МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«Інформаційна система адресної доставки товарів з використанням NGRX. Модуль нотифікацій користувача»

Завідувач випускаючої кафедри — **Довбиш А.С.**

Керівник роботи — **Берест О.Б.**

Студента групи ІН-м.92 — **Золотарьов І.О.**

СУМИ 2020

Сумський державний університет
(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*
Спеціальність *«Інформатика»*

Затверджую:
зав.кафедрою _____
_____
"_____" _____ 20___р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

Золотарьову Івану Олександровичу

1. Тема проекту (роботи) *Інформаційна система адресної доставки товарів з використанням NGRX. Модуль нотифікацій користувача*

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) Постановка завдання та формування чітких умов для веб-сайту. 3) Вибір вирішення поставленої задачі. 4) Налаштування середовища та розробка веб-сайту з використанням технології NGRX. 5) Аналіз виконаної роботи.*

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.	*Постановка завдання та формування чітких умов для веб-сайту*		
3.	*Вибір вирішення поставленої задачі*		
4.	*Налаштування середовища та розробка веб-сайту з використанням технології NGRX*		
5.	*Аналіз виконаної роботи*		
6.	*Оформлення пояснювальної записки до дипломної роботи*		

Студент – дипломник _____
(підпис)

Керівник проекту _____
(підпис)

РЕФЕРАТ

Записка: 78 стор., 37 рис., 1 таблиця, 6 додатків, 20 джерел.

Мета роботи — імплементація інформаційної системи адресної доставки товарів та модулю нотифікацій користувача із використанням технології NGRX.

Об'єкт дослідження — процес розробки інформаційної системи.

Предмет дослідження — платформа NGRX, її основні інструменти та їх рекомендовації із її використання.

Методи дослідження — побудова порівняльної таблиці систем управління станом; розробка односторінкового веб-сайту на основі технології NGRX.

Результати — проведено інсталяцію та конфігурацію середовища розробки на основі інструментів VS Code та VS Community; розроблена інформаційна система, що дозволяє зручно використовувати систему з адресної доставки товарів та послуг для всіх учасників процесу. Система представлена у вигляді веб сайту, де серверна частина реалізована на платформі .NET Core та на мові програмування C#, а клієнтська – на Angular та TypeScript.

ANGULAR, SINGLE PAGE APPLICATION, WEB-САЙТ,
STATE MANAGEMENT, SPA, NGRX STORE.

ЗМІСТ

ВСТУП

Технологічні надбання на сьогодні являються не тільки не рідкими, але їх можна назвати стандартом та ключовими для подальшого розвитку людини. Технологічній прогрес в IT-сфері є особливо вражаючим. Що починалося з виконання найпростіших математичних розрахунків – привело нас до куди більш вражаючих результатів. Тепер, більшість наших майнових, соціальних та інших повсякденних контактів ми проводимо без безпосереднього зв'язку. Один ыз найбільших внесків вносять веб-сайти. З'явившись в житті людини всього деяких два десятиліття тому вони впевнено зайняли спочатку нишу представництва компаній, а згодом стали ключовим джерелом інформації, а на сьогодні виконують незліченну кількість функцій, деякими з яких є:

- реклама товарів, послуг та ідей. Сучасно та зручно виконаний веб-сайт перехилить потенційного клієнта до вибору покупки товару, чи послуги, ідей, що пропонуються на ньому;
- продаж продукції, інформації, ідей, послуг. Для сучасної людини – час є однією із основних цінностей. Тож можливість заказу товарів і послуг, online, з телефону або компьютера – відкриває нові можливості як для продавця, так і для клієнта;
- безкоштовне надання деяких послуг та інформації є ефективним засобом залучення нових відвідувачів до деякого ресурсу для здобуття, статистичної інформації або показу реклами, якщо ресурс являється рекламним майданчом;
- підтримка клієнтів. Одна із додаткових послуг, але та, що значно впливає на імідж сучасної компанії.

Наслідком стрімкого зростанню популярності – зросли і вимоги до готового продукту. Що в свою чергу привело до вдосконалення методів розробки. Одним із основних подій у кругу розробників цього десятиліття стала

поява HTML5, що змінила сучасне представлення веб-сайту на, так званий, односторінковий сайт [7].

Зараз існує широкий вибір різних інструментів, що полегшують розробку. І, хоча бажаний результат можна досягти використовуючи будь-який із них, або не використовуючи жодний, виконання проекту у найкоротший термін є пріоритетом компаній та співробітників на рівні із якнайвищими показниками якості продукту, простоти масштаболваності та підтримки у подальшому.

Одним із високоефективних інструментів є засоби state management (управління станом). State – це інформація, що доступна програмі у деякий момент часу. Найчастіше, використання таких систем управління станом є необов'язковим на кожній із популярних платформ для розробки, тож вони є додатковими, опціональними бібліотеками для кожної із платформ розробки. Прикладами популярних зв'язок бібліотеки state management та платформи є NGRX для Angular, Redux для React та Vuex для Vue. Надалі, у процесі даної роботи буде проведено аналіз кожної із зазначених бібліотек, та зроблено вибір щодо використання такої, що є найкращою для задачі, та, використавши цю бібліоткеку - виконаємо реалізацію проекту.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Огляд бібліотек управління станом

State management бібліотеки є нещодавнім надбанням, тож популярність при виборі має значну роль, бо це прямо впливає ступінь підтримки бібліотеки, кількість вирішених питань та інформації по цьому засобу. Так,можна виділити три основні інструменти: Redux, NGRX та Vuex.

1.1.1 Redux

На сьогодні є безперечно найбільш популярною бібліотекою, для найбільш популярної платформи – React. Тож, базуючись тільки на цьому можна дивитися в його сторону. Але, варто зазначити, конкуренти мають достатню кількість користувачів, тож цей фактор не буде мати великої ваги.

Основні принципи Redux:

- Єдине джерело істини: стан всього додатка зберігається в дереві об'єктів в одному магазині.
- Стан доступний лише для читання: єдиний спосіб змінити стан - це визвати action, об'єкт, що описує те, що сталося.
- Зміни робляться чистими функціями: Щоб вказати, яким чином дерево стану змінюється внаслідок виклику action, ми пишемо чисті reducers[4].

Рисунок 1.1 - Взаємодія інструментів Redux

1.1.2 Vuex

Vuex - це патерн управління станом і бібліотека для додатків на Vue.js. Має найбільшу популярність на східному напрямку(Китай і т.д.). Він служить центральним сховищем даних для всіх компонентів програми та забезпечує передбачуваність зміни даних за допомогою певних правил[5].

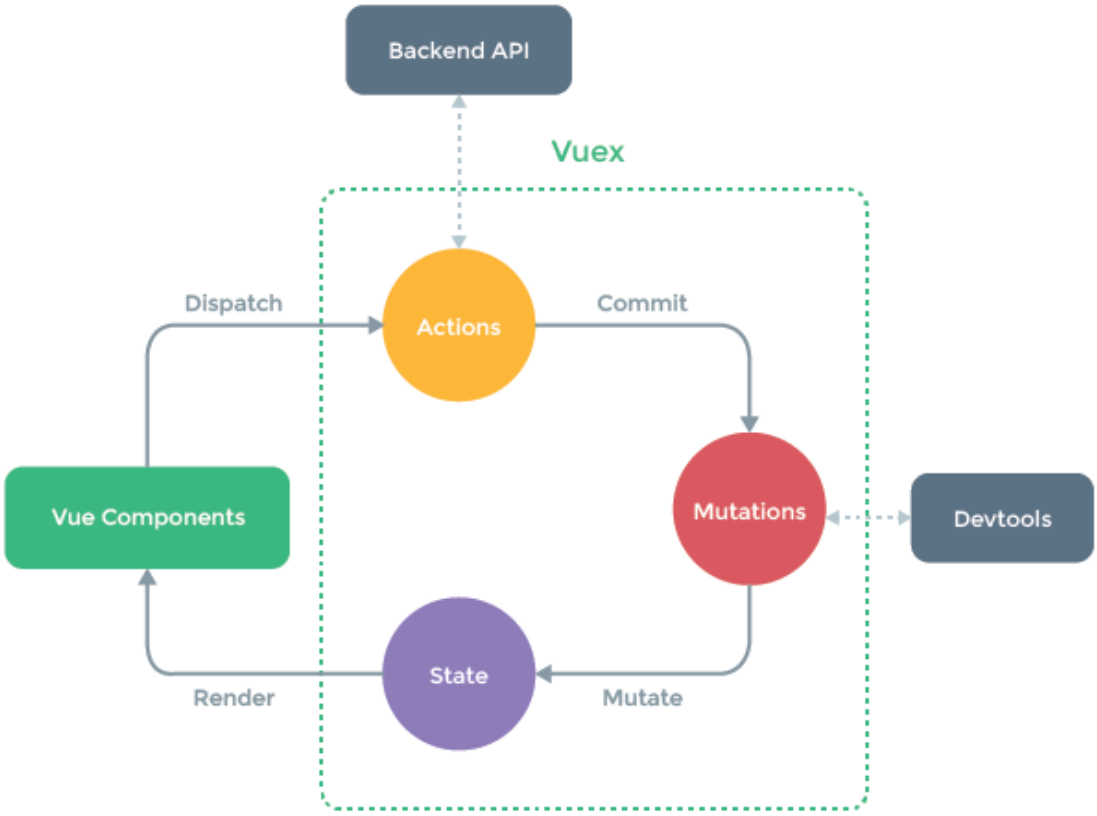

Рисунок 1.2 - Місце Vuex у додатку

1.1.3 NGRX

NgRx - це бібліотека керування станом для створення реактивних додатків на Angular. Вона забезпечує управління станом, ізоляцію побічних ефектів, управління колекціями об'єктів, прив'язку маршрутизатора, генерацію коду та інструменти розробника, які покращують досвід розробників при побудові

багатьох різних типів програм. Реалізує паттерн Redux. Переваги, що надає бібліотека NGRX:

- Оскільки у нас є одне джерело правди, і ми не можемо безпосередньо змінити стан, додатки будуть працювати більш злагоджено.
- Використання шаблону Redux дає нам багато цікавих функцій (dev tools), які полегшують налагодження.
- Тестування додатків стає простіше, оскільки ми вводимо чисті функції для обробки змін стану, а також тому, що і ngrx, і rxjs, мають безліч чудових можливостей для тестування.
- Як тільки освоюєш використання ngrx, розуміння потоку даних в додатках стає неймовірно простим і передбачуваним [14].

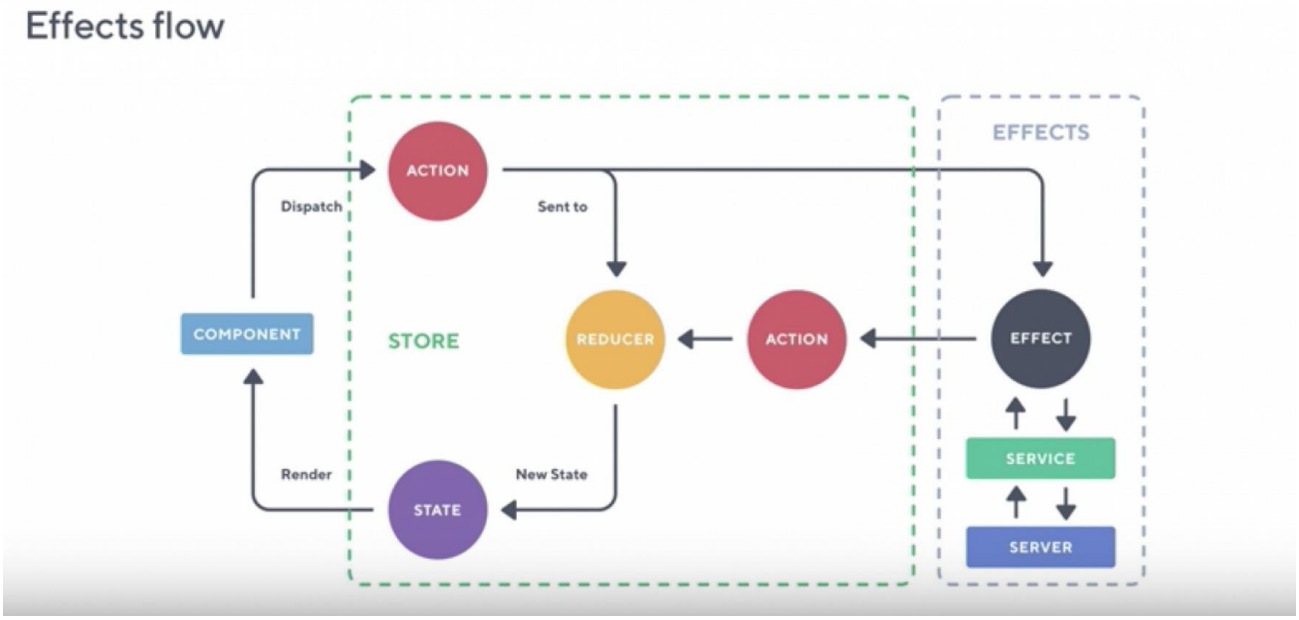

Рисунок 1.3 - NGRX flow

1.2 Детальний огляд бібліотеки NGRX

NGRX - це група бібліотек, «натхненна» бібліотекою Redux, яка, в свою чергу, «натхненна» шаблоном Flux. Це означає, що шаблон Redux є спрощеною

версією шаблону Flux, а NGRX є версією шаблону redux з використанням Angular і RxJS. Тож, NGRX є «Angular/RxJs»(рис 1.4) версією Redux.

«Angular» - тому що ngrx - це бібліотека для використання в Angular додатках.

«Rxjs» - тому що реалізація ngrx працює з використанням потоків observables, і різних операторів, що надаються «rxjs» [15].

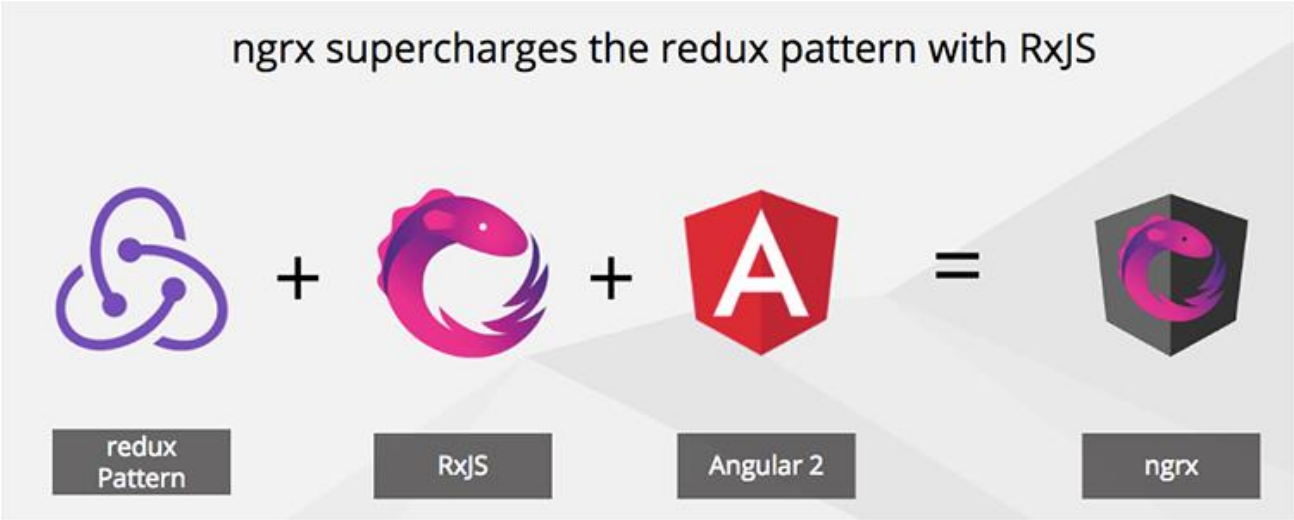

Рисунок 1.4 - Складові NGRX

Перейдемо до її інструментів та ключових засобів.

Використання store архітектури являє собою великий архітектурний зсув: з приходом односторінкових додатків ми перенесли перетворення model до view з сервера на клієнт.

Набагато більше, ніж набір бібліотек, екосистема NGRX дійсно є набором шаблонів розробки додатків: store, actions, reducers, selectors, всі ці засоби для реалізації цих шаблонів.

Всі концепти, пов'язані з NGRX store, такі як actions, reducers, тощо, простіше зрозуміти, якщо підходити до них як засіб для реалізації загального дизайну додатків.

Централізований store дизайн дуже ефективний у вирішенні багатьох проблем, які важко вирішити інакше, що це робиться шляхом введення деяких архітектурних компромісів.

Даний пакет модулів імплементує redux-патерн і дає можливість скористатися деякими прийомами, що допомагають зробити наш додаток більш гнучким і масштабованим. При використанні NGRX наш додаток буде мати:

- Єдино точний джерело даних і його зручне використання в компонентах;
- Зміну стану тільки за допомогою чистих функцій – reducers;
- Здійснення змін за допомогою відправки дій (actions) що описують тип змін і містять корисне навантаження (payload);
- 2 типи компонентів (контейнер \ уявлення);
- Ізоляцію side-effects;
- Потужний time-travelling дебагер.

Інші пункти опціональні (управління сутностями (entity) з коробки, синхронізація router і store, кодогенерацію і т.д.) [19].

Тож, що це означає для рядового розробника? А значить це наступне:

- Односпрямований і ясний потік даних (unidirectional data flow);
- Ясне розуміння змін стану програми;
- Поділ зміни стану від шару уявлення.

Так як redux дуже і дуже поширений патерн за останні кілька років у фронтенді, це певним чином уніфікує наші програми, що дуже корисно для бізнеса та розробників [11].

На рисунку 1.5 ми можемо бачити життєвий цикл NGRX, розглянемо кожний із складових детальніше.

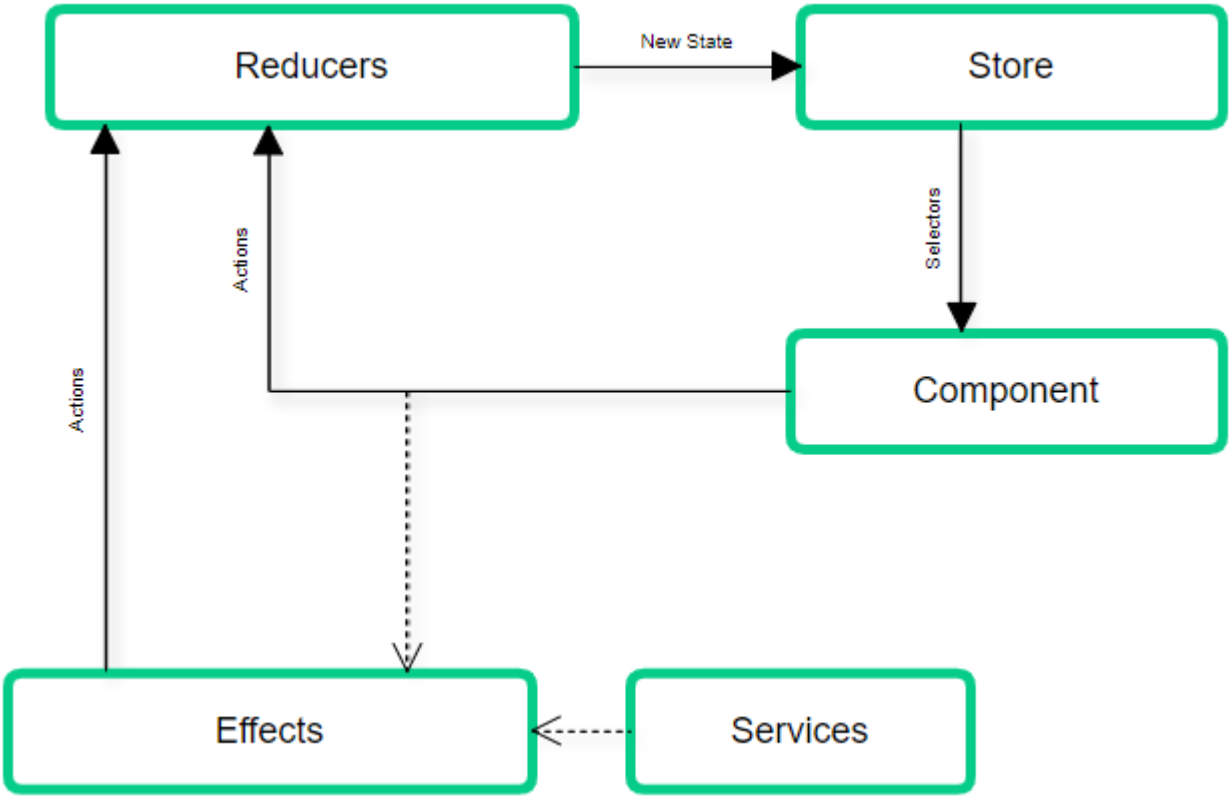

Рисунок 1.5 - Складові NGRX

1.3 Reducers

Reducers в NgRx відповідають за обробку переходів з одного стану в наступний стан у програмі. Reducer функції обробляють ці переходи, на основі типу action, що вони обробляють. Reducer функції є чистими функціями, оскільки вони виробляють один і той же висновок для даного входу. Вони без побічних ефектів і синхронно обробляють кожен стан переходу. Кожна reducer функція приймає останню відправлену дію, поточний стан, і визначає, чи слід повернути знову змінений стан або початковий стан. Є кілька послідовних частин кожного стану, керованого reducerом:

- Інтерфейс або тип, який визначає форму стану.
- Аргументи, що включають початковий стан або поточний стан і поточну дію.

Reducers використовують оператори switch в комбінації з дискримінованими об'єднаннями TypeScript, визначеними у діях, для забезпечення типо-безпечної обробки action у reducer. Switch конструкція використовується для визначення reducer, що відповідає типу action, що був визваний. Після виклику action всі зареєстровані reducers викликаються. Який reducer виконується визначається типом action, що визначений в умові випадку switch конструкції. З цієї причини кожен оператор switch завжди включає стандартний випадок, який повертає попередній стан, коли reducer функції не потрібно обробляти дію [12].

1.4 Actions

Actions є одним з основних будівельних блоків в NGRX. Вони виражають унікальні події, які відбуваються у додатку. Починаючи від взаємодії користувача зі сторінкою, зовнішньої взаємодії через мережеві запити та безпосередньої взаємодії з API пристроїв, ці та інші події описані за допомогою actions. Виділимо основні правила написання actions:

- Написання actions перед написанням основної імплементації, щоб зрозуміти і отримати спільне знання про функцію, що реалізується.
- Категоризування actions на основі джерела події.
- Actions швидко пишуться, тож чим на більшу кількість actions розділена імплементація, тим краще виражена і більш наглядною є реалізація додатку.
- Фіксація подій, а не команд, оскільки ми розділяємо опис події та обробку цієї події.
- Надайте контекст до action, з більш детальною інформацією, яку можна використовувати для налагодження з інструментами розробника [13].

1.5 Selectors

Дерево станів може стати досить великим об'єктом, і не має сенсу розміщувати весь цей об'єкт там, де потрібна тільки його частина.

Сховище NGRX надає нам функцію «селектор» для отримання фрагментів нашого сховища. А якщо нам потрібно застосувати деяку логіку до цього фрагменту перед використанням даних в компонентах? Тут в гру вступають селектори. Вони дозволяють нам обробляти дані фрагмента стану незалежно від компонента. Функція «select» сховища приймає в якості аргументу чисту функцію, вона і є нашим селектором.

Селектори - це в основному спосіб фільтрувати і отримувати дані з стану програми.

При використанні @ ngrx/entity, після створення адаптера, можна скористатися функцією getSelectors.

Але, якщо потрібно фільтрувати стан іншим способом, необхідно використовувати функцію createSelector для цього.

Ця функція приймає від 1 до 8 аргументів плюс функцію результату для виконання фільтра. Всі аргументи будуть доступні функції результату, можна використовувати їх разом для виконання фільтра [12].

1.6 Effects

У service-driven додатку Angular компоненти несуть відповідальність за взаємодію з зовнішніми ресурсами безпосередньо через сервіси. Натомість ефекти надають можливість взаємодіяти з цими сервісами та ізолювати їх від компонентів. Ефекти - це те місце, де ви виконуєте такі завдання, як вибірка даних, тривалі завдання, які створюють кілька подій, та інші зовнішні взаємодії, де компонентам не потрібне явне знання цих взаємодій.

Основні принципи:

- Ефекти виділяють побочні ефекти від компонентів, дозволяючи більш чисті компоненти, які беруть інформацію із state та викликають actions;
- Ефекти - це тривалі сервіси, які прослуховую кожний викликаний action, відправленого зі store;
- Ефекти фільтрують ці дії на основі типу дії, що їх цікавить. Це робиться за допомогою оператора;
- Ефекти виконують завдання, які є синхронними або асинхронними і повертають нову дію [12].

1.7 Observables

Observables схожі на масиви, за винятком того, що елементи не зберігаються в пам'яті, а надходять асинхронно з плином часу (потоки).

Ми можемо підписатися на observables і реагувати на їх події. JavaScript observables - це реалізація шаблону observer.

Реактивні розширення (зазвичай звані з префіксом Rx) надають observables для JS через RxJS.

Observables є потоками. Ми можемо спостерігати будь-який потік: від подій зміни розмірів в існуючих масивах до відповідей API.

Ми можемо створити observables практично з усього, що завгодно.

Promise - той же observable, але віддає тільки одне значення, а observables можуть повертати багато значень з плином часу.

Ми можемо працювати з observables по-різному. RxJS використовує безліч операторів.

Observables часто візуалізується за допомогою точок на лінії. Оскільки потік складається з асинхронних подій з плином часу, легко осмислювати це лінійним способом і використовувати саме такі зорові образи, щоб зрозуміти реактивні оператори [15].

1.8 Angular

Angular - це платформа для побудови клієнтських додатків на HTML і TypeScript. Angular написаний на TypeScript. Він реалізує основні та додаткові функції набором бібліотек TypeScript, які ви імпортуєте у свої програми.

Розглянемо основні тезиси, що стосуються даного фреймоворку.

В якості мови шаблону використовується Typescript, який представляє собою мову програмування від Microsoft.

Хоча Typescript і є основною мовою для Angular, додатки можна також писати за допомогою таких мов як Dart або JavaScript.

Angular підтримується Google.

Angular націлений на розробку односторінкових додатків, тобто SPA-рішень (Single Page Application). Як приклад можна привести популярні додатки для соціальних мереж (Twitter, Instagram і Facebook) .

Angular надає клієнтську MVC-інфраструктуру, яка допомагає у запуску і створенні динамічних додатків до сучасного рівня якості.

Програми, написані на Angular, сумісні з різними браузерами. Angular автоматично обробляє код JavaScript, відповідний для кожного браузера;

Чистий і точний дизайн користувальницького інтерфейсу.

Проста маршрутизація.

Структура Angular полегшує розширення синтаксису HTML і легко створює повторно використовувані компоненти по директивам.

В цілому, Angular - це фреймворк для створення великомасштабних, високопродуктивних і простих в обслуговуванні веб-додатків [14].

1.9 Постановка задачі

Проблема складності сучасних веб-сайтів є надзвичайно актуальною на теперішній час. Адже, коли не було необхідності зберігати та взаємодіяти з великою частиною даних на клієнтській частині – роль правильної архітектури

не мала вирішального значення. Але, сьогодні, з ускладненням браузерної частини додатка – правильно обрана платформа, бібліотеки та вірно побудована архітектура мають вирішальне значення для розробки легко масштабованих та підтримуваних додатків в умовах високої конкуренції аутсорсингових компаній.

Тож, перш ніж починати розробку, треба зрозуміти яка саме система буде найбільш актуальна та виправдана. Тому, буде логічним кроком буде порівняння конкурентних систем та вибір найкращої. Для виконання цієї цілі складемо порівняльну таблицю.

Після вибору системи, потрібно обрати, інсталювати та налаштувати середовище та програмні засоби, що будемо використовувати при розробці.

Створення загальної структури проекту, налаштування root-store, та реалізація першої його гілки, на прикладі дій логіну та регістрації.

Наступний крок – реалізація частини додатку, що призведе до виконання поставленої мети.

Отже, постанова задачі:

1. Огляд бібліотек клієнтської розробки веб додатків на основі state management. Вибір найбільш задовольняючої із них.
2. Налаштування середовища.
3. Створення структури проекту, root-store та реалізація state management для користувача.
4. Розробка інформаційної системи таргетної доставки товарів.

2 ВИБІР МЕТОДІВ ВИРІШЕННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

2.1 Вибір бібліотеки управління станом

Для порівняння бібліотек управління станом були сформовані наступні критерії, що були оцінені як ключові для вибору засобу:

- наявність даних по роботі;
- ступінь підтримки та оновлень;
- необхідність у часі для вивчення та розробки;
- розширюваність;
- інструменти для спостереження за станом;
- складність розробки мобільної версії на основі веб-сайту;
- вбудовані інструменти контролю побочних діх;

В результаті було побудовано порівняльню таблицю за обраними критеріями. (Таблиця 1).

Таблиця 1. Порівняльна таблиця бібліотек управління станом.

Бібліотека / **Особ-ливість**	**Redux**	**Vuex**	**NGRX**
Наявність даних по роботі	10	10	10
Ступінь підтримки та оновлень	10	10	10
Необхідність у часі для вивчення та розробки	4	7	2
Розширюваність	8	7	10

Інструменти для спостереження за станом	10	10	10
Складність розробки мобільної версії на основі веб-сайту	6	3	8
Вбудовані інструменти контролю побочних діх	10	0	10
Підсумок	58	47	60

У результаті порівняння бібліотек за встановленими критеріями було зроблено вибір у сторону NGRX, адже вона у найбільшій мірі задовольняє ключовим критеріям.

2.2 Вибір програмних засобів

2.2.1 TypeScript

TypeScript - це скриптова мова, що компілюється в JavaScript.

TypeScript починається і закінчується разом з JavaScript. Він використовує такий же синтаксис та семантику, який знають мільйони розробників JavaScript.

TypeScript компілюється до чистого, простого JavaScript коду, який працює на будь-якому браузері, в Node.js або в будь-якому механізмі JavaScript, який підтримує ECMAScript 3 (або новіше).

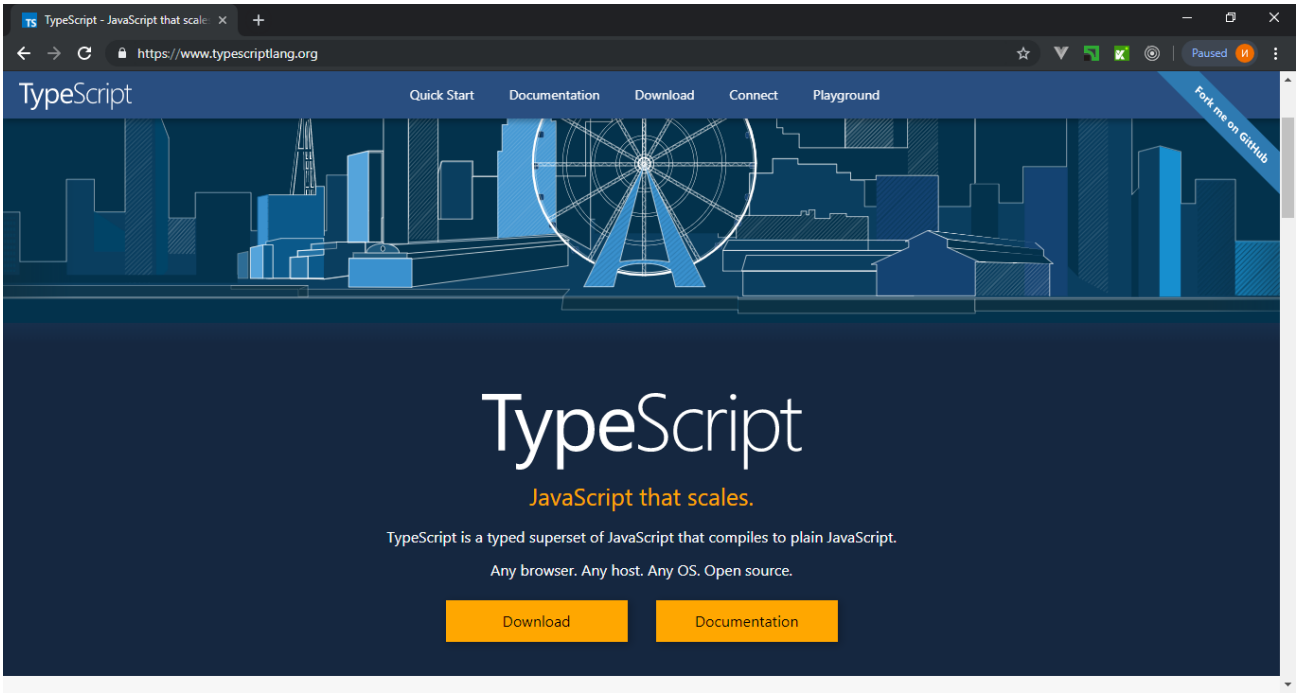

Рисунок 2.1 - Офіційний сайт TypeScript

Типи дозволяють розробникам JavaScript використовувати високопродуктивні засоби та практики розробки, такі як статична перевірка(до компіляції коду) та рефакторинг коду при розробці програм JavaScript.

TypeScript пропонує підтримку останніх і розвиваючихся функцій JavaScript, починаючи з ECMAScript 2015, такі як асинхронні функції та декоратори [9].

2.2.2 Signalr

Для спрощення роботи з комунікаціями реального часу була створена спеціальна бібліотека під назвою SignalR.

Вона надає простий API для створення функціоналу, який дозволяє викликати функції JavaScript на стороні клієнта з серверного коду, написаного за допомогою мов платформи .NET. SignalR значно спрощує роботу з комунікаціями реального часу. Бібліотека обробляє всі підключення і автоматично розсилає повідомлення всім підключеним клієнтам, або яким-небудь специфічним клієнтам.

Фактично бібліотека SignalR складається з API серверної сторони, який застосовується в коді на C #, і з клієнтських бібліотек JavaScript.

SignalR надає розробникам дві моделі: постійні підключення (Persistent Connection) і хаби (Hubs).

Для обміну даними між клієнтом і сервером SignalR використовує той спосіб передачі або той транспорт, який найбільше підходить в кожній ситуації. Однак розробники можуть мати найвищий пріоритет спосіб передачі. SignalR надає наступні типи технологій для взаємодії сервера і клієнта:

- WebSockets
- Server-sent events
- Forever Frames
- Long polling [20].

2.2.3 Visual Studio Code

Visual Studio Code це безкоштовний крос-платформний редактор коду, розроблений Microsoft. Програма має відкритий вихідний код. Виходячи з опитування, проведеного Stack Overflow в 2017 році, це один з найпопулярніших редакторів коду, яким користуються більше 24% розробників.

Він оснащений доступним набором інструментів для редагування та налагодження. Редактор легко інтегрується з іншими сервісами. Його власні властивості також легко розширити.

Нова функція Live Share надає можливості для парного програмування, завдяки чому можна з легкістю працювати над однією базою коду. Для цього не доводиться конфігурувати інструменти розробки або возитися з настройками оточення.

Крім того, серед особливостей VS Code ми бачимо Git-інтеграцію, IntelliSense (технологія автодоповнення), підсвічування синтаксису для найпопулярніших мов програмування і багато інших прекрасних функцій.

Існує можливість кастомізації плагінами, що поставляються Microsoft або створених спільнотою.

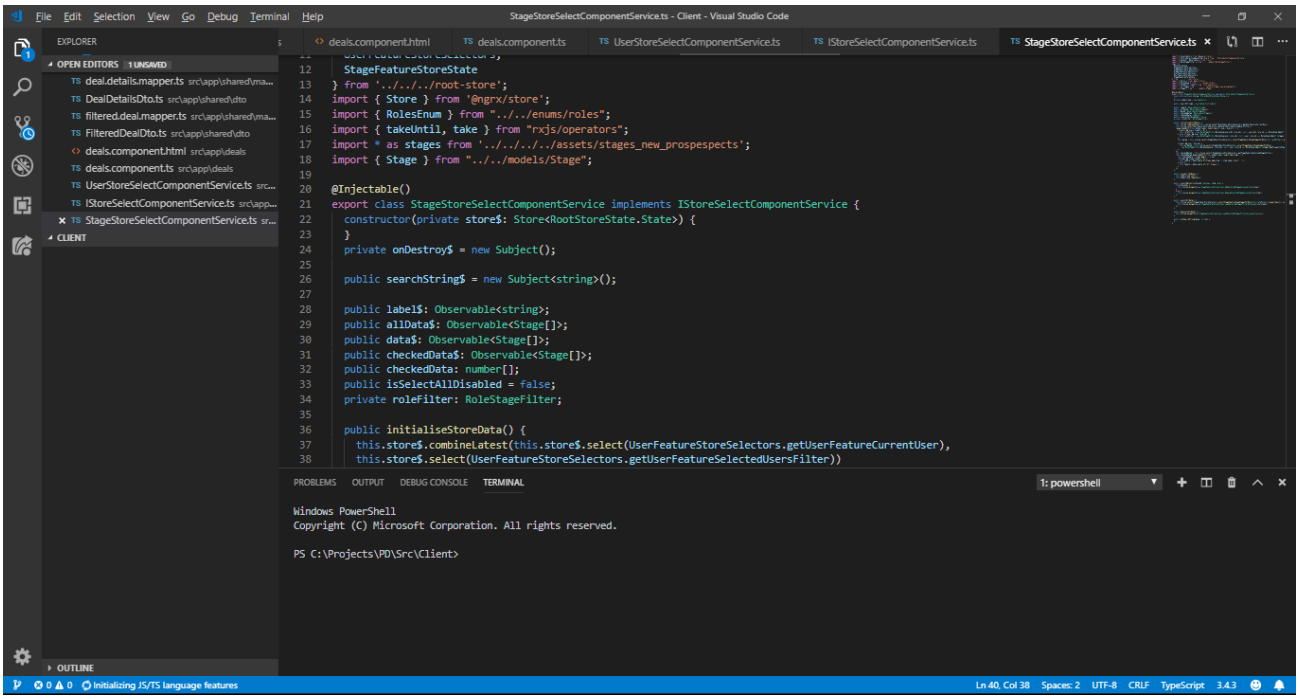

Рисунок 2.2 - Інтерфейс Visual Studio Code

2.2.4 Microsoft Visual Studio та C#

Програмне забезпечення Microsoft Visual Studio - це інтегроване середовище розробки (IDE). Microsoft Visual Studio містить набір інструментів, хмарних сервісів і розширень, які необхідні розробникам для створення додатків та ігор. Можна використовувати продукти Visual Studio з уже існуючими інструментами. Visual Studio допомагає редагувати, налагоджувати і створювати код, а потім публікувати додаток.

Microsoft Visual Studio включає в себе компілятори, засоби автодоповнення коду, візуальні редактори макетів і багато інших функцій, що полегшують процес розробки програмного забезпечення.

Основні можливості, що надає Microsoft Visual Studio IDE:

- Розробка додатків для Android, iOS, Mac, Windows, а також розробка хмарних і веб-додатків.

- Швидке написання коду.
- Легкі налагодження і діагностика.
- Часте тестування і впевнений випуск релізів.
- Розширення та налаштування відповідно до своїх потреб.
- Ефективна спільна робота [17].

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Створення проекту

Спершку нам необхідно інсталювати та сконфігурувати наші програмні засоби.

Закачати Visual Studio ми можемо із офіційного сайту(рисунок 3.1).

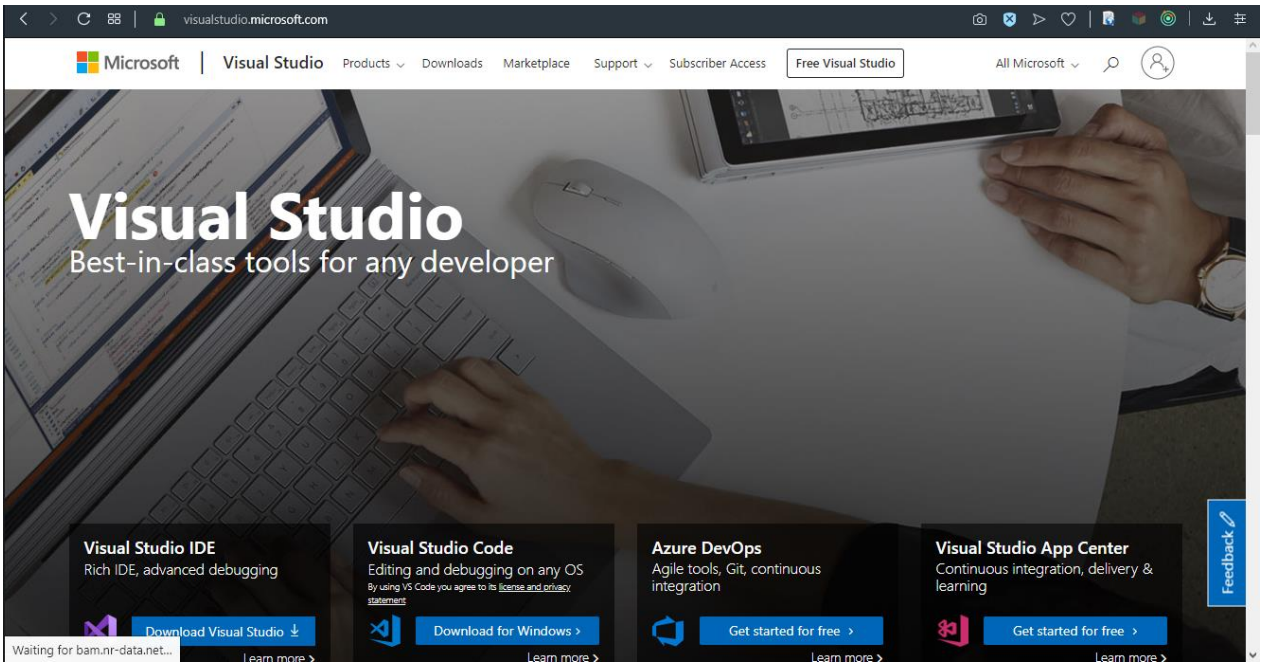

Рисунок 3.1 - Офіційний сайт Visual Studio

Завантажуємо з офіційного сайту та виконуємо установку nodejs, що необхідна для встановлення деяких залежностей. Після цього - інсталюємо Angular CLI. Для цього виконуємо наступну команду.

npm i -g @angular/cli

Відкриваємо VS Community IDE,, знаходимо шаблон ASP .NET Core Web Application(рисунок 3.2) та створюємо проект даного типу, обираючи Angular шаблон у наступному вікні(рисунок 3.3).

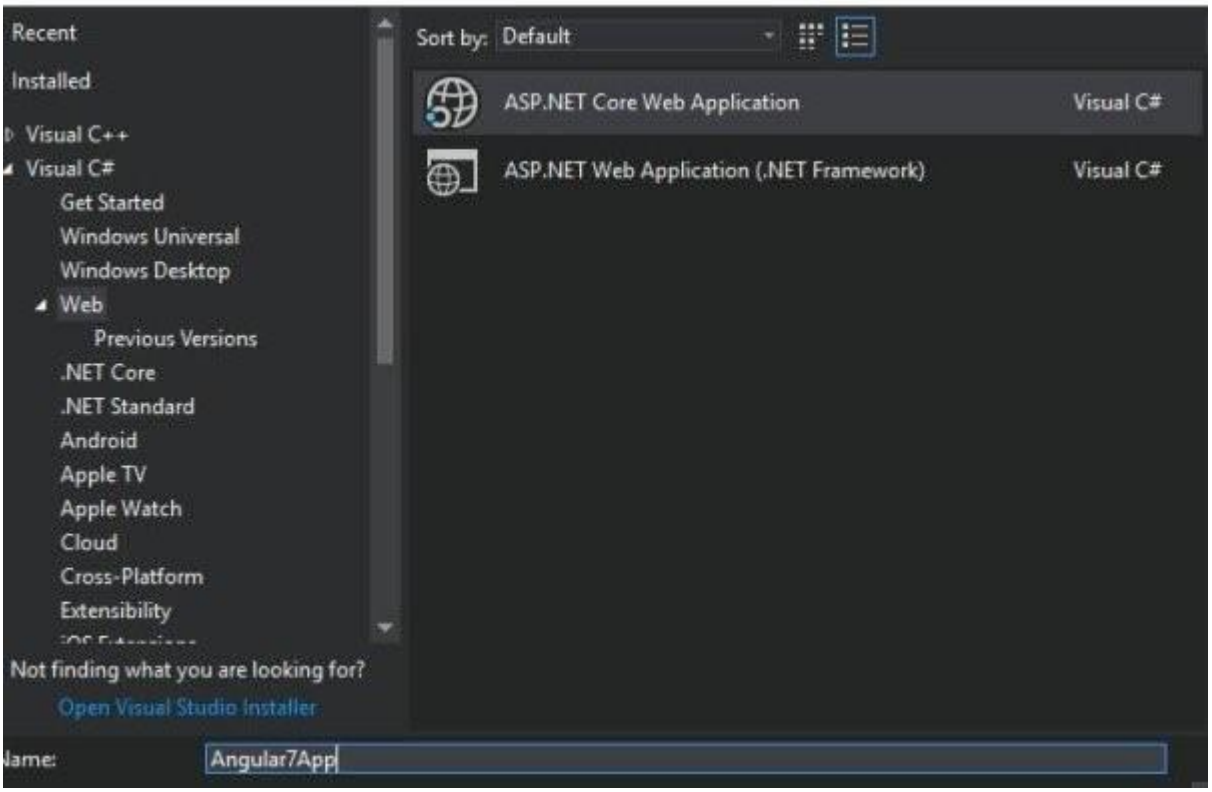

Рисунок 3.2 - Створення веб-додатка

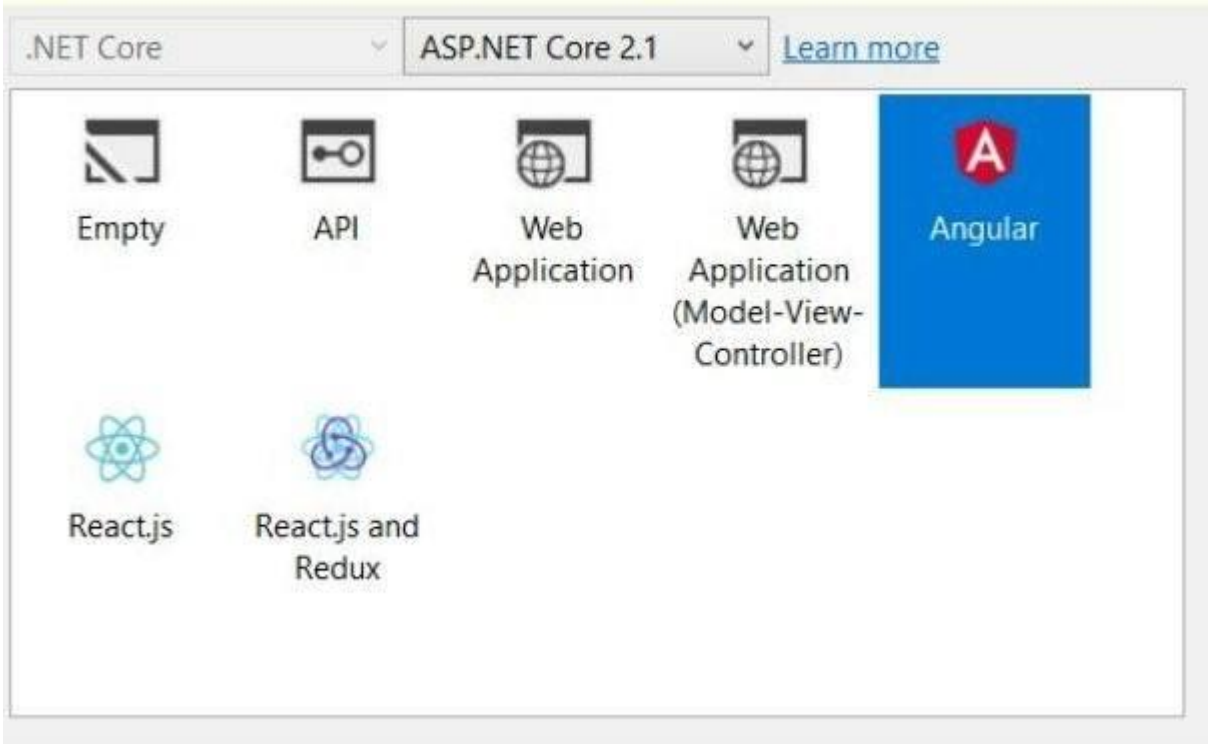

Рисунок 3.3 - Вибір Angular шаблону

Переходимо до папки ClientApp, що створена за замовчуванняи і буде містити «клієнтський» код та інсталюємо пакети NGRX за допомогою наступної команди:

npm i @ngrx/store-devtools @ngrx/effects @ngrx/entity @ngrx/entity ngrx/router-store @ngrx/store

Також, інсталюємо бібліотеку Angular Material, що надає нам доступ до компонентів у стилі material-дизайну. Команда для установки:

ng add @angular/material.

Проект запускається командою *ng serve.*

Наступний кроком є імплементація серверної частини проекту. Був обраний code-first підхід для роботи з міграціями та базою даних, та модель Domain-Driven-Design.

3.2 Імплементація state management

У ClientApp/src створюємо такі папки:

- guards;
- components;
- models;
- helpers;
- services;
- та store.

Надалі створюємо наступні компоненти: home, register, login, та alert за допомогою команди.

ng g c «component name».

Auth guard створюємо у guards, а у helpers – jwt та error інтерсертори,

створюємо сутність User у моделі. Вона має наступні поля.

id: number;

firstName: string;

lastName: string;

username: string;

token: string;

password: string;

Створюємо та імплементуємо сервіси користувачів та аутентифікації.

У них конфігуруємо взаємодію з серверними едпоінтами. Наприклад, метод register сервісу користувача виглядає так:

```
register (user: User) {
    return this.httpClient.post(`${environment.apiUrl}/user/register`, user);
  }
```

Результат – структура, що показана на рисунку 3.4.

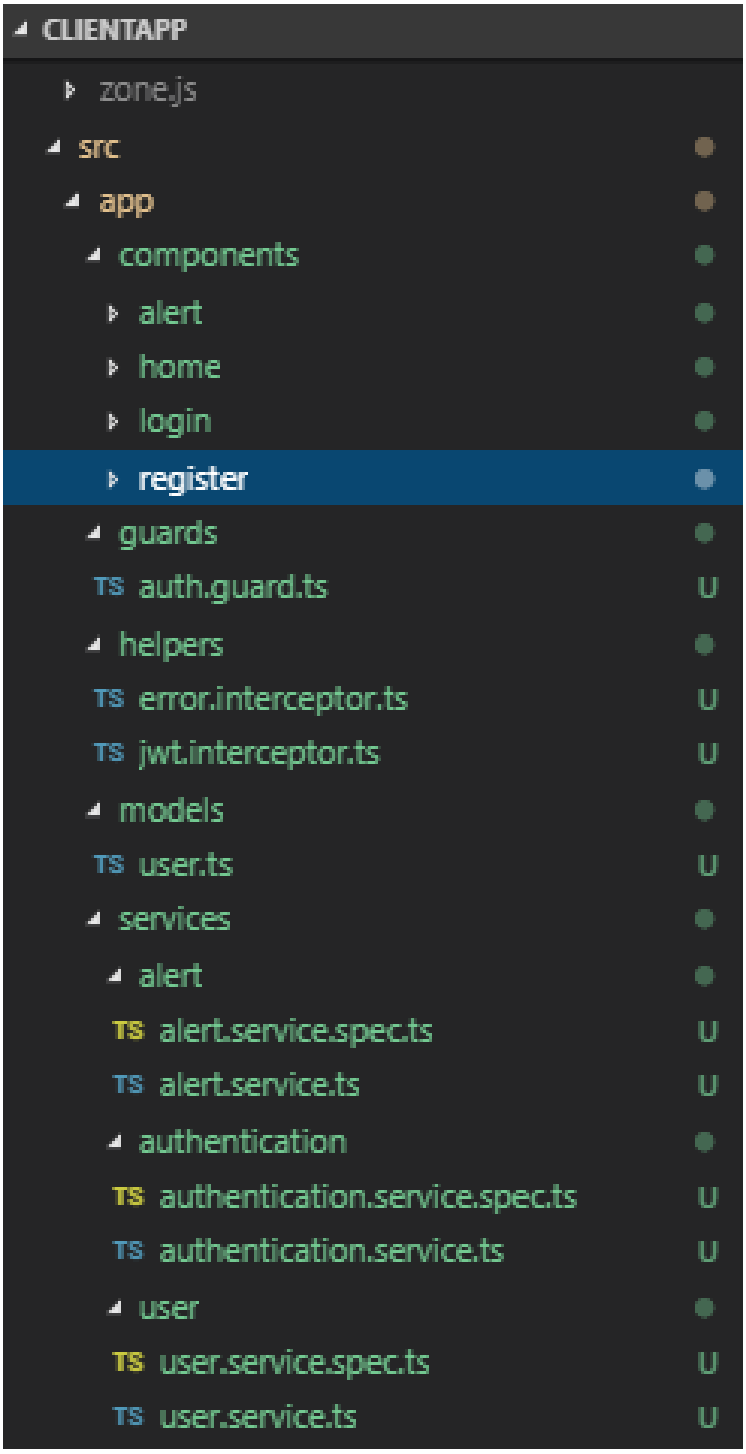

Рисунок 3.4 - Структура проекту

Alert є компонентом для зображення повідомлень стан виконання операції. Наприклад – повідомлення про невірні дані при спробі логіну на сайті.

У такому випадку ми бачимо наступне повідомлення «Provided login data is incorrent».

Login компонент ми використовуємо при логіні на веб-сайт. Також міститься кнопка переходу на сторінку для реєстрації. У проекті використовуємо localStorage для запису Json Web Token користувача. Якщо він вже присутній – користувач перенаправляється до сторінки home.

Компонент реєстрації(register) служить для створення нових користувачів. При успішній реєстрації користувач повертається на сторінку входу для подальшого процесу логіну.

Кнопка cancel – також повертає користувача до форми логіну.

Home компонент є компонентом на який ми потрапляємо після входу на сайт.

Після успішного логіну вгорі ми можемо побачити меню, що винонує функцію навігації або виходу користувача із системи(логауту).

Так, було наведено функції основних компонентів програми. Тепер розглянемо, як все працює.

Auth guard - основна його мета – заборонити доступ до недоступних для користувачів, що не залогінились, сторінок, та перенаправляти їх на сторінку виконання входу.

Jwt interceptor ми використовуємо для приєднання токену аутентифікації до заголовка запитів до серверної частини, щоб користувач отримав необхідний доступ до веб-серверу для отримання необхідних даних, що є захищеними від користувачів що не пройшли процес авторизації.

Перейдемо до сервісів.

Authentication сервіс - використовуємо для відправки реквестів для логіну на сервер, та інсталювання токену до локального хранилища, при успішному логіні.

User сервіс використовуємо для відправки реквестів реєстрації, отримання або видалення, та оновлення даних користувачів.

Імплементуємо Store.

Створимо root-state папку у папці store, а вже у root-state – нову папку user-feature-store.

У root-state додамо наступні файли: root-state.ts(визначає абстракцію стану), index.ts(використовується для більш зручного barrel-exporty), root-store-module.ts(окреслює структуру state, включаючи вкладені стани) та selectors.ts(селектори для отримання помилок). Створюємо папку selectors у user-feature-store, та наступні файли у новій папці: selectors.ts, effect, actions, index, state, user-feature-module та reducer. Маємо структуру файлі, що зображена на рисунку 3.5.

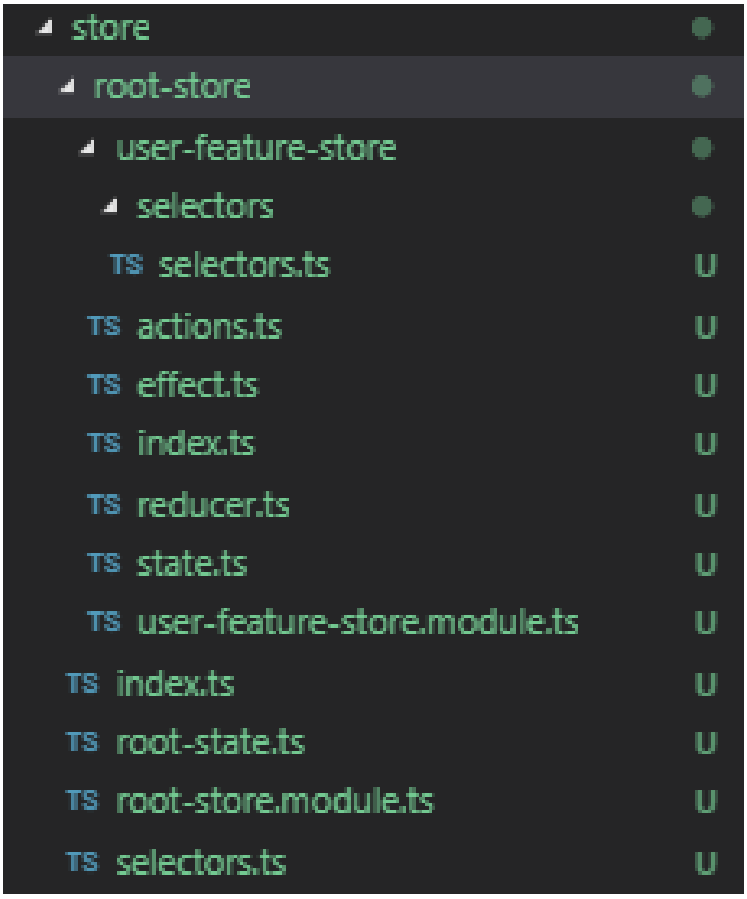

Рисунок 3.5 - Структура стану

У state.ts ми бачимо ініціалізацію користувацького стану, що зображена на рисунку 3.6.

Рисунок 3.6 - Імплементація state

Типізована ініціалізація через EntityState<Type> дозволяє не визначати явно зберігання об’єктів типу.

Також, можемо побачити поля зберігання поточного користувача, state помилок та завантаження.

Actions. Тут імплементовано enum, що описує дії, які можна буде побачити за допомогою моніторингу роботи store у redux-dev-tools додатку.

А також actions, як правило на одну дію є три actions. Перший для того, щоб отримувати у effect-ах. Він без префіксу, а також два для неуспішного та успішного виконання дії. Цей enum та імплементацію user-actions можна побачити на рисунку 3.7.

```
export enum ActionTypes {
  LOAD_REQUEST = '[User Feature] Load Users Request',
  LOAD_FAILURE = '[User Feature] Load Users Failure',
  LOAD_SUCCESS = '[User Feature] Load Users Success'
}
export class LoadUsersRequestAction implements Action {
  readonly type = ActionTypes.LOAD_REQUEST;
}
export class LoadUsersFailureAction implements Action {
  readonly type = ActionTypes.LOAD_FAILURE;
  constructor(public payload: { error: string }) {}
}
export class LoadUsersSuccessAction implements Action {
  readonly type = ActionTypes.LOAD_SUCCESS;
  constructor(public payload: User[]) {}
}
```

Рисунок 3.7 - Імплементація actions

У reducer ми бачимо конструкцію switch-case (рисунок 3.8), на кожний action якого реалізовано reducer, що кожний раз повертає новий state. Для reducer, що прив’язаний до дії без префіксу встановлено loading в true, та очищено поле помилки. При прив’язці до дії з success – виконуємо повернення стану з новими даними.

При прив`язці до дії з failure – встановлюємо error поле.

```
switch (action.type) {
  case ActionTypes.LOAD_REQUEST: {
    return {
      ...state,
      isLoading: true,
      error: null
    };
  }
  case ActionTypes.LOAD_SUCCESS: {
    return featureAdapter.addAll(action.payload, {
      ...state,
      isLoading: false,
      error: null
    });
  }
  case ActionTypes.LOAD_FAILURE: {
    return {
      ...state,
      isLoading: false,
      error: action.payload.error
    };
  }
```

Рисунок 3.8 - User Feature reducer.ts

Розглянемо Effects.

Відрізнити їх можна за характерними декораторами. Тут ми спостерігаємо ефекти на вхід користувача(рисунок 3.9), реєстрацію, вихід та завантаження користувачів. Кожний з них відповідно використовує сервіс, та повертає success або failure action в залежності від результатів запиту.

```
@Effect()
loginUserEffect$: Observable<Action> = this.actions$.pipe(
  ofType<UserFeatureStoreActions.LoginUserAction>(UserFeatureStoreActions.ActionTypes.LOGIN_USER),
  switchMap(action =>
    this.authService
      .login(action.payload.loginData.username, action.payload.loginData.password)
      .pipe(
        map(item => {
            action.payload.callback();
            this.router.navigate([this.route.snapshot.queryParams['returnUrl'] || '/']);
            return new UserFeatureStoreActions.SetCurrentUserSuccessAction(item);
          }),
          catchError(error => {
            this.alertService.error(error);
            return observableOf(new UserFeatureStoreActions.LoginUserActionFailure({ error }));
          })
      )
  )
);
```

Рисунок 3.9 - User Feature effect.ts

Селектори.

Були імплементовані селектори отримання станів Error та isLoading, поточного user, та user по Id(рисунок 3.10), а також завантаження всіх користувачів.

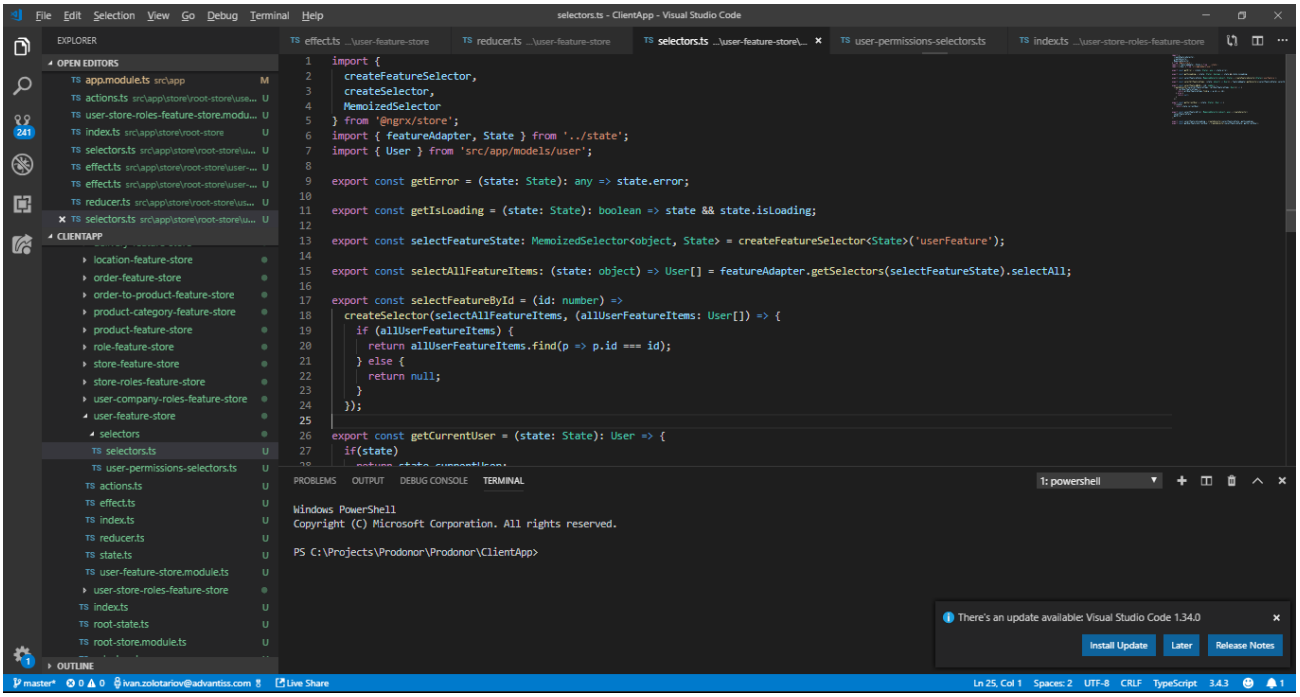

Рисунок 3.10 - User Feature selectors.ts

Завершивши реалізацію можна інсталювати Redux-dev-tools додаток для Chrome – та підключити до проекту, додавши до імпортів root-store-module – наступну строку StoreDevToolsModule.instrument().

Так store dev tools інструмент підключений до нашого проекту.

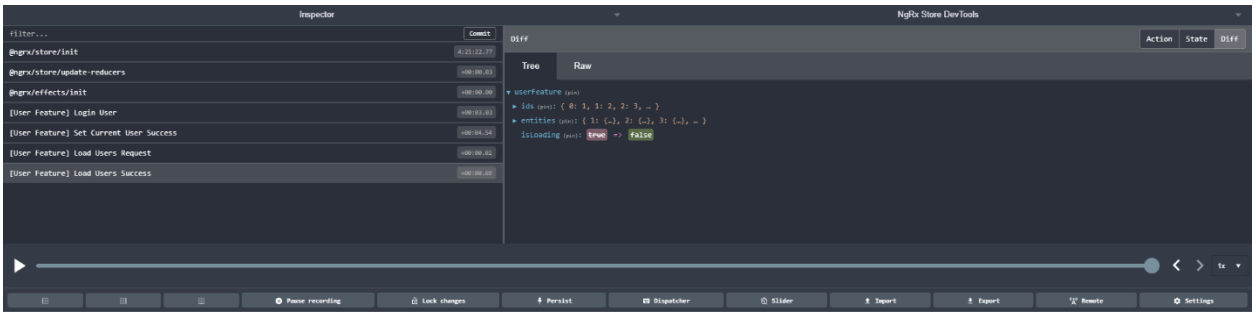

Рисунок 3.11 - Демонстрація роботи інструменту Store Dev Tools

Можна побачити череду викликів actions при виконанні входу користувача(рисунок 3.11). Також є можливістю time-travel по actions.

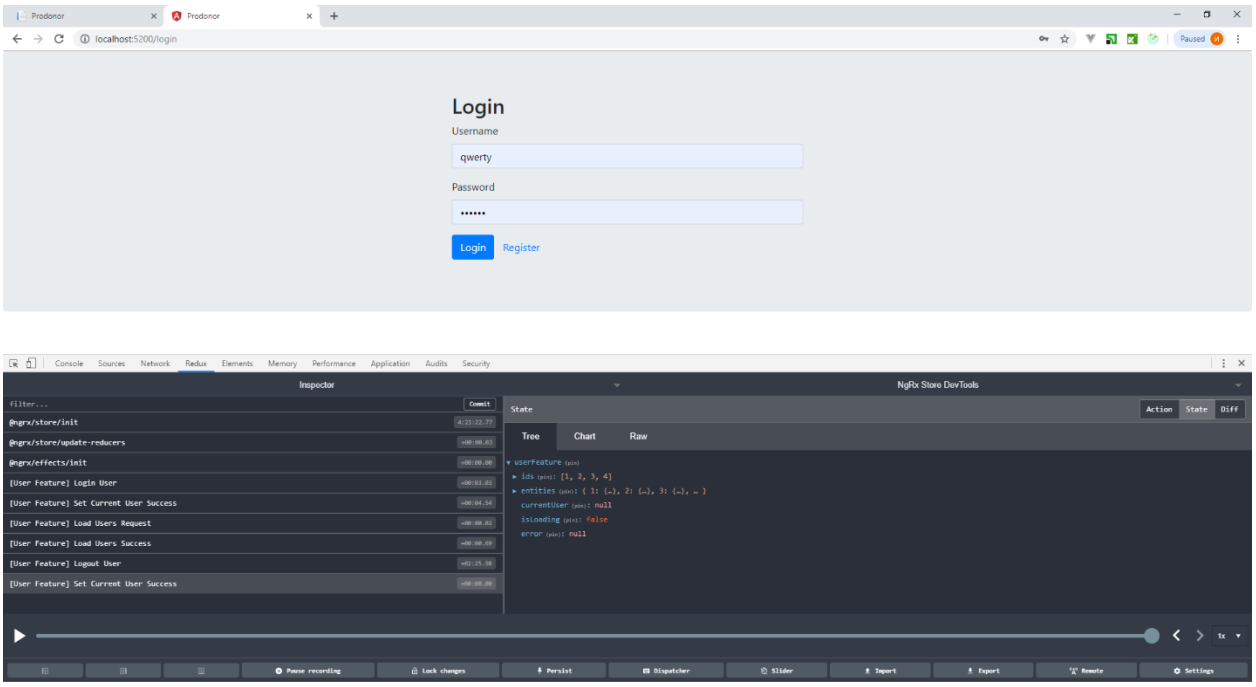

Рисунок 3.12 - Дані Store Dev Tools при виході користувача

Після виходу бачимо наступні дані(рисунок 3.12). Властивість currentUser встановлений до значення null.

3.3 Імплементація модулю нотифікацій

Переглянемо, як імплементувати максимально простий та масштабований модуль нотифікацій. Почнемо з того, що створимо alert компонент, що використаємо на найвищому клієнтського додатку – додамо його до app.component.html. Для нотифікацій – використаємо бібліотеку toastr у alert компоненті. Спосіб взаємодії можна побачити на рисунку 3.13. Ця бібліотека дозволяє показувати повідомлення користувачеві. Створимо alert.service для використання у новому компоненті. Реалізуємо методи, для додавання нових повідомлень - вони наведені на рисунку 3.14. А також публічний метод getMessageObservable для отримання об’єкту для підписки на отримання нових повідомлень, рисунок 3.15.

```
ngOnInit() {
    this.subscription = this.alertService.getMessageObservable().subscribe(message => {
        setTimeout(() => {
            if(message && message.type === 'success')
                this.toastr.success('Action succeeded!', message.text)
            else if(message && message.type === 'error')
                this.toastr.error('Action failed!', message.text)
        })
    });
}
```

Рисунок 3.13 – Використання бібліотеки toastr

```
success(message: string, keepAfterNavigationChange = false) {
    this.keepAfterNavigationChange = keepAfterNavigationChange;
    this.subject.next({ type: 'success', text: message });
}

error(message: string, keepAfterNavigationChange = false) {
    this.keepAfterNavigationChange = keepAfterNavigationChange;
    this.subject.next({ type: 'error', text: message });
}
```

Рисунок 3.14 – Методи додавання сповіщень

```
getMessageObservable(): Observable<any> {
    return this.subject.asObservable();
}
```

Рисунок 3.15 – Метод отримання об’єкту підписки на сповіщення

Таким чином ми створили найбільш простий і ефективний інструмент для управління нотифікаціями користувача, тепер перейдемо до використання цього інструмента.

Все що нам потрібно для використання реалізації нотифікацій – виконати inject створеного сервіса на рівні будь-якого конструктора, де ми хочемо використати нотифікації і, викликаючи методи для додавання повідомлень – показувати сповіщення користувачеві.

У нашому проекті даний модуль використано у зв’язці із NGRX Store та Signalr. Розглянемо як надсилати сповіщення із store. Все що нам потрібно – у кожному feature, де ми плануємо надсилати нотифікації про помилки, або ж про деякі успішно виконані запити, що потребують сповіщення – викликати alertservice.error або ж alertservice.success метод відповідно. Приклад використання – можна побачити на рисунку 3.16, де можна спостерігати додавання нового повідомлення про помилку кожного разу, сервер повертає результат із статусом, що не є успішним.

```
@Injectable()
export class UserFeatureStoreEffects {
  constructor(private userService: UserService, private authService: AuthenticationService, private actions$: Actions,
    private router: Router, private alertService: AlertService, private route: ActivatedRoute) {}
  @Effect()
  loadRequestEffect$: Observable<Action> = this.actions$.pipe(
    ofType<UserFeatureStoreActions.LoadUsersRequestAction>(UserFeatureStoreActions.ActionTypes.LOAD_REQUEST),
    switchMap(action =>
      this.userService
        .getAll()
        .pipe(
          map(items => new UserFeatureStoreActions.LoadUsersSuccessAction(items)),
            catchError(error => {
              this.alertService.error(error);
              return observableOf(new UserFeatureStoreActions.LoadUsersFailureAction({ error }));
            }
            )
        )
    )
  );
```

Рисунок 3.16 – Процес додавання сповіщень у store

```
@Injectable({
  providedIn: 'root'
})
export class SignalrService {
  private hubConnection: signalR.HubConnection;

  constructor(private alertService: AlertService) {
    this.startConnection();
  }

  public startConnection = () => {
    this.hubConnection = new signalR.HubConnectionBuilder()
                         .withUrl('http://localhost:14031', {
                           skipNegotiation: true,
                           transport: signalR.HttpTransportType.WebSockets})
                         .build();
    this.hubConnection
      .start()
      .then(() => console.log('Connection started'))
      .catch(err => console.log('Error while starting connection: ' + err));
  }

  public getMessage = () => {
    this.hubConnection.on('messageReceived', (data) => {
      if(data && data.type == 'error')
        this.alertService.error(data.message, true)
      else if(data && data.type == 'success')
        this.alertService.success(data.message, true)
    });
  }

  public userListererActivation(userName): void {
    if (this.hubConnection.state === signalR.HubConnectionState.Connected){
      this.hubConnection
      .invoke('UserListererActivation', userName)
      .catch(err => console.error('userListererActivation error:', err));
    }
  }
}
```

Рисунок 3.17 – Процес додавання сповіщень у signalr

Аналогічний принцип взаємодії із інструментом використовується і у signalr, показаний на рисунку 3.17. Signalr дозволяє сповіщати нашого користувача про зміни у стані системи, що спричинені іншими користувачами без перезавантаження сторінки, наприклад відхиленням створеного замовлення, або ж результат виконаної оплати, обробка якої може займати деякий час.

Результат імплементації даного модулю можна побачити на прикладах помилки аутентифікації(рисунок 3.18), а також при успішно опрацьованій оплаті з використанням signalr на рисунку 3.19.

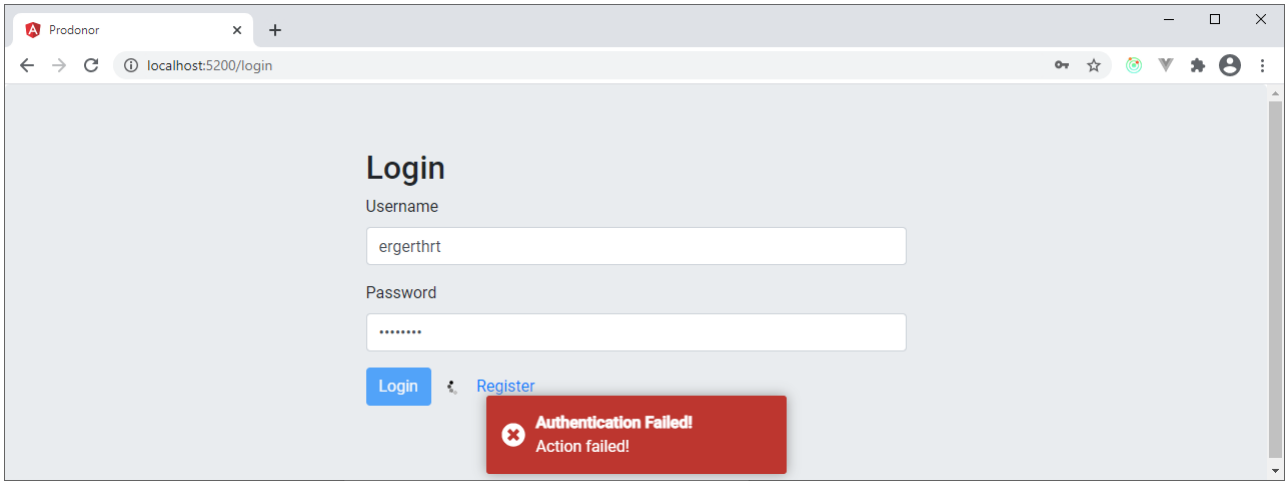

Рисунок 3.18 – Сповіщення помилки аутентифікації

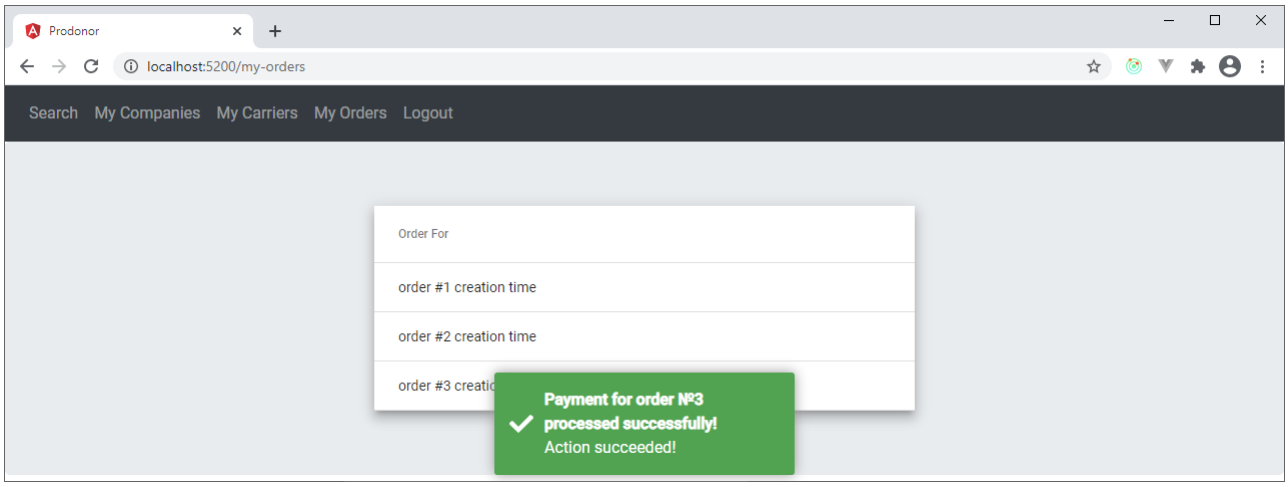

Рисунок 3.19 – Сповіщення про обробку платежу

Отже, було створено інструмент сповіщення користувача, що зручно використовувати у комбінації із будь-якою бібліотекою або компонентом.

3.4 Розробка проекту адресної доставки товарів

Коли серверні ендпоінти сконфігуровані – необхідно відтворити відповідну модель даних на клієнтській частині проекту(рисунок 3.20).

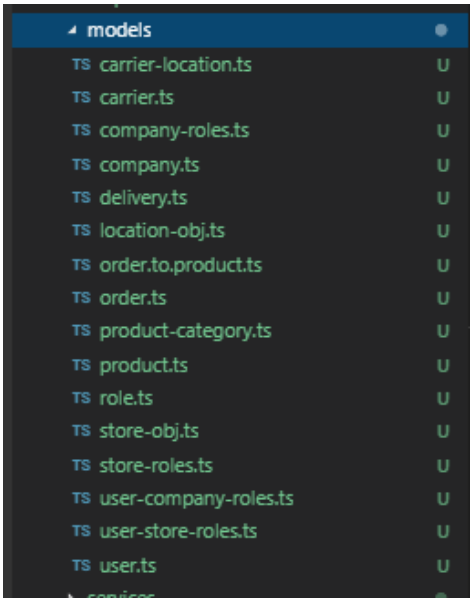

Рисунок 3.20 - Моделі даних

Створимо store-feature, сервіс до кожної моделі даних, та наступні компоненти: carrier, my-carriers, order, store, company, my-orders, product, my-companies. Для того щоб була понятною структура створених компонентів, переглянемо деякі сценарії користувача. З головної сторінки – користувачу доступні такі шляхи(після авторизації):

- «My-companies»(рисунок 3.21) - тут розташовані компанії у володінні користувача.
- «My-orders»
- «My-carriers»

Із «My-companies» можливо перейти до компанії, виконавши клік на обрану компанію із списку, також можна створити нову сутність. Перейшовшг на сторінку деякої компанії юзер бачить лістинг магазинів компанії(рисунок 3.19), а також поля оновлення даних, або додавання прав на редагування обраним користувачам.

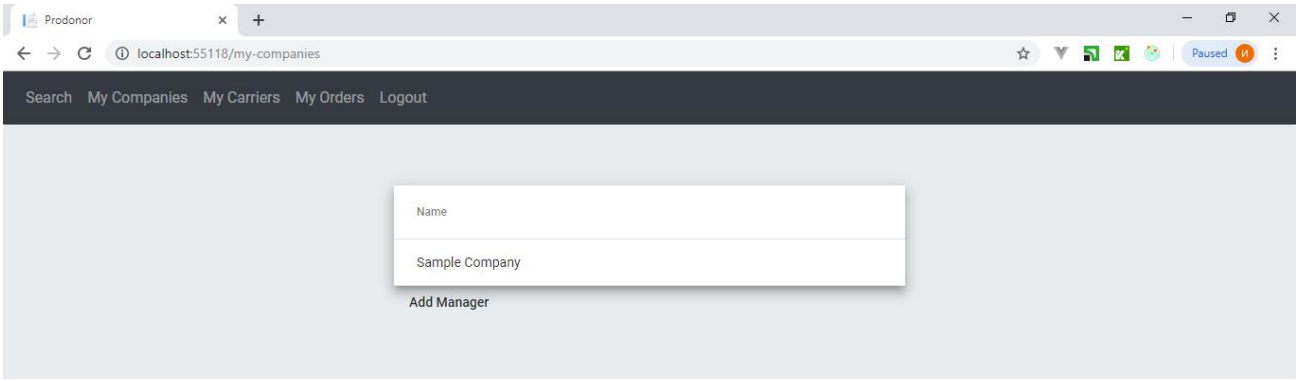

Рисунок 3.21 - Меню «My-companies»

На сторінці магазину(рисунок 3.23) за схожим принципом – є таблиця продуктів у магазині, та поля для оновлення адреси або імені, а також кнопка зміни прав на редагування, тепер уже на рівні магазину також присутня кнопка додавання продукту.

При переході на будь-який продукт – ми опиняємося на сторінці оновлення інформації про цей продукт(рисунок 3.24).

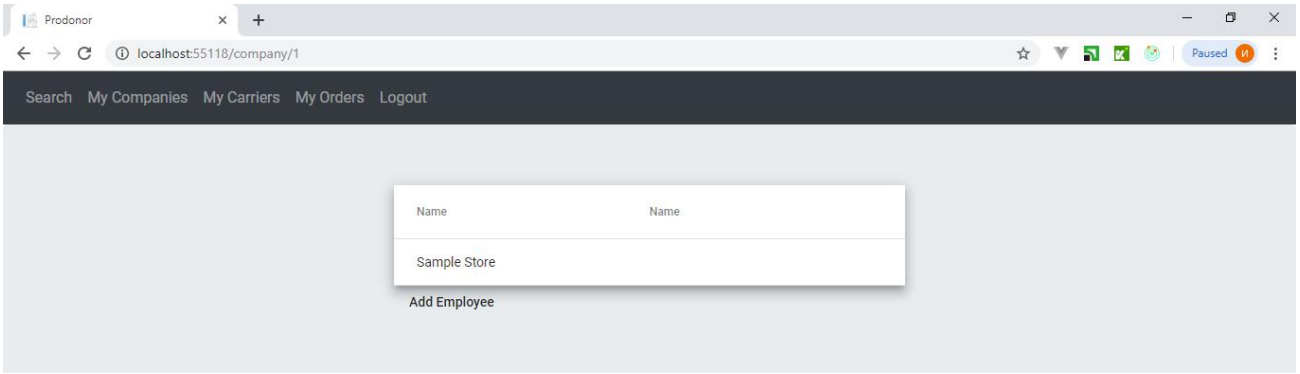

Рисунок 3.22 - Магазини компанії

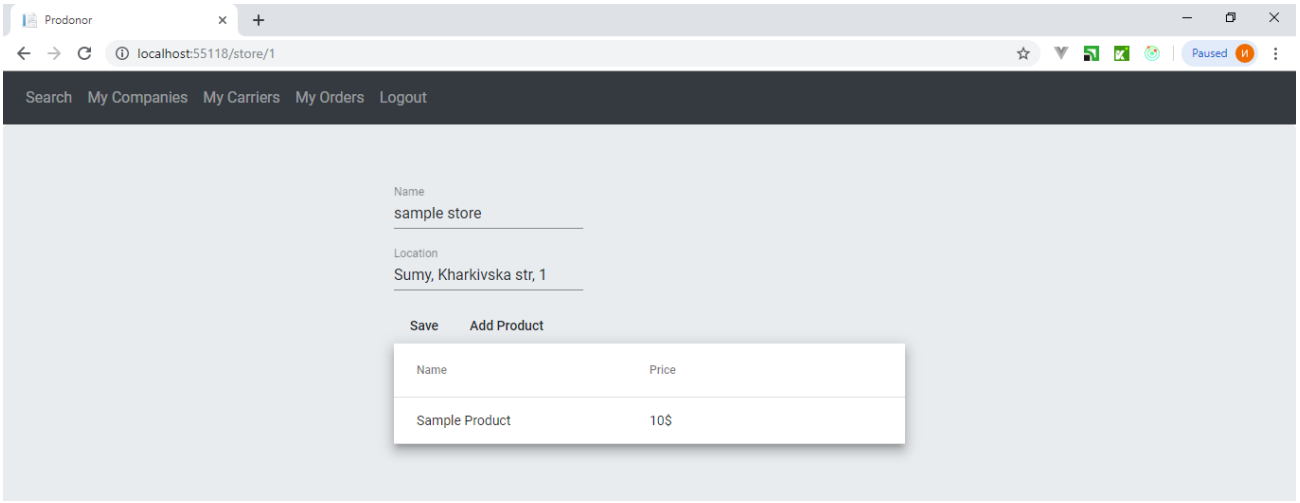

Рисунок 3.23 - Редагування магазину
та управління продуктами

На сторінці «My-carriers» (рисунок 3.25) присутні створені перевізники, при переході на сторінку яких – ми бачимо поточний список ордерів на доставку.

Із сторінки «My-orders» (рисунок 3.26) ми можемо переходити на сторінку створення замовлення, та спостерігати лист створених раніше. Для раніше створених замовлень – можна проводити оплату через інтеграцію із сервісом stripe. Приклад сторінки оплати можна побачити на рисунку 3.28. При створенні нового ордеру(рисунок 3.29) необхідно вказати телефон та адресу доставки.

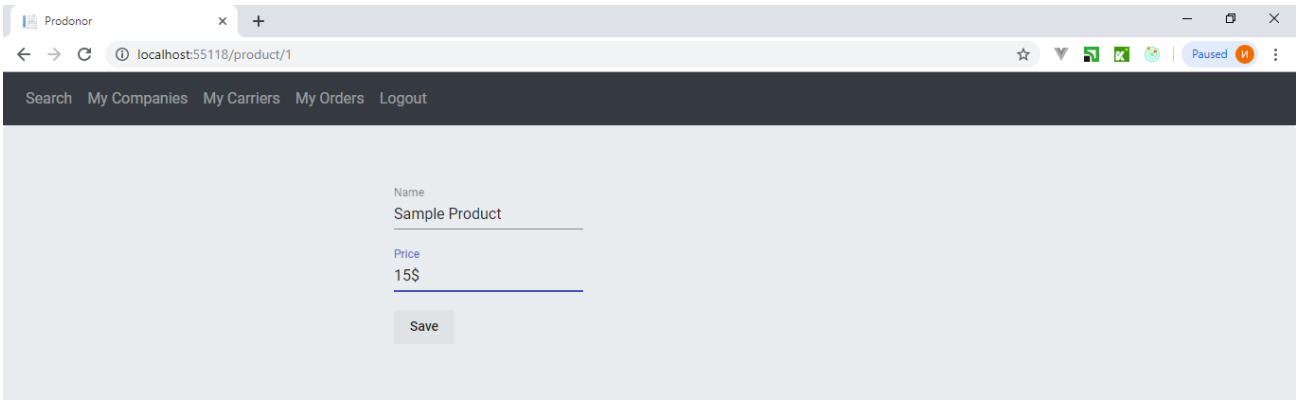

Рисунок 3.24 - Редагування продукту

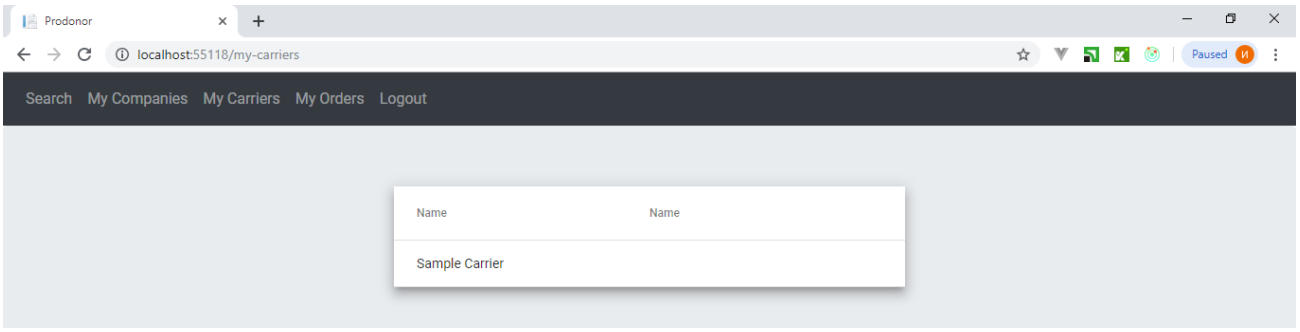

Рисунок 3.25 - Меню «My-carriers»

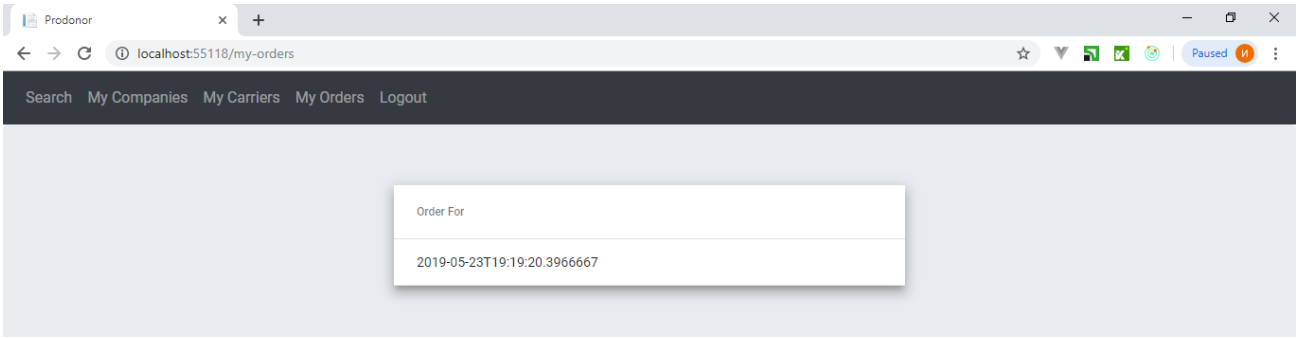

Рисунок 3.26 - Меню «My-orders»

Переглянемо обмін даними між різними features, що зображено на риснку 3.27. У папці selectors/user-company-roles-feature-store створимо новий селектор «user-to-feature-selectors». Для нього необхідний екземпляр поточного користувача, який не є екземпляром об’єктом поточної feature. Тому, ми виконуємо імпорт selectAll feature нашого селектора, та селектор для отримання поточного користувача, і створюємо інший із його використанням.

```
export const selectFeatureItemsForCurrentUser = (companyId: number) => createSelector(selectAllFeatureItems,
  UserCompanyCurrentSelectors.selectFeatureItemsForCurrentUser,
  (allUserFeatureItems: CompanyRoles[], userCompanyRoles: UserCompanyRoles[]) => {
    if (allUserFeatureItems && userCompanyRoles && userCompanyRoles.length > 0) {
      return allUserFeatureItems.filter(p => p.companyId === companyId &&
        userCompanyRoles.some(r => r.companyRolesId === p.id));
    } else {
      return null;
    }
  });
```

Рисунок 3.27 – Реалізація взаємодії між різними feature

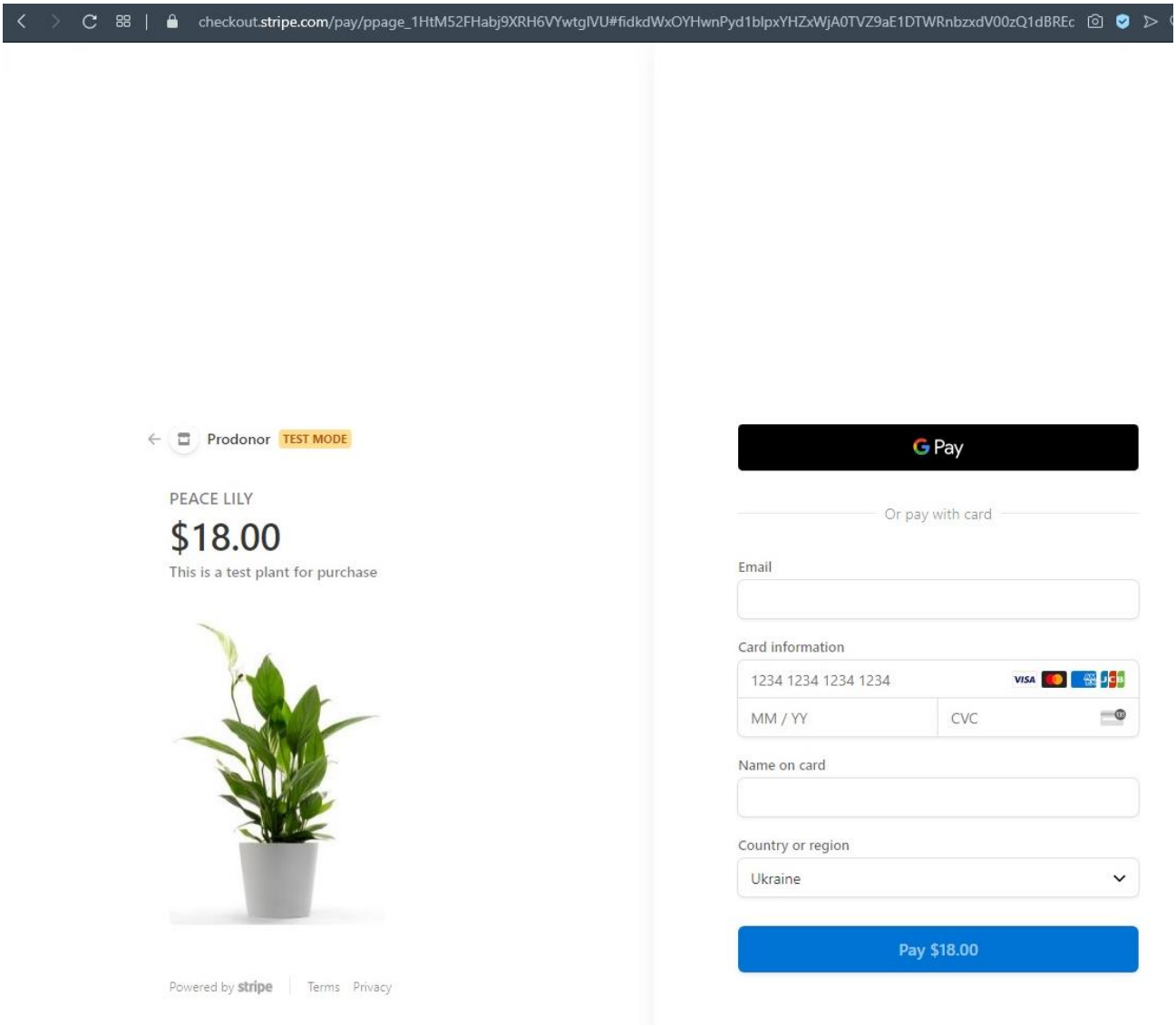

Рисунок 3.28 – Сторінка оплати із використанням stripe

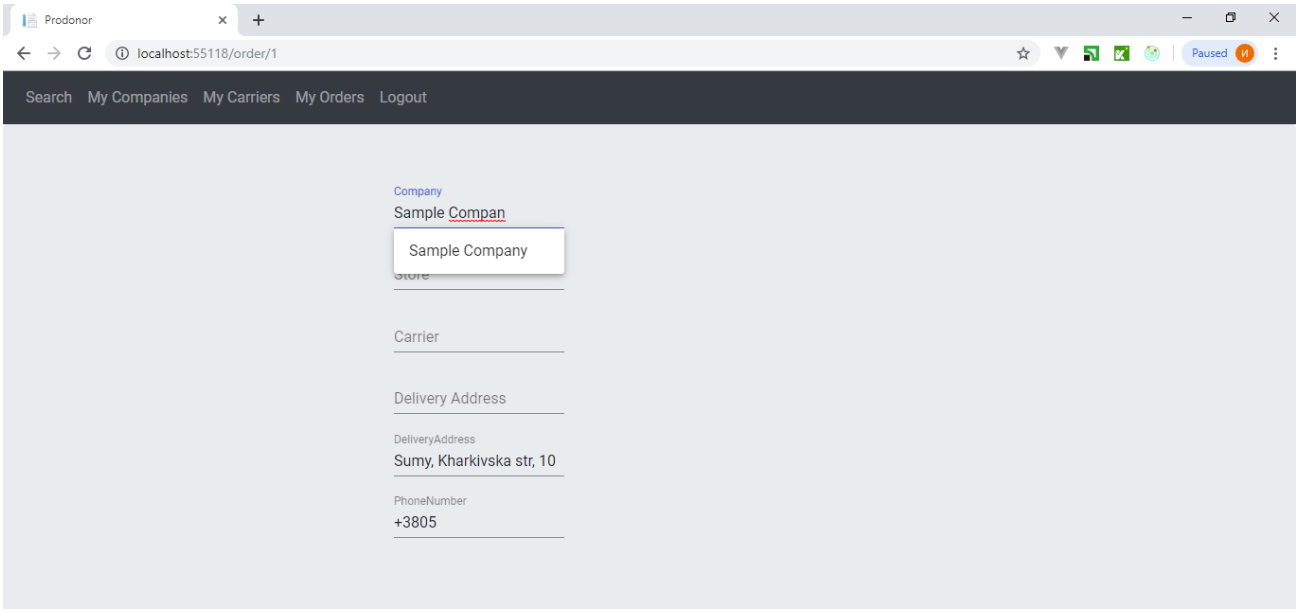

Рисунок 3.29 - Сторінка редагування ордеру

При такій реалізації вдається уникнути «cross-reference» ситуації, що наявна, коли у залежностях кожного із селекторів є інший.

Таким же чином створюємо actions інших модулів. Для їх виклику ми використаємо switchMap оператор(рисунок 3.30).

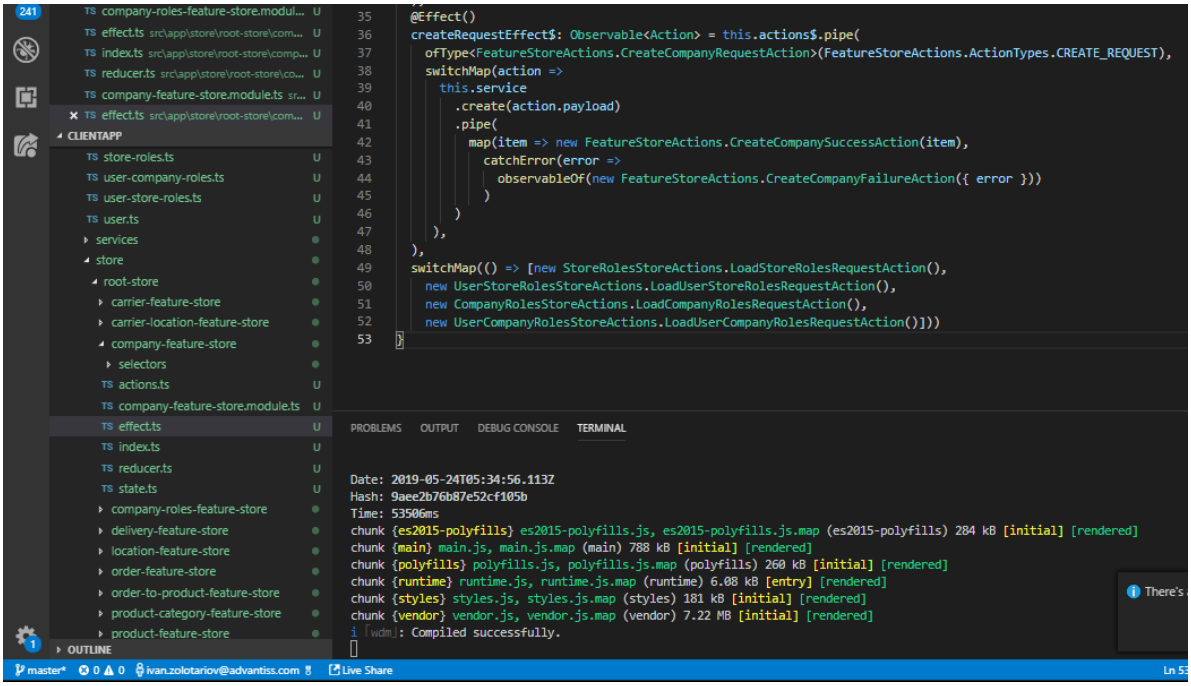

Рисунок 3.30 - Приклад effect, що викликає дії інших модулів

Підсумуємо, що результатом даної роботи є повністю функціонуючий сайт для створення заказів доставки товарів до дверей, який було створено на принципах надання щонайбільшої масштабованості, створенням features для кожної моделі даних та слідуванням ключовим принципам імплементації із використанням бібліотеки NGRX Store.

ВИСНОВКИ

Магістерську роботу можна назвати виконаною, адже її результатом є реалізація всіх завдань що були сформовані перед її початком. Підсумовуючи, можна зробити наступний висновок - бібліотека NGRX потребує значного порогу входу, тож недосвідчені розробники можуть відчути деякі складності на початку знайомства з даною бібліотекою, але при цьому вона надає, необхідні інструменти для створення сучасних, легко масштабованих single page applications ентерпрайз або середнього розміру. При цьому, спочатку може скластися враження, що налаштування потребує значних зусиль, але правильна імплементація з використанням інструментів бібліотеки дозволить не тільки реалізувати проект високої якості з точки зору подальшої підтримки, але і зекономити багато часу у подальшому. Тож результатом проекту є сучасна, високо-масштабована інформаційна система високого потенціалу, подальшими кроками якої повинні бути інтеграція із картографічними сервісами типу Google Maps API, реалізація пошуку продуктів та сервісів регістрації та авторизації за допомогою сторонніх систем або password-less логіну. В ході роботи опановано нові уміння в веб-розробці single page applications та state management, а також здобутий досвід розробки мовою TypeScript.

Як підсумок, у ході магістерської роботи:

1. Було переглянуто наступні платформи управління станом: NGRX, Vuex та Redux. Складено порівняльні таблиці бібліотек state management. За результатом яких визначено, що NGRX є найкращим варіантом ентерпрайз проектів. При цьому NGRX та Redux надають схожі можливості з керування станом, але NGRX має потужнішу систему контролю побічних дій, та є гарним варіантом при розробці мобільного додатку у майбутньому. Таким чином, обрана розробка із використанням бібліотеки NGRX на платформі Angular.

2. Було створено проект, проведена конфігурація середовища розробки Microsoft Visual Studio Community та Visual Studio Code.

3. Спланований, та втілений у життя проект із використанням сховища state management. Вдалося провести успішну реєстрацію та вхід користувача, коли ми могли спостерігати історію змін state за допомогою додатку Redux-dev-tools, де можна було впевнитися у правильній роботі системи.

4. Імплементована інформаційна система адресної доставки товарів, користувачі якої можуть або створити компанію із мережи магазинів або «компанію-перевізника» зі списком виконаних замовлень, а також робити заплановані закази до-дверей із вибором магазинів та перевізників.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. What are good JavaScript frameworks to design a single page application for both mobile and desktop? [Електронний ресурс] // Kuldeep Thakre – Режим доступу до ресурсу: https://www.quora.com/What-are-good-JavaScript-frameworks-to-design-a-single-page-application-for-both-mobile-and-desktop.

2. Angular in Action 1st Edition / Jeremy Wilken, 2018. – 320 с.

3. Angular. Architecture overview. [Електронний ресурс] – Режим доступу до ресурсу: https://angular.io/guide/architecture.

4. GetInstanse.info [Електронний ресурс] // – Режим доступу до ресурсу: https://getinstance.info/articles/react/learning-react-redux.

5. Управляем состоянием приложения с помощью Vuex [Електронний ресурс] // sfi0zy – Режим доступу до ресурсу: https://habr.com/ru/post/322354/.

6. Что такое Angular и зачем его все учат? [Електронний ресурс] // Kate Kucherenko – Режим доступу до ресурсу: https://agilie.com/ru/blog/chto-takoie-angular-i-zachiem-iegho-vsie-uchat.

7. Serverless Single Page Apps: Fast, Scalable, and Available / Ben Rady, 2016. – 214 с. – (Pragmatic Bookshelf).

8. Andrea D. V. Mastering Zabbix / D. V. Andrea, K. L. Stefano. – Birmingham: Packt Publishing, 2013. – 337 с. – (Packt Publishing Ltd.).

9. TypeScript. Official site. [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org/index.html.

10. Справочник современных концепций JavaScript: часть 1 [Електронний ресурс]/ Roman Ponomarev – Режим доступу до ресурсу: https://medium.com/devschacht/glossary-of-modern-javascript-concepts-1198b24e8f56.

11. Ngrx Store - An Architecture Guide [Електронний ресурс] – Режим доступу до ресурсу: https://blog.angular-university.io/angular-ngrx-store-and-effects-crash-course/.

12. Реактивные приложения на Angular/NGRX. Часть 1. Введение. [Електронний ресурс] // Игорь Демянюк – Режим доступу до ресурсу: https://medium.com/@demyanyuk/реактивные-приложения-на-angular-ngrx-часть-1-cb7b4f2852dc.

13. Angular Ngrx DevTools: Important Practical Tips [Електронний ресурс] // – Режим доступу до ресурсу: https://blog.angular-university.io/angular-ngrx-devtools/.

14. Architecting Angular Applications with Redux, RxJS, and NgRx: Learn to build Redux style high-performing applications with Angular 6 / Christoffer Noring, 2018. – 366c. – (Packt Publishing).

15. Reactive Programming with Angular and ngrx: Learn to Harness the Power of Reactive Programming with RxJS and ngrx Extensions 1st ed. Edition / Oren Farhi, – 2017. – 148c. – (Apress).

16. Visual Studio Code Distilled: Evolved Code Editing for Windows, macOS, and Linux / Alessandro Del Sole, – 2018. – 232c. – (Apress).

17. Professional C# 7 and .NET Core 2.0 7th Edition / Christian Nagel, - 2018. – 1440c. – (Wrox).

18. Patterns, Principles, and Practices of Domain-Driven Design/ Scott Millett, Nick Tune, 2015. – 800c. – (Wrox).

19. Інформаційна система адресної доставки товарів з використанням NGRX / Іван Золотарьов, - 2018. - 71с.

20. SignalR 2. [Електронний ресурс] – Режим доступу до ресурсу: https://metanit.com/sharp/mvc5/16.1.php.

ДОДАТКИ

Додаток А Лістинг Login Component

login.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { AuthenticationService } from 'src/app/services/authentication/authentication.service';
import {
    RootStoreState,
    UserFeatureStoreActions,
    UserFeatureStoreSelectors,
  } from '../../store/root-store';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { StoreHelperService } from 'src/app/services/storehelper/storehelper.service';
import { take } from 'rxjs/operators';
import { User } from 'src/app/models/user';

@Component({templateUrl: 'login.component.html'})
export class LoginComponent implements OnInit {
    loginForm: FormGroup;
    loading$: Observable<boolean>;
    submitted = false;
    previousUser: User;

    constructor(
        private storeHelper: StoreHelperService,
        private store$: Store<RootStoreState.State>,
        private formBuilder: FormBuilder,
        private router: Router,
        private authenticationService: AuthenticationService,
    ) {
        if (this.authenticationService.currentUserValue) {
            this.router.navigate(['/']);
        }
    }

    ngOnInit() {
        this.loginForm = this.formBuilder.group({
            username: ['', Validators.required],
            password: ['', Validators.required]
        });

        this.loading$ = this.store$.select(UserFeatureStoreSelectors.selectFeatureIsLoading);
```

```
this.store$.select(UserFeatureStoreSelectors.getUserFeatureCurrentUser).pipe(take(1)).subscribe(u
                                                    ser => {
        if(user && this.previousUser == null) {
            this.storeHelper.initialiseStoreData();
        }
        this.previousUser = user;
    });
  }

  get f() { return this.loginForm.controls; }

  onSubmit() {
    this.submitted = true;

    if (this.loginForm.invalid) {
        return;
    }

    let loginData = {
        username: this.f.username.value,
        password: this.f.password.value,
    }
    this.store$.dispatch(new UserFeatureStoreActions.LoginUserAction({loginData: loginData,
callback: function() {
        this.storeHelper.initialiseStoreData();
    }.bind(this)}));
  }
}
```

login.component.html

```
<h2>Login</h2>
<form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
  <div class="form-group">
    <label for="username">Username</label>
    <input type="text" formControlName="username" class="form-control" [ngClass]="{ 'is-
invalid': submitted && f.username.errors }" />
    <div *ngIf="submitted && f.username.errors" class="invalid-feedback">
      <div *ngIf="f.username.errors.required">Username is required</div>
    </div>
  </div>
  <div class="form-group">
    <label for="password">Password</label>
    <input type="password" formControlName="password" class="form-control" [ngClass]="{
'is-invalid': submitted && f.password.errors }" />
    <div *ngIf="submitted && f.password.errors" class="invalid-feedback">
      <div *ngIf="f.password.errors.required">Password is required</div>
    </div>
  </div>
  <div class="form-group">
    <button [disabled]="(loading$ | async) === true" class="btn btn-primary">Login</button>
    <img *ngIf="(loading$ | async) === true" class="pl-3"
src="
C05FVFNDQVBFMi4wAwEAAAAh/hpDcmVhdGVkIHdpdGggYWpheGxvYWQuaW5mbwAh+QQJ
CgAAACwAAAAAEAAQAAADMwi63P4wyklrE2MIOggZnAdOmGYJRbExwroUmcG2LmDEwnHQ
LVsYOd2mBzkYDAdKa+dIAAAh+QQJCgAAACwAAAAAEAAQAAADNAi63P5OjCEgG4QMu7Dm
ikRxQlFUYDEZIGBMRVsaqHwctXXf7WEYB4Ag1xjihkMZsiUkKhIAIfkECQoAAAAsAAAAABAAEA
AAAzYIujIjK8pByJDMlFYvBoVjHA70GU7xSUJhmKtwHPAKzLO9HMaoKwJZ7Rf8AYPDDzKpZBq
fvwQAIfkECQoAAAAsAAAAABAAEAAAAzMIumIlK8oyhpHsnFZfhYumCYUhDAQxRIdhHBGqRoK
w0R8DYlJd8z0fMDgsGo/IpHI5TAAAIfkECQoAAAAsAAAAABAAEAAAAzIIunInK0rnZBTwGPNMg
QwmdsNgXGJUlIWEuR5oWUIpz8pAEAMe6TwfwyYsGo/IpFKSAAAh+QQJCgAAACwAAAAAEAA
QAAADMwi6IMKQORfjdOe82p4wGccc4CEuQradylesojEMBgsUc2G7sDX3lQGBMLAJibufbSlKA
AAh+QQJCgAAACwAAAAAEAAQAAADMgi63P7wCRHZnFVdmgHu2nFwlWCI3WGc3TSWhUFG
xTAUkGCbtgENBMJAEJsxgMLWzpEAACH5BAkKAAAALAAAAAAQABAAAAMyCLrc/jDKSatlQtS
cKdceCAjDII7HcQ4EMTCpyrCuUBjCYRgHVtqlAiB1YhiCnlsRkAAAOwAAAAAAAAAAA==" />
    <a routerLink="/register" class="btn btn-link">Register</a>
  </div>
</form>
```

Додаток Б Лістинг Register Component

register.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { first } from 'rxjs/operators';
import { AuthenticationService } from 'src/app/services/authentication/authentication.service';
import { UserService } from 'src/app/services/user/user.service';
import { AlertService } from 'src/app/services/alert/alert.service';
import { Observable } from 'rxjs';
import {
    RootStoreState,
    UserFeatureStoreActions,
    UserFeatureStoreSelectors,
  } from '../../store/root-store';
import { Store } from '@ngrx/store';

@Component({templateUrl: 'register.component.html'})
export class RegisterComponent implements OnInit {
    registerForm: FormGroup;
    loading$: Observable<boolean>;
    submitted = false;

    constructor(
        private store$: Store<RootStoreState.State>,
        private formBuilder: FormBuilder,
        private router: Router,
        private authenticationService: AuthenticationService,
        private userService: UserService,
        private alertService: AlertService
    ) {
        // redirect to home if already logged in
        if (this.authenticationService.currentUserValue) {
            this.router.navigate(['/']);
        }
    }

    ngOnInit() {
        this.registerForm = this.formBuilder.group({
            firstName: ['', Validators.required],
            lastName: ['', Validators.required],
            username: ['', Validators.required],
            password: ['', [Validators.required, Validators.minLength(6)]]
        });
        this.loading$ = this.store$.select(UserFeatureStoreSelectors.selectFeatureIsLoading);
    }
```

```
    // convenience getter for easy access to form fields
    get f() { return this.registerForm.controls; }

    onSubmit() {
        this.submitted = true;

        // stop here if form is invalid
        if (this.registerForm.invalid) {
            return;
        }

        this.store$.dispatch(new
UserFeatureStoreActions.RegisterUserAction(this.registerForm.value));
    }
}
```

register.component.html

```
<h2>Register</h2>
<form [formGroup]="registerForm" (ngSubmit)="onSubmit()">
    <div class="form-group">
        <label for="firstName">First Name</label>
        <input type="text" formControlName="firstName" class="form-control" [ngClass]="{ 'is-
invalid': submitted && f.firstName.errors }" />
        <div *ngIf="submitted && f.firstName.errors" class="invalid-feedback">
            <div *ngIf="f.firstName.errors.required">First Name is required</div>
        </div>
    </div>
    <div class="form-group">
        <label for="lastName">Last Name</label>
        <input type="text" formControlName="lastName" class="form-control" [ngClass]="{ 'is-
invalid': submitted && f.lastName.errors }" />
        <div *ngIf="submitted && f.lastName.errors" class="invalid-feedback">
            <div *ngIf="f.lastName.errors.required">Last Name is required</div>
        </div>
    </div>
    <div class="form-group">
        <label for="username">Username</label>
        <input type="text" formControlName="username" class="form-control" [ngClass]="{ 'is-
invalid': submitted && f.username.errors }" />
        <div *ngIf="submitted && f.username.errors" class="invalid-feedback">
            <div *ngIf="f.username.errors.required">Username is required</div>
        </div>
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input type="password" formControlName="password" class="form-control" [ngClass]="{
'is-invalid': submitted && f.password.errors }" />
        <div *ngIf="submitted && f.password.errors" class="invalid-feedback">
            <div *ngIf="f.password.errors.required">Password is required</div>
```

```
<div *ngIf="f.password.errors.minlength">Password must be at least 6 characters</div>
      </div>
    </div>
    <div class="form-group">
      <button [disabled]="loading$ | async" class="btn btn-primary">Register</button>
      <img *ngIf="loading$ | async" class="pl-3"
src="
C05FVFNDQVBFMi4wAwEAAAAh/hpDcmVhdGVkIHdpdGggYWpheGxvYWQuaW5mbwAh+QQJ
CgAAACwAAAAAEAAQAAADMwi63P4wyklrE2MIOggZnAdOmGYJRbExwroUmcG2LmDEwnHQ
LVsYOd2mBzkYDAdKa+dIAAAh+QQJCgAAACwAAAAAEAAQAAADNAi63P5OjCEgG4QMu7Dm
ikRxQlFUYDEZIGBMRVsaqHwctXXf7WEYB4Ag1xjihkMZsiUkKhIAIfkECQoAAAAsAAAAABAAEA
AAAzYIujIjK8pByJDMlFYvBoVjHA70GU7xSUJhmKtwHPAKzLO9HMaoKwJZ7Rf8AYPDDzKpZBq
fvwQAIfkECQoAAAAsAAAAABAAEAAAAzMIumIlK8oyhpHsnFZfhYumCYUhDAQxRIdhHBGqRoK
w0R8DYlJd8z0fMDgsGo/IpHI5TAAAIfkECQoAAAAsAAAAABAAEAAAAzIIunInK0rnZBTwGPNMg
QwmdsNgXGJUlIWEuR5oWUIpz8pAEAMe6TwfwyYsGo/IpFKSAAAh+QQJCgAAACwAAAAAEAA
QAAADMwi6IMKQORfjdOe82p4wGccc4CEuQradylesojEMBgsUc2G7sDX3lQGBMLAJibufbSlKA
AAh+QQJCgAAACwAAAAAEAAQAAADMgi63P7wCRHZnFVdmgHu2nFwlWCI3WGc3TSWhUFG
xTAUkGCbtgENBMJAEJsxgMLWzpEAACH5BAkKAAAALAAAAAAQABAAAAMyCLrc/jDKSatlQtS
cKdceCAjDII7HcQ4EMTCpyrCuUBjCYRgHVtqlAiB1YhiCnlsRkAAAOwAAAAAAAAAAA==" />
      <a routerLink="/login" class="btn btn-link">Cancel</a>
    </div>
</form>
```

Додаток В Лістинг Alert Component

alert.component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';
import { AlertService } from 'src/app/services/alert/alert.service';


@Component({
   selector: 'alert',
   templateUrl: 'alert.component.html'
})


export class AlertComponent implements OnInit, OnDestroy {
   private subscription: Subscription;
   message: any;


   constructor(private alertService: AlertService,
      private toastr: ToastrService) { }


   ngOnInit() {
      this.subscription = this.alertService.getMessageObservable().subscribe(message => {
         setTimeout(() => {
            if(message && message.type === 'success')
               this.toastr.success('Action succeeded!', message.text)
            else if(message && message.type === 'error')
               this.toastr.error('Action failed!', message.text)
         })
      });
   }


   ngOnDestroy() {
```

```
this.subscription.unsubscribe();
    }
}
```

alert.component.html

```
<div *ngIf="message" [ngClass]="{ 'alert': message, 'alert-success': message.type
=== 'success', 'alert-danger': message.type === 'error' }">{{message.text}}</div>
```

Додаток Г Лістинг RootStore елементів

index.ts

```
import { RootStoreModule } from './root-store.module';
import * as RootStoreSelectors from './selectors';
import * as RootStoreState from './root-state';
export * from './user-feature-store';
export * from './carrier-feature-store';
export * from './carrier-location-feature-store';
export * from './company-feature-store';
export * from './company-roles-feature-store';
export * from './delivery-feature-store';
export * from './location-feature-store';
export * from './order-feature-store';
export * from './product-category-feature-store';
export * from './product-feature-store';
export * from './store-feature-store';
export * from './store-roles-feature-store';
export * from './user-company-roles-feature-store';
export * from './user-store-roles-feature-store';
export * from './role-feature-store';
export * from './order-to-product-feature-store';
export { RootStoreState, RootStoreSelectors, RootStoreModule };
```

root-state.ts

```
import { UserFeatureStoreState } from './user-feature-store';
import { CarrierFeatureStoreState  } from './carrier-feature-store';
import { CarrierLocationFeatureStoreState } from './carrier-location-feature-store';
import { CompanyFeatureStoreState } from './company-feature-store';
import { CompanyRolesFeatureStoreState } from './company-roles-feature-store';
import { DeliveryFeatureStoreState } from './delivery-feature-store';
import { LocationFeatureStoreState } from './location-feature-store';
import { OrderFeatureStoreState } from './order-feature-store';
import { ProductCategoryFeatureStoreState } from './product-category-feature-store';
import { ProductFeatureStoreState } from './product-feature-store';
import { StoreFeatureStoreState } from './store-feature-store';
import { StoreRolesFeatureStoreState } from './store-roles-feature-store';
import { UserCompanyRolesFeatureStoreState } from './user-company-roles-feature-store';
import { UserStoreRolesFeatureStoreState } from './user-store-roles-feature-store';
import { RoleFeatureStoreState } from './role-feature-store';
import { OrderToProductFeatureStoreState } from './order-to-product-feature-store';
export interface State {
  userFeature: UserFeatureStoreState.State;
  carrierFeature: CarrierFeatureStoreState.State,
  carrierLocationFeature: CarrierLocationFeatureStoreState.State,
  companyFeature: CompanyFeatureStoreState.State,
  companyRolesFeature: CompanyRolesFeatureStoreState.State,
```

```
deliveryFeature: DeliveryFeatureStoreState.State,
  locationFeature: LocationFeatureStoreState.State,
  orderFeature: OrderFeatureStoreState.State,
  productCategoryFeature: ProductCategoryFeatureStoreState.State,
  productFeature: ProductFeatureStoreState.State,
  storeFeature: StoreFeatureStoreState.State,
  storeRolesFeature: StoreRolesFeatureStoreState.State,
  userCompanyRolesFeature: UserCompanyRolesFeatureStoreState.State,
  userStoreRolesFeature: UserStoreRolesFeatureStoreState.State,
  roleFeature: RoleFeatureStoreState.State,
  orderToProductFeature: OrderToProductFeatureStoreState.State
}
```

root-store-module.ts

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { EffectsModule } from '@ngrx/effects';
import { StoreModule } from '@ngrx/store';
import { UserFeatureStoreModule } from './user-feature-store/';
import { CarrierFeatureStoreModule } from './carrier-feature-store';
import { CarrierLocationFeatureStoreModule } from './carrier-location-feature-store';
import { CompanyFeatureStoreModule } from './company-feature-store';
import { CompanyRolesFeatureStoreModule } from './company-roles-feature-store';
import { DeliveryFeatureStoreModule } from './delivery-feature-store';
import { LocationFeatureStoreModule } from './location-feature-store';
import { OrderFeatureStoreModule } from './order-feature-store';
import { ProductCategoryFeatureStoreModule } from './product-category-feature-store';
import { ProductFeatureStoreModule } from './product-feature-store';
import { StoreFeatureStoreModule } from './store-feature-store';
import { StoreRolesFeatureStoreModule } from './store-roles-feature-store';
import { UserCompanyRolesFeatureStoreModule } from './user-company-roles-feature-store';
import { UserStoreRolesFeatureStoreModule } from './user-store-roles-feature-store';
import { RoleFeatureStoreModule } from './role-feature-store';
import { OrderToProductFeatureStoreModule } from './order-to-product-feature-store';
import { StoreDevtoolsModule } from "@ngrx/store-devtools"
@NgModule({
  imports: [
    CommonModule,
    UserFeatureStoreModule,
    CarrierFeatureStoreModule,
    CarrierLocationFeatureStoreModule,
    CompanyFeatureStoreModule,
    CompanyRolesFeatureStoreModule,
    DeliveryFeatureStoreModule,
    LocationFeatureStoreModule,
    OrderFeatureStoreModule,
    ProductCategoryFeatureStoreModule,
    ProductFeatureStoreModule,
```

```
StoreFeatureStoreModule,
    StoreRolesFeatureStoreModule,
    UserCompanyRolesFeatureStoreModule,
    UserStoreRolesFeatureStoreModule,
    RoleFeatureStoreModule,
    OrderToProductFeatureStoreModule,
    StoreModule.forRoot({}),
    EffectsModule.forRoot([]),
    StoreDevtoolsModule.instrument()
  ],
  declarations: []
})
export class RootStoreModule {}
```

selectors.ts

```
import { createSelector, MemoizedSelector } from '@ngrx/store';
import { UserFeatureStoreSelectors} from './user-feature-store';
import { CarrierFeatureStoreSelectors} from './carrier-feature-store';
import { CarrierLocationFeatureStoreSelectors} from './carrier-location-feature-store';
import { CompanyFeatureStoreSelectors } from './company-feature-store';
import { CompanyRolesFeatureStoreSelectors } from './company-roles-feature-store';
import { DeliveryFeatureStoreSelectors } from './delivery-feature-store';
import { LocationFeatureStoreSelectors } from './location-feature-store';
import { OrderFeatureStoreSelectors } from './order-feature-store';
import { ProductCategoryFeatureStoreSelectors } from './product-category-feature-store';
import { ProductFeatureStoreSelectors } from './product-feature-store';
import { StoreFeatureStoreSelectors } from './store-feature-store';
import { StoreRolesFeatureStoreSelectors } from './store-roles-feature-store';
import { UserCompanyRolesFeatureStoreSelectors } from './user-company-roles-feature-store';
import { UserStoreRolesFeatureStoreSelectors } from './user-store-roles-feature-store';
import { RoleFeatureStoreSelectors } from './role-feature-store';
import { OrderToProductFeatureStoreSelectors } from './order-to-product-feature-store';

export const selectErrorFirst = createSelector(
  UserFeatureStoreSelectors.selectFeatureError,
  CarrierFeatureStoreSelectors.selectFeatureError,
  CarrierLocationFeatureStoreSelectors.selectFeatureError,
  CompanyFeatureStoreSelectors.selectFeatureError,
  CompanyRolesFeatureStoreSelectors.selectFeatureError,
  DeliveryFeatureStoreSelectors.selectFeatureError,
  RoleFeatureStoreSelectors.selectFeatureError,
  OrderToProductFeatureStoreSelectors.selectFeatureError,
  (userFeatureError: string,
    carrierFeatureError: string,
    carrierLocationFeatureError: string,
    companyFeatureError: string,
    companyRolesFeatureError: string,
```

```
deliveryFeatureError: string,
   roleFeatureError: string,
   orderToProductFeatureError: string,
  ) => {
   return userFeatureError || carrierFeatureError || carrierLocationFeatureError
   || companyFeatureError || companyRolesFeatureError || deliveryFeatureError ||
roleFeatureError || orderToProductFeatureError;
  }
  );

export const selectErrorSecond = createSelector(
  LocationFeatureStoreSelectors.selectFeatureError,
  OrderFeatureStoreSelectors.selectFeatureError,
  ProductCategoryFeatureStoreSelectors.selectFeatureError,
  ProductFeatureStoreSelectors.selectFeatureError,
  StoreFeatureStoreSelectors.selectFeatureError,
  StoreRolesFeatureStoreSelectors.selectFeatureError,
  UserCompanyRolesFeatureStoreSelectors.selectFeatureError,
  UserStoreRolesFeatureStoreSelectors.selectFeatureError,
  (locationFeatureError: string,
   orderFeatureError: string,
   productCategoryFeatureError: string,
   productFeatureError: string,
   storeFeatureError: string,
   storeRolesFeatureError: string,
   userCompanyRolesFeatureError: string,
   userStoreRolesFeatureError: string
   ) => {
   return locationFeatureError || orderFeatureError || productCategoryFeatureError
   || productFeatureError || storeFeatureError || storeRolesFeatureError
   || userCompanyRolesFeatureError || userStoreRolesFeatureError;
  });


export const selectFirstIsLoading: MemoizedSelector<object,boolean> = createSelector(
  UserFeatureStoreSelectors.selectFeatureIsLoading,
  CarrierFeatureStoreSelectors.selectFeatureIsLoading,
  CarrierLocationFeatureStoreSelectors.selectFeatureIsLoading,
  CompanyFeatureStoreSelectors.selectFeatureIsLoading,
  CompanyRolesFeatureStoreSelectors.selectFeatureIsLoading,
  DeliveryFeatureStoreSelectors.selectFeatureIsLoading,
  RoleFeatureStoreSelectors.selectFeatureIsLoading,
  OrderToProductFeatureStoreSelectors.selectFeatureIsLoading,
  (userFeatureLoading,
   carrierLoading,
   carrierLocationFeatureIsLoading,
   companyLoading,
   companyRolesLoading,
   deliveryLoading,
```

```
roleLoading,
   orderToProductLoading
   ) => {
   return userFeatureLoading || carrierLoading || carrierLocationFeatureIsLoading
   || companyLoading || companyRolesLoading || deliveryLoading || roleLoading ||
orderToProductLoading;
  });


export const selectSecondIsLoading: MemoizedSelector<object,boolean> = createSelector(
  LocationFeatureStoreSelectors.selectFeatureIsLoading,
  OrderFeatureStoreSelectors.selectFeatureIsLoading,
  ProductCategoryFeatureStoreSelectors.selectFeatureIsLoading,
  ProductFeatureStoreSelectors.selectFeatureIsLoading,
  StoreFeatureStoreSelectors.selectFeatureIsLoading,
  StoreRolesFeatureStoreSelectors.selectFeatureIsLoading,
  UserCompanyRolesFeatureStoreSelectors.selectFeatureIsLoading,
  UserStoreRolesFeatureStoreSelectors.selectFeatureIsLoading,
  (locationFeatureLoading,
   orderFeatureLoading,
   productCategoryFeatureLoading,
   productFeatureLoading,
   storeFeatureLoading,
   storeRolesFeatureLoading,
   userCompanyRolesFeatureLoading,
   userStoreRolesFeatureLoading
   ) => {
   return locationFeatureLoading || orderFeatureLoading || productCategoryFeatureLoading
   || productFeatureLoading || storeFeatureLoading || storeRolesFeatureLoading
   || userCompanyRolesFeatureLoading || userStoreRolesFeatureLoading;
  });

  export const selectError: MemoizedSelector<object, string> = createSelector(
   selectErrorFirst,
   selectErrorSecond,
   (errorFirst: string, errorSecond: string) => {
    return errorFirst || errorSecond;
   }
  );

  export const selectIsLoading: MemoizedSelector<object,boolean> = createSelector(
   selectFirstIsLoading,
   selectSecondIsLoading,
   (firstLoading,
    secondLoading
    ) => {
    return firstLoading || secondLoading;
   });
```

Додаток Д Лістинг UserFeatureStore

index.ts

```
import * as UserFeatureStoreActions from './actions';
import * as UserFeatureStoreSelectors from './selectors/selectors';
import * as UserFeatureStoreState from './state';
export {
  UserFeatureStoreModule
} from './user-feature-store.module';
export {
  UserFeatureStoreActions,
  UserFeatureStoreSelectors,
  UserFeatureStoreState
};
```

selectors.ts

```
import {
  createFeatureSelector,
  createSelector,
  MemoizedSelector
} from '@ngrx/store';
import { featureAdapter, State } from '../state';
import { User } from 'src/app/models/user';
export const getError = (state: State): any => state.error;
export const getIsLoading = (state: State): boolean => state && state.isLoading;
export const selectFeatureState: MemoizedSelector<object, State> =
createFeatureSelector<State>('userFeature');

export const selectAllFeatureItems: (state: object) => User[] =
featureAdapter.getSelectors(selectFeatureState).selectAll;
export const selectFeatureById = (id: number) =>
```

```
createSelector(selectAllFeatureItems, (allUserFeatureItems: User[]) => {
   if (allUserFeatureItems) {
     return allUserFeatureItems.find(p => p.id === id);
   } else {
     return null;
   }
 });
export const getCurrentUser = (state: State): User => {
  if(state)
   return state.currentUser;
}
export const selectFeatureError: MemoizedSelector<object, any> = createSelector(
  selectFeatureState,
  getError
);
export const selectFeatureIsLoading = createSelector(selectFeatureState, getIsLoading);
export const getUserFeatureCurrentUser = createSelector(selectFeatureState, getCurrentUser);
```

effect.ts

```
import { Injectable } from '@angular/core';
import { Actions, Effect, ofType } from '@ngrx/effects';
import { Action } from '@ngrx/store';
import { Observable, of as observableOf } from 'rxjs';
import { catchError, map, switchMap } from 'rxjs/operators';
import { UserService } from 'src/app/services/user/user.service';
import * as UserFeatureStoreActions from './actions';
import { AuthenticationService } from 'src/app/services/authentication/authentication.service';
import { AlertService } from 'src/app/services/alert/alert.service';
import { Router, ActivatedRoute } from '@angular/router';
```

```
  .pipe(
        map(item => {
          action.payload.callback();
          this.router.navigate([this.route.snapshot.queryParams['returnUrl'] || '/']);
          return new UserFeatureStoreActions.SetCurrentUserSuccessAction(item);
        }),
        catchError(error => {
          this.alertService.error(error);
          return observableOf(new UserFeatureStoreActions.LoginUserActionFailure({ error }));
        })
      )
    )
  );


        @Effect()
  logoutUserEffect$: Observable<Action> = this.actions$.pipe(
ofType<UserFeatureStoreActions.LogoutUserAction>(UserFeatureStoreActions.ActionTypes.LOG
OUT_USER),
    switchMap(action =>
      this.authService
        .logout()
        .pipe(
          map(item => {
            this.router.navigate(['/login']);
            return new UserFeatureStoreActions.SetCurrentUserSuccessAction(null);
          }),
          catchError(error => {
            this.alertService.error(error);
            return observableOf(new UserFeatureStoreActions.LoginUserActionFailure({ error }));
          })
```

```
  )));


        @Effect()
 registerUserEffect$: Observable<Action> = this.actions$.pipe(

ofType<UserFeatureStoreActions.RegisterUserAction>(UserFeatureStoreActions.ActionTypes.REG
ISTER_USER),
   switchMap(action =>
    this.userService
     .register(action.payload)
     .pipe(
      map(item => {
        this.alertService.success('Registration successful', true);
        this.router.navigate(['/login']);
        return new UserFeatureStoreActions.LoadUsersRequestAction();
       }),
       catchError(error => {
        this.alertService.error(error);
        return observableOf(new UserFeatureStoreActions.RegisterUserActionFailure({ error }));
       })
     )
   )
 );
}
```

reducer.ts

```
import { Actions, ActionTypes } from './actions';
import { featureAdapter, initialState, State } from './state';
export function featureReducer(state = initialState, action: Actions): State {
 switch (action.type) {
   case ActionTypes.LOAD_REQUEST: {
return {
```

```
    ...state,
    isLoading: true,
    error: null
  };
}
case ActionTypes.LOAD_SUCCESS: {
  return featureAdapter.addAll(action.payload, {
    ...state,
    isLoading: false,
    error: null
  });
}
case ActionTypes.LOAD_FAILURE: {
  return {
    ...state,
    isLoading: false,
    error: action.payload.error
  };
}
case ActionTypes.SET_CURRENT_USER: {
  return {
    ...state,
    isLoading: true,
    error: null
  };
}
case ActionTypes.SET_CURRENT_USER_SUCCESS: {
  return {
    ...state,
    currentUser: action.payload,
```

```
isLoading: false,
      error: null
    };
  }
  case ActionTypes.SET_CURRENT_USER_FAILURE: {
    return {
      ...state,
      isLoading: false,
      error: action.payload.error
    };
                }
  case ActionTypes.LOGIN_USER: {
    return {
      ...state,
      isLoading: true,
      error: null
    };
  }
  case ActionTypes.LOGIN_USER_FAILURE: {
    return {
      ...state,
      isLoading: false,
      error: action.payload.error
    };
  }
  case ActionTypes.LOGOUT_USER: {
    return {
      ...state,
      isLoading: true,
      error: null
```

```
  };
   }
   case ActionTypes.LOGOUT_USER_FAILURE: {
    return {
     ...state,
     isLoading: false,
     error: action.payload.error
    };
   }
   case ActionTypes.REGISTER_USER: {
    return {
     ...state,
     isLoading: true,
     error: null
    };
   }
   case ActionTypes.REGISTER_USER_FAILURE: {
    return {
     ...state,
     isLoading: false,
     error: action.payload.error
    };
   }
   default: {
    return state;
   }
        }
}
```

state.ts

```
import { createEntityAdapter, EntityAdapter, EntityState } from '@ngrx/entity';
import { User } from 'src/app/models/user';
export const featureAdapter: EntityAdapter<User> = createEntityAdapter<User>({
  selectId: model => model.id,
  sortComparer: (a: User, b: User): number =>
    a.username.localeCompare(b.username)
});
export interface State extends EntityState<User> {
  currentUser?: User;
  isLoading?: boolean;
  error?: any;
}
export const initialState: State = featureAdapter.getInitialState(
  {
    currentUser: null,
    isLoading: false,
    error: null
  }
);
```

actions.ts

```
import { Action } from '@ngrx/store';
import { User } from 'src/app/models/user';
export enum ActionTypes {
  LOAD_REQUEST = '[User Feature] Load Users Request',
  LOAD_FAILURE = '[User Feature] Load Users Failure',
  LOAD_SUCCESS = '[User Feature] Load Users Success',
  SET  CURRENT  USER = '[User Feature] Set Current User',
```

```
                    LOGIN_USER = '[User Feature] Login User',
  LOGIN_USER_FAILURE = '[User Feature] Login User Failure',
  LOGOUT_USER = '[User Feature] Logout User',
  LOGOUT_USER_FAILURE = '[User Feature] Logout User Failure',
  REGISTER_USER = '[User Feature] Register User',
  REGISTER_USER_FAILURE = '[User Feature] Register User Failure',
  SET_CURRENT_USER_SUCCESS = '[User Feature] Set Current User Success',
  SET_CURRENT_USER_FAILURE = '[User Feature] Set Current User Failure'
}
export class LoadUsersRequestAction implements Action {
  readonly type = ActionTypes.LOAD_REQUEST;
}
export class LoadUsersFailureAction implements Action {
  readonly type = ActionTypes.LOAD_FAILURE;
  constructor(public payload: { error: string }) {}
}
export class LoadUsersSuccessAction implements Action {
  readonly type = ActionTypes.LOAD_SUCCESS;
  constructor(public payload: User[]) {}
}
export class SetCurrentUserAction implements Action {
  readonly type = ActionTypes.SET_CURRENT_USER;
  constructor(public payload: any) {}
}
export class SetCurrentUserFailureAction implements Action {
  readonly type = ActionTypes.SET_CURRENT_USER_FAILURE;
  constructor(public payload: { error: string }) {}
}
export class SetCurrentUserSuccessAction implements Action {
```

```
    readonly type = ActionTypes.SET_CURRENT_USER_SUCCESS;
  constructor(public payload: User) {}
}
export class LoginUserAction implements Action {
  readonly type = ActionTypes.LOGIN_USER;
  constructor(public payload: any) {}
}
export class LoginUserActionFailure implements Action {
  readonly type = ActionTypes.LOGIN_USER_FAILURE;
  constructor(public payload: any) {}
}
export class LogoutUserAction implements Action {
  readonly type = ActionTypes.LOGOUT_USER;
}
export class LogoutUserActionFailure implements Action {
  readonly type = ActionTypes.LOGOUT_USER_FAILURE;
  constructor(public payload: any) {}
}
export class RegisterUserAction implements Action {
  readonly type = ActionTypes.REGISTER_USER;
  constructor(public payload: any) {}
}
export class RegisterUserActionFailure implements Action {
  readonly type = ActionTypes.REGISTER_USER_FAILURE;
  constructor(public payload: any) {}
}

export type Actions = LoadUsersRequestAction | LoadUsersFailureAction |
LoadUsersSuccessAction
| SetCurrentUserAction | SetCurrentUserFailureAction | SetCurrentUserSuccessAction |
```

```
LoginUserAction | LoginUserActionFailure | RegisterUserAction | RegisterUserActionFailure
| LogoutUserAction | LogoutUserActionFailure;
```

user-feature-store-module.ts

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { EffectsModule } from '@ngrx/effects';
import { StoreModule } from '@ngrx/store';
import { UserFeatureStoreEffects } from './effect';
import { featureReducer } from './reducer';
@NgModule({
  imports: [
    CommonModule,
    StoreModule.forFeature('userFeature', featureReducer),
    EffectsModule.forFeature([UserFeatureStoreEffects])
  ],
  providers: [UserFeatureStoreEffects]
})
export class UserFeatureStoreModule {}
```

Додаток Е Лістинг SignalrService

```
import { Injectable } from '@angular/core';
import * as signalR from '@aspnet/signalr';
import { BehaviorSubject, Observable } from 'rxjs';
import { AlertService } from '../alert/alert.service';

@Injectable({
 providedIn: 'root'
})
export class SignalrService {
 private hubConnection: signalR.HubConnection;

 constructor(private alertService: AlertService) {
  this.startConnection();
 }

 public startConnection = () => {
  this.hubConnection = new signalR.HubConnectionBuilder()
                    .withUrl('http://localhost:14031', {
                     skipNegotiation: true,
                     transport: signalR.HttpTransportType.WebSockets})
                    .build();
  this.hubConnection
    .start()
    .then(() => console.log('Connection started'))
    .catch(err => console.log('Error while starting connection: ' + err));
 }

 public getMessage = () => {
```

```
this.hubConnection.on('messageReceived', (data) => {
    if(data && data.type == 'error')
      this.alertService.error(data.message, true)
    else if(data && data.type == 'success')
      this.alertService.success(data.message, true)
  });
 }

 public userListererActivation(userName): void {
  if (this.hubConnection.state === signalR.HubConnectionState.Connected){
    this.hubConnection
    .invoke('UserListererActivation', userName)
    .catch(err => console.error('userListererActivation error:', err));
  }
 }
}
```