

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА
АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ВИКОРИСТАННЯ АРХІТЕКТУРНОГО ПІДХОДУ FLUX ДЛЯ
ПОБУДОВИ ВЕБ-ЗАСТОСУНКІВ НА ПРИКЛАДІ БІБЛІОТЕКИ REDUX

Виконав студент 3 курсу
БП «Інженерія програмного забезпечення»
Жулкевський Владислав Дмитрович

Керівник курсової роботи
старший викладач
Вовк Наталя Євгенівна

Київ 2020

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

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

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних
систем, доцент, к.ф.-м.н.

_____ О.П.Жежерун
„_____” _____ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту 3-го курсу, факультету інформатики
Жулкевському Владиславу Дмитровичу

Тема: Використання архітектурного підходу Flux для побудови веб-застосунків на прикладі бібліотеки Redux

Зміст ТЧ до курсової роботи:

Зміст

Анотація

Вступ

- 1 Аналіз архітектури Flux
- 2 Опис можливостей бібліотеки Redux
- 3 Розробка програмного застосунку “Сервіс опитування студентів”

Висновки

Список літератури

Додатки

Дата видачі „_____” _____ 2020 р. Керівник _____
(підпис)

Завдання отримав _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапу курсового проекту (роботи)	Термін виконання
1.	Отримання завдання на курсову роботу	04.10.2019
2.	Огляд літератури за темою роботи	14.10.2019- 25.12.2020
3.	Ініціалізація та розробка каскаду веб-застосунку: “Сервіс опитування студентів про якість викладання університетських курсів”	21.10.2019- 22.12.2020
4.	Написання розділу “Огляд існуючих способів організації веб-застосунків”	03.01.2020- 19.01.2020
5.	Написання розділів “Аналіз архітектури Flux” та “Опис можливостей бібліотеки Redux”	20.01.2020- 17.02.2020
6.	Створення сторінки адміністратора веб-застосунку	13.02.2020- 10.03.2020
7.	Створення сторінок для проходження опитування	11.03.2020- 29.03.2020
8.	Написання розділу “Розробка застосунку з використанням бібліотеки Redux”	30.03.2020- 12.04.2020
9.	Тестування застосунку	13.04.2020- 15.04.2020
10.	Створення презентації	16.04.2020- 17.04.2020
11.	Корегування роботи за результатами попереднього захисту.	18.04.2020- 25.04.2020
12.	Остаточне оформлення пояснювальної роботи та презентації	26.04-05.05.2020
13.	Подання роботи на кафедру для перевірки на плагіат	11.05.2020

14.	Захист курсової роботи	18.05.2020- 29.05.2020
-----	------------------------	---------------------------

ЗМІСТ

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ	2
КАЛЕНДАРНИЙ ПЛАН	3
АНОТАЦІЯ.....	6
ПЕРЕЛІК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ	7
ВСТУП.....	8
1 АНАЛІЗ АРХІТЕКТУРИ FLUX.....	10
1.1 Передумови виникнення підходу	10
1.2 Архітектура Flux.....	11
1.2.1 Диспетчер.....	12
1.2.2 Сховища	13
1.2.3 Дії.....	13
1.2.4 Представлення	14
1.2.5 Потік даних у Flux архітектурі	15
1.3 Переваги Flux.....	16
2 ОПИС МОЖЛИВОСТЕЙ БІБЛІОТЕКИ REDUX	19
2.1 Характеристики рис Redux.....	19
2.2 Бібліотека React Redux для зв'язування React.js із Redux.....	21
3 РОЗРОБКА ПРОГРАМНОГО ЗАСТОСУНКУ “СЕРВІС ОПИТУВАННЯ СТУДЕНТІВ”	25
3.1 Опис предметної області	25
3.2 Ініціалізація сховища Redux.....	25
3.3 Реалізація клієнтської архітектури сервісу на основі підходу Flux	27
ВИСНОВКИ	35
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	36
ДОДАТКИ	37
Додаток А	37
Додаток Б.....	40

АНОТАЦІЯ

У роботі детально розглянуто архітектуру Flux для побудови клієнтських застосунків, її складові компоненти та переваги даного підходу. Для програмної реалізації проекту було обрано бібліотеку Redux, яка реалізує даний тип архітектури. Також продемонстровано відмінності Redux від специфікації Flux та інструменти для використання Redux із React.js. На прикладі окремих компонентів проаналізовано потік даних при використанні досліджуваного підходу для організації архітектури застосунку.

ПЕРЕЛІК ТЕРМІНІВ ТА УМОВНИХ ПОЗНАЧЕНЬ

Фреймворк - це програмне середовище спеціального призначення, своєрідний каркас, який використовується для того, щоб істотно полегшити процес об'єднання певних компонентів при створенні програм.

Синглтон - це шаблон проектування, який гарантує, що певний клас матиме лише один екземпляр, і надає глобальну точку доступу до цього екземпляру.

Колбек, функція-колбек – (функція зворотного виклику) функція, яка викликається після того, як інша функція закінчила виконання.

Токен – унікальний ідентифікатор, представлений у вигляді стрічки.

Відлагодження, дебаг – процес виявлення помилок у програмі.

Хуки – можливості, реалізовані у React з версії 16.8, які дозволяють використовувати стани компонент без написання класів.

Пропс – властивість компонента.

ЗВО – заклад вищої освіти.

Pojo – plain old javascript object – звичайний JavaScript об'єкт.

ВСТУП

Архітектурний підхід Flux використовується у багатьох веб-застосунках для передбачуваного, одностороннього керування станом програми, що виділяє його поміж інших принципів та підходів. Також така бібліотека як Redux, яка була створена на основі цього архітектурного способу організації веб-застосунків, наразі є часто використовувана за даними npm trends[1]. У цій роботі висвітлено основні принципи використання цього підходу на прикладі вищезазначеної бібліотеки для React застосунків, зокрема для розробки клієнтської частини сервісу опитування студентів.

Мета дослідження полягає у ознайомленні із архітектурою Flux, принципами та специфікою цього підходу, а також у застосуванні цього підходу для розробки веб-застосунку. Основними завданнями даного дослідження є:

- висвітлення основних концепцій архітектури Flux;
- опис проблем та рішення, які надає підхід Flux;
- порівняння цього підходу із MVC архітектурою;
- аналіз бібліотек Redux та React Redux;
- створення веб-застосунку з використанням бібліотек React.js та Redux.

Об'єктом дослідження є архітектура Flux, запропонована компанією Facebook для розробки клієнтських веб-застосунків.

У цій роботі дослідження проводилися шляхом аналізу документації Flux та бібліотеки Redux, перегляд статей за темою дослідження та порівняння підходів MVC та Flux. Також було протестовано цей підхід через розробку застосунку на основі висвітлених даних у цій роботі. Створення схем відбувалося з допомогою онлайн-сервісу для побудови діаграм та схем Lucidchart.

До джерел дослідження відносяться такі електронні ресурси як офіційна документація опрацьованих у цій роботі бібліотек та електронні статті з теми дослідження, зазначені у списку використаних джерел.

Робота складається з 3 розділів.

Перший розділ призначено висвітленню та аналізу архітектури Flux та опису проблем, які вирішує даний підхід. Також у розділі з'ясовуються відмінності між цим підходом та підходом MVC.

В другому розділі наведено основні можливості бібліотеки Redux, яка є прикладом застосування досліджуваного підходу для організації веб-застосунків. Показано відмінності специфікації цієї бібліотеки від запропонованих підходом Flux. Розглянуто бібліотеку React Redux для зв'язування React.js та Redux.

У третьому розділі наводиться реалізація клієнтської частини сервісу опитування студентів з використанням Flux підходу. Висвітлюються потоки даних застосунку та описуються ключові аспекти у використанні бібліотеки Redux на практиці.

У результаті виконання дослідження створено програмний продукт, який призначений для створення платформи опитування студентів про якість викладання курсів у їх вищому навчальному закладі.

компонента Page до компонента Table через усі 3 дочірні компоненти, а в компонент FieldWithData потрібно передати функцію, яка присвоює отримані дані полю data компонента Page. Така організація створює надлишкові дані у всіх дочірніх компонентах, через які потрібно передавати ці дані. Окрім цього, розробникам потрібно подбати про передачу змінних та функцій дочірнім елементам, що у свою чергу робить код більш громіздким та важким для сприйняття та підтримки. Якщо знадобиться змінити структуру програми, то розробнику доведеться змінювати код майже у кожному дочірньому компоненті, що буде сповільнювати процес розробки програмного застосунку. І тому чим більше бізнес-логіки буде мати застосунок, тим важче стає реалізовувати той чи інший функціонал даного застосунку.

Таким чином, наведена проблема стала основною передумовою виникнення такої архітектури програмних застосунків як Flux.

1.2 Архітектура Flux

Flux – це архітектура, яка була представлена компанією Facebook у 2014 році. В офіційній документації Flux вказано, що це скоріше шаблон, а не формальний фреймворк, і ви можете почати безпосередньо використовувати Flux без додавання великої кількості нового коду [2]. Дійсно, Flux представляє собою певний шаблон організації потоку даних у програмному застосунку. Цікаво, що Facebook початково створював цей підхід, щоб використовувати його разом із власною бібліотекою React.js, проте, як ми побачимо далі, цей підхід не обмежується цією бібліотекою та може бути використаний разом із іншими бібліотеками та фреймворками. Проте, перед цим, давайте розглянемо складові компоненти архітектури Flux.

Отже, у архітектурі Flux виділяють такі три основні частини: dispatcher (диспетчер), store (сховище) та view (представлення). Також необхідною складовою Flux є поняття дії (action). Усі ці чотири складові забезпечують

однонаправлений потік даних нашого застосунку. На рис. 3 продемонстровано структуру Flux зі вищезазначеними елементами, але для того, щоб зрозуміти як функціонує ця структура загалом, проаналізуємо функції кожного компоненту окремо.

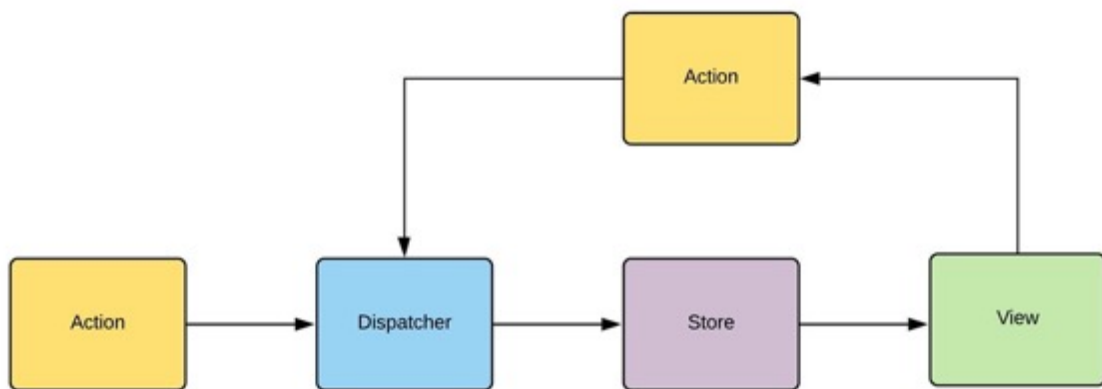


Рис. 2 Спрощена структура потоку даних Flux

1.2.1 Диспетчер

Диспетчер служить центральною ланкою у всьому потоці даних Flux. У загальному випадку він представляє собою синглтон, який містить асоціативний масив пар «ключ-значення». У ролі ключа виступає назва action, а у ролі значення – масив функцій – колбеків. Сам диспетчер реалізує метод `dispatch`, який, отримавши назву дії, синхронно проходиться по усьому масиву функцій, зв'язаних з назвою цієї дії, та викликає їх, передаючи дані, надіслані у об'єкті action, як параметри функції.

Для диспетчера існує п'ять основних методів, які використовуються для налаштування його роботи [2]:

- `register(function callback) : string` – реєструє колбек для виклику при отриманні даних отриманої дії. Повертає токен для використання у методі `waitFor()`;
- `unregister(string id) : void` – видаляє колбек із заданим у параметрах токеном;
- `waitFor(array<string> ids) : void` – чекає виклику та виконання колбеків із токенами, які присутні у масиві `ids`;
- `dispatch(object payload) : void` – передає диспетчеру об’єкт для виклику відповідного колбеку.
- `isDispatching() : boolean` – метод, що вказує чи виконується в даний момент диспетчеризація.

1.2.2 Сховища

Сховища являють собою об’єкти, які містять певний стан програми на конкретний період часу. У стані програми зберігаються дані, які пов’язані із предметною областю нашого застосунку, або містяться дані відображень, які не мають безпосереднього зв’язку із моделлю з предметної області. До прикладу, у сховищі з назвою “menu” можуть зберігатися дані про усі страви ресторану, дані про обрану користувачем страву та стан замовлення цієї страви, який буде висвічуватися у певній області сторінки інтерфейсу [3].

1.2.3 Дії

Дії – це методи, який надсилається диспетчеру здебільшого із представлень, щоб вказати, які зміни потрібно внести до поточного стану застосунку. Цей метод обов’язково має містити певний унікальний ідентифікатор, який диспетчер зможе використати для відповідного виклику пов’язаних функцій. В більшості реалізацій архітектури унікальним полем виступає рядковий літерал, який коротко описує зміну стану. Наприклад, ми можемо визначити action із назвою “FETCH_MEALS”, який буде додавати у

сховище “menu” дані про усі страви ресторану. Також дія може містити у собі дані, які будуть надаватися до функцій-колбеків диспетчера, щоб ті, у свою чергу, використали ці дані для зміни стану застосунку. Повертаючись до попереднього прикладу, коли диспетчер буде отримувати дію, у ній буде міститися окреме поле, назва якого визначається бібліотекою, яка реалізує Flux архітектуру. В цьому полі має міститися інформація, яка оброблюється у представленні та вже в обробленому вигляді відправляється диспетчеру разом з назвою дії.

Доволі часто для створення дій розробники визначають спеціальні функції, які називають генераторами дій (actionCreator). Ці функції приймають на вході деякі параметри, оброблюють їх в тілі функції та на виході повертають відповідні дії, які надсилаються диспетчеру. Таким чином, можна визначати наперед заготовлені функції у окремих файлах, а в представленнях користуватися ними для швидкого створення дій. В іншому випадку, розробник повинен визначити створення цих дій безпосередньо у самому представленні.

1.2.4 Представлення

Представлення у структурі архітектури Flux відіграють важливу роль, бо саме через них відбувається взаємодія з користувачем. У застосунках, зроблених за допомогою компонентно-орієнтованих бібліотек та фреймворків, роль представлень відіграють компоненти. При певних діях користувача представлення будуть генерувати дії для зміни стану застосунку. Після надсилання диспетчеру дії або після спрацювання відповідної функції для генерування дії, представлення може звертатися до сховищ для отримання та відображення оновлених даних в інтерфейсі програми. Такі представлення, які звертаються до сховищ стану програми, прийнято називати представлення-контролери (controller-view). Вони можуть передавати своїм дочірнім елементам отримані дані, щоб не пов’язувати усі компоненти зі сховищами стану програми.

1.2.5 Потік даних у Flux архітектурі

Отже, розглянувши призначення кожної компоненти досліджуваної архітектури окремо, зупинімося докладніше на тому, як відбувається взаємодія усіх компонент. Розширимо рис. 2 для того, щоб показати потоки даних сучасних веб-застосунків більш наочно.

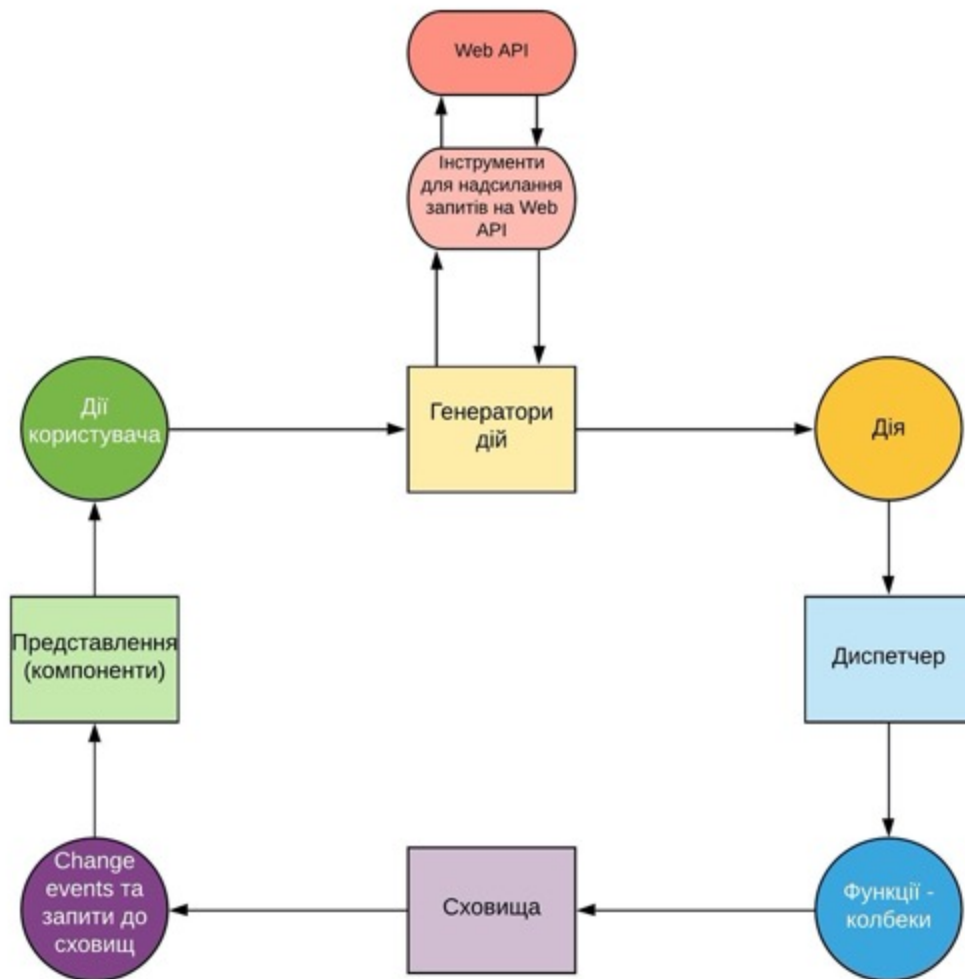


Рис. 3 Розширена модель потоку даних в архітектурі Flux

Спочатку користувач взаємодіє із певним представленням нашого застосунку, наприклад, сторінки з анкетой про проходження курсу. При натисканні на кнопку відправки форми анкети, викликається відповідний генератор дії, який в залежності від закладеного функціоналу має 2 сценарії виконання. Перший – якщо генератор отримав усю інформацію із самого представлення для зміни стану програми внаслідок натискання користувачем

кнопки, то генерується дія, яка потім передається диспетчеру на подальше опрацювання. Другий сценарій – перед зміною даних у сховищах робиться запит на певний Web API. Варто вказати, що за такого сценарію для даної архітектури встановлюють додаткові обробники для підтримки асинхронних функцій. Після того, як запит був надісланий та повернувся певний результат, у диспетчер надходить дія відповідно до тієї інформації, яка повернулася у результаті. Далі дія оброблюється диспетчером. Диспетчер викликає зареєстровані функції-колбеки, які відповідають назві переданої дії. Також за вказаним методом `waitFor(ids)` у пункті 1.2.1 функція може чекати, поки виконуються колбеки з вказаними в параметрах ідентифікаторами. У результаті опрацювання диспетчером функцій - колбеків у сховищах стану програми відбуваються відповідні зміни. Після того як сховища оновилися, до зв'язаних із сховищами компонентів сторінки надходять оновлені дані та компонент оновлює свій вміст або вміст дочірніх елементів. Також можливий варіант, коли оновлення вмісту відбуватиметься після спрацювання події, викликаного через втручання користувача (`change event`).

Підсумовуючи наведену інформацію у цьому підрозділі, можна встановити, що архітектура Flux забезпечує передбачуваний, односторонній потік даних, який був зазначений на рис. 3. Ця властивість робить парадигму Flux потужним інструментом для відслідковування стану застосунку, а також створює централізовану систему для доступу до даних програми.

1.3 Переваги Flux

У той час, як тільки виникла парадигма Flux, серед розробників був поширеним інший підхід для побудови клієнтської частини веб застосунків, а саме паттерн MVC. Згідно з ним, архітектура застосунку складалася із таких трьох основних частин, як моделі даних предметної області (Model), представлення (View) та контролери (Controller), які реагували на дії користувача шляхом маніпулювання моделями та змінювали вміст

представлень. Усі ці компоненти зв'язані наступним чином: користувач взаємодіє із сторінкою, натискаючи кнопку. Контролер, який реагує на дію користувача, робить запити на сервер, оновлюючи дані, та отримує ці дані від моделі, щоб потім оновити представлення. Схема потоку даних у цьому підході представлена на рисунку 4.

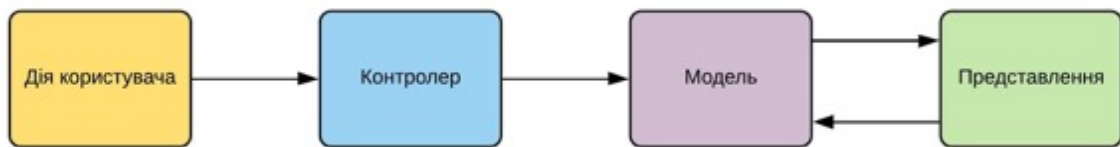


Рис. 4 Схема потоку даних у підході MVC

Аналізуючи рисунки 2 та 4, на яких зображені схеми потоків даних підходів Flux та MVC, можна побачити суттєві відмінності. Втім, давайте порівняємо реалізацію цих підходів для більш складних систем перед тим як робити висновки.

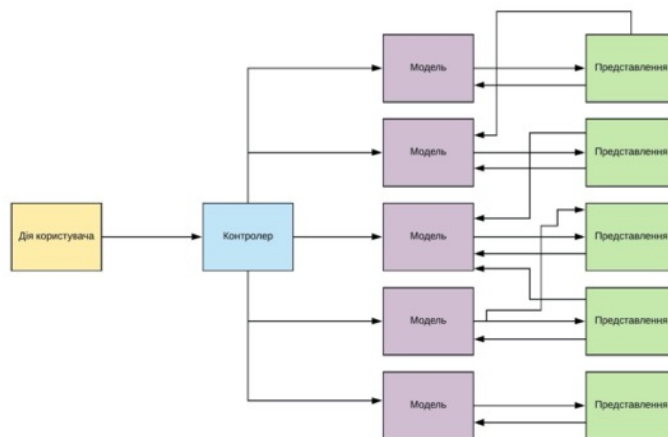


Рис. 5 Приклад більш складного потоку даних у MVC підході

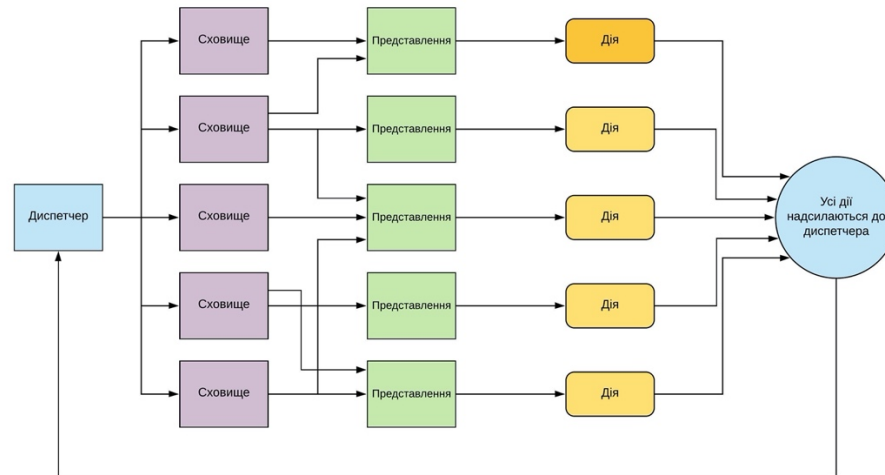


Рис. 6 Приклад більш складного потоку даних у Flux підході

Порівнюючи більш складні потоки даних у підходах, що розглядаються можна побачити, що із збільшенням функціоналу застосунку збільшується і кількість залежностей між компонентами. Проте із Flux підходом масштабування застосунку здійснюється легше. Це зумовлюється наявністю одностороннього потоку даних замість двостороннього, який використовується в архітектурі MVC [5]. Для Flux архітектури достатньо додатково створювати дії та реєструвати нові функції-колбеки для обробки дій диспетчером. Проте у MVC для масштабування потрібно буде створювати нові залежності між представленнями та моделями, що буде ускладнювати зв'язки між ними. Разом з цим підхід Flux полегшує дебаг застосунків через передбачувані структуру змін у потоці даних. Порівняно з Flux використовуючи MVC підхід до побудови застосунку відлагоджувати його стає дедалі важче і важче через двосторонній потік даних, що спричинює виникненню великої кількості залежностей між одними і тими самими частинами архітектури[6].

Підбиваючи підсумки зіставлення двох підходів, можна виділити основні переваги Flux підходу, а саме наявність одностороннього потоку даних, зміна стану програми через створення дій та їх передачі для обробки централізованим механізмом - диспетчером, а також легше масштабування та відлагодження програми.

2 ОПИС МОЖЛИВОСТЕЙ БІБЛІОТЕКИ REDUX

2.1 Характеристики рис Redux

Redux – це передбачуваний контейнер стану програми для JavaScript застосунків [6]. Бібліотека Redux була створена на основі Flux архітектури Деном Абрамовим та Ендрю Кларком у 2015 році. І хоча Redux по суті реалізує архітектуру Flux, у реалізації бібліотеки є деякі відмінності від тієї структури, яка була описана у розділі 1.

По-перше, Redux представляє нові компоненти, не передбачені у самому підході Flux, які називаються регуляторами (reducers). Вони являють собою чисті функції, які беруть поточний стан застосунку і надіслані у дії дані та використовують їх для створення та повернення нового стану програми. В регуляторах не рекомендується видозмінювати вихідні аргументи, виконувати фонові завдання, такі як надсилення запитів на Web API, викликати нечисті функції, наприклад *Date.now()* або *Math.random()*. Регулятори спроектовані для оновлення певних листків чи вузлів нашого об'єкту. Таким чином, ми можемо скомпонувати усі регулятори в один єдиний, який зможе оновлювати стан усього застосунку як тільки він отримуватиме певну дію.

По-друге, Redux спрощує концепцію Flux, усуваючи диспетчер (проте функція для диспетчеризації дій залишається) та замінюючи стан застосунку на єдиний об'єкт, в який трансформуються усі сховища даних. Тому в нас є одне єдине джерело, з якого ми можемо брати необхідні дані для відображення у представленнях. Цей об'єкт, тобто стан програми, можна представити у вигляді дерева, у якому вузли першого рівня репрезентують об'єкти, у яких зберігаються дані, пов'язані із певною моделлю даних предметної області. Наприклад, у вузлі під назвою *user* може зберігатися інформація про поточного користувача, список його повноважень, значення вподобаних кольорів, які користувач хоче використовувати у темі застосунку тощо.

По-третє, у Redux дії (actions) представлені у вигляді об'єктів. Для цих об'єктів обов'язково має визначитися поле з ключем *type*, значенням якого є

програмним забезпеченням. Наприклад, `redux-thunk` модуль надає підтримку асинхронних функцій при диспетчеризації. Також може проводитися логування роботи диспетчера, тобто вивід розробникам інформації про роботу застосунку, в даному випадку, вивід усіх дій та результатів диспетчеризації функцій. Або ж можна підключити програмний компонент, який буде надсилати певну метрику на обрані сервери.

Після диспетчеризації у дію вступають регулятори. Згідно з типом дії визначається яким чином буде змінено стан програми. Далі вони звертаються до попереднього стану, передивляються надіслану інформацію після диспетчеризації, та створюють новий стан застосунку, вказуючи нові дані на місці старих. Оновлення даних здійснюється у тих вузлах чи листках стану, які були вказані у регуляторі. Після цього представлення отримують оновлені дані через зв'язки із загальним станом програми, які визначаються безпосередньо розробниками. [4]

Підсумовуючи, хоч і `Redux` створювалася на основі `Flux` підходу, проте вона має зміни у деяких компонентах. Ці зміни були внесені до `Redux` для спрощення використання підходу та більш наочного відстежування даних кожного стану програми.

2.2 Бібліотека `React Redux` для зв'язування `React.js` із `Redux`

Так як у практичній частині курсової роботи було використано бібліотеку `React.js` для створення клієнтської частини застосунку, то надалі мова піде про те, як використовувати бібліотеку `Redux` разом із `React.js`. Варто зазначити, що бібліотека `Redux` може бути використана із будь-яким фреймворком або бібліотекою для написання користувацького інтерфейсу програми. Однак `Redux` наразі є найбільш вживаним із `React.js`. Це зумовлено тим, що `React`-компоненти також мають свої стани, і через зміну цих станів відбувається відображення потрібних для користувача даних. Також цікавим є той факт, що коли бібліотека `Redux` була представлена у 2015 році, компанія `Facebook` найняла обох

розробників цієї бібліотеки, Д. Абрамова та Е. Кларка, працювати разом із командою розробників React.

Далі у цьому підрозділі буде розглянуто іншу бібліотеку, React Redux, яку використовують при розробці застосунків із React.js та Redux. Згідно з офіційною документацією бібліотеки, React Redux – це офіційна бібліотека, розроблена командою Redux, створена для зв'язування сховища Redux із React компонентами. Основними характеристиками цієї бібліотеки є передбачуваність, енкапсульованість та оптимізованість. Передбачуваність виявляється у тому, що саме розробники визначають дані, які їм потрібно відображувати у React компонентах з глобального сховища Redux. React Redux надає власні компоненти – обгортки для керування сховищем стану програми, що і визначає таку рису як енкапсульованість. Оптимізованість виражається здатністю автоматично здійснювати оптимізацію процесів, пов'язаних із Redux сховищем, завдяки чому компоненти React оновлюються у тих випадках, коли тільки ті дані, які використовуються у самих компонентах, зазнали змін у сховищі стану програми [4].

Розглянемо детальніше як саме відбувається з'єднання компонент React із сховищем Redux за допомогою цієї бібліотеки. На офіційному сайті вказано два елементи, які приймають участь у цьому процесі.

Першим елементом є компонент Provider. Саме цей компонент пропагує сховище даних Redux усьому дереву компонент React. Так як майже кожен компонент може звертатися до сховища, у багатьох застосунках Provider визначають коренем дерева компонентів застосунку. Таким чином, якщо компонент не є дочірнім елементом Provider, то доступу до Redux сховища цей компонент не отримає. Компонент Provider було забезпечено такими пропсами, як `store`, `children` та `context`. У пропс `store` передається сховище із бібліотеки Redux. Пропс `children` відповідає за відображення усіх дочірніх компонентів. Пропс `context` використовується для впровадження контексту для дочірніх елементів. У контексті часто визначаються налаштування для користувацького інтерфейсу, наприклад, кольори, мови або ж інші статичні налаштування.

Другим елементом зв'язування сховища є функція `connect()`. Ця функція надає доступ компоненту до поточного стану сховища. При цьому компоненту повертаються тільки ті дані зі сховища, які вказуються у параметрах функції. Результатом `connect()` є інша функція, яка обгортає вказаний у параметрах компонент та вводить у пропси компонента дані, отримані зі сховища. Таким чином, виклики функції `connect()`, у загальних випадках, виглядають саме так: `connect(mapStateToProps, mapDispatchToProps)(MyComponent)`. Розглянувши призначення функції, перейдемо до параметрів цієї функції. Функція `connect()` має чотири необов'язкові параметри:

- *mapStateToProps* – функція, яка отримує параметром поточний стан сховища та повертає об'єкт, ключі якого записуються назвами у пропси компонента, а значення об'єкту є значеннями відповідних пропсів;
- *dispatchStateToProps* – може бути об'єктом або функцією з параметром `dispatch`, яка повертає об'єкт. Об'єкт містить функції, які передають диспетчеру сховища дії для зміни даних у сховищі;
- *mergeProps* – функція, яка вказує кінцеві пропси компоненту з пропсів, які були утворені в результаті функцій *mapStateToProps* та *mapDispatchToProps*, включаючи власні пропси компоненту.
- *options* – об'єкт для вказівки додаткових налаштувань функцій `connect`.

Окрім наведених двох елементів зв'язки з Redux сховищем можна виділити ще три так звані хуки, які забезпечує бібліотека React Redux.

Перший хук – `useSelector()`. Цей хук використовується для отримання у функціональний компонент значення зі сховища. Параметром хука є функція, у власному параметрі якої приймається поточний стан застосунку та повертаються потрібні дані для того, щоб компонент відобразив їх для користувача. При цьому оновлення цього поля буде здійснюватися кожного разу, коли диспетчер буде передавати оновлені дані регуляторам для зміни стану програми. Цей хук можна застосувати багато раз у одному і тому самому компоненті.

Другий хук – *useDispatch()*. Хук, який повертає відсилку на метод *dispatch()*. Цей метод надалі може використовуватися для передачі диспетчеру дій після того, як користувач взаємодіяв із компонентами або їх окремими елементами.



```
import React from 'react'
import { useDispatch } from 'react-redux'

export const CounterComponent = ({ value }) => {
  const dispatch = useDispatch()

  return (
    <div>
      <span>{value}</span>
      <button onClick={() => dispatch({ type: 'increment-counter' })}>
        Increment counter
      </button>
    </div>
  )
}
```

Рис. 8 Приклад застосування *useDispatch()*

На рисунку 8 можна побачити, що у змінна *dispatch* отримала відсилку на *dispatch* із сховища Redux. В визначенні колбеку події *onClick* теги *button* здійснюється передача диспетчеру дії з типом “increment-counter”. Після обробки дії регуляторами, у змінну *value* повернеться нове значення, збільшене на одиницю.

Третій та останній хук – *useStore()*. Цей хук повертає відсилку на те сховище, яке ми передавали компоненту *Provider*. Хоч цей хук існує в реалізації бібліотеки *React Redux*, сценарії, коли його потрібно використати, трапляються дуже рідко, зокрема коли є потреба у зміні регуляторів сховища. Тому розробники бібліотеки рекомендують використовувати *useSelector()* замість цього хука.

Отже, у цьому підрозділі було розглянуто основні принципи для поєднання сховища Redux із компонентами React. Ці принципи будуть використані у 3 розділі цієї курсової роботи для розробки програмного застосунку на базі архітектурного підходу Flux.

3 РОЗРОБКА ПРОГРАМНОГО ЗАСТОСУНКУ “СЕРВІС ОПИТУВАННЯ СТУДЕНТІВ”

3.1 Опис предметної області

Для розробки практичної частини курсової роботи було обрано створити клієнтський додаток, який буде відображати пройдені курси студентів за певний період навчання у ЗВО. Студенти зможуть проходити опитування по пройденим ними курсам. Дані будуть надсилатися на сервер через Web API, який буде надсилати, оброблювати та зберігати ці дані. Окрім студентів, додаток буде підтримувати функціонал для перегляду результатів опитування адміністрацією ЗВО. Також складовою цього застосунку буде сторінка адміністраторів ЗВО, які зможуть вносити та змінювати дані про студентів, викладачів, курси, структурні підрозділи, спеціальності тощо. Для кожної ролі буде призначено відповідні права доступу до елементів застосунку. Наприклад, студент не зможе передивитися результати опитування та не матиме доступу до сторінки адміністратора застосунку, в той час як адміністратори зможуть переглядати дані свого ЗВО на даній сторінці. В перспективах подальшої розробки проекту буде створено функціонал для проведення аналізу освітніх програм, які існують у ЗВО.

3.2 Ініціалізація сховища Redux

Отже, для розробки застосунку з використанням бібліотеки Redux потрібно зробити початкову конфігурацію.

```
import {applyMiddleware, compose, createStore} from "redux";
import thunkMiddleware from "redux-thunk";
import rootReducer from "./modules";

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

export default function store() {
  return createStore(
    rootReducer,
    composeEnhancers(applyMiddleware(thunkMiddleware))
  );
}
```

Рис. 9 Створення сховища стану програми

На рисунку 9 зображено код для створення сховища Redux. Функція *createStore* приймає у параметрах визначений нами регулятор, а також проміжне ПЗ за необхідності його використання. Таким чином, *rootReducer* – це окремий регулятор, який був скомпонований з усіх прописаних регуляторів в ході розробки програми, а *composeEnhancers* – це функція, яка за наявності встановленого розширення Redux DevTools підключає наше сховище для відображення у окремій вкладці інструментів розробника у використовуваному браузері. Якщо розширення не було завантажено, то використовується функція *compose*, яка компонує усі надані проміжні ПЗ, надані через *applyMiddleware* функцію, для підтримки додаткових функцій. У цьому застосунку використовується *redux-thunk*. Це ПЗ, як вже зазначалося у розділі 2.1, надає диспетчеру підтримку здійснення асинхронних функцій.

```
import institutions from "../institution";
import surveys from "../survey";
import results from "../result"
import admin_data from "../adminData"
import sidebar from "../sidebar"
import {reducer as formReducer} from "redux-form";

const appReducer = combineReducers( reducers: {
  auth: auth,
  institutions: institutions,
  surveys: surveys,
  results: results,
  admin_data: admin_data,
  sidebar: sidebar,
  form: formReducer
});

const rootReducer = (state, action) => {
  if (action.type === ACTION_TYPES.SIGN_OUT) {
    state = undefined;
  }
  return appReducer(state, action)
};

export default rootReducer;
```

Рис. 10 Об'єднання регуляторі

На рисунку 10 продемонстровано код з файлу *index.js* папки *redux*. Суть цього коду – це об'єднання усіх регуляторів в один, який буде передаватися сховищу Redux. Для цього використовується функція *combineReducers()*, яка приймає об'єкт з усіма визначеними регуляторами та повертає загальний регулятор. В кінці файлу ми експортуємо отриманий регулятор для імпорту його у наше сховище.

Остання ланка ініціалізації сховища для використання у компонентах – це передача підготовленої функції *store()* компоненту – обгортці кореневого

компонента App нашого застосунку, а саме Provider, який був детально розглянутий у розділі 2.2. Ці дії висвітлено на рисунку 11.

```
import React from "react";
import ReactDOM from "react-dom";
import {Provider} from "react-redux";
import store from "../redux/store";

import App from "../scenes/App/App";

ReactDOM.render(
  <Provider store={store()}>
    <App/>
  </Provider>,
  document.querySelector("#root")
);
```

Рис. 11 Передача компоненту Provider функції store() для створення сховища Redux

У результаті наших дій було сконфігуровано глобальне сховище, у якому будуть триматися дані про поточний стан нашого застосунку і до якого будуть звертатися наші компоненти для отримання та оновлення даних за принципами Flux архітектури.

3.3 Реалізація клієнтської архітектури сервісу на основі підходу Flux

У цьому розділі розкажується про потік даних застосунку на прикладі компонентів Survey та Surveys, які відповідають за показ пройдених студентами курсів та створення відгуків на дисципліни.

Інтерфейс користувача було розроблено у вигляді функціональних та класових компонент, щоб показати застосування і хуків бібліотеки Redux, і функції *connect()*.

Спочатку переглянемо реалізацію сторінки курсів студента. На рис. 12 наведено інтерфейс цієї сторінки. Отримання даних про курси відбувається

шляхом відправки HTTP запиту на сервер. Після успішної обробки відповіді, ці дані переходять у сховище. Так як у компоненті `Surveys` відбувається зв'язування з даними курсів у відповідному вузлі об'єкта стану, компонент отримує оновлені дані та відображає їх у вигляді таблиці. Далі студент обирає курс, на який він хоче написати відгук та переходить на сторінку з анкетою.

Назва курсу	Дії
Системне програмування	Редагувати
Технології сучасних дата - центрів	Редагувати
Базові мережні технології	Редагувати
Функціональне програмування	Редагувати
Логічне програмування	Заповнити
Комп'ютерна вірусологія	Заповнити
Веб-програмування	Заповнити
Теорія ймовірностей	Заповнити
Технологія веб-програмування Ruby on Rails	Заповнити
Методи об'єктно-орієнтованого програмування	Заповнити
Бази даних	Заповнити
Архітектура прикладних програм ріння підприємства	Заповнити
Вибрані питання програмої інженерії	Заповнити

Рис. 12 Сторінка курсів студента

Тепер опишемо детальніше, що відбувається у коді самого застосунку. Увесь процес починається при першому відображенні компоненту `Surveys` (Додаток А). Спочатку ми визначаємо, що наш класовий компонент буде отримувати через функції `connect()`, `mapStateToProps` і `mapDispatchToProps` такі дані зі сховища: `user` (інформація про поточного користувача), `surveys` (список усіх курсів) та `fetchSurveys` (генератор дій). Так як, потрапляючи перший раз на сторінку, ми не маємо дані про курси, то відображається спеціальний компонент `Loader`, який показує, що дані ще не підвантажилися з сервера. В цей час компонент за допомогою генератора дій `fetchSurveys()` (рис. 13) у так званому *lifecycle* методі `ComponentDidMount()`, який виконується тільки один раз після

першого відображення компоненту, продукує функцію та відправляє її на обробку диспетчером. Диспетчер за допомогою Redux-thunk та HTTP клієнту axios відправляє запит на сервер аби отримати інформацію про курси. Як тільки сервер надсилає відповідь, генерується дія з типом “FETCH_USER_SURVEYS” і даними про курси у полі payload та передається на обробку регулятора.

```
const doFetchUserSurveys = (id) => async dispatch => {
  api.get('/student/' + id + "/course/survey").then((response) => {
    dispatch({type: ACTION_TYPES.FETCH_USER_SURVEYS, payload: response.data.data})
  }).catch(() => {
    addNotification( message: "Не вдалося отримати курси студента!");
  });
};
```

Рис. 13 Генератор дії для передачі отриманих курсів в сховище програми

Отримавши дію, регулятор порівнює тип цієї дії із кожним значенням у конструкції switch. Якщо назва дії збігається із визначеною в case, тоді здійснюється “зміна” стану. Все, що було у попередньому стані, передається у новий із зміною потрібних полів. В даному випадку в нас виконуватиметься case з назвою FETCH_USER_SURVEY. Він збереже попередній стан сховища та оновить поле surveyList даними, переданими у дії в полі payload.

```
function reducer(state = initialState, action) {
  switch (action.type) {
    case ACTION_TYPES.GET_STUDENT_INFO:
      return {...state, student: action.payload};
    case ACTION_TYPES.FETCH_USER_SURVEYS:
      return {...state, surveysList: action.payload};
    case ACTION_TYPES.FETCH_USER_SURVEY:
      return {...state, currentSurvey: action.payload};
    case ACTION_TYPES.POST_USER_SURVEY:
      return {...state, currentSurvey: null, errors: null};
    case ACTION_TYPES.DISPLAY_ERRORS:
      return {...state, errors: action.payload};
    case ACTION_TYPES.REMOVE_ERROR:
      return {...state, errors: state.errors.filter((err)=>err !== action.payload)};
    default:
      return state;
  }
}
```

Рис. 14 Регулятор survey

Після цього оновлені дані надходять до компоненту Surveys та відображаються у вигляді таблиці. Для перегляду поточного стану сховища нашого застосунку скористаємось розширенням браузера Redux DevTools.

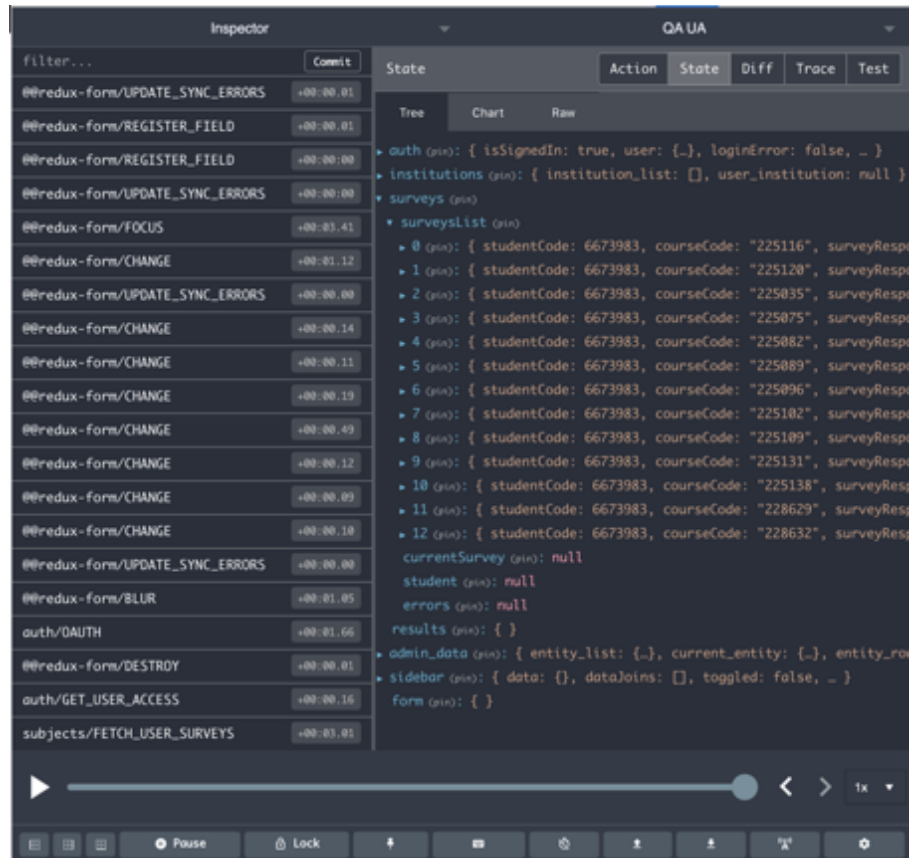


Рис. 15 Стан сховища у Redux DevTools

У такий же спосіб давайте проаналізуємо сторінку заповнення анкети, за відображення якої відповідає функціональний компонент Survey (Додаток Б). Інтерфейс сторінки продемонстровано на рис. 16. При початковому завантаженні цієї сторінки здійснюється отримання інформації про користувача, форму курсу та помилок зі сховища даних. Однак цього разу ми використаємо хуки *useSelector()* та *useDispatch()*, робота яких детально описані у пункті 2.2. На рисунку 17 вказано код компонента Survey, який відповідає за отримання вищезазначених даних.

Логічне програмування

09:00 13.04.2020 00:00 19.04.2020

ПІБ викладача(ки), який(я) читає(ла) лекції*

Оберіть...

Прізвище викладача(ки), який(я) проводить(ла) семінари або практичні*

Оберіть...

Тематика курсу зацікавила мене і видалася такою, що сприятиме моєму професійному та інтелектуальному розвитку.*

★★★★★

Мова викладання курсу (під час лекційних і семінарських занять) відповідає зазначеній в робочо-тематичному плані.*

★★★★★

Загалом я задоволений(а) якістю курсу.*

★★★★★

Я би порадив(ла) іншим прослухати цей курс.*

★★★★★

Викладач(ка) чітко пояснював(ла) матеріал на лекції.*

★★★★★

Начальні курси повинні становити інтелектуальний виклик і змушувати студентів докласти певних зусиль під час навчання, адже це сприяє інтелектуальному розвитку. Цей курс відповідає зазначеній вимозі.*

Рис. 16 Інтерфейс сторінки заповнення анкети

```
const Survey = (props) => {
  const user = useSelector(state=>state.auth.user);
  const survey = useSelector(state=>state.surveys.currentSurvey);
  const errors = useSelector(state=>state.surveys.errors);
  const dispatch = useDispatch();

  const st_id = user.person.students[0].code;
  const course_id = props.match.params.id;
  useEffect(  effect: () => {
    dispatch(actionCreators.doFetchUserSurvey(st_id, course_id));
  },  deps: [st_id, course_id, dispatch]);
}
```

Рис. 17 Використання хуків useSelector та useDispatch

Окрім цього, у компоненті використовується хук `useEffect` (рис. 17), представлений у бібліотеці `React`, принцип дії якого наступний: `useEffect()` приймає два параметри – колбек функцію для виклику та масив, у якому задаються залежні змінні. Кожного разу, коли змінюється одне із значень таких змінних, хук буде викликати функцію колбек та оновлювати компонент[9]. Функція-колбек обов'язково викликатиметься при початковому відображенні. Виходячи з цього, при першому завантаженні сторінки буде викликатися функція, яка буде генерувати дію `FETCH_USER_SURVEY`. Генерація відбуватиметься у методі `doFetchUserSurvey()`, який зображено на рис. 18.


```
const doFetchUserSurvey = (st_id, course_id) => async (dispatch) => {
  api.get('/student/' + st_id + '/course/' + course_id + '/survey').then((response) => {
    dispatch({type: ACTION_TYPES.FETCH_USER_SURVEY, payload: response.data})
  }).catch(() => {
    addNotification( message: "Виникла помилка при отриманні анкети.");
  });
};
```

Рис. 18 Генератор дії для передачі конкретного курсу в сховище програми

Далі відбуваються ті самі процеси, які вже описувалися вище у цьому розділі. Дія надходить до регулятора, він знаходить case для створення нового стану, дані у сховищі оновлюються та отримуються у компоненті після чого компонент відображає отримані дані на сторінці. У Redux DevTools можемо переглянути отримані дані та дії, які були передані диспетчеру та оброблені регулятором.

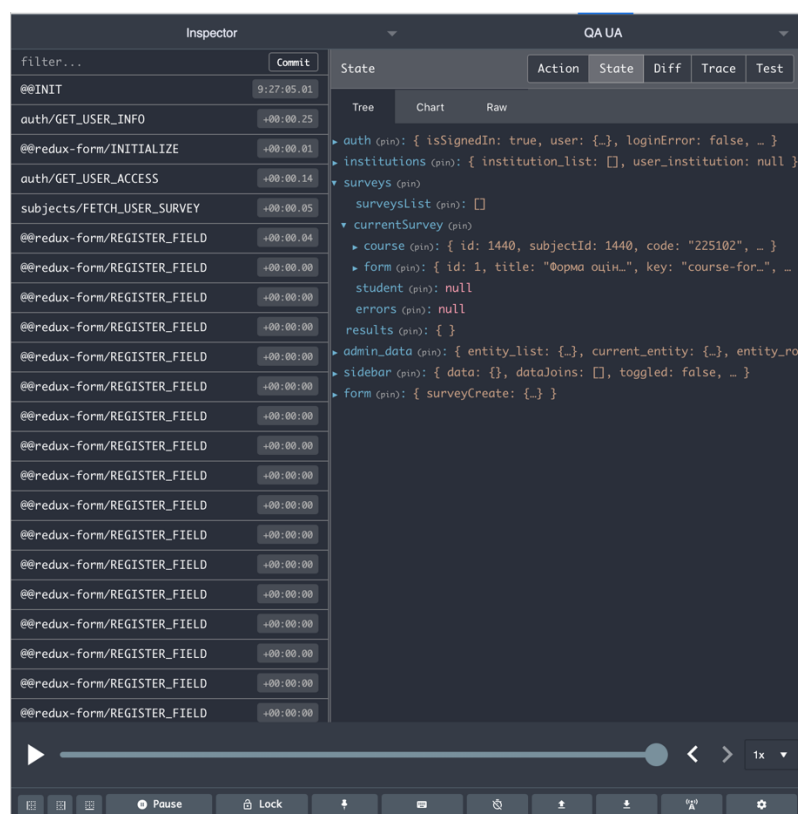


Рис. 19 Стан сховища Redux у DevTools після диспетчеризації дії
FETCH_USER_SURVEY

Крім цього функціоналу, компонент Survey містить дочірні компоненти Field з бібліотеки Redux Form. Ця бібліотека надає легкий спосіб керуванням

форми та збереження даних форми у стані сховища. У цій бібліотеці реалізується такий самий потік даних як і у Redux (рис. 19). Від розробників, які використовують цю бібліотеку, потрібно лише додати до загального регулятора регулятор із цієї бібліотеки та обгорнути компонент функцією `reduxForm()`, а в параметрах надати об'єкт з налаштуваннями форми, обов'язково вказавши поле `form` із унікальним значенням [10].

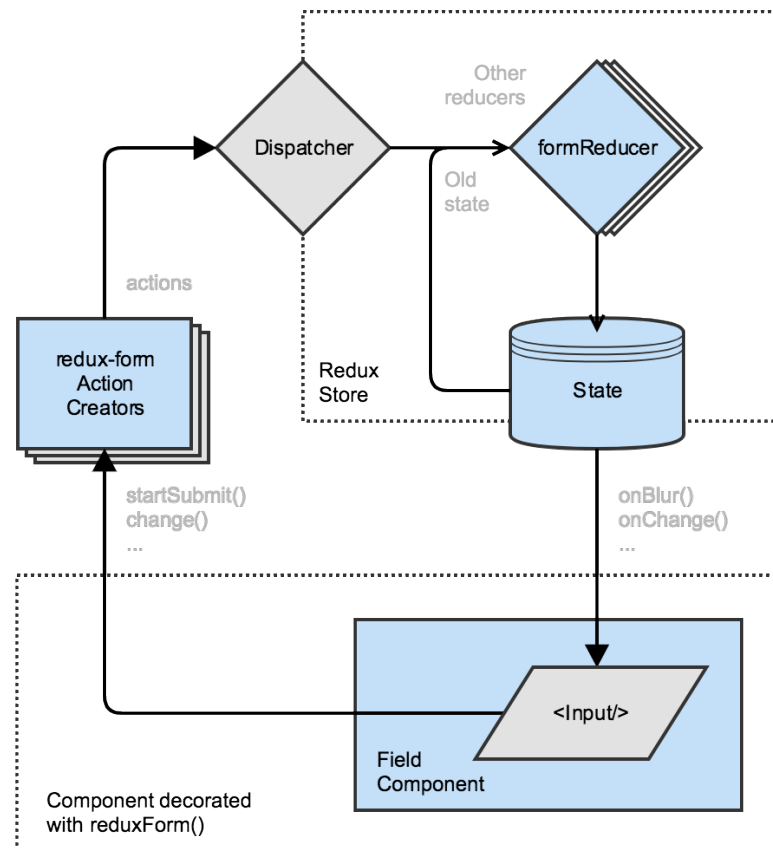


Рис. 20 Потік даних Redux Form [10]

Компонент `Field` реєструє поле у об'єкті форми, який є вузлом стану сховища, із назвою, зазначеною в атрибуті `name` цього компонента. Також у пропсах `Field` потрібно обов'язково вказати `component`, який буде відображатися для введення даних користувачем. Це можуть бути теги `input`, `textarea`, `select` або ж власний компонент.

Коли користувач буде взаємодіяти із компонентом `Field`, обгортка `redux-form` буде генерувати дії для зміни даних у стані сховища. При натисканні кнопки підтвердження відправки форми, ми викликаємо колбек функцію, яка має бути

обгорнута функцію *handleSubmit* для того, щоб дані форми (рис. 20) автоматично передалися у параметри нашого колбеку. Далі просто генерується дія, відправляється диспетчеру, той надсилає дані на сервер, і в залежності від відповіді сервера, змінює стан сховища через регулятори.

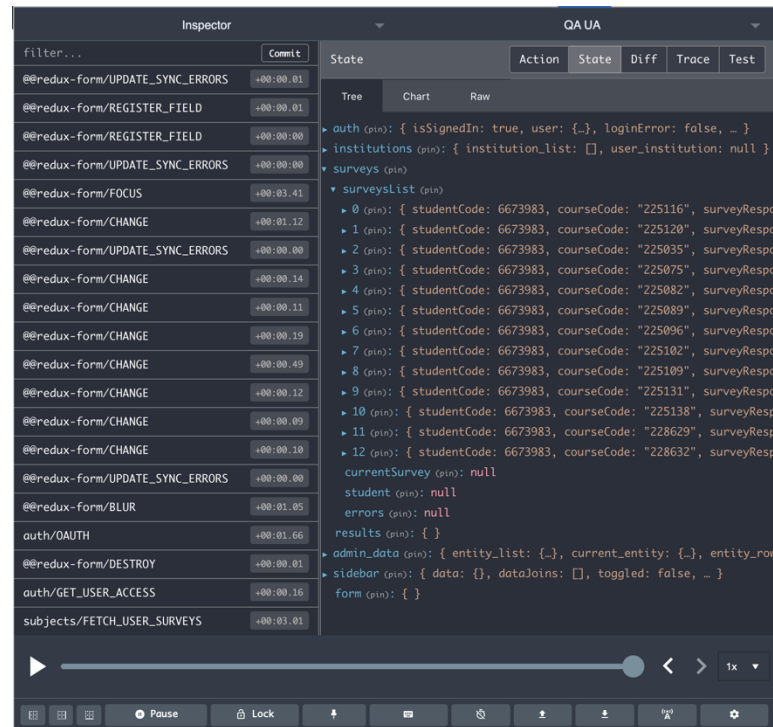


Рис. 21 Реєстрація полів форми та перегляд стану сховища у Redux DevTools

Підсумовуючи, ми бачимо, що архітектурний підхід Flux дає можливості для передбачуваного використання стану програми. За допомогою Redux DevTools можна переглядати дані у сховищі для кожного стану програми, а регулятори та генератори дій дають можливість для легкого масштабування нашого застосунку при збільшенні функціоналу нашого застосунку.

ВИСНОВКИ

У цій роботі було проведено детальний аналіз архітектури Flux. Цей архітектурний підхід виник як рішення проблеми з механізмом передачі даних по дереву компонент програмного застосунку. У цьому підході визначається чотири складові частини, а саме диспетчер, сховище, представлення та дії. Така структура забезпечує односторонній потік, що має переваги в порівнянні із двостороннім потоком. Зокрема із переваг можна виділити централізований доступ до даних, бо усі дані зберігаються у сховищі, зручніші можливості для дебагу застосунку та відносно легку масштабованість за рахунок створення дій та реєстрації колбеків у диспетчері.

Бібліотека Redux, яка взяла за основу цей підхід, зараз складає провідне місце у розробці клієнтських застосунків. Розробники внесли деякі корективи до складових цього підходу, проте потік даних зберігся такий самий, як і передбачається у Flux. Redux додав регулятори, які змінюють стан сховища, змінив представлення сховища до plain old javascript object. Теж саме можна сказати і про дії, які також перетворилися у роjo та використовуються для створення нового стану застосунку. Разом із бібліотекою React Redux, Redux створює зручні інструменти для реалізації клієнтських застосунків на основі Flux підходу.

І нарешті було розроблено програмний застосунок для опитування студентів про якість вищої освіти для різних ЗВО. У застосунку використовувався Redux для реалізації Flux-подібної архітектури та на прикладі двох сторінок було продемонстровано односторонній потік даних, який забезпечується обраною архітектурою. Ці властивості дозволяють підходу Flux забезпечувати сприятиме пришвидшенню додавання нового функціоналу та легку масштабованість застосунку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. NPM Trends [Електронний ресурс]. Режим доступу: <https://www.npmtrends.com/angular-vs-react-vs-vue-vs-redux-vs-react-redux>
2. Flux. Application architecture for building user interfaces [Електронний ресурс]. Режим доступу: <https://facebook.github.io/flux/>
3. Flux in Depth. Overview and components [Електронний ресурс]. Режим доступу: <https://blog.mgechev.com/2015/05/15/flux-in-depth-overview-components/>
4. What is the Flux Application Architecture? [Електронний ресурс]. <https://brigade.engineering/what-is-the-flux-application-architecture-b57ebca85b9e>
5. Flux Application Architecture and React [Електронний ресурс]. Режим доступу: <https://freecontent.manning.com/react-in-action-flux-application-architecture/>
6. Redux. A Predictable State Container for JS Apps [Електронний ресурс]. <https://redux.js.org/>
7. Alex Banks. Learning React: Functional Web Development with React and Redux: Підручник / Alex Banks, Eve Porcello. – К. : O'Reilly, 2017. – ст. 183 – 209.
8. React Redux. Official React bindings for Redux [Електронний ресурс]. Режим доступу: <https://react-redux.js.org/>
9. Огляд хуків. Документація React [Електронний ресурс]. Режим доступу: <https://uk.reactjs.org/docs/hooks-overview.html>
10. Redux-form documentation. Field component [Електронний ресурс]. Режим доступу: <https://redux-form.com/8.3.0/docs/api/field.md/>

ДОДАТКИ

Додаток А

(обов'язковий)

Лістинг програмного коду компонента Surveys у файлі Surveys.js

```

import React, {Component} from "react";
import {connect} from "react-redux";
import {Link} from "react-router-dom";
import {actionCreators} from "../../redux/modules/survey";
import {URL_SURVEYS} from "../../constants/url";
import Breadcrumbs from "../../components/Breadcrumbs";
import Loader from "../../components/Loader";

class Surveys extends Component {
  state = {
    crumbs: [
      {
        path: '/',
        name: 'Головна',
      },
      {
        path: '/surveys/',
        name: 'Опитування',
      }
    ]
  };

  componentDidMount() {
    this.props.fetchSurveys(this.props.user.person.students[0].code);
  }

  renderButton(survey) {
    return (
      <Link to={URL_SURVEYS + survey.course.code} className={`btn btn-
${survey.hasResponse? "primary":"info"}}`>
        {survey.hasResponse?"Редагувати":"Заповнити"}
      </Link>
    );
  }
}

```

```
}
```

```
renderList() {
  return this.props.surveys.map(survey => {
    return (
      <tr key={survey.course.id}>
        <td className="align-middle">{survey.course.subject.name}</td>
        <td className="text-center">{this.renderButton(survey)}</td>
      </tr>
    );
  });
}
```

```
renderTable() {
  return (<table className="my-4 table table-striped table-bordered table-
hover">
    <thead>
      <tr>
        <th scope="col" className="h3 text-center">Назва курсу</th>
        <th scope="col" className="h3 text-center">Дії</th>
      </tr>
    </thead>
    <tbody>
      {this.renderList()}
    </tbody>
  </table>)
}
```

```
render() {
  const {user} = this.props;
  let students = user ? user.person.students : null;

  if (!students)
    return <Loader/>;
  return <div className="table-responsive">
    <Breadcrumbs crumbs={this.state.crumbs}/>
    <div className="my-4 text-center">
      <h2>Список дисциплін</h2>
    </div>
```

```

    {this.props.surveys ? this.renderTable() :
      <div className="d-flex justify-content-center">
        <Loader/>
      </div>}
  </div>;
}
}

const mapStateToProps = state => {
  return {
    surveys: state.surveys.surveysList,
    user: state.auth.user
  };
};

const mapDispatchToProps = (dispatch) => {
  return {
    fetchSurveys: (code) => dispatch(actionCreators.doFetchUserSurveys(code))
  }
};

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Surveys);

```

Додаток Б

(обов'язковий)

Лістинг програмного коду компонента Survey у файлі Survey.js

```
import React, {Fragment, useEffect, useState} from "react";
import {Field, reduxForm} from "redux-form";
import {connect, useDispatch, useSelector} from "react-redux";
import {ACTION_TYPES, actionCreators} from "../../redux/modules/survey";
import {CheckBoxField, CheckBoxRadioField, ListField, NotesField, required,
ScoreField} from "../components";
import {surveyQuestionsType} from "../../constants";
import Breadcrumbs from "../../components/Breadcrumbs";
import {Link} from "react-router-dom";
import {URL_SURVEYS} from "../../constants/url";
import Loader from "../../components/Loader";
import moment from "moment";
import {DATE_TIME} from "../../constants/formats";

const sorting = (a, b) => {
  let keyA = a.order,
      keyB = b.order;
  if (keyA < keyB) return -1;
  if (keyA > keyB) return 1;
  return 0;
};

const Survey = (props) => {
  const user = useSelector(state=>state.auth.user);
  const survey = useSelector(state=>state.surveys.currentSurvey);
  const errors = useSelector(state=>state.surveys.errors);
  const dispatch = useDispatch();
  const [crumbs, setCrumbs] = useState([]);

  const st_id = user.person.students[0].code;
  const course_id = props.match.params.id;

  useEffect( () => {
    dispatch(actionCreators.doFetchUserSurvey(st_id, course_id));
```



```
}, [st_id, course_id, dispatch]);
```

```
useEffect(()=>{
  if (survey)
    setCrumbs([
      {
        path: "/",
        name: "Головна"
      },
      {
        path: "/surveys/",
        name: "Опитування"
      },
      {
        path: `/surveys/${survey.id}`,
        name: `${survey.form.title}`
      }
    ]);
}, [survey]);
```

```
const onSubmit = val => {
  const data = val.data.reduce(function(acc, cur, i) {
    acc[i] = cur.toString();
    return acc;
  }, {});
  dispatch(actionCreators.doPostSurvey(st_id, course_id, {data}));
};
```

```
const renderQuestions = () => {
  const data = survey? survey.data: null;

  const questions = survey.form.questions;

  return questions.sort(sorting).map(question => {
    const toValidate = [];

    const error = errors? errors.find((err)=>err.questionId ===question.id):null;
```

```

switch (question.type) {
  case surveyQuestionsType.INPUT:
    if (question.required) toValidate.push(required);
    return (
      <Fragment key={question.id}>
        <div className="my-2">
          <Field
            name={"data."+question.id}
            component={NotesField}
            title={question.title}
            inputText={question.inputText}
            required={question.required}
            validate={toValidate}
            initialValue={data?.[question.id]?data[question.id]:""}
            async_error={error}
          />
        </div>
        <div className="border-top my-3"/>
      </Fragment>
    );
  case surveyQuestionsType.SCORE:
    if (question.required) toValidate.push(required);
    return (
      <Fragment key={question.id}>
        <div className="my-2">
          <Field
            name={"data."+question.id}
            component={ScoreField}
            title={question.title}
            maxRating={question.max}
            minRating={question.min}
            defaultRating={data?.[question.id]?data[question.id]:0}
            required={question.required}
            validate={toValidate}
            async_error={error}
          />
        </div>
        <div className="border-top my-3 "/>
      </Fragment>
    );
}

```

```

);
case surveyQuestionsType.RADIO:
  if (question.required) toValidate.push(required);
  return (
    <Fragment key={question.id}>
      <div className="my-2">
        <Field
          name={"data."+question.id}
          variant_name={question.id}
          component={CheckBoxRadioField}
          title={question.title}
          variants={question.variants}
          required={question.required}
          validate={toValidate}
          async_error={error}
        />
      </div>
      <div className="border-top my-3 "/>
    </Fragment>
  );
case surveyQuestionsType.CHECKBOX:
  if (question.required) toValidate.push(required);
  return (
    <Fragment key={question.id}>
      <div className="my-2" >
        <Field
          name={"data."+question.id}
          check_name={question.id}
          component={CheckBoxField}
          title={question.title}
          answer={question.answer}
          required={question.required}
          validate={toValidate}
          async_error={error}
        />
      </div>
      <div className="border-top my-3 "/>
    </Fragment>
  );

```

```

case surveyQuestionsType.LIST:
  if (question.required) toValidate.push(required);
  return (
    <Fragment key={question.id}>
      <div className="my-2" >
        <Field
          name={"data."+question.id}
          component={ListField}
          title={question.title}
          options={question.options}
          required={question.required}
          validate={toValidate}
          enableCustomValue={question.enableCustomValue}
          initialValue={data?.[question.id]?data[question.id]:""}
          async_error={error}
        />
      </div>
      <div className="border-top my-3 "/>
    </Fragment>
  );
default:
  return <h1>Undefined Header Type</h1>;
}
});
};

if (
  !survey ||
  !survey.form.questions ||
  !crumbs
) {
  return (
    <div>
      <Loader/>
    </div>
  );
}
const {surveyAvailableFrom, surveyAvailableUntil} = survey.course;

```

```

return (
  <div>
    <Breadcrumbs crumbs={crumbs}
onClick={()=>dispatch({type:ACTION_TYPES.POST_USER_SURVEY})}/>
    <div className="row justify-content-center">
      <div className="col-md-10">
        <div
          id="survey"
          className="card"
          style={{
            borderTopRightRadius: "10px",
            borderTopLeftRadius: "10px"
          }}
        >
          <div
            className="card-header"
            style={{
              backgroundColor: "#000000",
              borderTopRightRadius: "10px",
              borderTopLeftRadius: "10px"
            }}
          >
            <h3 className="card-title" style={{color: "#ffffff"}}>
              {survey.course.subject.name}
            </h3>
            <span className="badge badge-info py-2 mx-1 my-2">
              <i className="fa fa-hourglass-start mx-1"/>
                {moment(surveyAvailableFrom).format(DATE_TIME)}
            </span>
            <span className="badge badge-info py-2 mx-1 my-2">
              <i className="fa fa-hourglass-end mx-1"/>
                {moment(surveyAvailableUntil).format(DATE_TIME)}
            </span>
          </div>
          <div className="card-body" style={{backgroundColor:
"#e9ecef"}}>
            <form
              id="survey-form"
              onSubmit={props.handleSubmit(onSubmit)}

```

```

    >
    <div>{renderQuestions()}</div>
    <button className="btn btn-primary" type="submit">
      Зберегти
    </button>
    <Link
      id="button-cancel"
      to={URL_SURVEYS}
      className="btn btn-danger float-right"

onClick={()=>dispatch({type:ACTION_TYPES.POST_USER_SURVEY})}
    >
      Скасувати

    </Link>
  </form>
</div>
</div>
</div>
</div>
</div>
);

};

const mapStateToProps=(state)=>{
  const init = state.surveys.currentSurvey?.data;
  const initValues = init ? Object.entries(init).reduce((acc,[i,val])=>{
    acc[i]=val;
    return acc;
  },[]):null;
  return {
    initialValues: {data:initValues}
  }
};

const formWrapped = reduxForm({
  form: "surveyCreate",
  enableReinitialize: true

```

```
})(Survey);
```

```
export default connect(mapStateToProps)(formWrapped);
```