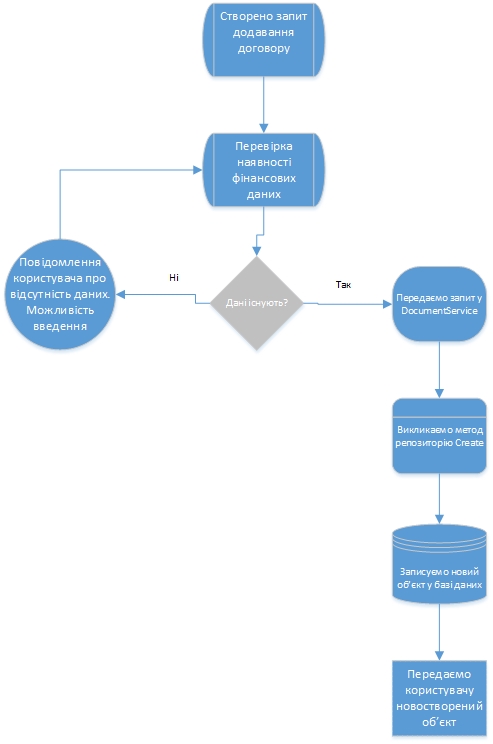
В разі наявності таких даних відправляється запит на серверну частину програмного продукту, у якому знаходиться унікальний ідентифікатор користувача, токен життя веб-сторінки та сам запит на створення нового договору. Запит надсилаєтьсся на клієнтську частину використовуючи HTTP-PUT метод, оскільки згідно з існуючими принципами побудови додатків саме цей метод необхідно використовувати для створення або розміщення нових об’єктів.

Після надходження запиту у відповідний метод контроллера, викликається DocumentManager, який передає необхідні деталі запиту у DocumentService.

Всередині сервісу ми викликаємо метод репозиторію, що відповідає за створення нових об’єктів, і передаємо отриманий об’єкт наверх ієрархії, до вихідної точки контрроллера.

В разі виникнення якої помилки по ходу роботи цього алгоритму, викидається виняткова ситуація і припиняється обробка конкретного запиту.

На рисунку 3.11 схематично зображено хід роботи даного алгоритму.



*Рис 3.11 Алгоритм створення договору.*

**3.2.4 Схема додавання іншого користувача до числа підписантів угоди.**

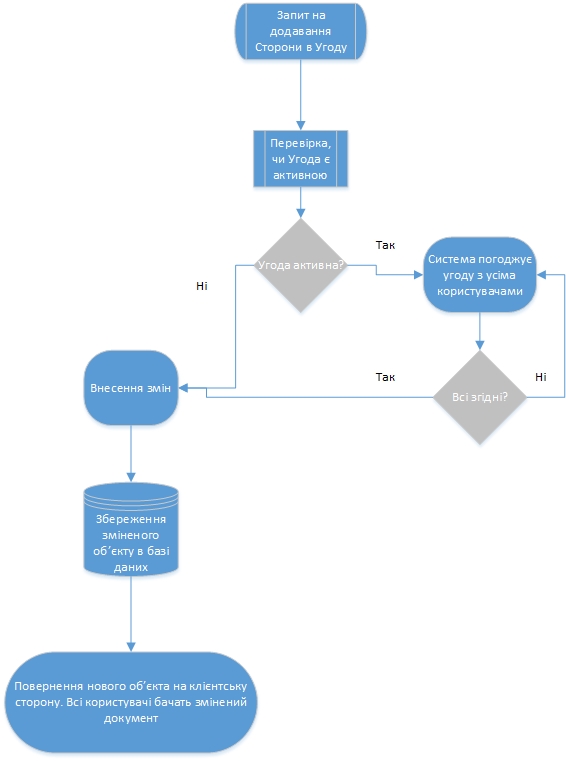
Для внесення змін у поки недіючий договір, системі достатньо авторизаційних прав одного користувача. Після того, як до нас надійшло повідомлення від клієнтського застосунку, разом із ідентифікатором користувача, якого перший юзер хоче долучити до документу, а також із типом ролі, в межах якої можливості доданого користувача відносно цього документу будуть обмежені. Також у запиті передається ідентифікатор користувача, який виступив ініціатором додавання іншого, а також унікальний визначник документу.

В разі недіючого договору, ми знаходимо у базі неактивних договорів угоду з переданим нам визначником. При цьому ми використовуємо інтерфейс репозиторію для отримання об’єкта із бази даних, а потім сервіс, для оперування цим об’єктом. Ми додаємо до мапи користувачів ідентифікатор коричтувача та його роль, і викликаючи метод репозиторію save(), зберігаємо змінений об’єкт в базі даних, та надсилаємо його клієнтському застосунку для оновлення відображення даних.

Якщо ж договір активний на даний момент, для додавання нового користувача система проводить розсилку повідомлень інший сторонам даного договору, які можуть погодити додавання або ж відмовити у цьому. У разі цілковитої згоди всіх сторін, система додає користувача.

В іншому разі система не робить жодних дій, допоки запит абсолютно не погоджено або не скасовано.

Схему додавання користувача до договору зображено на рисунку 3.12



*Рис 3.12 Додавання користувача в договір*

**3.2.5 Алгоритм редагування угоди, коли вона вже є активною.**

1. Після надсилання запиту від клієтської частини програми разом із відредагованим документом, він надходить на вхідну точку програми, де з допомогою Dispatcher Servlet перенаправляється на Document Controller.

1. Після надходження об’єкту запиту в документ контроллер, відбувається перевірка дозволів користувача, від імені якого цей запит було надіслано у систему.

Вона проходить таким чином: спершу в UserManager надходить сигнал про здійснення перевірки автенифікації користувача за його ідентифікатором, паролем та токеном, що видається при авторизації та зберігається у заголовку кожного запиту.

UserManager почергово викликає методи із UserService, який надає методи доступу до ресурсів UserRepository.

Викликаються такі методи як validateIsUserAuthorised,

validateTokenFreshness, validateUserExistense, checkListOfBannedUsers.

Після того, як вся перевірка була пройдена, DocumenManager отримує дозвіл на проведення зміни, і розсилає всім іншим підписантам цього договору його нову версію.

Всі вони можуть вносити свої побажання, які будуть доступні всім при перегляді загальної версії.

Коли система отримала схвалення остаточної версії усіма користувачами, вона редагує копію документа у головному сховищі, та оновлює дату його актуальності – внесення змін завершено.

**3.2.6 Варіанти доступу до фінансових даних користувача.**

Фінансові дані користувача (до прикладу, його банківська картка) можуть бути доступними системі двома способами:

1. Система у якості провайдера фінансових даних. В такому разі, номер картки розділяється алгоритмом на дві частини, які зберігаються в базі даних окремо одна від одної. Унікальними ідентифікаторами обох частин є визначник користувача.

Решта даних картки, такі як термін дії та її CVV-код зберігаються також окремо, на різних таблицях бази даних. Для шифрування кожної таблиці використовуються різні секретні ключі, які відрізняються способом формування, довжиною, але є однаково надійними.

Для доступу до даних банківської картки і проведення із нею операцій, користувачу, окрім авторизації в системі, необхідно також підтвердити свій доступ до системи із допомогою одноразово згенерованого коду, що надсилається на його фінансовий номер телефону, та має бути введеним протягом 3хв від моменту надсилання.

Якщо користувач заходить у систему із нестандартної мережі, або публічної, тоді системі знадобиться додаткове підтвердження особи користувача, таке як введення CVV коду його картки та порівняння його зі збереженим у базі даних.

1. Система у якості менеджера фінансових даних.

Якщо користувач користується системою через сервіси Google, то його фінансові дані вже є збереженими в Google Pay, відповідно, у системі не зберігається його жодна фінансова інформація, і вся взаємодія із разунками проходить через посередництво Google Pay.

**3.2.7 Блокування аккаунту**

Алгоритм роботи системи для розпізнання необхідності блокування аккаунту та умови, потрібні для його розблокування.

У випадку, якщо системою реєструється підвищена активність або підозріла поведінка одного з користувачів, програмою приймається рішення про блокування акаунту даного користувача.

Таке обмеження може бути спровоковане у наступних випадках:

* Більше 1го запиту в секунду із одного акаунту протягом 2 хвилин

( в даному випадку обмеження в доступі активовується задля уникнення можливості потрапляння системи під DDOS атаку.)

* Підозрілі дії користувача – видалення кількох довогорів підряд, намагання зупинити дію існуючого договору без попередження інших учасників, спроба позбавити інших сторін договору їх прав, або зміна їх ролей шляхом, що є непередбаченим інструкцією до користування системою
* Скарги інших користувачів на дії конкретного юзера – образливі висловлювання в коментарях до документів, уникнення відповідальності, взятої на себе умовами договорів.

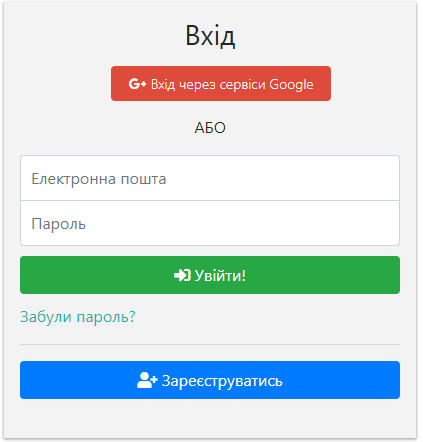
Якщо користувача заблоковано тимчасово, йому необхідно лише дочекатись закінчення терміну блокування.

У разі, якщо системою користувача було визначено як ненадійного, його акаунт вважається замороженим. Розморозка відбувається якщо користувач має змогу офіційно запевнити, що виконує свої обов’язки, а також за умови, що інші користувачі підтримують розблокування заблокованого.

**4 . Використання розробленої системи.**

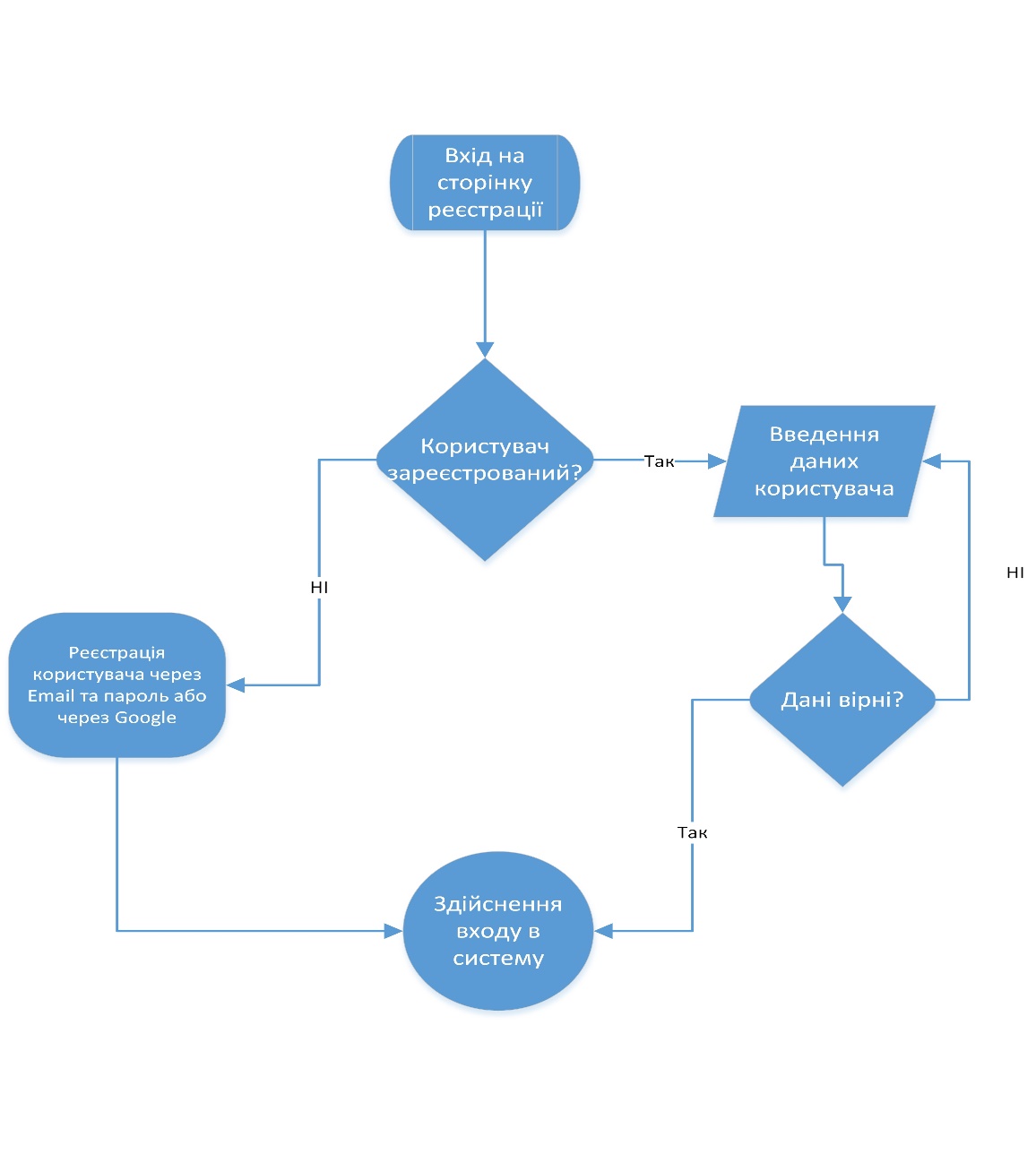
**Схеми дій користувача та скріншоти системи**

На наступному рисунку зображено скріншот віджета авторизації користувача.



*Рис 4.1 Віджет авторизації користувача*

На наступній схемі зображено алгоритм авторизації та/або реєстрації користувача у системі.

**

*Рис 4.2 Алгоритм здійснення входу в систему*

Після того, як користувач скористався системою авторизації та отримав унікальний ідентифікатор наявності до системи, він може приступати до виконання необхідних йому дій.

**4.1 Створення договору**

Для створення договору користувачу необхідно зайти на вкладку Договори,

Створити новий договір.

Після цього користувач вводить назву угоду, і має можливість одразу ж додати користувачів до неї.

Якщо користувач вже має у своєму розпорядженні необхідний електронний примірник угоди, він може його завантажити у цьому ж меню. Після збереження угоди, її копія завантажується на сервери програми, а назва починає бути у списку неактивних договорів.

Також користувачу часто знадобиться можливість додавати інших користувачів після того, як угоду було створено.

**4.2 Додавання нового учасника договору**

Для того, щоб додати до договору іще одного суб’єкта, необхідно виконати на стороні користувача наступні дії:

* Зайти у вкладку Договори
* Обрати підгрупу Неактивні
* Обираємо необхідний договір, після відображення договору на правій бічній панелі знаходиться кнопка «Додати сторону»
* Обираємо роль, яка буде у сторони, відповідно до основних видів ролей, передбачених у договорі – Замовник, Виконавець, Спів-замовник, Спів-виконавець, Поручитель, Арбітр.
* Якщо додається суб’єкт, що має працювати з фінансами, отримувати або надавати кошти іншим учасникам договору, його фінансові дані мають бути актуальними на момент додавання його до договору.

Якщо договір вже активний, тоді необхідно обрати підпункт Активні договори, знайти необхідний договір, і пройти той же алгоритм операцій, що й для неактивних.

Єдина відмінність в тому, що коли договір ще неактивний, то додати до нього сторону може будь-який учасник, як і видалити її. Опісля підписання і переходу в активний стан договір може бути змінено лише за погодження всіх учасників угоди, або ж однієї зі сторін та арбітра, якщо під час імплементації договору було виявлено певні невідповідності або неточності, що були завадою до виконання угоди в повній мірі.

**4.3 Схема роботи із віджетом планових платежів**

Окрім роботи із договорами замовників та виконавців, система пропонує користувачам можливість здійснювати безумовні планові платежі без їхньої участі чи особистого підтвердження (користувач має можливість вказати обов’язковість попередження його системою про здійснення певних платежів).

Без необхідності укладати угоду, користувачу необхідно здійснити простий алгоритм дій, для налаштування відповідного сервісу.

Після входу в систему користувач переходить на вкладку Платежі, далі на підпункт Планові платежі, і тисне кнопку Створити новий.

Після цього система перенаправляє користувача на сторінку створення планового платежу.

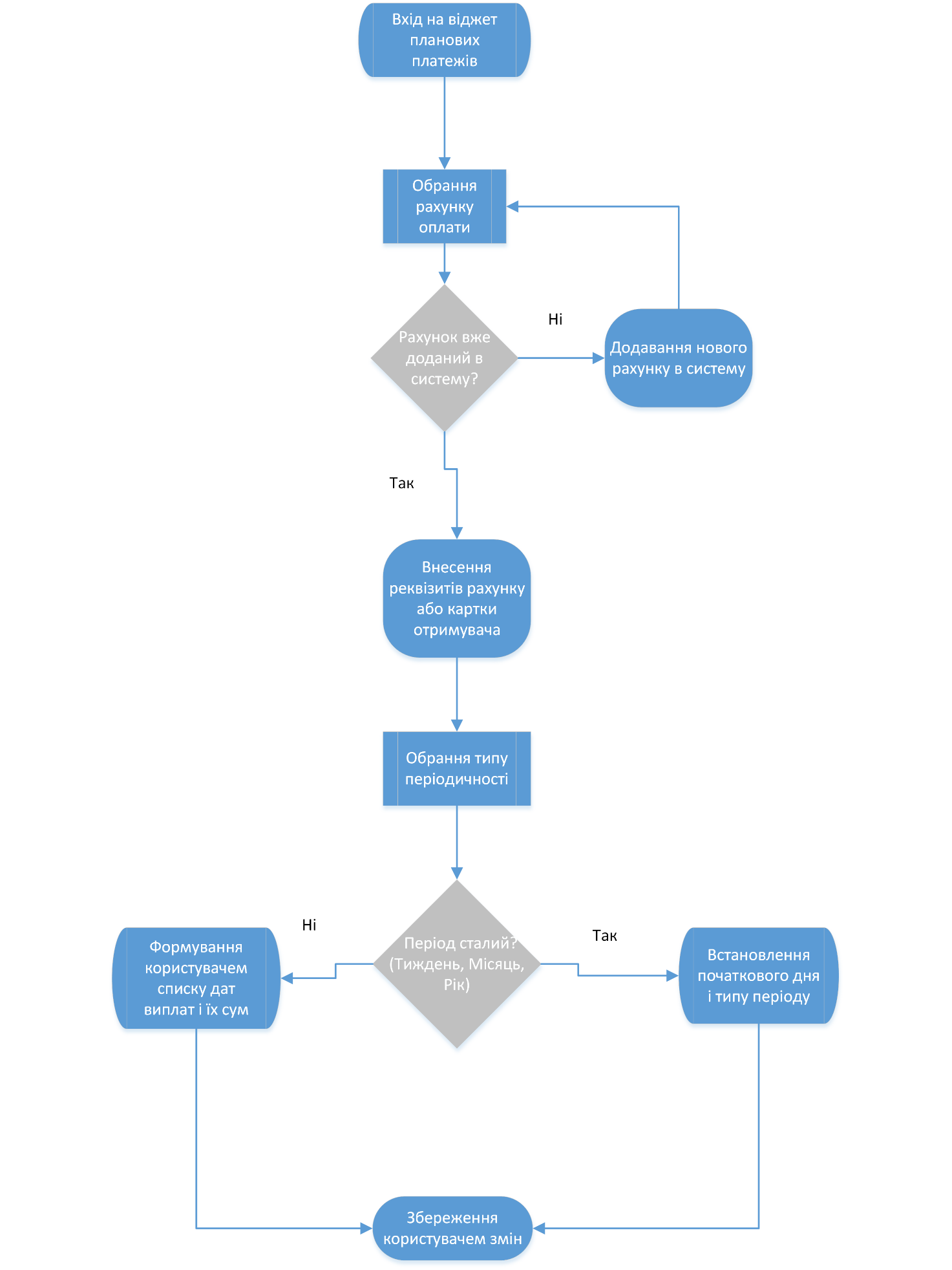
Це може бути як переказ між рахунками, так і оплата надання послуг мобільного зв’язку або комунальних послуг.

Алгоритм роботи з віджетом простий:

1. Користувач вказує, з якого існуючого рахунку або картки будуть здійснюватися відрахування.

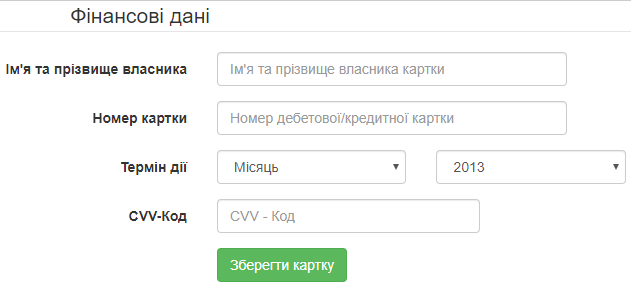
Якщо жоден з попередньо доданих способів оплати його не влаштовує, він має можливість додати дані банківької картки не виходячи із даного віджету.

1. Користувач вводить реквізити або номер банківської картки суб’єкта – отримувача.
2. Користувач вводить тип періодичності і день місяця, коли необхідно здійснити перший платіж. Також йому доступна можливість просто сформувати власноруч список дат і суми, які мають бути перераховані у ці дати.



*Рис 4.2 Схема роботи із плановими платежами*

На рисунку 4.3 зображено базовий віджет для заповнення даних банківської картки.

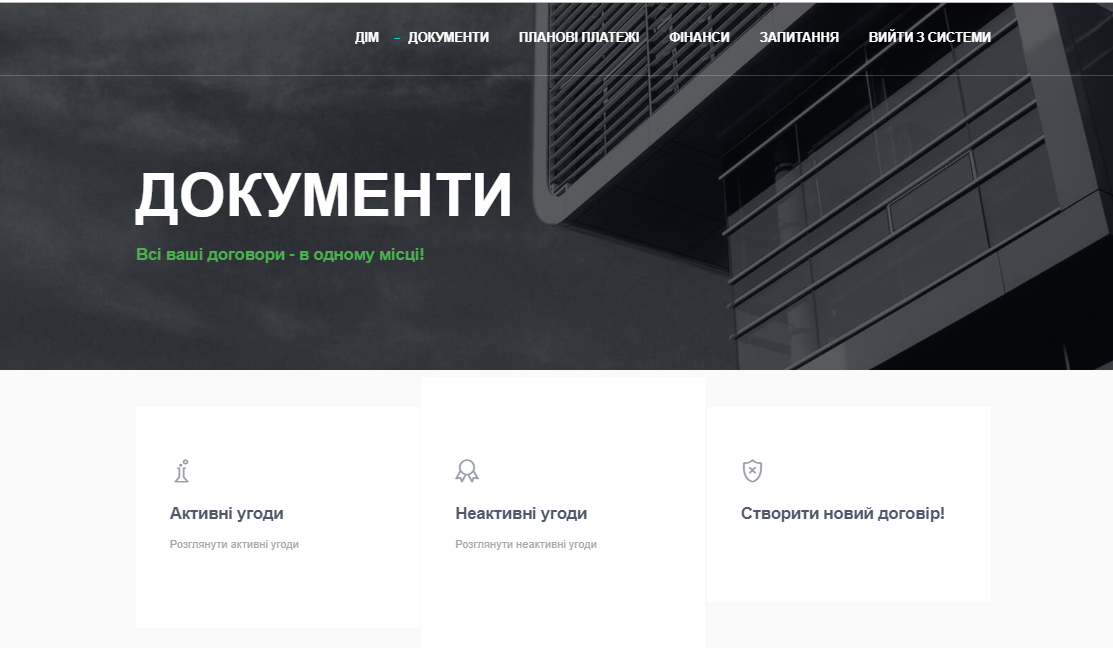


*Рис 4.3 Віджет для збереження даних банківської картки.*

На наступному рисунку (4.4) зображено скріншот стартової сторінки програмного продукту із вже зареєстрованим користувачем а також скріншот вкладки «Документи» на рисунку 4.5.



*Рис 4.4 Стартова сторінка сайту*



*Рис 4.5 Вигляд вкладки Документи*

Висновки:

Під час виконання цієї бакалаврської кваліфікаційної роботи мною було виконано роботу із аналізу різноманітних варіантів систем, що надають фінансові послуги користувачам. Розглянуто декілька найпоширеніших способів шифрування даних, а також види архітектури, необхідної для успішного розроблення системи. Спроектовано та розроблено систему для забезпечення безпечного контролю фінансових операцій між різними суб’єктами. В результаті розробки системи покращено навички програмування, організації часу, роботи із документацією.