СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

Кваліфікаційна робота магістра

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ КЕРУВАННЯ СТАНОМ В ОДНОСТОРІНКОВИХ ЗАСТОСУНКАХ

Здобувач освіти гр. ІН.мдн-12С Данило ШАМАЄВ

Науковий керівник, Сергій ШАПОВАЛОВ
кандидат ф.-м. наук,
доцент

В.о. завідувач кафедри, Ігор ШЕЛЕХОВ
кандидат технічних наук,
доцент

СУМИ 2023

Сумський державний університет

Факультет *ЦЗДВФН* Кафедра *Комп'ютерних наук*
Спеціальність *122 «Комп'ютерні науки»*

Затверджую:
зав. кафедри _____

"_____" _____ 20___р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ
Шамаєв Данило Вікторович
(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) Інформаційна технологія керування станом в односторінкових застосунках затверджую наказом по інституту від "____" _____ 20 ___ р. № _____

2. Термін здачі студентом закінченого проекту (роботи) _____

3. Вхідні данні до проекту (роботи) _____
_____
_____
_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити) *1) Інформаційний огляд. 2) Вибір програмних засобів. 3) Практична реалізація*

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) _____
_____
_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання
_____

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1	*Інформаційний огляд*		
2	*Вибір програмних засобів*		
3	*Практична реалізація*		
4	*Оформлення кваліфікаційної магістерської роботи*		

Студент – дипломник _____
(підпис)

Керівник проекту _____
(підпис)

РЕФЕРАТ

Записка: 46 стор., 17 рис., 2 табл., 2 додатки, 15 джерел.

Об’єкт дослідження — архітектурні підходи в односторінкових застосунках.

Мета роботи — порівняння підходів до управління даними у front-end застосунках.

Методи дослідження — описовий, абстрактно-логічний, порівняльний.

Результати — розроблено і доопрацьовано програмне забезпечення (веб-застосунок) для управління персональними фінансами і проаналізовано різницю між двома версіями застосунку.

ВЕБ-ЗАСТОСУНОК, ПРИКЛАДНИЙ ПРОГРАМНИЙ ІНТЕРФЕЙС, ВЕБ-ТЕХНОЛОГІЇ, УПРАВЛІННЯ СТАНОМ, ОДНОСТОРІНКОВІ ЗАСТОСУНКИ

ЗМІСТ

ВСТУП

На сьогоднішній день розробка веб-застосунків є як ніколи складною. Через розвиток обчислювальних потужностей з’явилось більше архітектурних можливостей і, паралельно цьому, зросли потреби бізнесу та вимоги до таких застосунків. На заміну відносно простим і примітивним програмам для настільних персональних комп’ютерів, які часто напряму взаємодіяли з віддаленими БД, прийшли великі застосунки, часто розділені формально на дві частини – front-end і back-end, де front-end може бути веб сторінками, мобільним додатком або програмою для ОС Windows, а back-end – майже чим завгодно, від популярного REST API до serverless.

Ця випускна робота присвячена проблемі управління станом в сучасних односторінкових front-end застосунках. Це є актуальною проблемою, оскільки з еволюцією web-розробки і появою односторінкових застосунків (Single Page Application) складність розробки значно виросла, отже, виросли і вимоги до архітектури таких застосунків.

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ІНСТРУМЕНТІВ

1.1 Вибір фреймворку

В сучасній сфері розробки веб-застосунків існує безліч рішень, бібліотек, фреймворків та інших інструментів. Серед найпоширеніших і найпопулярніших фреймворків [1] виділяються React, Vue та Angular. В контексті цієї роботи буде розглянуто Angular.

Angular – це сучасний фреймворк призначений для розробки односторінкових веб-застосунків [2] і написаний на мові TypeScript. Цей фреймворк має широкий набір інструментів для створення великих і середніх застосунків з потенціалом масштабування.

Одна з головних переваг Angular в порівнянні з такими фреймворками як React та Vue – це наявність Angular CLI. За допомогою цього потужного інструменту інтерфейсу командного рядка, можна вирішувати широкий спектр задач, з якими зустрічається розробник застосунку: створити новий проєкт та його базову структуру і архітектуру, згенерувати нову сутність, отримати підказки, запустити проєкт, збудувати проєкт, запустити автоматичні тести та багато іншого. Через наявність такого потужного інструменту, Angular іноді називають не просто фреймворком, а платформою для веб-застосунків.

Як зазначено вище, нові проєкти за замовчуванням створюються з визначеною архітектурою і структурою директорій та файлів (в англомовній спільноті це також називають scaffolding – риштування). Це теж є перевагою Angular [3], оскільки дозволяє зекономити час і зусилля впродовж всіх етапів розробки застосунків, незалежно від цільової платформи (веб, мобільні пристрої, настільні комп’ютери).

В основі архітектури багатьох сучасних frontend фреймворків лежить компонентний підхід [4], в тому числі в Angular. В останньому випадку компонент являє собою настроюваний HTML елемент, який, як правило, має

бізнес-логіку, описану мовою TypeScript, HTML шаблон і набір CSS стилів. На рис. 1.1 візуалізовано розділення змісту веб-сторінки на різні компоненти.

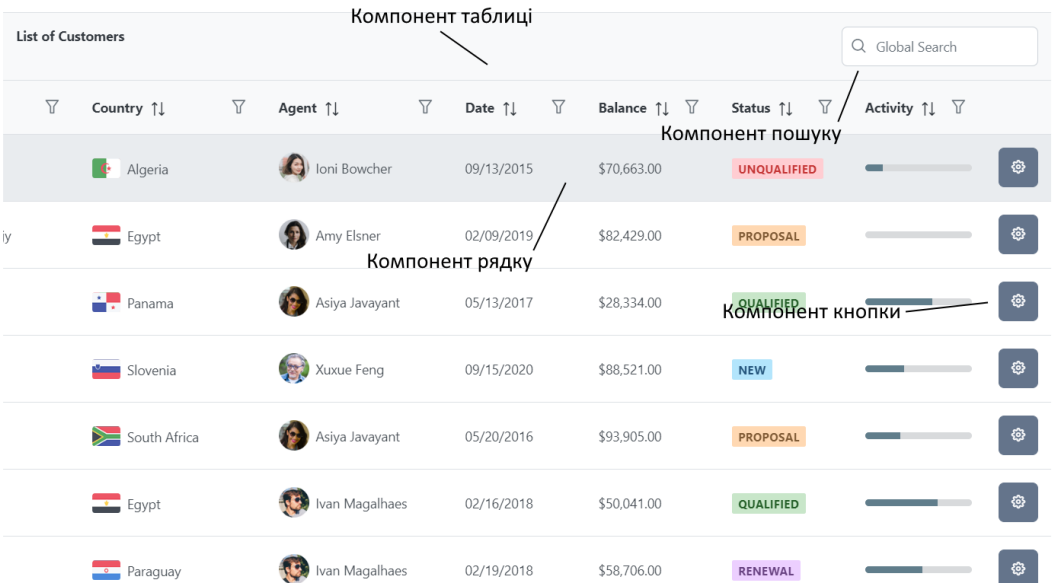

Рисунок 1.1 – Приклад розділення сторінки на компоненти

Hyper-Text Markup Language (HTML) – це мова розмітки, яка використовується для опису структури веб-сторінки та визначає її зміст [5]. HTML-документ складається з елементів (тегів) які інтерпретуються браузером і відображаються на веб-сторінці.

Cascading Style Sheets (CSS) є мовою таблиць стилів, які описують зовнішній вигляд документу [6], в свою чергу описаного за допомогою мови розмітки HTML.

Варто зазначити, що стилізація за допомогою CSS не є єдиною опцією в Angular. Також підтримуються такі препроцесори як Sass і Less. Вони розширюють можливості CSS [7] за допомогою вкладених правил, міксінів [8], модулів, наслідування та інших прийомів. Оскільки в цій роботі не планувалось використовувати багато стилізації, доцільніше було відмовитись від препроцесорів.

Окрім компонентів, в Angular існують такі сутності як модулі, сервіси, директиви, інтерфейси, сторожові класи (guards), та багато інших, які разом складають застосунки на основі цього фреймворку.

Однією з бібліотек, які входять в інфраструктуру Angular, є RxJs. Ця бібліотека реалізує один з підходів до програмування, а саме *реактивний* [7]. Ця парадигма побудована на асинхронних потоках даних і розповсюдженні змін, її також іноді називають подієво-орієнтованим програмуванням. На практиці RxJs реалізує такі сутності:

- Observable: являє собою абстрактну обгортку для потоку (колекції) даних або подій;
- Observer: набір callback-функцій (next, error, complete), що реагують на дані, видані Observable;
- Subscription: підписка, що викликає виконання Observable;
- Operators: чисті функції, які маніпулюють над даними в потоці і над ходом виконання потоку;
- Subject: спеціальний тип Observable, що дозволяє розсилати дані багатьом Observer'ам, оскільки звичайний Observable для кожного Observer являє унікальний контекст виконання потоку;

та деякі інші, що неактуальні для цієї задачі. Варто відмітити різницю Observable та об'єктом Promise, що реалізований в мові JavaScript за замовчуванням: Promise завжди виконується одразу, натомість Observable не буде виконуватись, поки на нього не підписатися за допомогою Subscription. Це дає більш широкі можливості для реалізації асинхронного програмування у front-end застосунках.

TypeScript – це мультипарадигмальна мова програмування, яка компілюється в JavaScript і є її надмножиною [8]. Вона додає до JavaScript [9] строгу типізацію, аналогічну системам типів в об'єктно-орієнтованих мовах програмування таких як Java та C#. На таблиці 1.1 продемонстровані основні відмінності систем типів TypeScript і JavaScript.

Таблиця 1.1 Порівняння мов програмування

	JavaScript	TypeScript
Зв’язування типів	Динамічне	Статичне
Автоматична конвертація типів	Так	Ні (в більшості випадків)
Перевірка типів	Під час виконання	Під час компіляції
Виявлення помилок	Переважно під час виконання	Переважно під час компіляції

1.2 Вибір бібліотеки для управління даними

В якості рішення для управління даними було використано NgRx – фреймворк, який створений для поліпшення розробки реактивних застосунків у Angular. В основі цього фреймворку використовується Flux – архітектура, що являє собою підхід у побудові користувацьких веб-інтерфейсів за допомогою однонаправлених потоків даних [10]. В мінімальному варіанті, Flux реалізує три шари:

- Actions (*дії*),
- Stores (*сховища*),
- Views (*подання*).

На рис. 1.2 зображена візуальна репрезентація цих шарів і їх взаємодії. На практиці, NgRx розвиває ці ідеї і має більш складну ментальну модель та декілька компонентів.

Рисунок 1.2 – Візуальна репрезентація Flux-архітектури

Головним компонентом, який використовується в даній роботі, є @ngrx/store. Store (сховище) – це глобальне управління станом для Angular застосунків, що реалізує Flux-архітектуру і являє собою контрольований контейнер стану [11]. Ключовими концептами NgRx Store є:

- Actions (*дії*), що описують унікальні події і відправляються з компонентів та сервісів;
- Reducers (редуктори) – чисті функції, що оброблюють зміни стану, беручи поточний стан і останню дію (Action) для обчислення нового стану;
- Selectors (селектори) – чисті функції, що можуть вибирати, виводити і складати частини стану;
- Store (сховище), що надає доступ до стану за допомогою Observable і спостерігає за діями;
- Effects (ефекти) – модель побічних ефектів, які ізольовані від компонентів Angular і виконують роботу зі зовнішніми API за допомогою сервісів.

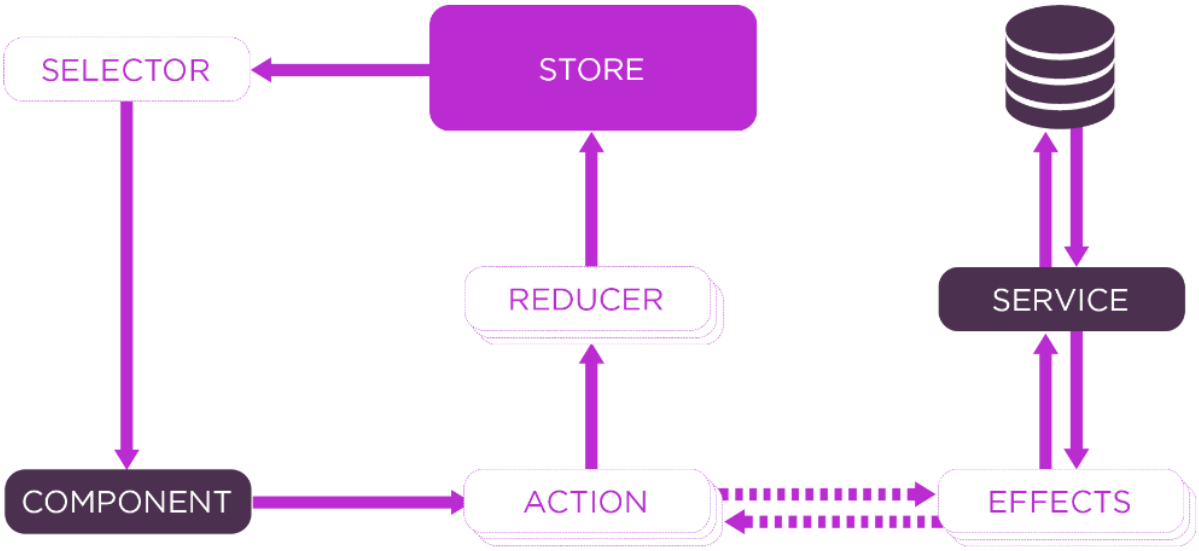

Рисунок 1.3 – Схема взаємодії компонентів архітектури NgRx

1.3 Інші допоміжні технології

Для збереження і синхронізації користувацьких даних було обрано платформу Firebase. Ця платформа надає широкий спектр хмарних послуг [12], в тому числі нереляційну документ-орієнтовану базу даних Firestore Database та прикладний програмний інтерфейс (API) до неї. Всі дані в цю базу даних записуються у вигляді об’єктів (документів), які зберігаються в колекціях. Завдяки такому гнучкому підходу нема потреби в проєктуванні сутностей і відносин між ними, як в реляційних базах даних, наприклад SQL.

В якості бібліотеки компонентів було обрано PrimeNG [13] і Primeflex в якості утиліти для CSS. Приклад одного з таких компонентів приведено на рис. 1.1. Таке рішення дозволяє пришвидшити процес розробки і не фокусувати його на проблемах стилізації, але при цьому зберегти можливості для глибокого налаштування у разі такої потреби.

Поміж інших, також використовувались програмно-прикладні інтерфейси браузеру, такі як localStorage та IndexedDB [9], для збереження локальних даних користувача.

1.4 Постановка задачі

Перед початком роботи була поставлена задача розробити програмне забезпечення у вигляді односторінкового веб-застосунку для керування персональними фінансами. Серед інших вимог були: здатність авторизації або реєстрації за допомогою e-mail, Google або Facebook, а також збереження користувацьких даних у віддаленому сховищі.

В якості об’єкта дослідження було визначено архітектурні підходи управління даними у застосунку, з метою аналітичного порівняння таких підходів.

2 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ

2.1 Ініціалізація проєкту

Перед початком роботи було встановлено Node.js 18.12.1 LTS з офіційного сайту. Менеджер пакетів npm який потрібний для завантаження потрібних модулів вже включено до Node.js.

Ініціалізація проєкту відбувалася за допомогою інструмента Angular CLI наступною командою:

```
npm install -g @angular/cli && ng new personal-budget-app
```

після чого менеджер пакетів npm завантажить всі необхідні для застосунку модулі і помістить їх в папку node_modules в корені проєкта.

Також за допомогою цього ж інструменту було згенеровано початкову структуру застосунку, яка складається з компонентів, сервісів і модулів, за допомогою наступної команди:

```
ng generate module accounts --module app --routing true --route accounts && ng generate module login --module app --routing true --route login && ng generate module register --module app --routing true --route register && ng generate module settings --module app --routing true --route settings && ng generate module transactions --module app --routing true --route transactions && ng generate component --module accounts --route new-account && ng generate component new-transaction --module transactions --route new-transaction && ng generate service accounts --path src/app/services --skipTests && ng generate service auth --path src/app/services --skipTests && ng generate service settings --path src/app/services --skipTests && ng generate service messaging --path src/app/services --skipTests && ng generate service settings --path src/app/services --skipTests && ng generate service theme --path src/app/services --skipTests && ng generate service toast --path src/app/services --skipTests
```

Для подальшої роботи з системою типів були створені дефініції моделей (інтерфейси) сутностей Account, Transaction і User:

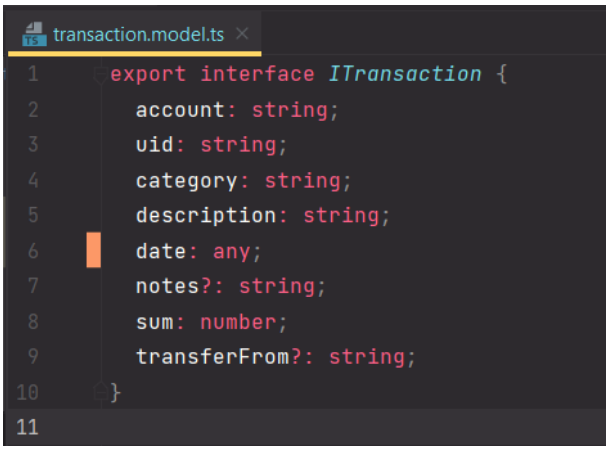

Рисунок 2.1 – Вихідний код моделі сутності Transaction.

Також за вимогою замовника були створені окремо налаштовані стилі з різними палітрами для темного і світлого режимів. Вони розташовані в /src/app/styles, а для контролю перемикання між цими режимами було реалізовано сервіс ThemeService, який здійсняє зберігання обраного режиму в localStorage. Методи цього сервісу безпосередньо використовуються в компоненті SettingsComponent.

2.2 Сервіси для взаємодії з серверною частиною

Для взаємодії з серверною частиною додатку, а точніше з програмно-прикладним інтерфейсом Firebase було реалізовано ряд сервісів. Серед них – AccountsService, який має набір методів для роботи з рахунками і транзакціями в застосунку. На прикладі методу getUserTransactions можна побачити, яким чином здійснюється отримання списку транзакцій користувача. (src/app/services/accounts.service.ts)

```
public getUserTransactions(accountID: string): Observable<any> {
  return this.authService.user$.pipe(
    concatMap((user) => {
      return from(
        collectionData(
          query(
            collection(this.db, 'transactions'),
            where('uid', '==', user?.uid),
            where('account', '==', accountID)
          )
```

```
        )
      );
    })
  );
}
```

this.authService.user$ представляє собою об’єкт типу Observable, що містить в собі потік з об’єктом User. Коли в потоці з’являється User, за допомогою оператору concatMap здійснюється переключення на інший потік, створений за допомогою функції from(…). В цій функції використовуються безпосередньо функції API Firestore, що виділяють з колекції ‘transactions’ всі транзакції, що належать (where) поточному користувачу і рахунку, переданому в параметрах. Таким чином, метод повертає об’єкт Observable, який і «обгортає» потік цих даних.

2.3 Модулі застосунку

Було реалізовано такі модулі як LoginModule, RegisterModule, SettingsModule, TransactionsModule і AccountsModule. Кожен з них мав одноіменний компонент, а TransactionsModule та AccountsModule включали по два компоненти, які входили до схожої області використання. Розглянемо останній модуль.

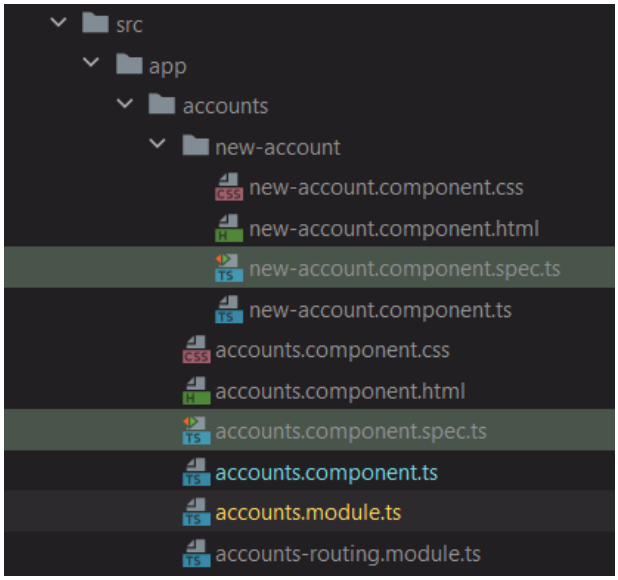

Рисунок 2.2 – Структура директорій і файлів модулю accounts.

AccountsModule складається з двох компонентів: AccountsComponent, NewAccountComponent, а також їх стилів, шаблонів, автоматично згенерованих тестових специфікацій, які не розглядаються в даній роботі, та модулю маршрутизації.

(src/app/accounts/accounts.module.ts)

```
import {NgModule} from '@angular/core';
import {CommonModule} from '@angular/common';
import {ReactiveFormsModule} from '@angular/forms';
import {ButtonModule} from 'primeng/button';
import {InputTextModule} from 'primeng/inputtext';
import {RippleModule} from 'primeng/ripple';

import {AccountsRoutingModule} from './accounts-routing.module';
import {AccountsComponent} from './accounts.component';
import {NewAccountComponent} from './new-account/new-account.component';

@NgModule({
  declarations: [AccountsComponent, NewAccountComponent],
  imports: [
    CommonModule,
    AccountsRoutingModule,
    ButtonModule,
    RippleModule,
    InputTextModule,
    ReactiveFormsModule,
  ],
})
export class AccountsModule {}
```

На прикладі цього модулю можна побачити як імпортуються всі залежності, необхідні для компонентів цього модулю, а також яким чином фіксується приналежність компонентів за допомогою спеціального декоратору @NgModule().

2.4 Компоненти застосунку

Розглянемо компонент AccountsComponent. У файлі accounts.component.ts можна побачити безпосередню реалізацію логіки і посилання на шаблон та стилі компоненту.

(src/app/accounts/accounts.component.ts)

```
import {Component, OnDestroy} from '@angular/core';
import {Router} from '@angular/router';
import {Observable, retry, Subject, takeUntil} from 'rxjs';
import {IAccount} from '../models/account.model';
import {AccountsService} from '../services/accounts.service';

@Component({
  selector: 'app-accounts',
  templateUrl: './accounts.component.html',
  styleUrls: ['./accounts.component.css'],
})
export class AccountsComponent implements OnDestroy {
  public accounts$: Observable<IAccount[]>;
  public balance$: Observable<number>;
  private unsubscribe$: Subject<void>;

  constructor(
    private accountsService: AccountsService,
    private router: Router
  ) {
    this.unsubscribe$ = new Subject<void>();
    this.accounts$ = this.accountsService
      .getUserAccounts()
      .pipe(retry(1), takeUntil(this.unsubscribe$));
    this.balance$ = this.accountsService
      .getUserTotalBalance()
      .pipe(takeUntil(this.unsubscribe$));
  }
```

```
  ngOnDestroy(): void {
    this.unsubscribe$.next();
    this.unsubscribe$.unsubscribe();
  }

  public openAccount(account: string): void {
    this.router.navigate(['transactions', account]);
  }

  public addAccount(): void {
    this.router.navigate(['accounts', 'new']);
  }
}
```

Декоратор @Component(…) потрібний для перетворення звичайного класу на клас-компонент і призначення йому html-тегу (selector), шаблону (templateUrl) та стилів (styleUrls).

Розберемо методи класу:

- constructor(…) приймає об’єкти accountsService і router відповідних типів в якості залежностей (за допомогою механізму Dependency Injection) та ініціалізує члени класу;
- ngOnDestroy() викликається перед руйнуванням об’єкту класу на останньому етапі життєвого циклу і передає пусте значення в Subject unsubscribe$, що дозволяє в купі з оператором takeUntil() дозволяє ефективно уникати витоків пам’яті і відписуватись від Observable;
- openAccount(account: string) визивається з шаблону компонента і приймає в якості параметру ID рахунку користувача, після чого здійснює навігацію на сторінку відповідного рахунку;
- addAccount() аналогічно методу вище, здійснює навігацію на сторінку створення нового рахунку.

На прикладі цього компоненту можна побачити яким чином працює внутрішня логіка застосунку. Розглянемо як описаний шаблон розмітки компоненту:

(src/app/accounts/accounts.component.html)

```
<div class="text-center">
  <h6>BALANCE</h6>
  <h2>{{balance$ | async}}</h2>
</div>
<button
  pButton
  pRipple
  label="Add Account"
  icon="pi pi-plus"
  class="block ml-auto"
  (click)="addAccount()"
></button>
<div>
  <div
    pRipple
    class="p-card my-2"
    *ngFor="let account of accounts$ | async"
    (click)="openAccount(account.id)"
  >
    <div class="p-card-header text-lg font-medium p-2">
      {{account.name}}
    </div>
    <div class="p-card-content px-2 py-2">
      {{account.balance}}
    </div>
  </div>
</div>
```

Як видно, цей код дещо відрізняється від стандартного HTML. Можна зустріти синтаксис {{…}}, що містить в собі TypeScript код. Цей прийом в

Angular називається *інтерполяцією*. Також тут використовуються численні директиви:

- pButton – перетворює звичайний HTML елемент <button> на розширений компонент з бібліотеки PrimeNG
- pRipple – додає до елементу так званий «ripple» ефект який впливає на візуальний стиль
- *ngFor – структурна директива, що аналогічна конструкції for… і породжує нові елементи на сторінці з заданого масиву.

Іншим примітним явищем є використання async pipe, що дозволяє відображати на сторінці дані, які знаходяться в асинхронному потоці Observable. Це заміняє потребу в subscribe() методі і створенню додаткового коду в компоненті, роблячи код таким чином більш шаблонно-орієнтованим.

2.5 Запуск і перевірка роботи застосунку

Для того, щоб запустити проєкт в конфігурації для розробки, потрібно виконати команду ng serve. Після чого Angular CLI за допомогою компілятора TypeScript завершить збірку проєкту і запустить сервер в локальному режимі. Типовою адресою локального серверу є http://localhost:4200.

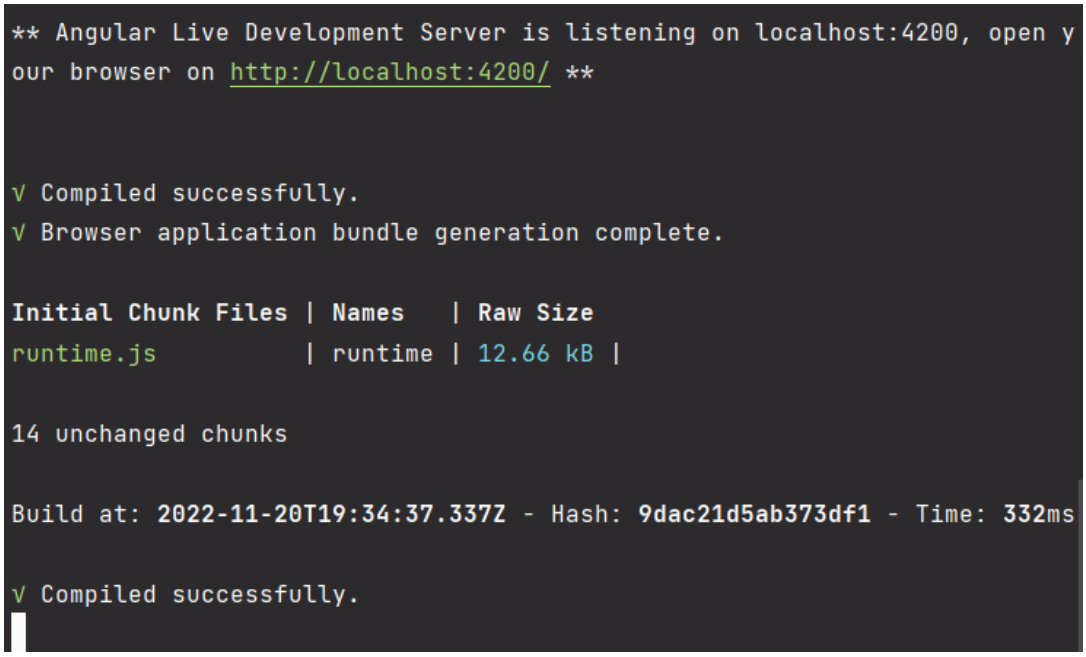

Рисунок 2.3 – Результат запуску локального веб-серверу за допомогою команди ng serve

Відкривши дану адресу в браузері, можна побачити працюючий застосунок з формою авторизації та можливостями входу через соціальні сервіси Google та Facebook.

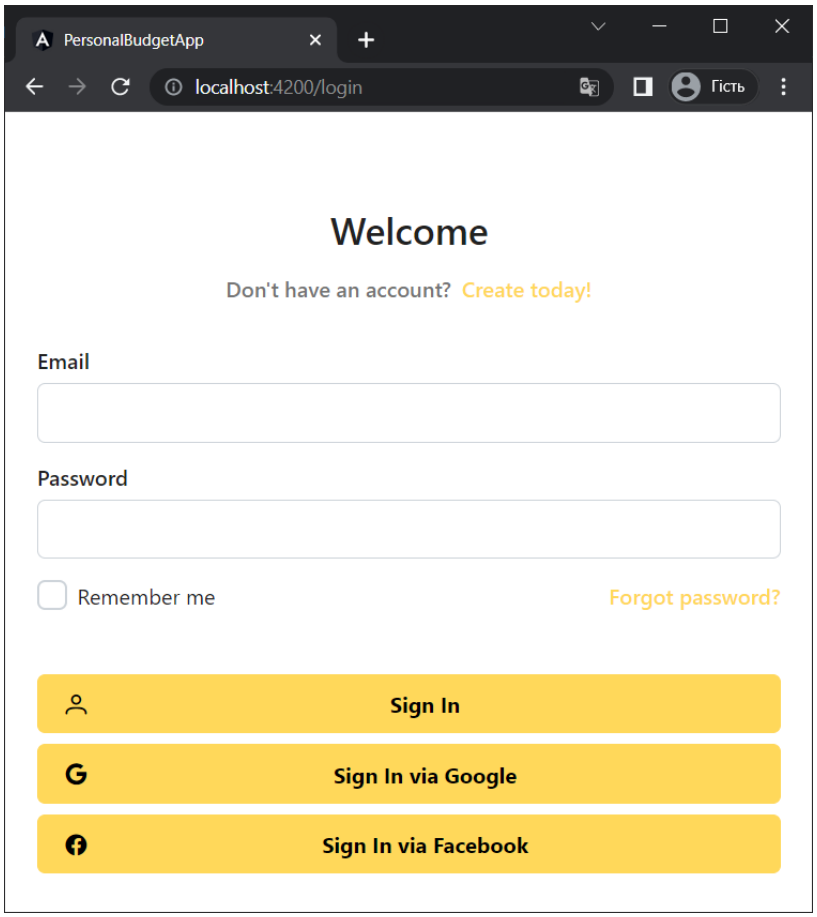

Рисунок 2.4 – Сторінка авторизації

Після авторизації за допомогою зв’язки e-mail/пароль, або одного з сервісів (рекомендовано), відкриється головна сторінка з рахунками користувача. Якщо користувач ще не створював рахунків, то список і баланс будуть пустими.

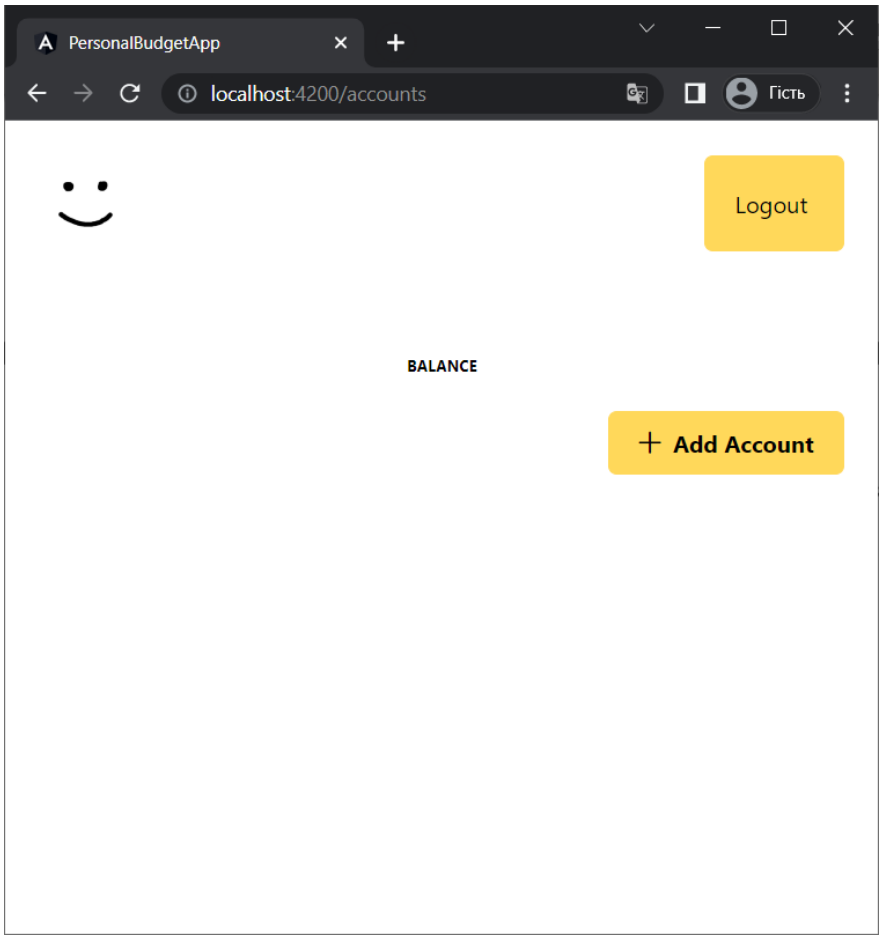

Рисунок 2.5 – Порожня сторінка з рахунками

Натиснувши «Add» відкриється форма створення нового рахунку (рис. 2.6). Після введення даних, оновлений список рахунків і баланс можна спостерігати на головній сторінці (рис. 2.7). Щоб додати нову трансакцію до рахунку, потрібно обрати карточку і здійсниться перехід на сторінку цього рахунку (рис. 2.8), після чого треба натиснути «Add Transaction».

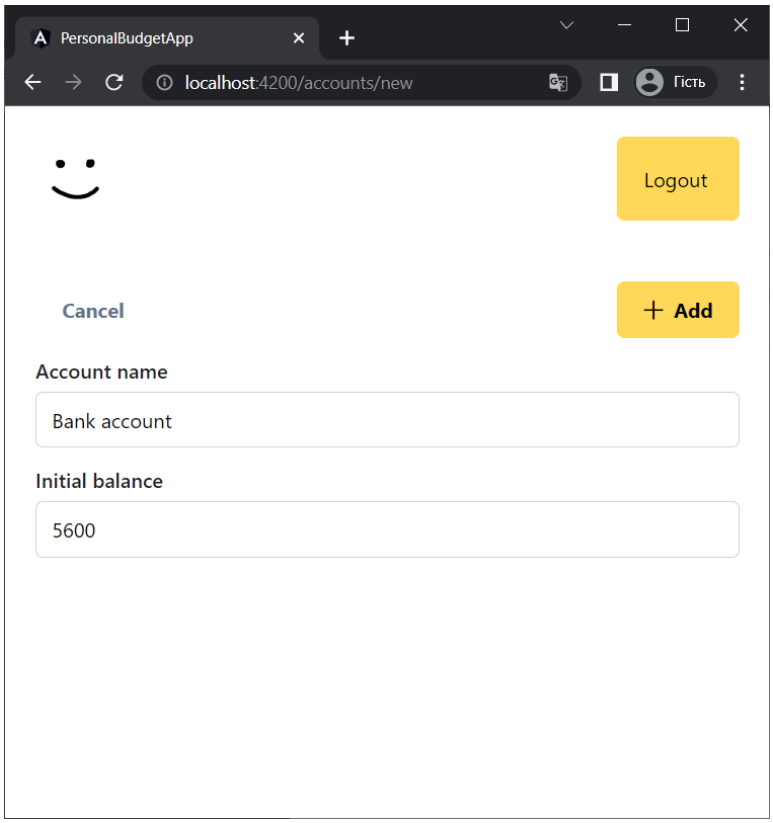

Рисунок 2.6 – Форма створення нового рахунку

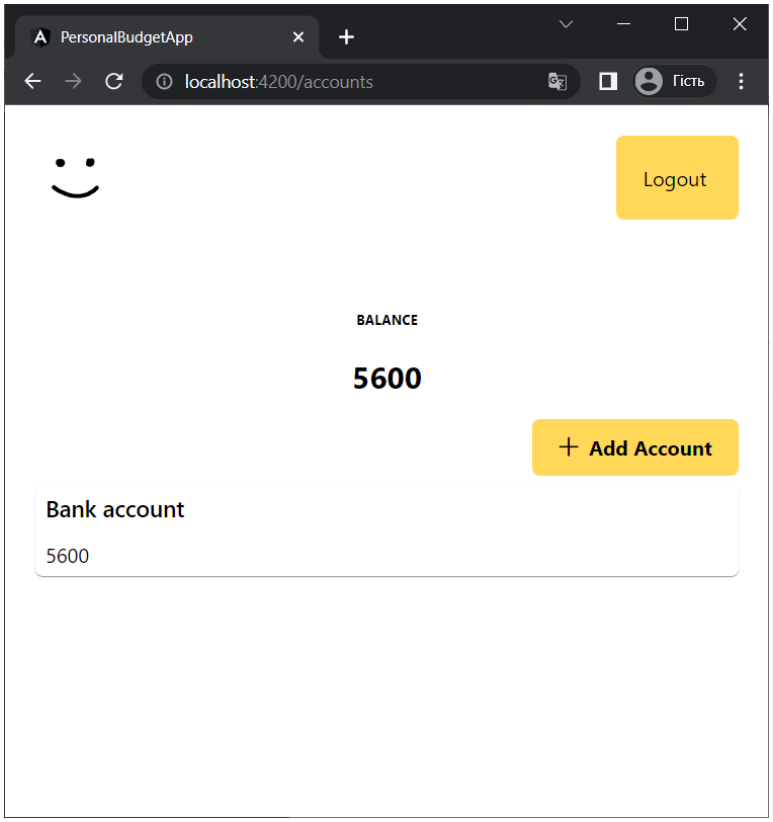

Рисунок 2.7 – Оновлений список рахунків і баланс

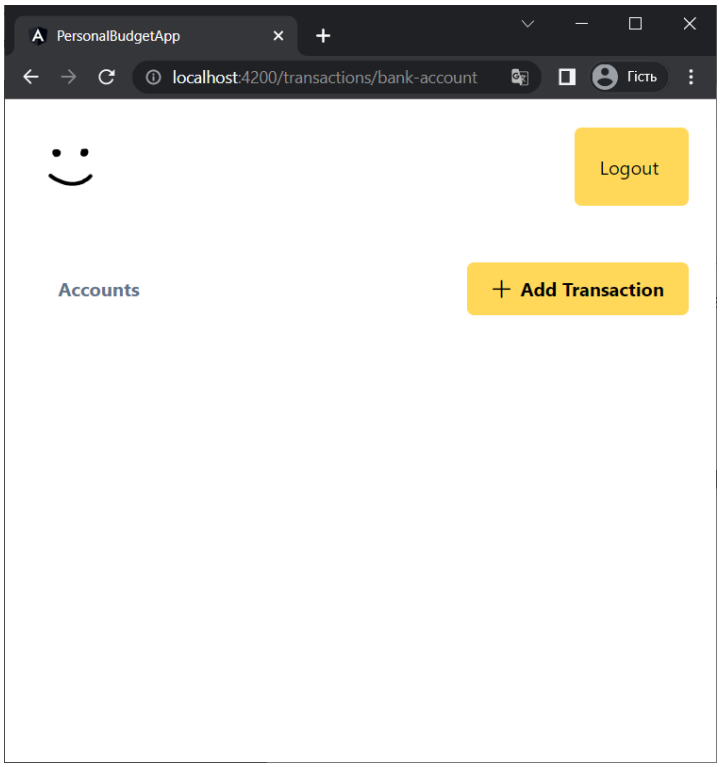

Рисунок 2.8 – Пуста сторінка з транзакціями рахунку Bank account

Після заповнення форми нової транзакції (рис. 2.9) і натиснення «Add», нова транзакція з’явиться в списку (рис. 2.10).

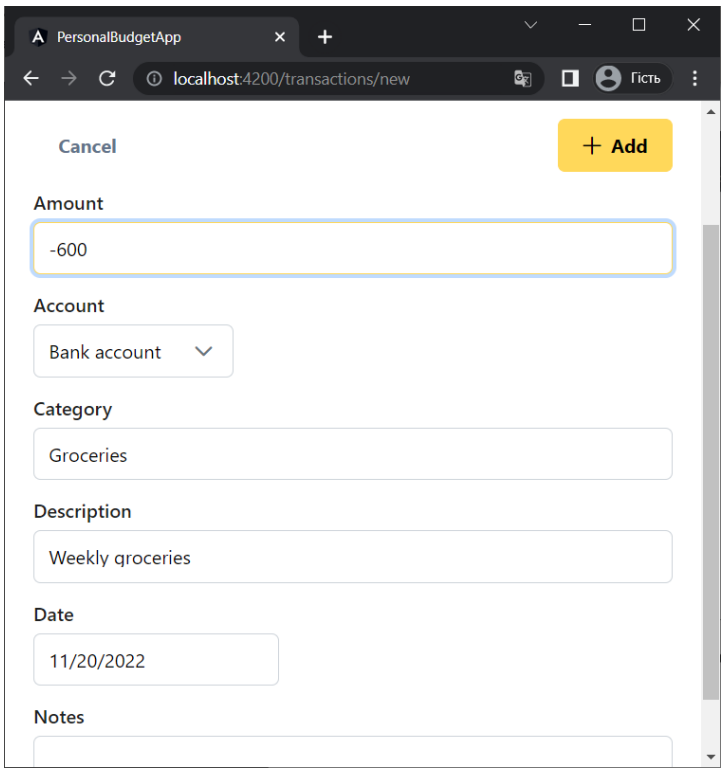

Рисунок 2.9 – Форма створення нової транзакції

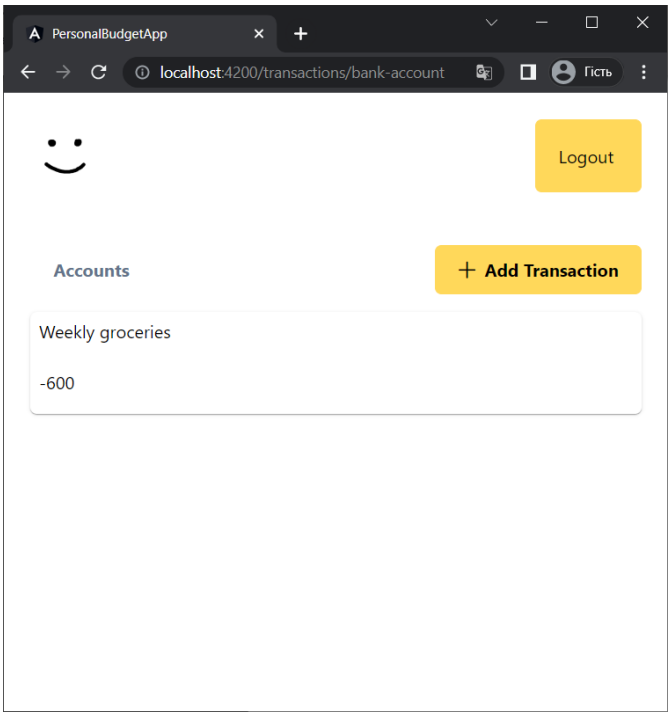

Рисунок 2.10 – Оновлений список транзакцій рахунку Bank account

Для того, щоб перейти на сторінку налаштувань користувача, потрібно натиснути на його фото профілю у верхньому лівому кутку. В налаштуваннях є можливість оновити фото профілю, ім’я та перемикнути світлий/темний режим за бажанням.

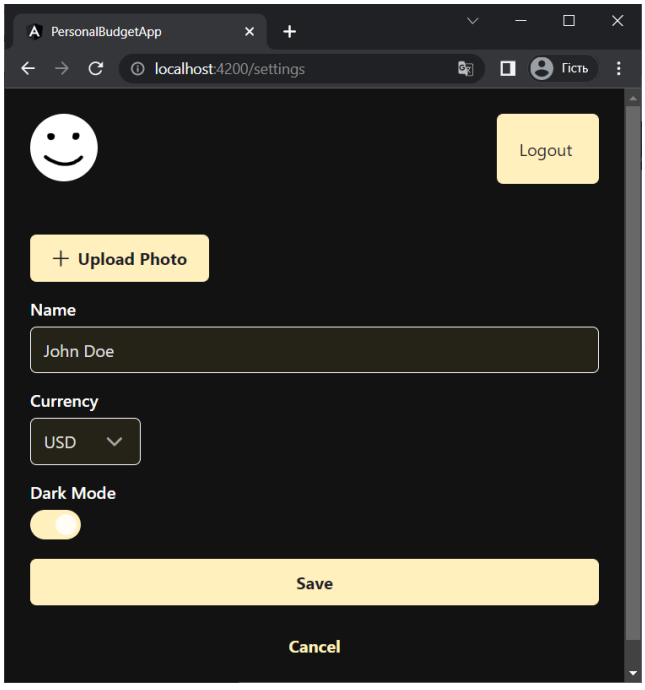

Рисунок 2.11 – Демонстрація вигляду темного режиму

3 РЕАЛІЗАЦІЯ І АНАЛІЗ НОВОЇ АРХІТЕКТУРИ

3.1 Встановлення і конфігурація необхідних пакетів

Для того, щоб почати реалізацію однонаправленої архітектури Flux на практиці, потрібно встановити необхідні пакети NgRx. Це було зроблено за допомогою наступної команди Angular CLI:

```
ng add @ngrx/store@14.2.7
```

При виконанні даної команди, з’являється інтерактивний режим командного рядку, в якому потрібно обрати конфігурацію ESLint (було обрано «recommended»).

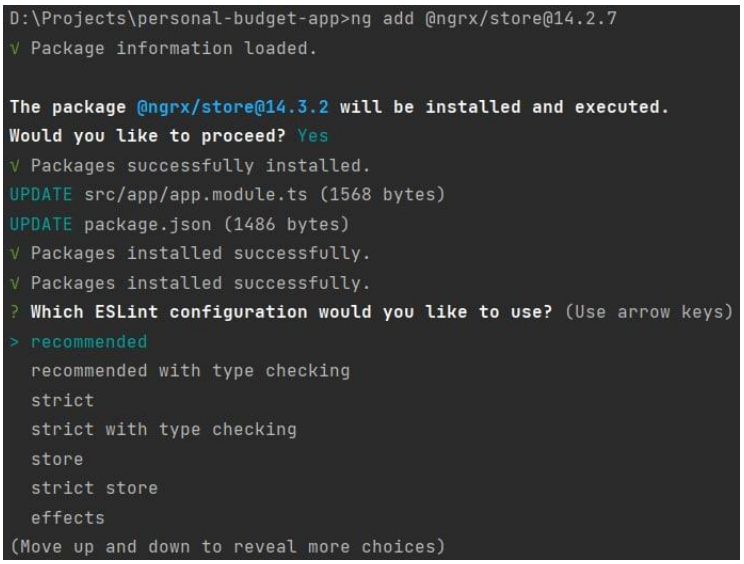

Рисунок 3.1 – Процес додавання NgRx до проєкту і його налаштування

Наступним кроком було описано інтерфейс глобального стану AppState, пусту мапу редукторів і передано її до StoreModule в якості параметру методу .forRoot() при об’явлені imports модуля AppModule. Пізніше, AppState буде містити в собі стани інших частин застосунку, а reducers – всі редуктори. (src/app/state/index.ts)

```
import {ActionReducerMap} from '@ngrx/store';

export interface AppState {}

export const reducers: ActionReducerMap<AppState, any> = {};
```

Також при більш детальному аналізі наявного функціоналу і механізмів роботи NgRx було прийнято рішення скористатись ще одним компонентом екосистеми – Effects. Це здійснилося за допомогою додавання і встановлення пакету @ngrx/effects відповідної версії, а також внесення EffectsModule з методом .forRoot() до списку імпортів AppModule (відривок коду, що наведений вище).

Останнім кроком підготовки конфігурації проєкту було розширення можливостей Chromium DevTools для наглядності процесу роботи Store і всіх змін стану застосунку. Пререквізитом цього кроку являється встановлення розширення для браузерів на базі Chromium – Redux DevTools, або аналогічне для браузеру Firefox. Потрібно додати до проєкту пакет @ngrx/store-devtools потрібної версії і додати StoreDevtoolsModule з відповідною конфігурацією до імпортів AppModule.

(src/app/app.module.ts)

```
import {NgModule} from '@angular/core';
import {initializeApp, provideFirebaseApp} from '@angular/fire/app';
import {getAuth, provideAuth} from '@angular/fire/auth';
import {getFirestore, provideFirestore} from '@angular/fire/firestore';
import {getStorage, provideStorage} from '@angular/fire/storage';
import {BrowserModule} from '@angular/platform-browser';
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';
import {EffectsModule} from '@ngrx/effects';
import {StoreModule} from '@ngrx/store';
import {StoreDevtoolsModule} from '@ngrx/store-devtools';
import {MessageService} from 'primeng/api';
import {AvatarModule} from 'primeng/avatar';
import {ButtonModule} from 'primeng/button';
import {InputSwitchModule} from 'primeng/inputswitch';
import {RippleModule} from 'primeng/ripple';
import {ToastModule} from 'primeng/toast';
import {environment} from '../environments/environment';

import {AppRoutingModule} from './app-routing.module';
```

```
import {AppComponent} from './app.component';
import {reducers} from './state';
import {AccountsEffects} from './state/accounts/accounts.effects';

@NgModule({
      declarations: [AppComponent],
      imports: [
        BrowserModule,
        BrowserAnimationsModule,
        AppRoutingModule,
        provideFirebaseApp(() => initializeApp(environment.firebase)),
        provideAuth(() => getAuth()),
        provideFirestore(() => getFirestore()),
        provideStorage(() => getStorage()),
        ToastModule,
        InputSwitchModule,
        ButtonModule,
        RippleModule,
        AvatarModule,
        StoreModule.forRoot(reducers, {}),
        EffectsModule.forRoot([]),
        StoreDevtoolsModule.instrument({
          maxAge: 25,
          logOnly: environment.production,
        }),
      ],
      providers: [MessageService],
      bootstrap: [AppComponent],
})
export class AppModule {}
```

На цьому етапі, не можна не відмітити збільшену кількість імпортованих модулів, класів, функцій, тощо, в проєкті. Це – зворотна сторона сучасної розробки односторінкових застосунків, і така проблема не специфічна для фреймворку Angular.

3.2 Процес впровадження нової архітектури

Однією з переваг NgRx є те, що впроваджувати цю технологію можна поступово, тому з наступними кроками створено стан, редуктори, дії, селектори і ефекти. Перш-на-перш було описано інтерфейс стану рахунків користувача AccountsState, а також його початковий стан і редуктор.

(src/app/state/accounts/accounts.reducers.ts)

```
import {createReducer, on} from '@ngrx/store';
import {IAccount} from '../../models/account.model';
import {loadAccountsSuccess} from './accounts.actions';

export interface AccountsState {
  accounts: IAccount[];
}

const initialState: AccountsState = {
  accounts: [],
};

export const accountsReducer = createReducer(
  initialState,
  on(loadAccountsSuccess, (state, action): AccountsState => {
    return {
      ...state,
      // @ts-ignore
      accounts: [...action.payload.accounts],
    };
  })
);
```

Виклик функції on(…) у вищезазначеному фрагменті коду дозволяє описати асоціацію між дією loadAccountsSuccess, яка буде описана далі, і лямбда-функцією редукції цієї дії з подальшою зміною стану.

Для того, щоб стан AccountsState і його редуктор було включено до стану застосунку AppState, потрібно було додати його до інтерфейсу AppState і константи reducers, яка є мапою редукторів.

(src/app/state/index.ts)

```
import {ActionReducerMap} from '@ngrx/store';
import {accountsReducer, AccountsState} from './accounts/accounts.reducers';

export interface AppState {
  accounts: AccountsState;
}

export const reducers: ActionReducerMap<AppState, any> = {
  accounts: accountsReducer,
};
```

Дії (actions), що пов’язані з рахунками, були описані у файлі src/app/state/accounts/accounts.actions.ts. Для стислості, нижче буде приведено опис лише однієї з таких дій.

```
export const loadAccountsSuccess = createAction(
  '[Accounts] Load Accounts Success',
  props<{accounts: IAccount[]}>()
);
```

Сама дія являє собою текстовий рядок з коротким описом того, що відбувається, та областю виконання в квадратних дужках – це звичайна практика називання таких дій в патерні Flux/Redux. Також, у випадку loadAccountsSuccess, ця дія має об’єкт властивостей, які можуть нести з собою будь які дані. Оскільки ця дія описує успіх виконання іншої дії, loadAccounts, то логічно, що вона повинна мати масив завантажених рахунків. Але яким чином ці рахунки завантажуються?

Оскільки редуктори є чистими функціями і не можуть мати залежностей у вигляді сервісів за визначенням, то для роботи з сервісами безпосередньо існують ефекти (Effect). Ефекти були описані у файлі

src/app/state/accounts/accounts.effects.ts. Розглянемо один з цих ефектів:

```
loadAccounts$ = createEffect(() =>
    this.actions$.pipe(
      ofType(loadAccounts.type),
      mergeMap(() =>
        this.accountsService.getUserAccounts().pipe(
          map((accounts) => ({
            type: loadAccountsSuccess.type,
            payload: {accounts},
          })),
          catchError(() => EMPTY)
        )
      )
    )
  );
```

Отже, розберемо даний фрагмент коду:

- createEffect() – спеціальна функція, що повертає Observable з дією у відповідь на іншу дію;
- this.actions$ – змінна-Observable типу Actions, що містить в собі потік дій всього застосунку;
- ofType() – спеціальний pipe-оператор, який фільтрує події в потоці за типом дії;
- mergeMap() – оператор, який перемикає потік на новий Observable;
- this.accountsService.getUserAccounts() – метод сервісу AccountsService, який здійснює запит до віддаленого серверу (у випадку даного застосунку – до хмарної бази даних Firebase Firestore) і повертає Observable;
- map() – оператор, який в наведеному випадку конвертує масив accounts в об’єкт типу Action з навантаженням у вигляді оригінального масиву рахунків accounts, який в подальшому буде передано відповідному редуктору;
- catchError – оператор, який перемикає хід потоку у випадку помилки.

Також важливо додати AccountsEffects до масиву ефектів, що передається в якості аргументу EffectsModule.forRoot(…) в AppModule, для коректної роботи.

Для того, щоб мати можливість отримати дані зі Store про рахунки в будь-якому компоненті застосунку, потрібно було створити селектори для стану AccountsState. Всі селектори accounts були описані у файлі accounts.selectors.ts у вже встановленій директорії. Далі наведено їх опис їх дії і призначення:

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import {AccountsState} from './accounts.reducers';

export const selectAccountsState =
  createFeatureSelector<AccountsState>('accounts');

export const selectAccounts = createSelector(
  selectAccountsState,
  (state) => state.accounts
);

export const selectTotalBalance = createSelector(selectAccountsState, (state)
=>
  state.accounts.reduce(
    (prevValue, curValue) => prevValue + curValue.balance,
    0
  )
);
```

- selectAccountsState – feature-селектор, який виділяє і повертає стан AccountsState з глобального стану AppState;
- selectAccountsState – селектор, що виділяє і повертає масив accounts, базуючись на селекторі стану, що описаний вище;
- selectTotalBalance – виділяє стан з рахунками і повертає обрахований загальний баланс за допомогою стандартної функції JavaScript reduce().

Наступним кроком було впровадження Store в компоненті AccountsComponent. Виклики функцій сервісу accountsService було замінено на виклики селекторів сховища застосунку, а ініціалізація завантаження даних була реалізована за допомогою диспетчування відповідної дії викликом методу dispatch(). Отже, всі зміни стосуються лише конструктору AccountsComponent (уривок коду наведений нижче), і зміни в бізнес логіці застосунку зводяться до мінімуму, натомість сама структура коду стає більш зрозумілою і послідовною з точки зору розробника.

```
constructor(
  private accountsService: AccountsService,
  private router: Router,
  private store: Store
) {
  this.store.dispatch(loadAccounts());
  this.unsubscribe$ = new Subject<void>();
  this.accounts$ = this.store
    .select(selectAccounts)
    .pipe(retry(1), takeUntil(this.unsubscribe$));
  this.balance$ = this.store
    .select(selectTotalBalance)
    .pipe(takeUntil(this.unsubscribe$));
}
```

3.3 Аналіз результатів

Для того щоб запевнитись, що впровадження NgRx здійснилось успішно, застосунок був запущений в режимі розробки за допомогою команди ng serve. З точки зору користувача, в застосунку нічого не змінилося функціонально, і він працює та виглядає так, як і раніше. Але перевірити роботу NgRx можна за допомогою Redux DevTools. Для цього потрібно відкрити DevTools > вкладка “Redux”.

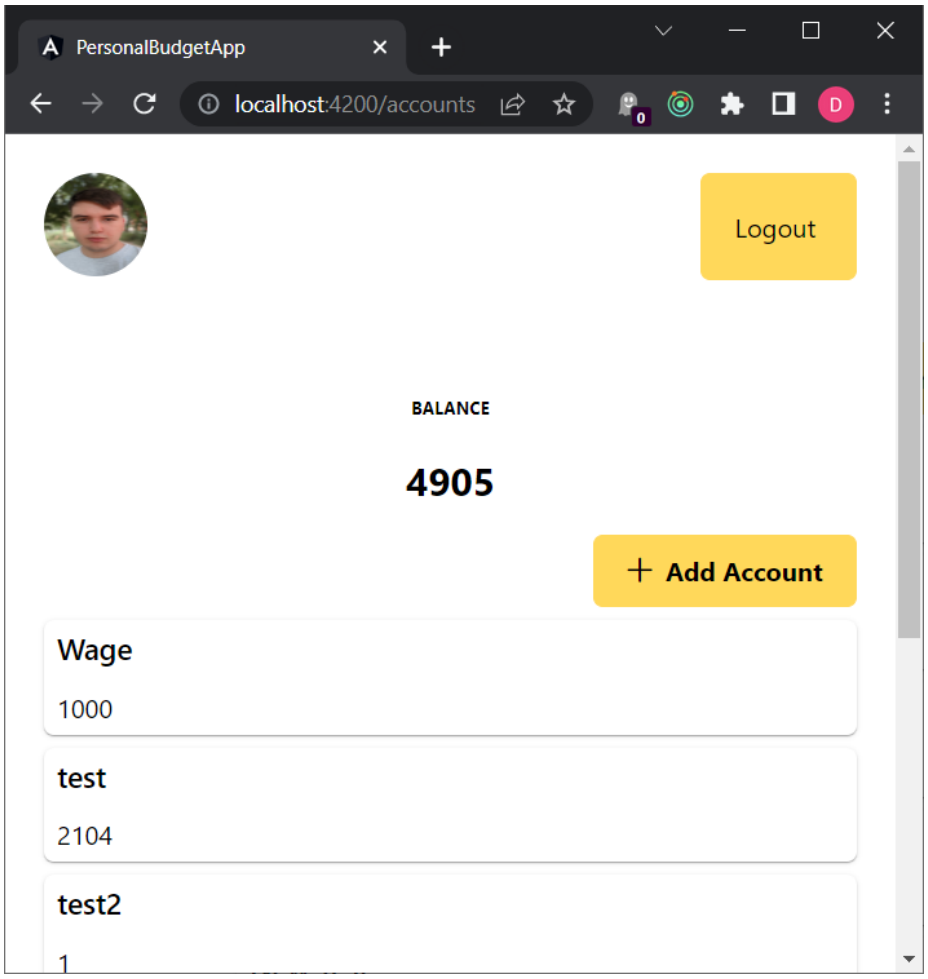

Рисунок 3.2 – демонстрація сторінки з рахунками користувача

Як можна побачити на рис. 3.2, все працює справно, і можна, серед інших можливостей, спостерігати за процесом обігу даних в застосунку (рисунок 3.3).

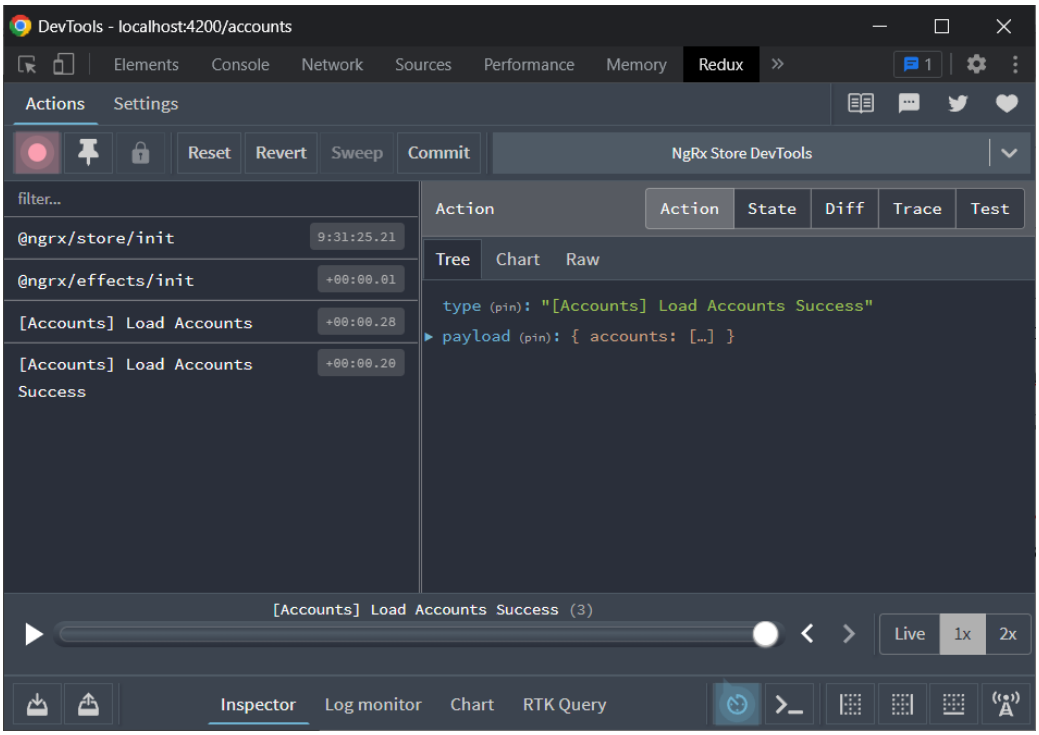

Рисунок 3.3 – Демонстрація виводу Redux DevTools

Щоби порівняти версію застосунку до змін в архітектурі і після, то було прийнято рішення почати порівняння з коду. Компонент AccountsComponent не зазнав суттєвих змін, чого не можна сказати про AppModule. В останньому прибавилось 9 рядків програмного коду, які в купі з вже існуючими дещо ускладнили читабельність цього файлу. Також було додано 5 файлів, які всього складають 116 рядків коду. Це досить великий об’єм, враховуючи, що функціональні можливості програми не змінились, та є ідентичними попередній версії. Але це все не впливає прямо на досвід роботи користувача з програмою, лише на досвід розробника.

Наступним параметром в порівнянні був розмір застосунку, оскільки такий параметр безпосередньо впливає на досвід користувача. Для збірки застосунку в конфігурації production, тобто призначеній для повноцінного використання, було використано команду ng build. Результати збірки застосунку до і після змін можна побачити в таблиці 3.1.

Таблиця 3.1 Результати замірів до змін в застосунку і після

	До змін	Після змін (з NgRx)
Розмір збірки	1.14 мегабайт	1.19 мегабайт
Час збірки	17.7 секунд	23.3 секунди

Отже, розмір збірки не зазнав суттєвих змін, але швидкість збірки уповільнилась майже на 5 секунд, що в контексті невеликого застосунку вже є досить вагомою зміною.

ВИСНОВКИ

В даній кваліфікаційній магістерській роботі було проаналізовано підходи до управління станом в сучасних веб-застосунках на прикладі реального застосунку.

В ході роботи було проведено збір та структуризацію матеріалів про сучасну розробку веб-застосунків, аналіз інструментів і готових рішень з областей front-end розробки.

Створено працюючий веб-застосунок для управління персональними фінансами в двох версіях з принциповими відмінностями архітектури і проведено аналіз цих відмінностей.

Наступним кроком у розвитку цього веб-застосунку має бути повне переведення його на Flux архітектуру, включно зі всіма компонентами застосунку, задля архітектурної уніфікації і поліпшення користувацького досвіду.

СПИСОК ЛІТЕРАТУРИ

1. E. Wohlgethan, «Supporting Web Development Decisions by Comparing Three Major JavaScript Frameworks: Angular, React and Vue.js,» *Bachelorarbeit,* с. 41-41, 2018.
2. J. Wilken, Angular in Action, Shelter Island, NY: Manning Publications Co., 2018.
3. D. Uluca, Angular for Enterprise-Ready Web Applications: Build and deliver production-grade and cloud-scale evergreen web apps with Angular 9 and beyond, 2nd Edition, Birmingham: Packt Publishing, 2020.
4. S. J. A. Emmit, SPA Design and Architecture: Understanding single-page web applications, Shelter Island, NY: Manning Publications Co., 2015.
5. C. Musciano та B. Kennedy, HTML & XHTML: The Definitive Guide (6th Edition), Sebastopol, CA: O'Reilly Media, Inc., 2006.
6. E. Meyer та E. Weyl, CSS: The Definitive Guide: Visual Presentation for the Web, Sebastopol, CA: O'Reilly Media, Inc., 2017.
7. L. Watts, Mastering Sass, Birmingham: Packt Publishing, 2016.
8. Less core team, «Features In-Depth | Less.js,» [Онлайновий]. Режим доступу: https://lesscss.org/features/#mixins-feature. [Дата звернення: 12 12 2022].
9. P. Daniels та L. Atencio, RxJS in Action, Shelter Island, NY: Manning Publications Co., 2017.
10. N. Rozentals, Mastering TypeScript: Build enterprise-ready, modular web applications using TypeScript 4 and modern frameworks, 4th Edition, Birmingham: Packt Publishing, 2021.
11. D. Flanagan, JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language, Sebastopol, CA: O'Reilly Media, Inc., 2020.
12. M. Garreau та W. Faurot, Redux in Action, Shelter Island, NY: Manning Publications Co., 2018.

13. O. Farhi, Reactive Programming with Angular and ngrx: Learn to Harness the Power of Reactive Programming with RxJS and ngrx Extensions, Lod, Israel: Apress, 2017.
14. L. Moroney, The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform, Seattle, WA: Apress, 2017.
15. PrimeTek, «PrimeNG,» [Онлайновий]. Режим доступу: https://www.primefaces.org/primeng/. [Дата звернення: 01 12 2022].

ДОДАТОК А

Вихідний код першої і другої версії застосунків можна знайти у відкритому доступі за посиланнями https://github.com/extrawest/angular_personal_budget_app та https://github.com/danyloshamaiev/personal-budget-app/tree/other відповідно. Дата останнього звернення: 12.12.2022.

ДОДАТОК Б

Приклади впровадження нової архітектури:

Файл src/app/state/index.ts:

```
import {ActionReducerMap} from '@ngrx/store';
import {accountsReducer, AccountsState} from './accounts/accounts.reducers';

export interface AppState {
  accounts: AccountsState;
}

export const reducers: ActionReducerMap<AppState, any> = {
  accounts: accountsReducer,
};
```

Файл src/app/state/accounts/accounts.reducers.ts:

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import {AccountsState} from './accounts.reducers';

export const selectAccountsState =
  createFeatureSelector<AccountsState>('accounts');

export const selectAccounts = createSelector(
  selectAccountsState,
  (state) => state.accounts
);

export const selectTotalBalance = createSelector(selectAccountsState, (state)
=>
  state.accounts.reduce(
    (prevValue, curValue) => prevValue + curValue.balance,
    0
  )
);
```

Файл src/app/state/accounts/accounts.actions.ts:

```
import {createAction, props} from '@ngrx/store';
import {IAccount} from '../../models/account.model';

export const loadAccounts = createAction('[Accounts] Load Accounts');
export const loadAccountsSuccess = createAction(
  '[Accounts] Load Accounts Success',
  props<{accounts: IAccount[]}>()
);
export const addAccount = createAction(
  '[Accounts] Add Account',
  props<{account: Partial<IAccount>}>()
);

export const addAccountSuccess = createAction('[Accounts] Add Account
Success');
```

Файл src/app/state/accounts/accounts.selectors.ts:

```
import {createFeatureSelector, createSelector} from '@ngrx/store';
import {AccountsState} from './accounts.reducers';

export const selectAccountsState =
  createFeatureSelector<AccountsState>('accounts');

export const selectAccounts = createSelector(
  selectAccountsState,
  (state) => state.accounts
);

export const selectTotalBalance = createSelector(selectAccountsState, (state)
=>
  state.accounts.reduce(
    (prevValue, curValue) => prevValue + curValue.balance,
    0
  )
);
```

Файл src/app/state/accounts/accounts.effects.ts:

```
import {Injectable} from '@angular/core';
import {Actions, createEffect, ofType} from '@ngrx/effects';
import {catchError, EMPTY, mergeMap, tap} from 'rxjs';
import {map} from 'rxjs/operators';
import {AccountsService} from '../../services/accounts.service';
import {
  addAccount,
  addAccountSuccess,
  loadAccounts,
  loadAccountsSuccess,
} from './accounts.actions';

@Injectable()
export class AccountsEffects {
  constructor(
    private actions$: Actions,
    private accountsService: AccountsService
  ) {}

  loadAccounts$ = createEffect(() =>
    this.actions$.pipe(
      ofType(loadAccounts.type),
      mergeMap(() =>
        this.accountsService.getUserAccounts().pipe(
          map((accounts) => ({
            type: loadAccountsSuccess.type,
            payload: {accounts},
          })),
          catchError(() => EMPTY)
        )
      )
    )
  );

  addAccount$ = createEffect(() =>
```

```
    this.actions$.pipe(
      ofType(addAccount.type),
      mergeMap(({account}) => {
        return (
          this.accountsService
            // @ts-ignore
            .addUserAccount(account.name, +account.initialBalance)
            .pipe(
              map(() => ({
                type: addAccountSuccess.type,
              })),
              catchError(() => EMPTY)
            )
        );
      })
    )
  );
}
```

Файл src/app/accounts/account.component.ts:

```
import {Component, OnDestroy} from '@angular/core';
import {Router} from '@angular/router';
import {Store} from '@ngrx/store';
import {Observable, retry, Subject, takeUntil} from 'rxjs';
import {IAccount} from '../models/account.model';
import {AccountsService} from '../services/accounts.service';
import {loadAccounts} from '../state/accounts/accounts.actions';
import {
  selectAccounts,
  selectTotalBalance,
} from '../state/accounts/accounts.selectors';

@Component({
  selector: 'app-accounts',
  templateUrl: './accounts.component.html',
  styleUrls: ['./accounts.component.css'],
```

```
})
export class AccountsComponent implements OnDestroy {
  public accounts$: Observable<IAccount[]>;
  public balance$: Observable<number>;
  private unsubscribe$: Subject<void>;

  constructor(
    private accountsService: AccountsService,
    private router: Router,
    private store: Store
  ) {
    this.store.dispatch(loadAccounts());
    this.unsubscribe$ = new Subject<void>();
    this.accounts$ = this.store
      .select(selectAccounts)
      .pipe(retry(1), takeUntil(this.unsubscribe$));
    this.balance$ = this.store
      .select(selectTotalBalance)
      .pipe(takeUntil(this.unsubscribe$));
  }

  ngOnDestroy(): void {
    this.unsubscribe$.next();
    this.unsubscribe$.unsubscribe();
  }

  public openAccount(account: string): void {
    this.router.navigate(['transactions', account]);
  }

  public addAccount(): void {
    this.router.navigate(['accounts', 'new']);
  }
}
```