КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики
Кафедра теоретичної кібернетики

Кваліфікаційна робота
на здобуття ступеня бакалавра
за спеціальністю 122 Комп'ютерні науки
на тему:

Розробка онлайн сервісу трекінгу бюджету

Виконав студент 4-го курсу
Грогуль Юрій Васильович _____ (підпис)

Науковий керівник:
професор, доктор фіз.-мат. наук
Крак Юрій Васильович _____ (підпис)

Засвідчую, що в цій курсовій
роботі немає запозичень з праць
інших авторів без відповідних посилань.

Студент _____ (підпис)

Роботу розглянуто й допущено до захисту
на засіданні кафедри теоретичної
кібернетики
«____» _____ 2022 р.,
протокол № ___

Завідувач кафедри
доктор фіз.-мат. наук, професор
Юрій КРАК _____ (підпис)

КИЇВ-2022

РЕФЕРАТ

Обсяг роботи 59 сторінок, 30 ілюстрацій, 1 таблиця, 16 джерел посилань.

Ключові слова: .NET, C#, ASP.NET CORE, WEBAPI, EF CORE, CQRS, MVC, POSTGRESQL, NODE.JS, JAVASCRIPT, SVELTE, DAISYUI , TAILWIND, PRIVAT24, CLOUDFLARE, NGINX, FASTFOREX.

Метою роботи є створення та розгортання зручного вебсервісу, який дозволить легко та успішно керувати особистим або сімейним бюджетом з покращенням аналізу своїх доходів та витрат їх візуалізацією, плануванням та прогнозуванням, а також підхід, при якому з персонального комп’ютера можна зробити вебсервіс.

Результати роботи: розроблено вебсервіс за допомогою сучасних підходів, а саме розроблено клієнтську та серверну частини на основі сучасних підходів проєктування та застосування сучасних та актуальних середовищ програмування з інтеграцією популярних сервісів таких як банківська система privat24 для відстежування обігу банківської карти та FastForex для реалізації власного сервісу конвертації валют.

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

DNS (Domain Name System) – система доменних імен

IP (Internet Protocol) – інтернет протокол

HTTP (HyperText Transfer Protocol) – протокол передачі даних

API (Application Programming Interface) – інтерфейс програмування додатків

ASP.NET Core (Active Server Pages для .NET Core)

CSS (Cascading Style Sheets) – каскадні таблиці стилів

HTML (HyperText Markup Language) – мова розмітки гіпертекстових документів

JS (JavaScript)

JSON (JavaScript Object Notation) – об'єктній запис JavaScript

CQRS (Command and Query Responsibility Segregation)

MVC (Model-view-controller) – модель–представлення–контролер

SQL (Structured query language) – мова структурованих запитів

EF Core (Entity Framework Core)

ORM (Object-relational mapping) – об'єктно-реляційна проєкція

ПЗ – програмне забезпечення

БД – база даних

СУБД – система управління базами даних

НСЗК – навігаційна сутність зовнішнього ключа

ЗМІСТ

ВСТУП

Багато людей постають перед фінансовими труднощами і як складова частка подібної проблеми пов'язана саме з тим, що люди не планують свій бюджет. Сімейний бюджет – це план прибутків та витрат сім’ї за певний проміжок часу. Основна мета контролю над власними коштами – отримати розгорнуту картину власних фінансів для більш ефективного використання. Неконтрольоване використання фінансів стає причиною перевищення витрат над доходами. Як наслідок, це змушує брати кредити, ставати боржником, що тільки ускладнює ситуацію та погіршує добробут.

Сьогодні існує багато методів обігу коштів, яких не було ще 20 років тому, що дозволяють автоматизувати грошовий обіг до такого рівня, що не потрібно вміти додавати 2+2, щоб оплатити будь-яку послугу, оскільки за вас все зробить сервіс, яким ви користуєтесь (наприклад, банківська система автоматично керує сумою безготівкових коштів банківських рахунків). Все йде до того, що готівка з часом буде витіснена різними віртуальними сервісами, чи то банківська система, чи то криптовалютна, чи така, про яку ми сьогодні навіть не здогадуємось. І всі ці системи потрібно відстежувати, контролювати, підводити статистику, і вочевидь якщо всі подібні процеси пов'язані з персональним чи промисловим бюджетом будуть уніфіковані та зібрані в одній трекінговій системі яка буде підтримувати багато банків, в якій будуть зручні інструменти для управління, планування, групування, багатокористувацьке управління, візуальне відображання статистики та запорукою всього зручний інтерфейс призведе до ефективності та продуктивності ведення бюджету.

Основна ціль роботи – зосередитись на побудові повноцінного самостійного сервісу та інтегрувати в систему відстежування безготівкових коштів (пластикові, віртуальні, валютні картки) українського банку “ПриватБанк” та конвертер валют (при інтеграції з FastForex) за

допомогою їх API. При розробці спираючись на сучасні технології та методи розгортання вебсервер на персональному комп'ютері, реалізацію новітньої архітектури та налаштування мережевої архітектури для доступу до вебресурсу.

РОЗДІЛ 1. ВЕДЕННЯ БЮДЖЕТУ В СУЧАСНОМУ СВІТІ

1.1. Що таке бюджет, його переваги та кому він потрібен.

Важливість складання бюджету – це фінансовий урок, який важко переоцінити. Якщо ви та ваша сім'я бажаєте фінансової безпеки, єдиною відповіддю є дотримання бюджету.

До епохи пластикових карток люди, як правило, знали, чи живуть вони за коштами. Наприкінці місяця, якщо у них залишалося достатньо грошей, щоб оплатити рахунки та отримати частину заощаджень, вони були на правильному шляху. У наші дні люди, які зловживають кредитними картками, не завжди усвідомлюють, що витрачають зайві кошти, поки не потонуть у боргах.

Однак, якщо ви створюєте і дотримуєтеся бюджету, ви ніколи не опинитеся в такому небезпечному становищі. Ви точно знатимете, скільки грошей ви заробляєте, скільки можете дозволити собі витрачати щомісяця і скільки вам потрібно заощадити. Звичайно, підраховувати цифри та стежити за бюджетом не так весело, як безсоромні шопінг. Але подивіться на це так: коли наступного року ваші друзі, щасливі витрачати гроші, призначаються на зустріч із боргом консультанта, ви кинетеся в ту європейську пригоду, на яку Ви заощаджували, або, ще краще, перейдете до свого новий будинок.

Скажімо, Ви відповідально витрачаєте гроші, дотримуєтеся бюджету до і ніколи не маєте боргів по кредитній картці. Добре для вас! Але ти щось не забуваєш? Наскільки важливо сьогодні розумно витрачати гроші, економія також має вирішальне значення для вашого майбутнього.

Бюджет може допомогти вам зробити це. Важливо передбачити інвестиційні внески у свій бюджет. Якщо ви щомісяця відкладатимете частину свого заробітку для внеску до недержавних пенсійних фондів, ви врешті-решт створите гарну подушку. Хоча зараз Вам, можливо,

доведеться трохи пожертвувати, у майбутньому це буде того варте. Зрештою, Ви б воліли провести свою пенсію, граючи в гольф і ходячи на пляж, або працюючи зустрічальником у місцевому продуктовому магазині, щоб звести кінці з кінцями? Точно.

Життя сповнене несподіваних сюрпризів, одні кращі за інших. Коли ви звільняєтеся, захворієте або отримуєте травму, розлучаєтеся або хтось помирає в сім'ї, це може призвести до серйозних фінансових потрясінь. Звичайно, здається, що такі надзвичайні ситуації завжди виникають у найгірший можливий час – коли у вас вже немає грошей. Саме для цього кожному потрібен надзвичайний фонд.

Ваш бюджет має включати фонд на випадок надзвичайних ситуацій, який складається щонайменше з трьох-шести місяців витрат на проживання. Ці додаткові гроші гарантують, що ви не потрапите в глибину боргів після життєвої кризи. Звичайно, потрібен час, щоб заощадити витрати на проживання за три-шість місяців.

Не намагайтеся відразу скинути більшу частину своєї зарплати до свого фонду надзвичайних ситуацій. Включіть це у свій бюджет, поставте реалістичні цілі та почніть з малого. Навіть якщо ви щотижня відкладете лише 100 гривень, ваш екстрений фонд буде повільно накопичуватися.

Складання бюджету змушує вас уважно придивитися до своїх звичок витрачати гроші. Ви можете помітити, що витрачаєте гроші на речі, які вам не потрібні. Чи ви чесно дивитесь всі 100 каналів за своїм дорогим розширеним тарифним планом або підпискою на кілька потокових? Тобі справді потрібно 30 пар чорних черевиків? Бюджетування дозволяє вам переосмислити свої звички витрачати гроші та перефокусувати свої фінансові цілі.

Отримання бюджету також допоможе вам зловити більше закритих очей. Скільки ночей ви хвилювалися про те, як ви збираєтеся оплачувати рахунки? Люди, які втрачають сон через фінансові проблеми, дозволяють

своїм грошам контролювати їх. Якщо ви плануєте свої гроші розумно, ви більше ніколи не втратите сон через фінансові проблеми.

Звичайно, це лише верхівка айсберга. Дотримання бюджету має безліч інших переваг. Так чому ж чекати? Час починати складати бюджет!

1.2. Статистика та результати ведення бюджету.

При спробах дослідити статистику в Україні стає очевидно, що вона відсутня, швидше за все через те, що ведення бюджету не є популярним у громадян нашої країні, крім цього бракує якісних ресурсів для його ведення. Це підтверджує актуальність створення нашого продукту.

Тому ми візьмемо статистику американського «Бюро статистики та праці» за 2022 рік [3]. Результати наведені у таблиці 1.

Таблиця 1.

Витрати	**Середня річна вартість**	**% бюджету**
Житло	$21,409	30%
Перевезення	$9,826	14%
Податки на прибуток	$9,402	13%
Харчування	$7,316	10%
Особисте страхування, соціальне страхування та внески до пенсійного плану	$7,246	10%
Охорона здоров'я	$5,177	7%

Витрати	**Середня річна вартість**	**% бюджету**
Розваги	$3,341	5%
Грошові внески	$2,283	3%
Одяг та послуги	$1,434	2%
Освіта	$1,271	2%
Різне	$907	1%
Товари та послуги особистої гігієни	$646	1%
ЗАГАЛЬНО	$70,258	100%*

*У зв’язку з округленням відсотки не складають 100%.

1.3. Рекомендації по веденню бюджету.

Ключ до складання бюджету полягає в дотриманні основного правила – витрачайте менше, ніж заробляєте.

Один із способів розпочати складання бюджету – це вказати, що ви заробляєте, витрачаєте гроші та заборгуєте. Це може допомогти переглянути відомості про заробітну плату, виписки про виплати, рахунки, виписки з банку та виписки з кредитної картки. Якщо ви витрачаєте або заробляєте гроші іншим способом, обов’язково подивіться і на це.

Спробуйте переглянути достатню кількість рахунків і виписок за минулий рік, щоб зрозуміти свої звички щодо заробітку та витрат. Варто зазначити, що деякі рахунки вищі в різні пори року. Наприклад, рахунки за електроенергію взимку часто вищі через опалення.

Після того, як ви врахували найнеобхідніші та надзвичайні ситуації,

ваша мета полягає в тому, щоб залишилися гроші, щоб витратити їх на те, що ви хочете.

Розрахунок того, що ви витрачаєте це перший крок до управління грошима. Однією з найскладніших речей у складанні бюджету та управлінні грошима може бути відстеження того, що ви витрачаєте. Витрати можуть бути регулярними (постійні витрати) або нерегулярними або одноразовими (змінні витрати).

Постійні витрати, які ви можете включити в сімейний бюджет:

- погашення іпотеки або оренди;
- комунальні послуги – газ, світло, вода, телефон, інтернет;
- комісійні збори та земельні податки;
- плата за навчання в школі або вищому навчальному закладі;
- медичне, автомобільне та побутове страхування;
- витрати на громадський транспорт;
- погашення кредитних карток та кредитів фізичних осіб.

Змінні витрати, які ви можете включити до сімейного бюджету:

- їжа;
- утримання будинку та побутові товари;
- шкільну форму, підручники та канцтовари;
- медичні та стоматологічні збори;
- ремонт автомобілів та бензин;
- особисті речі, такі як одяг і стрижки;
- реєстраційні внески та обладнання – наприклад, для спортивних, музичних або танцювальних програм;
- свята;
- розваги;
- подарунки – наприклад, на день народження чи весілля;
- інші речі, наприклад спеціальні частування для вас і вашої родини.

Якщо ваш дохід дозволяє, то навмисне переоцінювання грошей, необхідних для рахунків, може допомогти вам знайти додаткові гроші.

Ваш бюджет покаже вам, витрачаєте ви зараз більше чи менше, ніж заробляєте. Якщо ви зараз витрачаєте більше, це може допомогти сісти всією сім'єю і подумати, де можна заощадити гроші. І якщо ви вже витрачаєте менше, ніж заробляєте, ви можете подивитися, як заощадити і як використовувати свої заощадження.

Ось декілька важливих порад для створення.

Перегляньте свої витрати. З'ясуйте, чи економите ви стільки, скільки можете. Чи могли б ви витрачати менше на певні предмети? Чи є у вас кредитні картки з високими відсотками чи інші позики? Чи не могли б ви погасити їх якнайшвидше і розглянути більш підходящі варіанти кредиту або позики? Бажано робити це регулярно.

Створіть буфер заощаджень. Перш ніж почати заощаджувати для своїх потреб, дуже важливо зберегти додаткові заощадження на випадок фінансових ситуацій. Наприклад, ви можете поставити собі за мету зберегти частину грошей на окремому ощадному рахунку на випадок надзвичайних ситуацій.

Вирішіть, на чому ви економите. Які ваші цілі? Скільки потрібно заощадити, щоб їх досягти?

Встановіть кінцевий термін для досягнення мети. Приділіть собі багато часу – здається, що економія може зайняти вічність. Але будьте реалістами, і ви уникнете тиску.

Відкрийте безкоштовний банківський рахунок, який відокремлений від вашого основного рахунку. Ви можете використовувати цей обліковий запис лише для заощадження для досягнення своєї мети. Ви можете налаштувати прямий дебет зі свого основного рахунку, щоб регулярно переказувати встановлену суму заощаджень.

Подивіться на інші варіанти, наприклад, попросіть роботодавця

розділити вашу зарплату, щоб частина з них надходила на ваш окремий ощадний рахунок.

Зверніться до свого банку, фінансової установи чи фінансового консультанта, якщо вам потрібна додаткова консультація.

1.4. Висновки ведення бюджету.

Після того, як ви побудували план заощаджень, перед початком роботи варто переглянути переваги та недоліки. Таким чином ви дізнаєтеся, як це вплине на ваше сімейне життя. Якщо ви невпевнені в певних частинах вашого плану, зверніться за порадою або ще раз перевірте свої розрахунки.

Якщо ви вже сформували та утвердили план, чудово було б мати сучасний інструмент, який би спростив задачу ведення бюджету. Наша ціль – розробити подібний інструмент, систему, яка популяризує та допоможе громадянам нашої країни та не тільки покращити практику ведення бюджету.

РОЗДІЛ 2. АРХІТЕКТУРА ТА ВИМОГИ ДО СИСТЕМИ

2.1. Загальна архітектура системи

Розберемось з загальною архітектурою сервісу, на нашому хості (персональному комп’ютері) повинні бути front-end, back-end, NGINX, також потрібен сервер бази даних (у нашому випадку це буде PostgreSQL). За межами хоста, що є частиною мережевої архітектури залишається CloudFlare, через який будуть проходити всі запити від браузера користувача. Та не забуваємо про інтеграцію з privat24 та fastforex. Отож візуалізуємо нашу загальну архітектуру на рисунку 1, для кращого сприйняття. Зауважимо, що перелічені компоненти, котрі розташовані у нашому випадку на хості, можуть бути розташовані абсолютно в різних місцях, на різних комп’ютерах, серверах.

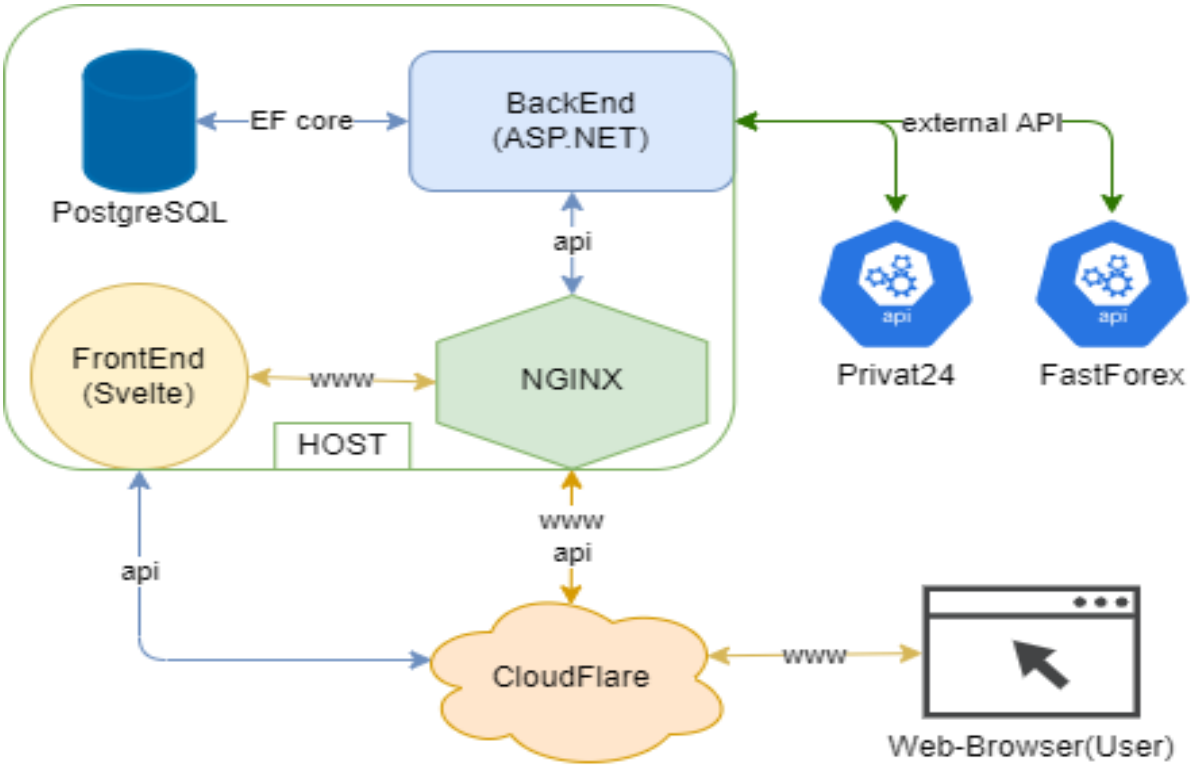

Рисунок 1 – Загальна архітектура сервісу.

2.2. Огляд технологій для клієнтської частини

Клієнтська частина сервісу містить інтерфейс, який дозволяє користувачеві взаємодіяти з системою за допомогою браузера та формувати запити до серверної частини застосунку. Клієнтська частина, або front-end пишеться мовою розмітки HTML, мовою стилю сторінок CSS та мовою програмування Javascript. Існує багато платформ, фреймворків і бібліотек для розробки клієнтської частини. Наприклад, основні платформи це Node.js, PHP, Python, .NET, Golang та інші. Ми будемо використовувати Node.js, для якого основними фреймворками є Vue.js, Angular.js, React.js, але ми будемо використовувати новоспечений Svelte. Завжди при використанні Node.js не обійдеться без використання менеджера пакунків, за замовчуванням це npm, його ми й будемо використовувати для встановлення бібліотек для нашого Node.js проєкту.

Node.js – платформа для виконання мережевих застосунків, написаних мовою JavaScript з відкритим кодом, розвивається з 2009-го року.

Svelte.js – це так званий JavaScript-фреймворк, що зникає. За допомогою власного компілювання, перетворює декларативні Svelte-компоненти в чистий JavaScript код, який точково оновлює нативний DOM браузеру.

Svelte це новий, “магічний” фреймворк на рівні популярних фреймворків, таких як Vue.js, Angular.js, React.js, оскільки не будує абстракції поверх JS, а компілює компоненти в нативний JS, від чого й отримав назву фреймворку, що зникає [8]. Використовуючи Svelte, ви пишете менше коду, не використовуєте Virtual DOM (який дещо сповільнює роботу сайту та збільшує його розмір) та використовуєте знайомі для будь-якого front-end розробника інструменти у Svelte компоненті, а саме HTML, CSS, JS, а не розширення, такі як JSX. Структура проєкту складається з двох типів файлів, це *.js та *.svelte, у

файли *.js може виноситись логіка, яка буде спільно використовуватись в файлах *.svelte. Файли *.svelte це компоненти, що інкапсулюють HTML, CSS та JS, у результаті компіляції проєкту, всі компоненти будуються в нативний JavaScript- файл, який і буде відображати всю логіку всіх Svelte компонент.

З мінусів Svelte це те, що він доволі молодий, хоч і набирає популярність, але мало залучений на корпоративному рівні, що заважає йому набирати популярність, оскільки у програмістів постає питання, навіщо мені вчити та розбиратись, якщо мене з цим не візьмуть на роботу, оскільки в трендах у роботодавців популярні інші фреймворки, такі як Vue.js, Angular.js, React.js, котрі зарекомендували себе протягом тривалого часу. Але це нас не зупинить, оскільки ми готові використовувати інноваційні технології, ті що набирають популярності, щоб розвивати їх в маси.

Для того, щоб стилізувати наш сайт, будемо використовувати daisyui, це компонентна бібліотека, яка інтегрує в середовище розробки фреймворку популярний плагін Tailwind CSS, який має готові CSS компоненти, що забезпечує нам використання потрібних нам CSS класів, для стилізації нашої вебсторінки. Детальніше ознайомитись можна за посиланням *https://tailwindui.com/*, де можна побачити приклади застосування та документацію по його використанні.

2.3. Огляд технологій для серверної частини

Серверна частина застосунку, або back-end виконує основну логіку та обробляє запити від клієнта. Реалізація серверної частини даної системи представлена у вигляді API(прикладний програмний інтерфейс). При використанні API в контексті веб розробки, як правило, воно визначається набором повідомлень запиту HTTP, також визначається структура повідомлень-відповідей. Дані можуть передаватись у різних форматах,

часто це JSON(JavaScript Object Notation), а також XML, HTML, plain text. У випадку серверної частини варіантів платформ для реалізації ще більше, ніж у клієнтської частини. Важко виділити найбільщ популярну платформу з наявних, таких як .NET, Node.js, PHP, Java, Python, Golang, Ruby, та інші.

Для розробки back-end будемо використовувати .NET 6 платформу, та мову програмування C#, фреймворк ASP.NET Core. Зі структурою застосунку ASP.NET Core можна ознайомитись на рисунку 2. [9] Зазвичай при розробці API використовується паттерн проєктування MVC, ідея якого – розділити застосунок на три основні групи компонент: моделі, представлення, контролери. Це дозволяє реалізувати принцип розділення задач. Завдяки цій структурі всі запити направляються в контролери, які відповідають за роботу з моделями для виконання запланованої логіки та побудови відповіді.

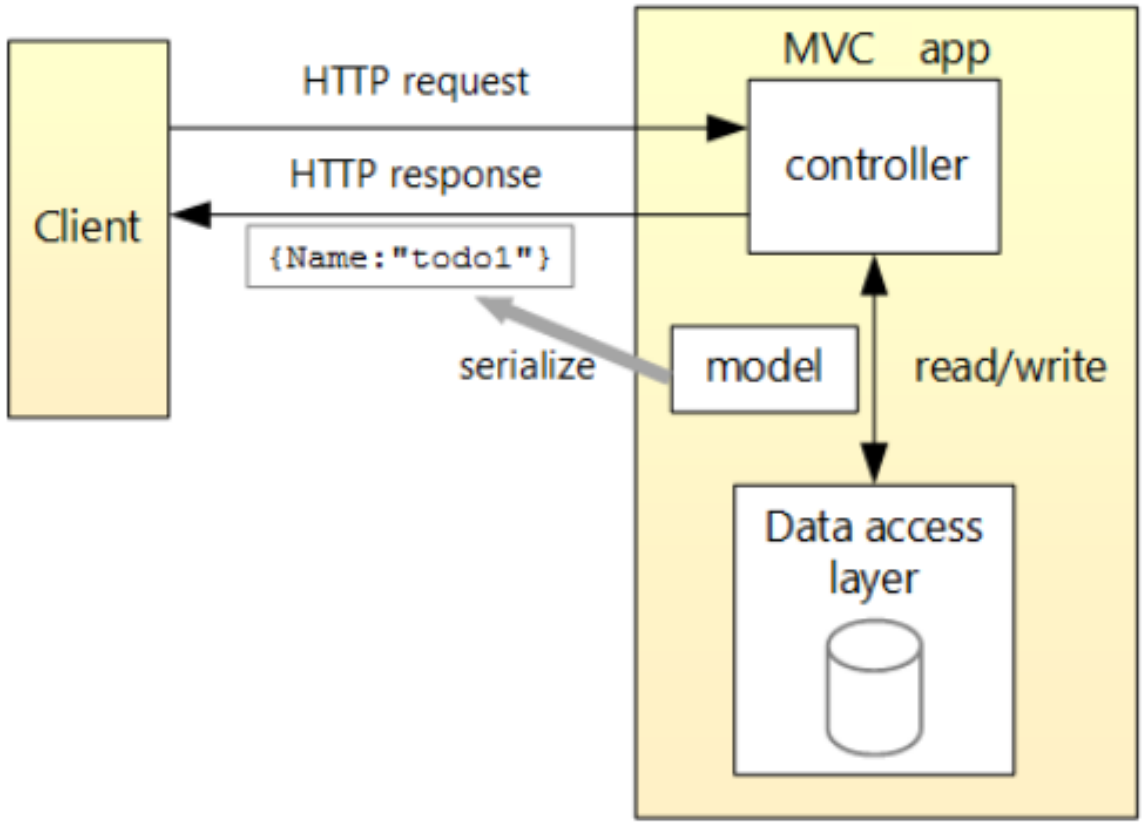

Рисунок 2 – Структура застосунку ASP.NET Core. (Рисунок запозичений з онлайн ресурсу https://docs.microsoft.com/)

Але на цьому зупинятись ми не будемо, оскільки ми застосуємо CQRS підхід, який дозволить нам розвантажити контролери та поділити операції на два типи, а саме на команди та на запити, реалізовувати шаблон проєктування будемо за допомогою бібліотеки MediatR. На рисунку 3 візуально зображена структура підходу. Принцип CQRS поділяє операції на два типи:

- **Команди (Command)** – певні дії для зміни стану застосунку, які мають повноваження зчитувати, записувати, редагувати БД.
- **Запити (Queries)** – мають повноваження лише на зчитування БД, та повернення даних, без зміни стану застосунку.

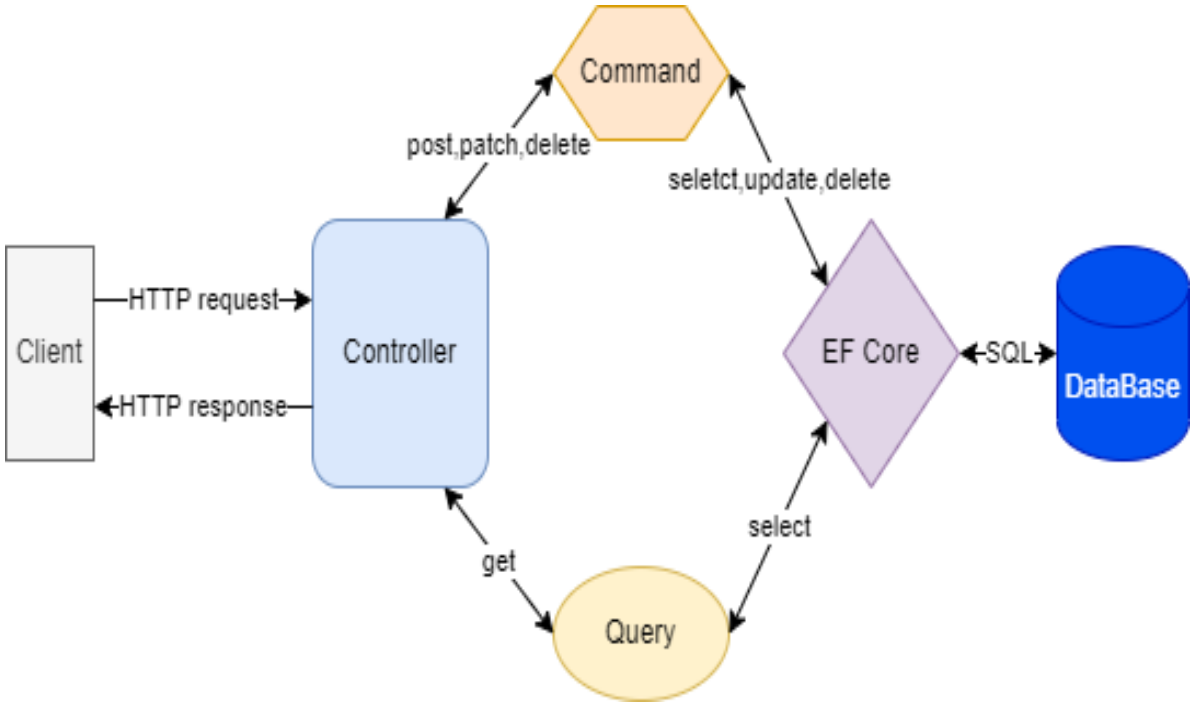

Рисунок 3 – Структура CQRS підходу.

Зокрема на рисунку 3 можемо спостерігати наявність EF Core, це структура об'єктно-реляційного відображення (ORM) з відкритим сирцевим кодом, яка зв’язує базу даних з концепціями об'єктноорієнтованих мов програмування (в нашому випадку C#)

створюючи віртуальну об’єктну базу даних, до якої ми звертаємось за допомогою написаних програмістом моделей. Існує два основні підходи для організації взаємодії EF Core з БД:

- **Code-first** – У цьому випадку розробник розробляє виключно моделі з можливостями налаштування зв’язків між моделями, вказанням типів, властивостей полів таблиці тощо. EF Core генерує базу даних, відштовхуючись від розроблених моделей. Цей підхід дозволяє розробляти базу даних прямо з коду і не потребує маніпуляцій з СУБД.
- **Database-first** – Відштовхуючись від назви, розуміємо, що підхід заснований на першочерговому створенні БД за допомогою певних СУБД, після чого потрібно розробити моделі для взаємодії з БД, які грають роль проміжного інтерфейсу між кодом застосунку та запитами до БД, завдяки прихованій реалізації EF Core. Цей підхід дещо складніший за Code-first, оскільки потребує гарні знання в побудові БД, SQL та постійного оновлення моделей в коді, після оновлення бази даних (додавання, зміни, редагування таблиць). Відповідно розробка застосунку займає більше часу та ускладнює підтримку.

Ми використаємо Code-first підхід, оскільки додаткова задача побудови БД нас мало цікавить, все що нам потрібно це побудувати таблиці, в яких будуть наявні зв’язки багато до багатьох та один до багатьох. Ці зв’язки неявно та надзвичайно просто реалізуються за допомогою EF Core.

СУБД, які підтримуються EF Core: MySQL, Oracle, Progress, PostgreSQL, Firebird, SQLite, VistaDB, Devart, OpenLink, SQL Anywhere, Sybase, Synergex. В нашому випадку вибір не принциповий, оскільки якщо до проєкту буде налаштований той чи інший провайдер EF Core, це не вплине на зміну коду моделей. Тобто якщо у вас є віддалений сервер

MySQL, ви запросто можете використовувати його. У нашому випадку ми використаємо локальний на комп'ютері сервер PostgreSQL.

2.4. Вимоги до системи

Ціль системи: забезпечення громадянам різних країн зручним інструментом по веденню бюджету. Допомога у відстежувані транзакцій, їх класифікація, візуальна статистика, планування бюджету. Розподілення або поширення з іншими користувачами, для їх спільного використання, планування та ведення бюджету.

Основні модулі системи:

- Авторизація
- Управління бюджетами
- Управління транзакціями бюджету
- Управління можливими категоріями транзакцій
- Планувальник
- Статистика
- Сервіс конвертації валют(інтеграція з FastForex)
- Інтеграція з ПриватБанк

Авторизація. Для користування системою необхідно мати обліковий запис. Без авторизації користувач не має права відвідувати ніякі сторінки окрім як головну, на якій він може лише пройти авторизацію.

Сутність користувача (AppIdentityUser) має наступні поля: електронна адреса (Email), ім’я (FirstName), прізвище (LastName), пароль (Password), посилання на фотографію облікового запису (PhotoUrl).

Новому користувачу доступна реєстрація у системі, якщо вказати ім’я, прізвище, електронну адресу, пароль. Пароль повинен хешуватись у базі даних. Також повинна бути обробка помилок при некоректних діях реєстрації нового користувача, таких як введення вже зареєстрованої

електронної адреси, порожні поля та помилки при авторизації в наявний обліковий запис – порожні поля, неправильно введений електронна адреса або пароль.

Управління бюджетами. Після успішної авторизації користувач потрапляє на головну сторінку сервісу, яка відображає бюджети користувача. На ній він повинен мати змогу приєднатись до бюджету іншого користувача, створити власний бюджет та побачити список бюджетів у вигляді: назва, баланс, валюта, код запрошення (якщо він активований), кількість учасників, можливість виходу з бюджету та перехід на сторінку з управлінням транзакціями та можливістю керувати гаманцями інтегрованих банків. Варто додати, що видалення бюджету можливе лише в тому випадку, якщо всі користувачі вийдуть з нього.

Сутність бюджету (Budget) має наступні поля: Назва (Name), баланс (Balance, НСЗК), код запрошення (InviteToken).

Варто зазначити та запам’ятати:

1) У всіх сутностей є первинний ключ ID, який відповідає формату GUID який не буде згадуватись в описі сутностей.
2) Баланс – це навігаційна властивість EF Core, на практиці – це зовнішня сутність, тобто фактично це зовнішній ключ на сутність баланс (Balance), яка має 2 поля: сума (Amount), валюта (Currency).
3) Надалі навігаційні сутності EF Core будуть позначатись в дужках, що вони являються навігаційною сутністю зовнішнього ключа (НСЗК), і мають інший вигляд в базі даних (Як приклад, Balance стане BalanceID).

Для того, щоб приєднатись за допомогою коду запрошення, потрібно його отримати від іншого користувача та ввести в відповідне поле. Код запрошення може бути недійсним у різних випадках, наприклад, якщо користувач, який запросив вас – видалив бюджет (якщо всі користувачі

бюджету з нього вийшли) або деактивував код запрошення. Також користувач не може повторно приєднатись в один і той же бюджет. Все це супроводжується відповідними повідомленнями та попередженнями.

При створенні бюджету потрібно вказати ім’я бюджету, обрати валюту зі списку та можливість вказати, чи генерувати код для запрошення стороннього користувача.

Оскільки в одного користувача може бути багато бюджетів. А в одному бюджеті може бути багато користувачів – видніється зв’язок багато до багатьох. Це єдиний випадок такого зв’язку у всій нашій системі. Подібний зв’язок буде позначений навігаційним полем Users.

Управління транзакціями бюджету. Цей розділ відповідає за обіг коштів бюджету. Відповідно сторінка повинна відображати список транзакцій, можливість додавання, видалення (якщо вони не автоматично згенеровані) транзакцій.

Сутність транзакцій (TransactionDescription) має наступні поля: дата (Date), замітки (Notes), автоматично згенерована (AutoGen), баланс (Balance, зовнішній ключ), категорію (TransactionDescriptionCategory, НСЗК), бюджет (Budget, НСЗК). AutoGen – позначка, що транзакцію створив не користувач, а ПриватБанк або інший інтегрований банк. Якщо користувач при створенні транзакції не вказує валюту, вона повинна автоматично визначатись відносно основної валюти бюджету.

Управління можливими категоріями транзакцій. Найменший розділ, котрий відповідає за категорії для транзакцій. Відповідно повинен містити додавання, видалення, перегляд списку категорій.

Сутність категорій (TransactionDescriptionCategory) має наступні поля: назва (Name), дохід (Income), колір (Color), бюджет (Budget, НСЗК). Оскільки сума транзакції не може бути від’ємною, поле Income позначає, чи це дохід, чи витрата відносно балансу бюджету. Колір – це hex або rgb

формат. Назва повинна бути унікальною серед інших категорій, які відносяться до того ж бюджету.

Планувальник. Розділ, відповідно від назви дозволяє користувачу створити абстрактні плани бюджету. Сторінка повинна містити список планів, можливість створення нового плану, видалення та редагування.

Сутність Плану (PlannedBudget) має наступні поля: дата початку (DateStart), дата закінчення (DateEnd), заголовок (Title), опис (Description), запланований бюджет (PlannedBalance, НСЗК), реалізований бюджет (RealizeBalance, НСЗК), категорія (TransactionDescriptionCategory, НСЗК), бюджет (Budget, НСЗК). Якщо користувач не вказує категорію, він може вільно керувати реалізованим балансом та зберегти його. У випадку, коли користувач вказав категорію, то транзакції в інтервалі вказаних користувачем початкової та кінцевої дати підтягуються самостійно, та коригують створений план без можливості редагувати реалізований бюджет.

Також варто виділяти користувачу плани, які вже досягли мети по збору коштів, або прострочили збір.

Наприклад, користувач планує подорож Україною в кінці літа, тому йому до цієї пори потрібно накопичити кошти, він вказує дату до якої потребує накопичень, дату від якої розпочинає накопичення, та суму яку планує накопичити, не вказуючи категорію. Тепер протягом потрібного періоду, він може змінювати реалізований баланс (накопичень).

Також планувальник може відігравати роль фільтра. Наприклад користувач хоче виміряти, скільки коштів він витрачає у квітні грошей на їжу. Він створює план, вказавши початкову дату на перше число квітня, а кінцеву – на останній день квітня, після чого обирає категорію “їжа”. Відповідно він зможе побачити, скільки коштів було витрачено транзакціями по даній категорії в зазначений проміжок часу.

Сервіс конвертації валют(інтеграція з FastForex). Єдиний розділ з вище перечислених, до якого користувач не матиме прямого доступу, його ціль надати сервіс для серверної частини застосунку для зручного конвертування валют. Користувач не явно стикається лише у тому випадку, коли вказує валюту транзакції, що відрізняється від валюти бюджету.

Сервіс повинен надати зручний інтерфейс для обмінів валюти з транзакції, до валюти бюджету, при цьому по курсу, який був зафіксований на момент проведення транзакції.

Основна задача інтеграції – розробити систему, яка в окрему від основної схеми бази даних застосунку буде записувати курси всіх доступних валют сервісу FastForex, та надавати зручний доступ для конвертації валюти. Тобто створити окрему базу даних для курсу валют, яка буде оновлюватись постійно через заданий інтервал звертаючись до FastForex API.

Як додаткова можливість для сторонніх розробників можна додати власний API, який дозволяє подивитись всі підтримуючі валюти, курс між двома валют, та конвертацію суму однієї валюти в іншу.

Інтеграція з Приватбанк. Розділ, який впливає на транзакції бюджету. Користувач, на головній сторінці повинен мати змогу підключити та відключити раніше підключені картки Приватбанку. Інтеграція повинна реалізовувати добавляння транзакцій в бюджет з привязаних карток. Назва категорії транзакцій повинна відповідати назві “Приват24” зеленого кольору. Деталі транзакцій отримувати за допомогою Приват24 API. Дані для запитів зберегти в окремій таблиці.

Сутність Приват24Даних (Privat24Credential) має наступні поля: ідентифікатор мерчанту (MerchantID), пароль мерчанту (MerchantPassword), номер картки (CardNumber), дата початку зчитування (StartDate).

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ

В даному розділі ми розберемо налаштування мережевої інфраструктури використовуючи CloudFlare та NGINX, розробку front-end на основі Node.js, Svelte, DaisyUI (Tailwind) та back-end на основі ASP.NET Core, EF Core, з реалізацією CQRS та бази даних – PostgreSQL.

3.1. Налаштування мережевої інфраструктури

Завдання мережевої інфраструктури – виділення унікального доменного імені, налаштування мережевого доступу до нього (в нашому випадку будемо забезпечувати доступ користувачам до нашої системи).

3.1.1. Резервування доменного імені

Першим кроком буде резервація власного доменного імені. “Домен – частина простору ієрархічних імен мережі Інтернет, що обслуговується групою серверів системи доменних імен (DNS-серверів) та централізовано адмініструється” [2]. Його можна зарезервувати написавши в будь-якому пошуковику: “зарезервувати домен”, та перейти за будь-яким посиланням та придбати його. Як приклад ми зарезервуємо домен на вебресурсі *https://porkbun.com/*. Потрібно зареєструвати обліковий запис, після чого ввести потрібний нам домен в строку запиту на сайті. Наш буде “budgetfrog.space”, обираємо її якщо вона попередньо не зарезервована іншим користувачем, або обираємо доступний домен (Рис. 4).

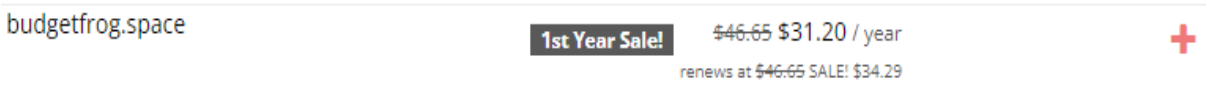

Рисунок 4 – Елемент зі списку доступних доменів (Porkbun).

Переходимо в кошик та оплачуємо резервацію. Після успішної оплати ми зможемо налаштувати nameserver (Як DNS-сервер будемо

використовувати CloudFlare, деталі налаштування будуть згодом). “DNS – ієрархічна розподілена система перетворення імені хоста (комп'ютера або іншого мережевого пристрою) в IP-адресу” [6]. Для того, щоб вказати Nameservers переходимо в розділ Domain Management, обираємо попередньо зарезервований домен та відкриваємо “Details” де змінюємо NAMESERVERS на адресу (naomi.ns.cloudflare.com, walt.ns.cloudflare.com) CloudFlare Nameservers, щоб мати доступ до управління (Рис. 5).

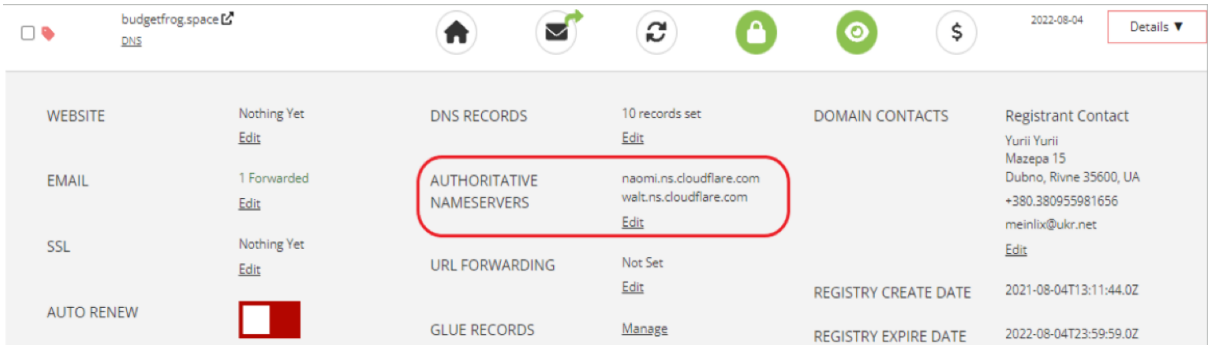

Рисунок 5 – Деталі зарезервованого домену (Porkbun).

За потреби є можливість активувати “AUTO RENEW” який буде знімати щорічно плату за резервацію домену, оскільки його неможливо придбати назавжди. Це потрібно робити, оскільки надзвичайно часто в автоматичному режимі різні брокери скуповують протерміновані домени, за яких вчасно не внесли кошти, після чого вони потребують десятикратні винагороди для того, щоб власники доменів змогли викупити їх назад у власність.

3.1.2. Налаштування DNS для зарезервованого домену

CloudFlare – компанія, яка надає мережеві послуги доставлення контенту, роботи служб безпеки в Інтернеті, захоплює статистику вашого вебресурсу та багато іншого. Вона працює як проміжний сервер між пристроєм користувача який робить запит до нашого ресурсу та хост-сервером [5]. Хостинг – це послуга надання комп’ютерних ресурсів у

безперервному віддаленому доступі. У нашому випадку хостингом буде персональний комп’ютер.

Тож налаштуємо DNS в CloudFlare для нашого зарезервованого домену. Спершу пройдемо реєстрацію нового акаунта. Після чого переходимо на сторінку “Add site” та вводимо наш домен (Рис. 6)

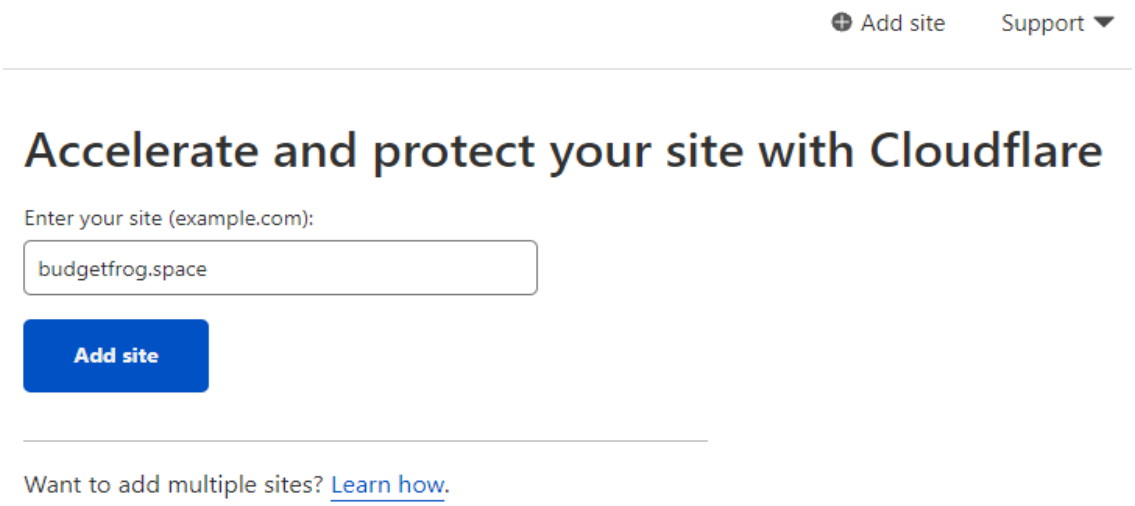

Рисунок 6 – Сторінка додавання сайту (CloudFlare).

Оскільки ми самі будемо для себе хостингом, нам потрібно отримати статичну IP адресу для того, щоб вона не змінювалась (потрібно звертатись до провайдера). Наступний крок – дізнатись IP адресу комп’ютера в глобальній мережі. Найпростіше – це виконати в будь-якому пошуковику запит: “Моя IP адреса”. Наприклад *https://2ip.ua/*, заходимо на нього, та бачимо на головній сторінці нашу публічну IP-адресу (Рис. 7).

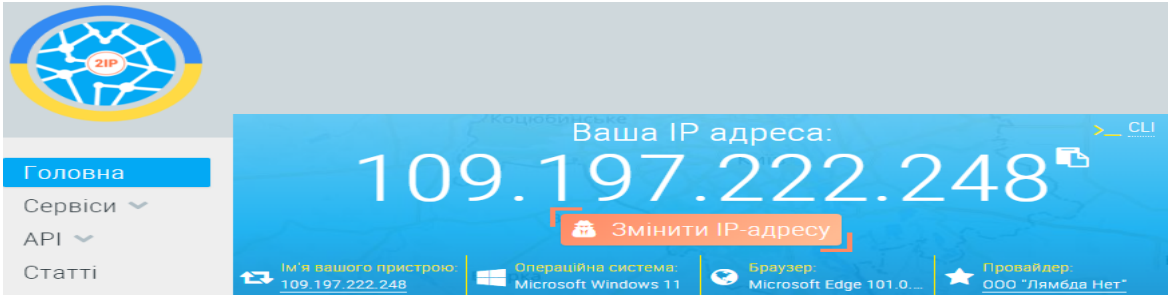

Рисунок 7 – Головна сторінка *https://2ip.ua/*.

Відразу після того, як ми додали сайт та дізнались нашу IP адресу, переходимо до налаштування DNS (Рис. 8). Нам потрібно редагувати та видалити всі записи (якщо такі присутні) та додати власні, натиснувши на кнопку “Add record”. В DNS обов’язковим є внесення запису про зарезервований домен як це зображено на рисунку 8, а саме “budgetfrog.space” в полі **Name**. [6] Далі розбираємось з записами:

- **Type** “A” (Адресний запис, встановлює відповідність між ім'ям і IP-адресою) Потрібно вказати у всіх випадках.
- **TTL** “Auto” – це час кешування на серверах інформації про ваші записи DNS.
- **Proxy status** можемо встановлювати будь-який, але якщо ви в процесі розробки встановите “Proxied” будьте готові, що зміни які ви внесете будуть оновлюватись тривалий час. Ця функція потрібна лише тоді, коли ваш вебсайт буде працювати без активної розробки, вона покращує швидкість відповіді, кешуючи попередню відповідь на запит.
- В полі **Content** вказуємо IP-адресу, де знаходитиметься сервери нашого сервісу. І останній параметр Name відповідає за субдомени, наприклад “api” зареєструє нам домен “api.budgetfrog.space” і всі запити по даній адресі будуть перенаправлятись на введену IP-адресу.

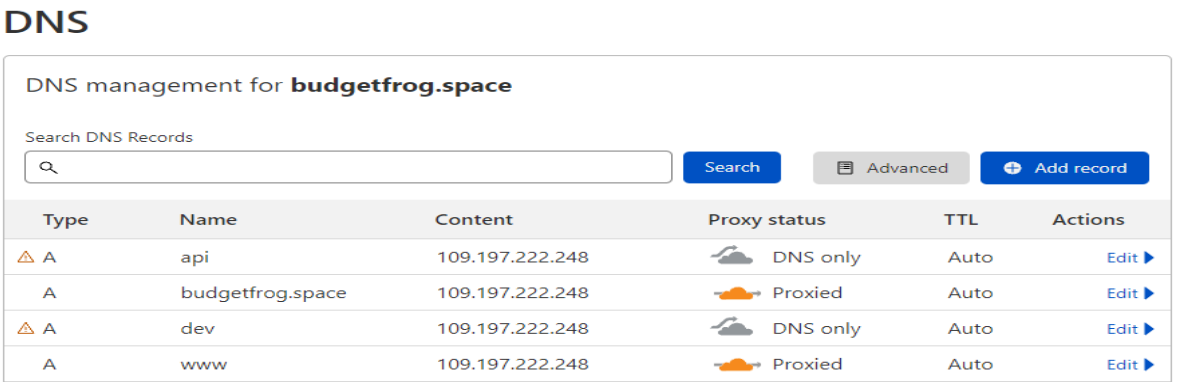

Рисунок 8 – Сторінка налаштування DNS (CloudFlare).

3.1.3 Налаштування зворотного проксі за допомогою NGINX

NGINX – це безкоштовне ПЗ, http-сервер, балансер, популярний і відомий як проксі-сервер, оскільки спочатку розроблявся для зворотного проксі [7], для чого ми його і будемо використовувати в межах локальної мережі.

“Зворотний проксі-сервер – тип проксі-сервера, який ретранслює запити клієнтів із зовнішньої мережі на один або кілька серверів, логічно розташованих у внутрішній мережі” [2].

Для початку потрібно його завантажити з офіційної сторінки під нашу операційну систему (windows 11). Після чого переходимо до налаштування зворотного проксі на комп'ютері. Для цього потрібно відкрити конфігураційний файл (який знаходиться в теці з встановленим програмним забезпеченням “.../conf/nginx.conf”) та вказати наступну конфігурацію (Рисунок 9.1 та 9.2) знаючи, що наша клієнтська частина працюватиме на порті 6060, а серверна на 7070.

```
http {
  include mime.types; # Підключаємо тип контенту, що передається.
  default_type application/octet-stream;
  sendfile on; # Дозволяємо передавати файл.
  gzip on; # Зжимаємо відповіді у формат gzip.

  server { #Розпочинаємо прослуховувати запити на клієнтську частину
    listen 80; #Прослуховуємо порт 80 (стандартний порт)
    # Якщо вхідний запит прийде з наступних доменів, тоді
    #перенаправляємо на адресу, що вказана в proxy_pass
    server_name budgetfrog.space;
    access_log logs/FRONTproxy.log; # Вказуємо логування роботи
    location / {
```

Рисунок 9.1. – Конфігураційний файл NGINX, частина 1.

```
        proxy_pass http://127.0.0.1:6060; # Адрес перенаправлення
      }
    }

    #Аналогічна конфігурації для серверної частини, з іншим server_name
    #та proxy_pass,
    server {
      listen 80;
      server_name api.budgetfrog.space;
      access_log logs/APIproxy.log;
      server_tokens off;
      location / {
        proxy_pass https://127.0.0.1:7070;
      }
    }
}
```

Рисунок 9.2. – Конфігураційний файл NGINX, частина 2.

В результаті зберігаємо конфігурацію, та запускаємо NGINX.

3.1.4 Висновки налаштування мережевої інфраструктури

Ми налаштували мережеву інфраструктуру таким чином, що коли нам з різних доменів приходять запити – вони розподіляються відповідно конфігурації NGINX, за допомогою зворотного проксі.

Якщо ми на персональному комп'ютері запустимо front-end на порті 6060 всі запити з адреси “*budgetfrog.space*” будуть перенаправлятись відповідно до нього. Аналогічно з back-end, якщо він буде запущений на порті 7070, він буде отримувати всі запити за адресою “*api.budgetfrog.space*”.

3.2. Розробка серверної частини

Для розробки back-end були використані наступні інструменти:

1) Visual Studio 2022 – інтегроване середовище розробки.
2) Postman – платформа для тестування API.
3) PostgreSQL – система управління базами даних,
4) Git – розподілена система керування версіями файлів.

Дослідити готовий код серверної частини можна за посиланням *https://github.com/MeinLiX/BudgetFrog*.

3.2.1. Реалізація серверної архітектури.

Наша ціль розробити REST API [9], з уніфікованими відповідями, основну логіку нашого сервісу. Для цього розпочнемо зі створення WebAPI проєкту в Visual Studio (Рис. 10). Під час створення залишаємо стандартні налаштування.

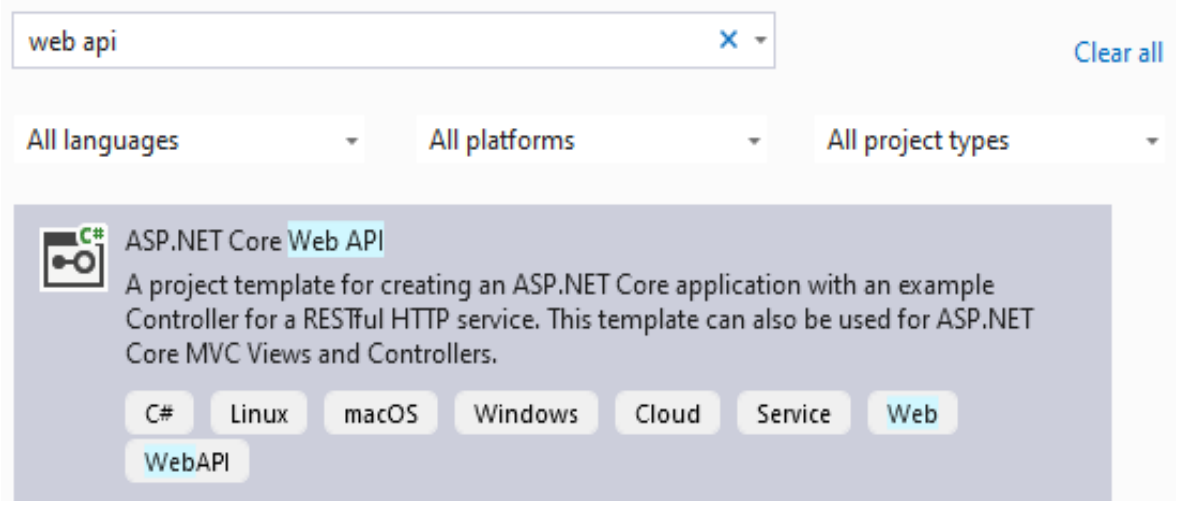

Рисунок 10 – Вибір шаблону WebAPI в Visual Studio.

Після створення отримуємо шаблон фактично пустого проєкту з одним контролером “WeatherForecastController” котрий відразу видаляємо.

Для реалізації CQRS та підключення СУБД до нашого проєкту за допомогою EF Core потрібно встановити відповідні пакети, за допомогою NuGet. У Visual Studio існує візуальний менеджер пакетів, тому відкриємо

його за допомогою правого кліку по назві проєкту, та виберемо “Manage NuGet Packages”, та встановимо наступні пакети як на рисунку 11.

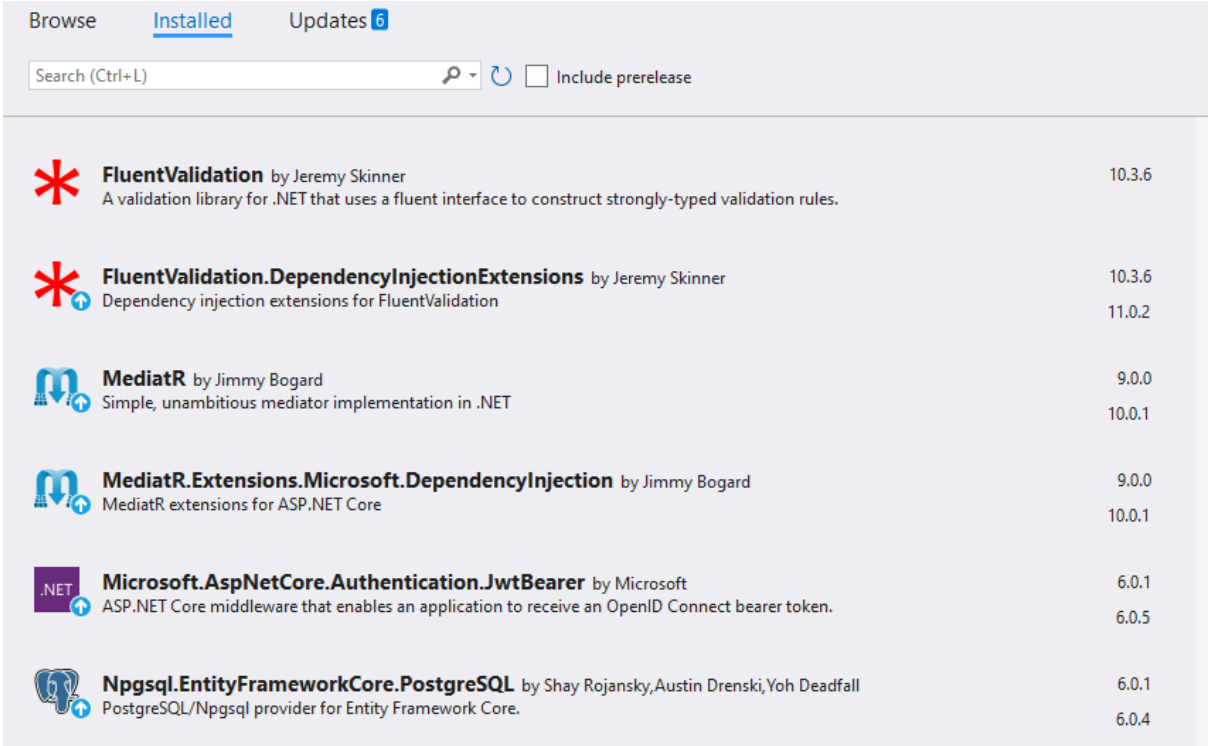

Рисунок 11 – Список NuGet пакетів, потрібних для нашого WebAPI.

FluentValidation – пакет, котрий допоможе зручно валідувати моделі запитів.

MediatR – за допомогою нього будемо реалізовувати розподілення контролерів на команди та запити.

JwtBearer – оскільки ми будемо використовувати аутентифікацію за допомогою JWT токену, цей чудовий пакет нам допоможе в цьому.

Npgsql.EntityFrameworkCore.PostgreS – провайдер PostgreSQL для EF Core, як ми вже зазначали у вимогах, тут може бути інший провайдер, але це не змінить код проєкту.

Після того, коли ми розібрались з потрібними пакетами, перейдемо до структури проєкту, створимо відповідні директорії, в яких буде наступне:

1) **Context** – налаштування провайдерів EF Core.
2) **Controller** – контролери, котрі описують endpoints.
3) **Features** – мікросервіси та реалізація CQRS (команди та запити).
4) **Middleware** – мікросервіси на рівні ASP.NET.
5) **Models** – моделі, котрі будуть описувати нашу базу даних.
6) **PipelineBehaviours** – мікросервіси, на рівні MediatR.
7) **Utils** – допоміжні утиліти для застосунку.

Підключимо MediatR, FluentValidation та JwtBearer, для цього в Program.cs додамо наступні рядки, які дозволяють увімкнути аутентифікацію за допомогою JWT токену, створювати класи для команд, запитів та для валідації моделей:

```
builder.Services.AddAuthentication(JwtBearerDefaults.Authentic
ationScheme).AddJwtBearer(jwtBearerOptions =>
jwtBearerOptions.TokenValidationParameters =
AuthEngine.TokenValidationParameters);
builder.Services.AddMediatR(Assembly.GetExecutingAssembly());
builder.Services.AddValidatorsFromAssembly(typeof(Program).Ass
embly);
```

Зразу для уніфікацій відповідей від сервера створимо обгортку моделі відповіді. Відповідно додамо в Utils теку Wrapper з інтерфейсом IResult та клас Result (Додаток А), котрі будуть екземпляром відповіді. Вони будуть містити поля про успішність операції, час виконаної операції, повідомлення за потреби, та поле в якому будуть передаватись різні об’єкти. Надалі при будь-якому повернені відповіді з сервера передаємо її за допомогою створеного класу Result.

3.2.2. Побудова моделей та додавання контексту бази даних

На даному етапі нам потрібно створити та додати всі потрібні моделі, які будуть описувати базу даних, відповідно до тих, які прописані нами в розділі вимоги до системи.

Створюємо модель користувача. Всі моделі будуть успадкованими від ModelBase (Додаток А), це дозволяє не вказувати первинний ключ у всіх моделях.

```
public class AppIdentityUser : ModelBase
{
    8 references
    public string Email { get; set; }

    2 references
    public string FirstName { get; set; }

    2 references
    public string LastName { get; set; }

    5 references
    public string Password { get; set; }

    1 reference
    public string PhotoUrl { get; set; }

    1 reference
    public List<Budget> Budgets { get; set; } = new();
}
```

Рисунок 12 – Клас, що описує модель користувача.

Модель для EF Core – це стандартний клас, в котрого будуть зчитуватися лише поля описуючи таблицю бази даних. Також можна помітити поле “Budgets”, яка має тип “List<Budget>”, своєю чергою у моделі “Budget” буде поле “Users”, це вказує на зв’язок багато до багатьох, та являється навігаційною сутністю. Вочевидь можна обійтись без будівельного класу, тоді EF Core самостійно підбере типи колонок, відповіді типам полів класу, але ми будемо контролювати цей процес, створивши “AppIdentityUserBuilder” (Рис. 13):

- “Property” – вказує поле, котре налаштовуємо.
- “HasColumnType” – вказує на тип, котрий потрібно встановити.
- “IsRequired” – вказує чи може колонка приймати значення NULL.

```
public static class AppIdentityUserBuilder
{
    1 reference
    public static void BuildAppIdentityUser(this ModelBuilder MB)
    {
        MB.BuildModelBase<AppIdentityUser>();

        MB.Entity<AppIdentityUser>()
          .Property(u => u.Email)
          .HasColumnType("VARCHAR(128)")
          .IsRequired();

        MB.Entity<AppIdentityUser>()
          .Property(u => u.FirstName)
          .HasColumnType("VARCHAR(16)")
          .IsRequired();

        MB.Entity<AppIdentityUser>()
          .Property(u => u.LastName)
          .HasColumnType("VARCHAR(16)")
          .IsRequired();

        MB.Entity<AppIdentityUser>()
          .Property(u => u.Password)
          .HasColumnType("VARCHAR(64)")
          .IsRequired();

        MB.Entity<AppIdentityUser>()
          .Property(u => u.PhotoUrl)
          .HasColumnType("TEXT");
    }
}
```

Рисунок 13 – Будівельний клас моделі користувача.

По аналогії розробляємо всі потрібні моделі, після створення всіх моделей для реалізації вимог до системи, отримаємо відповідність до рисунка 14, детальніше дослідити моделі можна за допомогою GitHub.

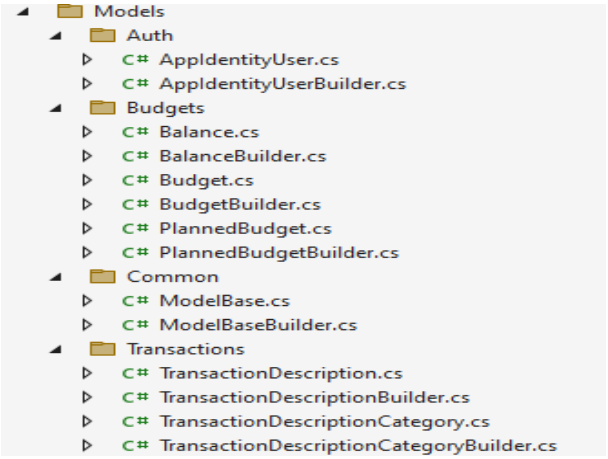

Рисунок 14 – Тека Models після створення моделей на вимоги системи.

Для того, щоб всі розроблені класи з розділу “Models” ініціалізувати в EF Core, потрібно створити відповідний клас контексту, який буде реалізувати собою схему в базі даних встановленого провайдеру. Та підключити його

до ASP.NET Core middleware в файлі “Program.cs”.

Перший крок це створення контексту (Рис. 15). Він охоплює всі описані моделі, відповідно один “DbSet” – окрема таблиця. Але кращим варіантом буде описання інтерфейсу, та після – його реалізацію, оскільки це може знадобитись, якщо в проєкті буде декілька провайдерів EF Core, та розробник побажає переналаштувати певні залежності тощо. Також в “OnModelCreating” потрібно викликати написані нами розширення для “ModelBuilder”.

```
public class BudgetAppContext : DbContext, IBudgetAppContext
{
    0 references
    public BudgetAppContext(DbContextOptions<BudgetAppContext> options)
        : base(options)
    {
        //Database.EnsureDeleted();
        Database.EnsureCreated();
    }

    6 references
    public DbSet<AppIdentityUser> AppIdentityUsers { get; set; }
    2 references
    public DbSet<Balance> Balances { get; set; }
    20 references
    public DbSet<Budget> Budgets { get; set; }
    9 references
    public DbSet<PlannedBudget> PlannedBudgets { get; set; }
    9 references
    public DbSet<TransactionDescription> TransactionsDescription { get; set; }
    12 references
    public DbSet<TransactionDescriptionCategory> TransactionDescriptionCategories { get; set; }
    2 references
    public DbSet<Privat24Credential> Privat24Credentials { get; set; }


    20 references
    public async Task<int> SaveChangesAsync() => await base.SaveChangesAsync();

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.BuildAppIdentityUser();
        modelBuilder.BuildBalance();
        modelBuilder.BuildBudget();
        modelBuilder.BuildPlannedBudget();
        modelBuilder.BuildTransactionDescription();
        modelBuilder.BuildTransactionDescriptionCategory();
        modelBuilder.BuildPrivat24Credential();
    }
}
```

Рисунок 15 – Клас контексту основної бази даних сервісу.

Вирішальний крок по інтеграції контексту EF Core буде додавання його до сервісу ASP.NET Core в “Program.cs” наступні строки коду, які додають в сервіси можливість звернутися до бази даних, отримуючи її в

будь-якому виконувальному класі через конструктор класу, вказуючи аргументом потрібний контекст:

```
builder.Services.AddDbContext<BudgetAppContext>(options =>
options.UseNpgsql(builder.Configuration["ConnectionStrings:BudgetApp
IdentityDB"]));
builder.Services.AddScoped<IBudgetAppContext>(provider =>
provider.GetService<BudgetAppContext>() ?? throw new
NullReferenceException());
```

Але щоб попередньо доданий код працював, нам потрібно в конфігураційному файлі “appsettings.json” вказати “ConnectionStrings”, а відповідно в ньому “BudgetAppIdentityDB” з наступним значенням (в нашому випадку PostgreSQL встановлений за замовчуванням на тому ж комп’ютері, тобто локально):

```
"ConnectionStrings":
{
"BudgetAppIdentityDB":"Host=localhost; Port=6666;
Database=BudgetAppIdentity; Username=postgres; Password=password"
}
```

На цьому, контекст успішно під'єднано, ми тепер можемо у контролерах підключати базу даних, а точніше ORM EF Core, який буде з нею працювати. У всіх випадках робота з контекстом буде реалізовуватись за допомогою інтерфейсу Linq to object, що дозволить реалізувати весь потрібний функціонал з базою даних зручним методом.

3.2.3. Інтеграція та розробка відповідних бібліотек для взаємодії зі сторонніми застосунками.

Ціль інтеграцій – розширення функціональності власного застосунку. Нас очікує дві інтеграції, одна з яких реалізація власного конвертера валют який буде зберігати дані в окремій базі даних, курс яких буде витягувати з FastForex API, та інтеграція з ПриватБанком, для

отримання транзакцій.

Конвертер валют. Для його реалізації додамо мікросервіс, котрий буде запускатись при запуску нашого застосунку та працювати паралельно з ним. Для цього в “Program.cs” зразу допишемо:

```
builder.Services.AddScoped<ExchangeRateService>();
builder.Services.AddHostedService<ExchangeRatesFFUpdaterHostedServic
e>();
```

“ExchangeRatesFFUpdaterHostedService” буде відсилати в певний проміжок часу запит до FastForex API для актуалізації курсу валют, та зберігати його до бази даних. Тому додаємо ще одну базу даних “ExchangeRateContext” аналогічно тому, як ми це робили в попередньому пункті (модель потрібних таблиць можна взяти з їх сайту). Для зручності додамо в базову модель логіку конвертації, щоб не описувати її кожен раз при спробі конвертації валют (Рис. 16).

```
public decimal GetRate(string from, string to)
{
    if (from == to)
        return 1;

    if (from == @base && to != @base)
        return results[to];

    if (from != @base && to == @base)
        return 1 / results[from];

    return results[to] / results[from];
}

public decimal Exchange(decimal amount, decimal rate) => amount * rate;

public decimal Convert(string from, string to, decimal amount) => Exchange(amount, GetRate(from, to));
```

Рисунок 16. – Методи розширення базової моделі FastForex.

В результаті отримуємо наступний сервіс “ExchangeRateService”, котрий буде нам допомагати конвертувати валюти скрито від користувача. Головний метод сервісу буде знаходити потрібний нам курс по найближчій заданій даті та змінювати стару валюту на нову. На рисунку 17 можемо побачити логіку звернення методу до бази даних, щоб отримати актуальний курс приближений до дати переданого параметра

“dateExchange”.

```
public FFbase GetExchange(DateTime dateExchange)
{
    try
    {
        return _ER_context.FFbase
                          .Include(ffb=>ffb.results)
                          .ToList()
                          .OrderBy(ffb => TimeRange(dateExchange,ffb.date))
                          .First();
    }
    catch
    {
        throw new AppException("[Currency service] Exchange rates not found.");
    }
}
```

Рисунок 17 – Метод сервісу “ExchangeRateService” для отримання курсу.

Наступний метод “ChangeCurrency” сервісу конвертує валюту переданого балансу на нову (назва нової валюти передається в аргументі “to”), за курсом актуальним на момент часу вказаним в “data” (Рис. 18). Відповідно, ми тепер можемо викликати сервіс “ExchangeRateService” в потрібному місці та без дублювання логіки викликати потрібні методи, які швидко та зручно конвертують валюту, навіть без доступу до FastForex.

```
public async Task<Balance> ChangeCurrency(Balance bal, string to, DateTime date)
{
    Balance balance = new() { Amount = bal.Amount, Currency = bal.Currency, ID = bal.ID };
    var exchangeRates = GetExchange(date);

    balance.Amount = exchangeRates.Convert(balance.Currency, to, balance.Amount);
    balance.Currency = to;

    return balance;
}
```

Рисунок 18 – Метод сервісу “ExchangeRateService” для конвертації валюти.

Інтеграція з ПриватБанк. Інтеграція з банком – це вже трохи нестандартний процес, оскільки взаємодія з ПриватБанк API відбувається у форматі XML, а також всі запити та відповіді захищаються відповідною сигнатурою, котра розраховується з md5 та sha1. Через це створимо окрему бібліотеку спеціально для реалізації запитів до ПриватБанку. Відвідавши вебсторінку документації для виписки транзакцій за рахунком:

https://api.privatbank.ua/#p24/orders, можемо зробити висновок про структуру запиту та відповіді. Всі параметри запиту описані в документації.

Для реалізації задуманого створюємо новий проєкт privat24.NET, додаємо всі потрібні моделі для XML. Для того, щоб швидко та без помилок описати моделі можна скористатися функціональністю Visual Studio, яка дозволяє, маючи приклад структури відповіді та запиту, згенерувати класи, які будуть серіалізуватись з об'єкта в XML та навпаки. Для цього скопіюємо в буфер обміну потрібну структуру, відкриємо Visual Studio > "Edit" > "Paste Special" > "Paste XML As Classes", після чого ми отримуємо згенеровані класи, котрі залишилось розбити по файлах, або залишити в одному файлі. Далі створюємо метод, який буде генерувати сигнатуру, та реалізуємо методи, які будуть звертатись по API та отримувати інформацію по транзакціях. Інтеграція надалі потребуватиме виклику розробленої нами бібліотеки при запиті користувача на додавання нового рахунку ПриватБанку, та при запиті користувача на список транзакцій (Якщо до бюджету буде під'єднаний рахунок банку, тоді ми змушені будемо відправити запит до банку, щоб він нам повернув виписку по карті, яку ми форматуватимемо під власну структуру транзакцій). Для того, щоб сервер міг надсилати запити до ПриватБанку, потрібно мати "MerchantID", "MerchantPassword", "NumCard", відповідно для збереження цих даних потрібно додати нову таблицю.

3.2.4. Розробка основної логіки.

Зазвичай вся логіка відбувається та реалізується в контролерах, чим нагромаджує їх, ускладнює подальшу підтримку, збільшує ризики дублювання коду тощо. Саме для цього ми реалізуємо підхід CQRS, за допомогою використання MediatR, що дозволить розділити всі запити на 2 типи та розвантажити контролер. Контролер буде тепер відповідати лише

за налаштування endpoints (REST API) та в ньому будуть використовуватись потрібні команди чи запити, котрі ми будемо реалізовувати в теці Features. Ми розберемо написання валідації, команд та запитів для категорій, а всі інші контролери будуть доступні для перегляду в GitHub репозиторії.

Приклад класу, який реалізує команду (Рис. 19). Бачимо, що ми реалізуємо інтерфейс IRequest та IRequestHandler (MediatR). В основному класі NameCommand ми можемо додати публічні поля, котрі повинні бути в запиті, для яких налаштуємо потрібну валідацію за допомогою FluentValidation в окремому класі, називаючи його NameCommandValidator. Варто зазначити, що на рівні реалізації Command та Query ідентичні.

```
public class NameCommand : IRequest<Result<object>>
{
    1 reference
    public string BudgetID { get; set; } = "";
    0 references
    private Guid GetBudgetID { get => Guid.Parse(BudgetID); }

    1 reference
    public class NameCommandHandler : IRequestHandler<NameCommand, Result<object>>
    {
        private readonly IBudgetAppContext _context;
        0 references
        public NameCommandHandler(IBudgetAppContext context)
        {
            _context = context;
        }
        0 references
        public async Task<Result<object>> Handle(NameCommand command, CancellationToken cancellationToken)
        {
            //command.GetBudgetID;
            return Result<object>.Success();
        }
    }
}
```

Рисунок 19 – Вигаданий приклад класу типа команди з полем “BudgetID”.

Додамо директорію “TransactionDescriptionCategoryFutures”, далі розіб’ємо її на 3 тека “Commands”, “Queries”, “Validators” відповідно назвам розуміємо, що там повинно знаходитись. Створимо запит, який буде повертати всі доступні категорії (список категорій), для цього він повинен від користувача отримати ідентифікатор бюджету (Рис. 20) та виконати запит до бази даних, вказавши пошук серед всіх транзакцій такі, які

належать до заданого бюджету, та користувач до них має доступ (користувач находиться в бюджеті), сортуємо за іменем та повертаємо результат.

```
public class GetBudgetCategoriesQuery :
        IRequest<Result<List<TransactionDescriptionCategory>>>
{
    3 references
    public string BudgetID { get; set; }
    1 reference
    private Guid GetBudgetID { get => Guid.Parse(BudgetID); }

    1 reference
    public class GetBudgetCategoriesQueryHandler :
            IRequestHandler<GetBudgetCategoriesQuery,
            Result<List<TransactionDescriptionCategory>>>
    {
        private readonly IBudgetAppContext _context;
        private readonly SignInManagerService _signInManager;
        0 references
        public GetBudgetCategoriesQueryHandler(IBudgetAppContext context, SignInManagerService signInManager)
        {
            _context = context;
            _signInManager = signInManager;
        }
        0 references
        public async Task<Result<List<TransactionDescriptionCategory>>> Handle(GetBudgetCategoriesQuery query,
                                                                               CancellationToken cancellationToken)
        {
            var user = await _signInManager.GetUser();
            List<TransactionDescriptionCategory> categories =
                _context.TransactionDescriptionCategories
                        .Where(c => c.Budget.ID == query.GetBudgetID && c.Budget.Users.Contains(user))
                        .OrderBy(t => t.Name)
                        .ToList();

            return Result<List<TransactionDescriptionCategory>>.Success(categories);
        }
    }
}
```

Рисунок 20 – Запит на список категорій.

На рисунку 21 зображено код контролеру, який викликатиме розроблений запит. Поглянувши на метод контролеру, ми можемо спостерігати встановлення за допомогою атрибутів ASP.NET Core статус кодів, котрі може повернути метод, та метод запиту (get), також атрибутом позначено, що budgetID є частиною path. (це означає, що користувач передає ідентифікатор бюджету в URL, приклад: “*localhost:7070/category/123456*”, де 123456 – потрібний нам ідентифікатор) Дослідити BaseController можливо в Додатку А, він лише ініціалізує атрибути, котрі вказують, що ми розробляємо API контролер, та URL до контролера буде залежати від його назви.

```
public class TransactionDescriptionCategoryController : BaseController
{
    private readonly IMediator _mediator;

    public TransactionDescriptionCategoryController(IMediator Mediator)
    {
        _mediator = Mediator;
    }

    /// <summary>
    /// Get list of categories.
    /// </summary>
    [HttpGet("{budgetID}")]
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status400BadRequest)]

    public async Task<IActionResult> GetCategories([FromRoute] string budgetID)
    {
        return Ok(await _mediator.Send(new GetBudgetCategoriesQuery() { BudgetID = budgetID }));
    }
```

Рисунок 21 – Контроллер категорій з методом GET.

Наступний крок тестування створеного endpoint за допомогою Postman. Для чого створимо новий запит в Postman, додамо посилання "*https://localhost:7070/TransactionDescriptionCategory/{budgetID}*", де {budgetID} – це ідентифікатор бюджету формату GUID. При виконанні запиту отримуємо результат (Рис 22). Надалі для тестування валідації вказуємо різні поля, обов'язкові та необов'язкові, вказуємо різні значення, які коректні та ні, та спостерігаємо за бізнес-логікою, якщо відповідь некоректна – шукаємо причину, та виправляємо проблему.

Рисунок 22 – GET запит до списку категорій, за допомогою Postman.

Повна колекція всіх можливих запитів у Postman, яка використовувалась для тестування back-end є доступною для перегляду на GitHub репозиторії за посиланням *https://github.com/MeinLiX/BudgetFrog*.

3.3. Розробка клієнтської частини

Для розробки front-end були використані наступні інструменти:

1) WebStorm 2022 – інтегроване середовище розробки.
2) NPM – менеджер пакунків.
3) Microsoft edge DevTools.
4) Git.

3.3.1. Реалізація клієнтської архітектури.

Першим кроком потрібно створити Node.js проєкт та встановити всі потрібні пакети для розробки Svelte застосунку. Для цього скористаємось NPM, введемо в терміналі команду ініціалізації проєкту, а саме: `npm init`, та пройдемо поетапне налаштування, яке буде в терміналі. Далі за допомогою `npm install <PackageName>`, встановлюємо потрібні пакети.

Основні пакети котрі потрібно встановити: svelte, rollup, svelte-spa-router, tailwindcss, daisyui, axios. Весь перелік можна побачити в конфігураційному файлі “package.json” (Додаток Б), також з нього беремо скрипти, для компілювання та налагодження, розробки нашого front-end проєкту – “build”, “dev”, “start”. Надалі для під час розробки запускаємо за допомогою терміналу проєкт `npm run dev`, це дозволить нам не перезапускати самостійно проєкт, оскільки він буде самостійно перезапускатись, якщо буде змінено будь-який файл.

Після того, коли ми розібрались з потрібними пакетами, перейдемо до структури проєкту, створюємо головну теку “src” та в ній відповідні директорії, в яких буде наступне:

1) **components** – готові компоненти, які використовуватимуться з views.
2) **routes** – налаштування доступних сторінок для відвідування.
3) **services** – описуємо потрібні для використання сервіси, винесені загальні функції.
4) **stores** – описуєм доступу до об’єктів LocalStorage та svelte/store.
5) **views** – головні svelte компоненти для відображення сторінок.

В результаті можемо реалізувати MVC підхід, де контролером буде виступати функція, яка буде звертатись до нашого сервера та обробляти відповідь. Моделі всі описані на back-end частині, та компоненти View відповідно повинні бути описані в розділі views.

3.3.2. Розробка модуля для запитів до серверної частини.

Оскільки ми знаємо всі доступні endpoints в нашому сервері, організуємо функцію для запитів, та побудуємо об’єкт, який буде приймати у вигляді параметрів потрібні поля для запиту, надсилати його (відносно місця розташування) та повертати відповідь від сервера.

На рисунку 23 можемо дослідити структуру головної функції, а саме “Request”, яка реалізує запити до серверної частини. Для аутентифікації в серверній частині використовуємо додатковий заголовок запиту “Authorization” та JWT ключ. Якщо JWT дійсний, тоді ми зможемо мати доступ до всіх запитів, інакше – отримаємо відповідь, що користувач не авторизований, якщо будемо звертатись до методів, які потребують аутентифікацію. JWT ключ зберігається в “LocalStorage” браузера під час реєстрації чи авторизації. (Методи для взаємодії з “LocalStorage” можна переглянути в Додатку Б). Запит здійснюється за допомогою бібліотеки “axios”, що дозволяє нам зручно передавати параметри, встановлювати метод запиту. Функція повертає Promise, що дозволяє використовувати тіло `try{}catch{}` та виконувати паралельні запити, використовувати `async/await`. В Додатку Б можна ознайомитись зі структурою методів, які

побудовані на основі об’єкта, для зручної навігації під час розробки для виконання запитів до серверної частини.

```
export const Request = async (path :string  = `/`, method :string  = `get`, data :null  = null, params :null  = null) => {
    try {
        const res = await axios({
            url: `${api}${path}`,
            method: method,
            data: data,
            params: params,
            headers: {
                'Authorization': `bearer ${LS.Get( key: `jwt`)}`
            }
        });
        if (res.data?.succeeded === false) {
            return Promise.reject( reason: {...res.data, status: res.status});
        }
        return Promise.resolve( value: {...res.data});
    } catch (err) {
        if (err.response.status === 401) {
            LS.Set("jwt", null)
        }
        return Promise.reject( reason: {...err.response.data, status: err.response.status});
    }
};
```

Рисунок 23 – Функція “Request” для запитів до серверної частини.

3.3.3. Розробка компонентів.

Наша ціль розробити сторінки, якими буде приємно та зручно користуватись, саме для цього будемо використовувати компонентну бібліотеку DaisyUI (Tailwind), яку ми описували в розділі 2.3. Огляд технологій для клієнтської частини.

Створимо компоненту “Root.svelte”, яка буде відображатись при перегляді головної сторінки, та Budgets, яка є другою частиною компоненти “Root”, але винесена в окремий файл, оскільки доступна лише для авторизованих користувачів. “Root.svelte” повинна відображати 3 стани:

1. Авторизація користувача.
2. Реєстрація користувача.
3. Головна сторінка авторизованого користувача(компонента Budgets).

Для реалізації переліченої логіки потрібно створити 3 компоненти, одна для авторизації (SignIn), реєстрації (SignUp), та Budgets (їх реалізація доступна в GitHub репозиторії).

```
<script>
    import {auth} from "../stores";
    import Budgets from "./Budgets.svelte";
    import SignIn from "../components/auth/SignIn.svelte";
    import SignUp from "../components/auth/SignUp.svelte";

    let registrationField = false;

    const switchRegField = () => (registrationField = !registrationField);
</script>

{#if !$auth}
    <div class="root">
        <div class="root__illustration"/>
        <div class="root__container">
            <div>
                {#if !registrationField}
                    <SignIn/>
                    <label on:click={switchRegField} class="label-text text-center mt-5" style="..."> Sign up for a new account </label>
                {:else}
                    <SignUp/>
                    <label on:click={switchRegField} class="label-text text-center mt-5" style="..."> Sign in already have account </label>
                {/if}
            </div>
        </div>
    </div>
{:else}
    <Budgets/>
{/if}
```

Рисунок 24 – Код компоненти “Root.svelte”.

На рисунку 24 можемо побачити код компоненти, інтеграцію зовнішніх компонент та можливості Svelte. Для перевірки, чи являється користувач авторизованим – використовуємо об’єкт “auth”, який являє собою реалізацію “svelte/store”, що дозволяє нам його змінювати та відстежувати у всіх компонентах (які звертаються до даного об’єкту) в режимі реального часу, а не лише під час оновлення сторінки.

Конструкція {#if вираз} дозволяє відображати компоненті Svelte тіло (в нашому випадку з виразом auth – це елемент “div” з класом “root” та весь її вміст) якщо вираз є дійсним, інакше показуємо компоненту “budgets”. Змінна “registrationField” відповідає за те, яку компоненту показувати, для реєстрації чи авторизації.

“on:click” є частиною Svelte та являє собою властивість “onclick” стандартного HTML. Ми ж передаємо елементу“label” функцію “registrationField”, для зміни відображення компоненти з авторизації на реєстрацію, та навпаки, якщо користувач натисне на елемент “label”. Для

того, щоб ми могли в компоненту передати власну властивість, потрібно в ній і блоці “script” оголосити змінну з ключовим словом export (приклад. `export let lyrics;`).

3.4. Робота сервісу

Розберемо основний функціонал вже готового додатку. Для доступу до ресурсу скористаємось посиланням budget frog.space, яке ми налаштовували в розділі налаштування мережевої інфраструктури.

Весь код доступний в GitHub репозиторії за посиланням: https://github.com/MeinLiX/BudgetFrog.

Неавторизований користувач не може відвідувати сторінки сайту, йому буде лише доступна авторизація та реєстрація (Рис. 25).

Рисунок 25 – Головна сторінка для неавторизованого користувача.

Головна сторінка зображена на рисунку 26 (авторизованого користувача) відображає список бюджетів користувача, можливість

створити новий бюджет або приєднатися до існуючого бюджету за допомогою токену запрошення (Рис. 26).

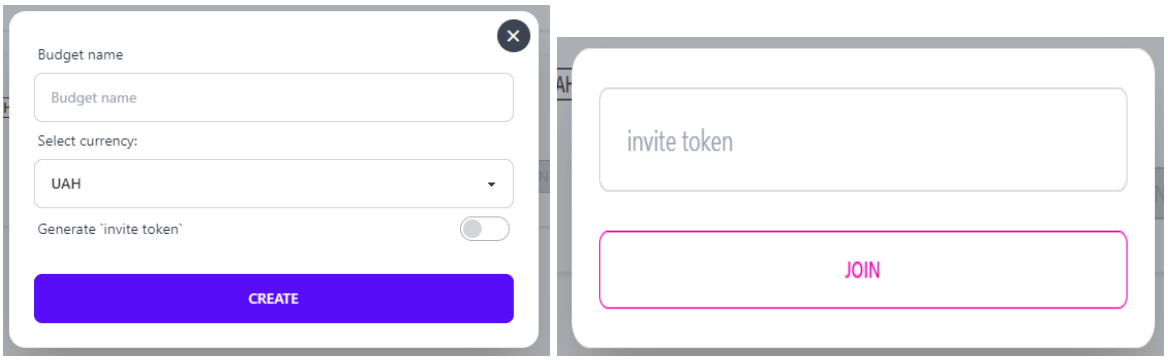

Рисунок 26 – Модальні вікна для створення та приєднання бюджету.

Кожен бюджет дозволяє дізнатися кількість його користувачів, якщо навести курсором мишки на його назву. Також функціональні кнопки “Leave” – залишити бюджет, “INVITE TOKEN” – скопіювати токен для запрошення, “OPEN” – відкрити транзакції бюджету. Якщо користувач натисне на логотип ПриватБанку, він зможе добавити або видалити прив’язані картки.

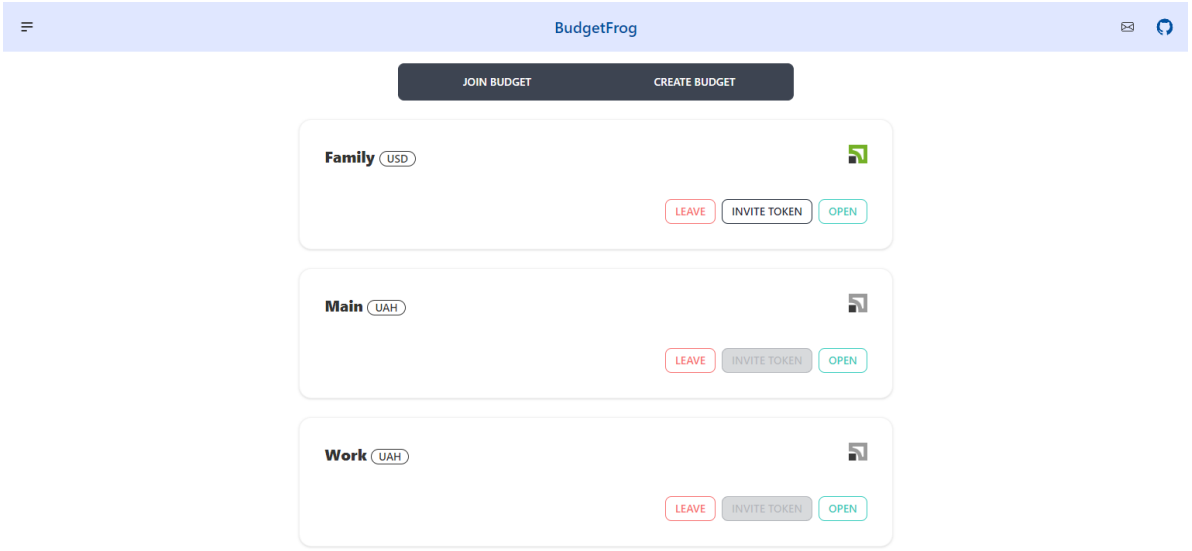

Рисунок 27 – Головна сторінка, список бюджетів.

Перейдемо на сторінку транзакцій, за допомогою натискання на елемент “OPEN”. Потрапивши на сторінку з транзакціями (Рис. 28) додається додаткова навігаційна панель жовтого кольору, за допомогою ми

можемо відкрити плани, категорії, статистику обраного бюджету (його назва відображається з самого ліва панелі, натиснувши на неї відкривається сторінка зі списком транзакцій).

За допомогою елементу “ADD TRANSACTION” користувач може додати нову транзакцію, вказавши категорію, опис (не обов’язково), дату транзакції, суму та валюту транзакції, після чого вона відобразиться в загальному списку. Також на рисунку 28 видно, що транзакції, які були зроблені за допомогою прив’язаної карточки ПриватБанку – редагування не підлягають, подібні транзакції можна забрати лише всі одночасно, відв’язавши картку від бюджету на головній сторінці. Також варто зазначити, що у користувача є можливість прив’язати декілька карт до одного бюджету, що може бути зручно для деяких задач.

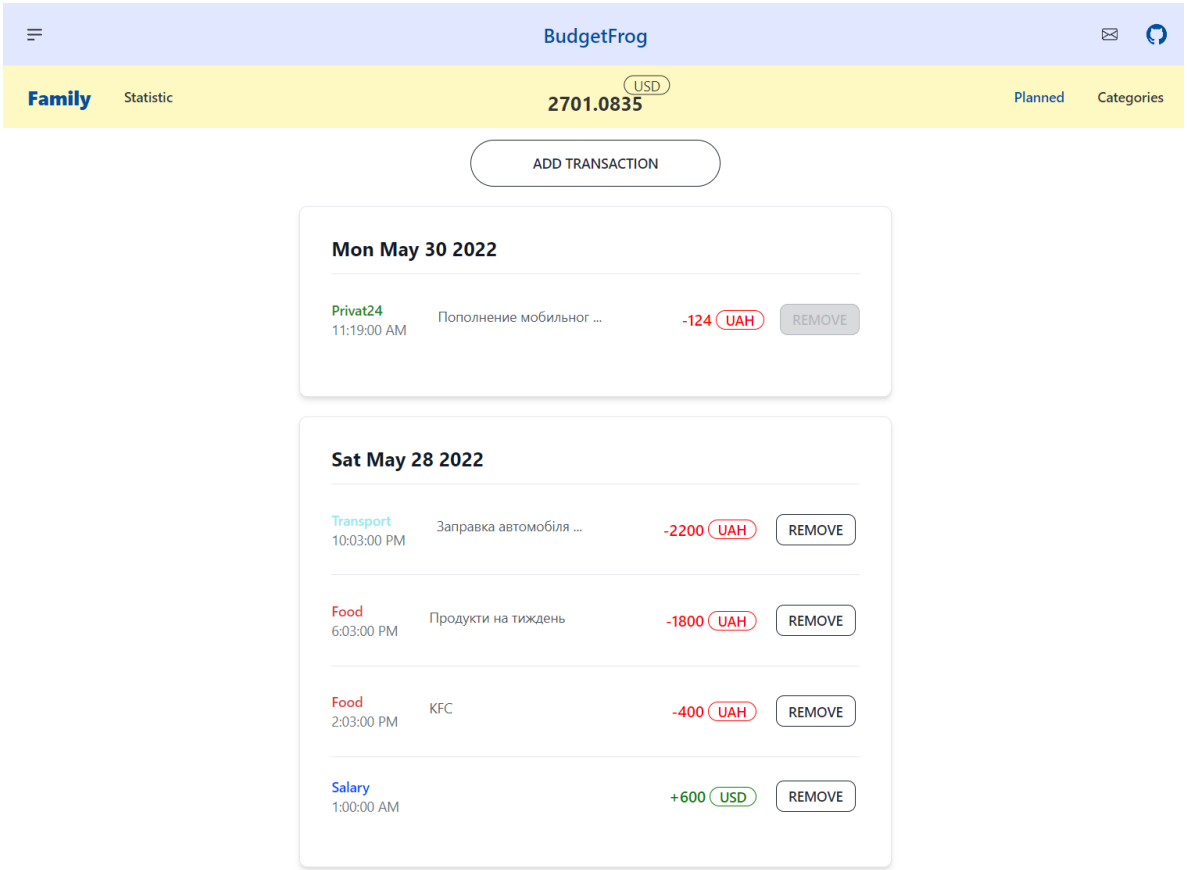

Рисунок 28 – Список транзакцій бюджету “Family”.

Планувальник бюджету унікальна можливість розробленого нами сервісу, яка дозволяє слідкувати за витратами, планувати накопичення тощо. Приклад використання можна побачити на рисунку 29. У користувача є можливість самостійно редагувати реалізований бюджет, якщо він не вказував категорію. Зелені – виконані, червоні – прострочені.

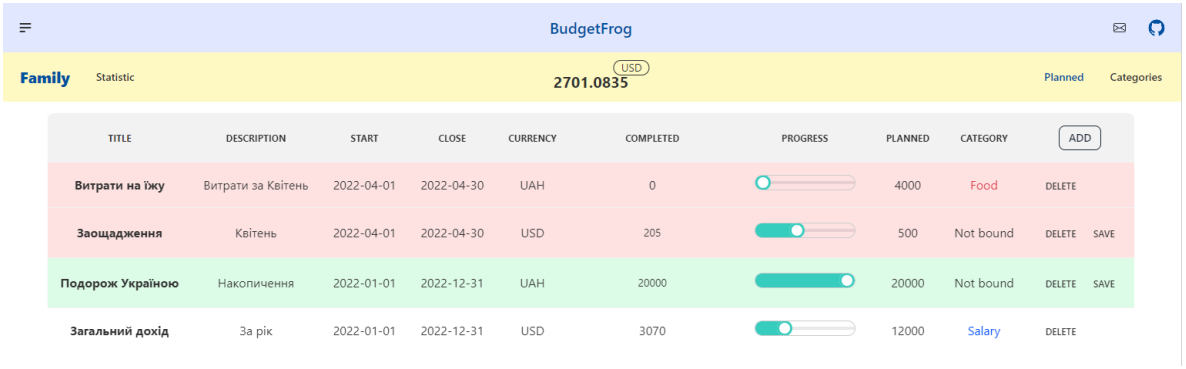

Рисунок 29 – Планувальник бюджету “Family”.

Для кожного користувача доступна статистика бюджету (Рис. 30), де відображається у вигляді графіку частота транзакцій з різними категоріями протягом року. В поле для вводу, потрібно обрати рік, котрий потрібно проаналізувати.

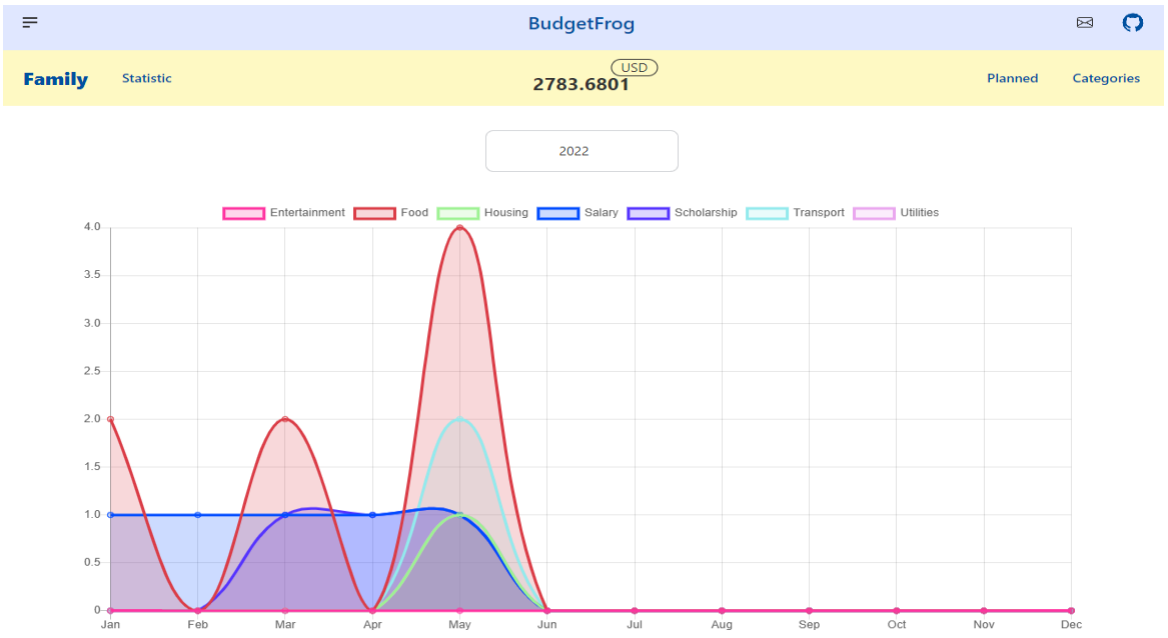

Рисунок 30 – Статистика бюджету “Family”.

ВИСНОВОК

Попри те, що існують різні системи відстежування бюджету, вони недосконалі, далеко не всі вони підтримують інтеграції з українськими банками та мають деякі недоліки, такі як відсутність планування, використання однієї валюти та тому подібне. Чим більше буде існувати сервісів для трекінгу бюджету і чим інтуїтивнішими та зрозумілими вони будуть для використання непідготовленим користувачам, тим вище буде рівень планування бюджету у населення, зокрема буде покращуватись добробут людей, що і ставили за мету.

У результаті виконання дипломної роботи була побудована послідовність кроків для повного налаштування сервісу з нуля, а саме налаштування мережевої архітектури та розробки системи з використанням популярних технологій, власною реалізацією CQRS підходу на серверній частині та MVC на клієнтській. Ми розібрались з новим підходом до проектування back-end архітектури CQRS та його програмною реалізацією на базі ASP.NET Core, яка дозволяє розподілити та розвантажити контролери на “Query” та “Command” та розробили моделі для EF Core і контролери, які реалізують REST API. Також ми використали з метою розповсюдження та популяризації нового фреймворку Svelte для побудови клієнтського застосунку на основі MVC патерну. Був розроблений застосунок, готовий та зручний для його подальшого розширення, покращення, удосконалення.

СПИСОК ЛІТЕРАТУРИ

1. ВІКАРЧУК, О. І.; НІКОЛАЄНКО, С. М.; КАЛІНІЧЕНКО, О. О. Фінансова грамотність–запорука успішного населення. Економіка. Управління. Інновації. Серія: Економічні науки, 2019, 1.
2. Wikipedia, The Free Encyclopedia. [online] Available at: https://www.wikipedia.org/
3. U.S. Bureau of Labor Statistics. [online] Available at: https://www.bls.gov/
4. MAZKO, E.; MEDVEDENKO, D. Family budget—nature, premises and benefits. The main budget rules. 2020.
5. DEWI ESTRI, J. H.; UMAR, Rusydi; RIADI, Imam. Implementation of Cloudflare Hosting for Speeds and Protection on The Website. Universitas Ahmad Dahlan, 2019.
6. COX, Russ; MUTHITACHAROEN, Athicha; MORRIS, Robert T. Serving DNS using a peer-to-peer lookup service. In: International Workshop on Peer-To-Peer Systems. Springer, Berlin, Heidelberg, 2002. p. 155-165.
7. REESE, Will. Nginx: the high-performance web server and reverse proxy. Linux Journal, 2008, 2008.173: 2.
8. LEVLIN, Mattias. DOM benchmark comparison of the front-end JavaScript frameworks React, Angular, Vue, and Svelte. 2020.
9. MASSE, Mark. REST API design rulebook: designing consistent RESTful web service interfaces. " O'Reilly Media, Inc.", 2011.
10. DE OLIVEIRA, Jason; BRUCHET, Michel. Learning ASP. NET Core 2.0: Build Modern Web Apps with ASP. NET Core 2.0, MVC, and EF Core 2. Packt Publishing Ltd, 2017.
11. KABBEDIJK, Jaap; JANSEN, Slinger; BRINKKEMPER, Sjaak. A case study of the variability consequences of the CQRS pattern in online

business software. In: Proceedings of the 17th European Conference on Pattern Languages of Programs. 2012. p. 1-10.

12. TROELSEN, Andrew; JAPIKSE, Philip. Pro C# 10 with .NET 6 Foundational Principles and Practices in Programming. 2021.
13. LOELIGER, Jon; MCCULLOUGH, Matthew. Version Control with Git: Powerful tools and techniques for collaborative software development. " O'Reilly Media, Inc.", 2012.
14. PERER, Adam; SHNEIDERMAN, Ben. Integrating statistics and visualization: case studies of gaining clarity during exploratory data analysis. In: Proceedings of the SIGCHI conference on Human Factors in computing systems. 2008. p. 265-274.
15. HWANG, Seokyon. A Bayesian approach for forecasting errors of budget cost estimates. Journal of Civil Engineering and Management, 2016, 22.2: 178-186.
16. HYNDMAN, Rob J.; ATHANASOPOULOS, George. Forecasting: principles and practice. OTexts, 2018.

ДОДАТОК А. КОД СЕРВЕРНОЇ ЧАСТИНИ

Автоматично згенерований шаблонний код Program.cs (WebAPI)

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllers();
var app = builder.Build();
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run(); .
```

Код інтерфейсу IResult

```
public interface IResult
{
      List<string>? Messages { get; set; }
      bool Succeeded { get; set; }
      DateTime Date { get; set; }
}

public interface IErrorResult<T> : IResult<T>
{
      public string? Exception { get; set; }
      public string? ErrorId { get; set; }
      public int StatusCode { get; set; }
}

public interface IResult<out T> : IResult
{
      T? Data { get; }
}
```

Базова модель (ModelBase) та її будуючий клас.

```
public abstract class ModelBase
{
      public Guid ID { get; set; }
}

public static class ModelBaseBuilder
{
      public static void BuildModelBase<T>(this ModelBuilder MB)
where T : ModelBase
      {
            MB.Entity<T>()
              .HasKey(m => m.ID);
            MB.Entity<T>()
              .Property(m => m.ID)
              .HasColumnType("VARCHAR(36)");
      }
}
```

Базовий контролер (BaseController).

```
[Route("[controller]")]
[ApiController]
public class BaseController : ControllerBase
{
}
```

ДОДАТОК Б. КОД КЛІЄНТСЬКОЇ ЧАСТИНИ

Конфігураційний файл package.json.

```
{
      "name": "BudgetFrog",
      "version": "2.0.0",
      "scripts": {
         "build": "rollup -c",
         "dev": "rollup -c -w",
         "start": "sirv public --no-clear --port 6060"
      },
      "devDependencies": {
         "@rollup/plugin-commonjs": "^17.0.0",
         "@rollup/plugin-node-resolve": "^11.0.0",
         "@rollup/plugin-replace": "^4.0.0",
         "autoprefixer": "^10.4.7",
         "axios": "^0.27.2",
         "daisyui": "^2.15.0",
         "dotenv": "^16.0.0",
         "postcss": "^8.4.13",
        "postcss-load-config": "^3.1.4",
        "rollup": "^2.3.4",
        "rollup-plugin-css-only": "^3.1.0",
        "rollup-plugin-livereload": "^2.0.0",
        "rollup-plugin-postcss": "^4.0.2",
        "rollup-plugin-svelte": "^7.0.0",
        "rollup-plugin-terser": "^7.0.0",
        "svelte": "^3.48.0",
        "svelte-preprocess": "^4.10.6",
        "svelte-spa-router": "^3.2.0",
        "tailwindcss": "^3.0.24"
    },
    "dependencies": {
        "sirv-cli": "^1.0.0"
    }
}
```

Функції для керування “LocalStorage” браузера.

```
export const LocalStorage = {
    Get: (key) => {
        const item = localStorage.getItem(key);
        if (item) {
            return item;
        } else {
            if (key == "jwt") {
                auth.set(false);
            }
            return null;
        }
    },
    Set: (key, value) => {
        localStorage.setItem(key, value);
        if (key == "jwt") {
            auth.set(value != null);
        }
    }
}
```

Частина об’єкта для зручної навігації запитів до серверної частини (RequestController).

```
export default {
    status: {
        ping: () => Request(`/status/ping/`, `get`),
        status: () => Request(`/status/`, `get`)
    },
    user: {
        me: () => Request(`/user/me/`, `get`),
        login: ({Email, Password}) =>
            Request(`/user/login/`, `post`, {Email, Password}),
           }, /*...*/
    },
    budget: {/*...*/ },
}
```