OVERVIEW FLUX-ARCHITECTURE ON THE EXAMPLE OF STATE STORAGE OF JAVA-SCRIPT APPLICATION REDUX

Ovcharuk I.
PhD, Associate Professor of the Department of Information Technologies and Design, State University of Infrastructure and Technologies, Kyiv, Ukraine
Martyniuk A.
Master's degree student of the Department of Information Technologies and Design, State University of Infrastructure and Technologies, Kyiv, Ukraine

ОГЛЯД FLUX-АРХІТЕКТУРИ НА ПРИКЛАДІ КОНТЕЙНЕРУ СТАНІВ JAVA-SCRIPT ДОДАТКУ REDUX

Овчарук І.
к.т.н., доцент, доцент кафедри інформаційних технологій та дизайну, Державний університет інфраструктури та технологій, Київ, Україна
Мартинюк А.
магістрант кафедри інформаційних технологій та дизайну, Державний університет інфраструктури та технологій, Київ, Україна
DOI: 10.24412/3453-9875-2021-73-1-70-74

Abstract
The article provides a detailed analysis of MVC technology and Flux architecture. The problems that arise when using the MVC (Model-View-Controller) template are considered, namely when actions lead to cascading updates, which, in turn, leads to unpredictable results and code that becomes difficult to debug. The article discusses the advantages of Flux-architecture, which eliminates these problems. The results are demonstrated on the developed application.

Анотація
В статті докладно надано аналіз технології MVC та Flux-архітектури. Розглянуто проблеми, що виникають при використанні шаблону MVC (Model-View-Controller), а саме, коли дії викликають каскадні оновлення, що, в свою чергу, призводить до непередбачуваних результатів і коду, який стає важко відлагоджувати. В статті розглянуто переваги Flux-архітектури, яка дозволяє усунути зазначені проблеми. Результати продемонстровані на розробленому застосунку.

Keywords: Flux-architecture, application state repository, controller, model, data section (view), repository, dispatcher.
Ключові слова: Flux-архітектура, сховище станів додатку, контролер, модель, розріз даних (view), сховище, диспетчер.

Вступ. Flux-архітектура – архітектурний підхід або набір патернів проектування для побудови користувацького інтерфейсу веб-застосунку, в поєднанні з реактивним програмуванням. Flux-архітектура побудована на односпрямованих потоках даних.

Головною відмінною рисою Flux є односторонній напрямок передачі даних між компонентами архітектури Flux. Архітектура накладає обмеження на потік даних, зокрема, виключаючи можливість оновлення стану компонентів самостійно. Такий підхід робить потік даних передбачуваним і полегшує відстеження причин можливих помилок в програмному забезпеченні.

Порівняння Flux та MVC архітектур. Щоб краще описати потік, порівняємо його з однією з провідних клієнтських архітектур: MVC. У клієнтському застосунку MVC (рис.1) взаємодія користувача запускає код у контролері. Контролер знає, як координувати зміни в одній або декількох моделях, викликаючи відповідні методи . Коли моделі змінюються, вони повідомляють один або декілька розрізів даних (View), які, в свою чергу, читають нові дані з моделей і оновлюють себе відповідно, щоб користувач міг бачити ці нові дані.

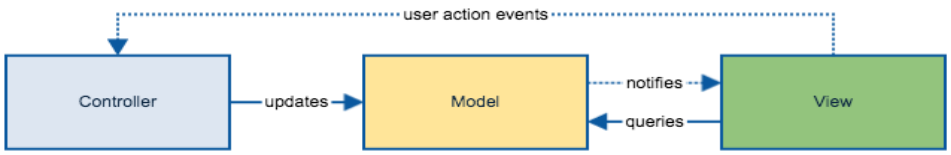

Рис. 1. Схематичне зображення простого додатку MVC-архітектури.

Оскільки програма MVC зростає, а контролери, моделі та розрізи даних (View) додаються, залежності стають складнішими. З додаванням всього трьох розрізів даних (View) (рис. 2), одного контролера і однієї моделі, графік залежності вже важче простежити. Коли користувач взаємодіє з інтерфейсом, виконуються кілька шляхів коду розгалуження, і виникає проблема з налагодженням у стані програми, яка пов'язана зі з'ясуванням того, який модуль (або модулі) в одному (або декількох) з цих потенційних шляхів коду містить помилку. У гіршому випадку взаємодія з користувачем спричинить оновлення, які, в свою чергу, викличуть додаткові оновлення, що призводить до помилок та каскадних ефектів [1, с. 105].

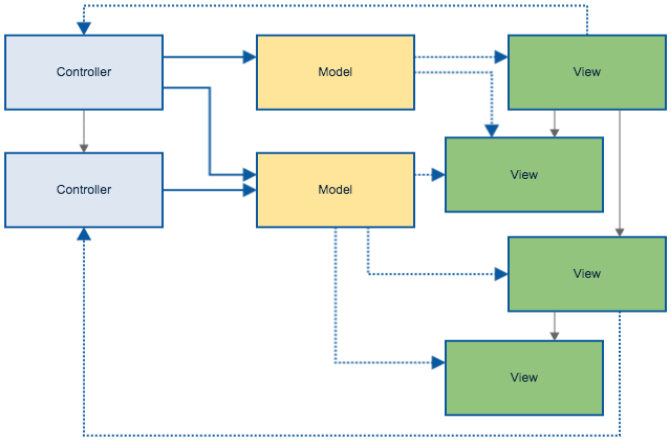

Рис. 2. Схематичне зображення складного додатку MVC-архітектури.

Flux уникає цієї конструкції на користь одностороннього потоку даних. Усі взаємодії користувачів у розрізі даних викликають *дію (Action Creator)*, що призводить до того, що подія *Action* викликається диспетчером (*Dispatcher*). Диспетчер є однією точкою виклику для всіх дій у Flux-додатку. Дія відправляється від диспетчера в *сховища (Store)*, які оновлюються у відповідь на дію (рис. 3).

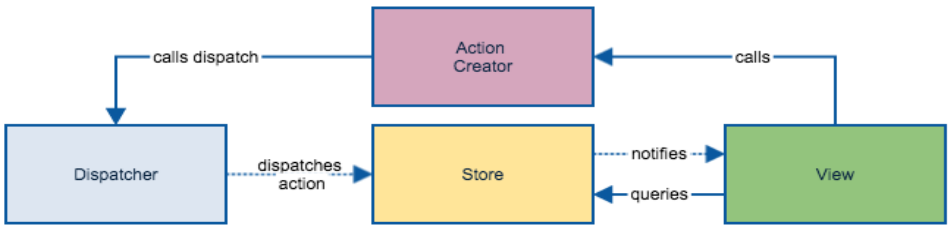

Рис. 3. Схематичне зображення простого додатку Flux-архітектури.

Потік істотно не змінюється для додаткових станів або розрізів даних (View). Диспетчер просто відправляє всі дії в *усі* стани в додатку. Причому, він не містить знань про те, як насправді оновити стан – саме сховище містить цю бізнес-логіку. Кожен стан відповідає за домен програми і оновлює себе тільки у відповідь на дії (рис.4).

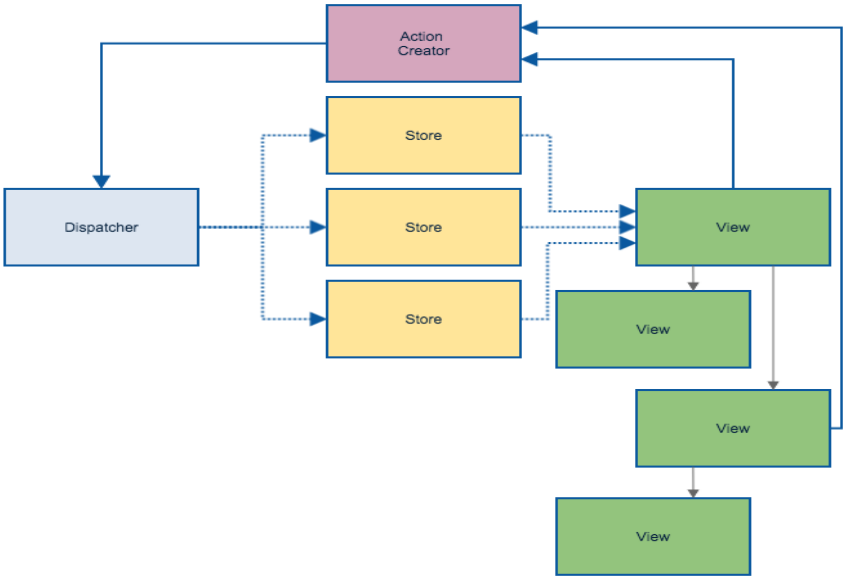

Рис. 4. Схематичне зображення складного додатку Flux-архітектури.

Коли стан оновлюється, він запускає подію зміни. У багатьох додатках React спеціальні розрізи даних (View) (відомі іноді як "Controller-View") відповідають за спостереження за цією подією змін, читання нових даних станів та передачу цих даних через властивості до дочірніх розрізів даних (View). У програмі React для події зміни стану, щоб викликати повторний рендеринг подання верхнього рівня, ефективно ре-рендерить всю ієрархію подання (яка ефективно обробляє React). Це повністю дозволяє уникнути складних помилок і проблем з продуктивністю, які можуть виникнути через спроби спостерігати за конкретними змінами властивостей моделей і лише незначно змінювати частини розрізів даних (View) [2, с. 23].

Ключові властивості. Архітектура потоку забезпечує потік даних явним, легко зрозумілим, надає можливість збільшити локалізацію помилок і має наступні властивості:

– Примусова синхронність. Дії (*Actions*) та їх обробники (*Handlers*) всередині сховища синхронні. Усі асинхронні операції мають викликати дію, яка повідомляє системі про результат операції. Хоча дії можуть робити асинхронні запити на сервер (API-call), обробники дій у сховищі в ідеалі цього не зроблять. Це правило робить потік інформації в додатку надзвичайно явним; налагодження помилок у стані застосунку просто включає в себе з'ясування того, яка дія викликала поганий стан, а потім знайти неправильну логіку, яка відповіла на цю дію [3, с. 10].

– Інверсія контролю. Оскільки стани оновлюються всередині сховища у відповідь на дії (а не оновлюються ззовні контролером або подібним модулем), жодна інша частина системи не повинна знати, як змінити стан програми. Вся логіка оновлення стану міститься в самому сховищі. І, оскільки сховище тільки коли-небудь оновлюються у відповідь на дії і тільки синхронно, то під час тестування сховища постає питання: поставити їх у початковий стан (INITIAL_STATE), надіслати їм дію та перевірити, чи змінився правильно стан.

– Семантичні дії. Оскільки сховище повинно оновлювати себе у відповідь на дії, то дії, як правило, є семантично описові. Наприклад, у Flux-програмі форуму, щоб позначити повідомлення як прочитане, можна надіслати дію з відповідним типом MARK_THREAD_READ. Дія (і компонент, що генерує дію) не знає, *як* виконати оновлення, але *описує,* що він хоче, щоб сталося. Через цю властивість рідко доводиться змінювати типи дій, лише те, як на них реагує сховище. До тих пір, поки ваша програма має концепцію "потоку" і у вас є кнопка або інша взаємодія, яка повинна позначати повідомлення як прочитане, тип дії семантично дійсний [10, с. 116].

– Без каскадних дій. Потік забороняє відправку другої дії в результаті відправки дії. Це допомагає запобігти каскадним оновленням, які важко налагоджувати, і допомагає вам думати про взаємодію у вашому додатку з точки зору семантичних дій.

Реалізація Flux-архітектури. На сьогоднішній день є багато реалізацій Flux-архітектури. Ось найпоширеніші з них:

– Redux – це бібліотека JavaScript з відкритим кодом для керування станом програми. Найчастіше використовується в поєднанні з React або Angular для розвитку клієнтської частини. Містить ряд інструментів, які значно полегшують передачу даних сховища через контекст.

– Fluxxor – це набір інструментів для полегшення побудови шарів даних JavaScript за допомогою Flux-архітектури шляхом об'єднання багатьох основних Flux-концепцій. Він особливо добре працює в поєднанні з React і містить кілька помічників, щоб полегшити інтеграцію з додатками React.

– MobX – це автономна бібліотека для управління front-end станом програми. MobX забезпечує узгодженість внутрішнього стану інтерфейсного додатку, надаючи зручні інструменти для його зміни. Спрощено, MobX дозволяє реалізувати ланцюжок: «Виконати дію» → «Змінити стан» → «Змінити вигляд». При цьому зміни відбуваються атомарно і автоматично – в результаті гарантовано, що не буде моменту, коли стан буде не узгодженим [11, с. 123].

– VueX – шаблон керування станом застосунку + бібліотека для Vue.js додатків. Він служить централізованим магазином для всіх компонентів у додатку, з правилами, які гарантують, що стан може бути мутований лише передбачуваним чином.

– NgRx – це фреймворк для побудови реактивних додатків в Angular. NgRx надає бібліотеки для: управління глобальним та локальним станом, ізоляція побічних ефектів для просування більш чистої архітектури компонентів, управління зборами сутностей, інтеграції з маршрутизатором. А також є інструментом, який покращує досвід розробників при створенні багатьох різних типів додатків.

Приклад. Створимо додаток (візуальна частина не входить до розгляду статті):

– Створимо пустий проект та виконаємо команду *create-react-app*

– За допомогою NPM інсталюємо ще низку бібліотек (рис. 5):

```
"dependencies": {
  "ramda": "^0.27.1",     // Набір корисних функцій
  "react": "^17.0.2",     // Реакт вже встановленний командою create-react-app
  "react-dom": "^17.0.2",  // Реакт-дом вже встановленний командою create-react-app
  "react-scripts": "4.0.3",  // Реакт-скриптс вже встановленний командою create-react-app
  "react-redux": "^7.2.5",   // бібліотека для зв'язки Реакт та Редакс
  "redux": "^4.1.1",         // Redux.
  "redux-thunk": "^2.3.0",   // бібліотека для можливості зв'язки із сервером.
  "reselect": "^4.0.0",      // бібліотека для зручності написання селекторів
},
```

Рис. 5. Список залежностей додатку.

– Створимо всі файли, які нам знадобляться (рис. 6):

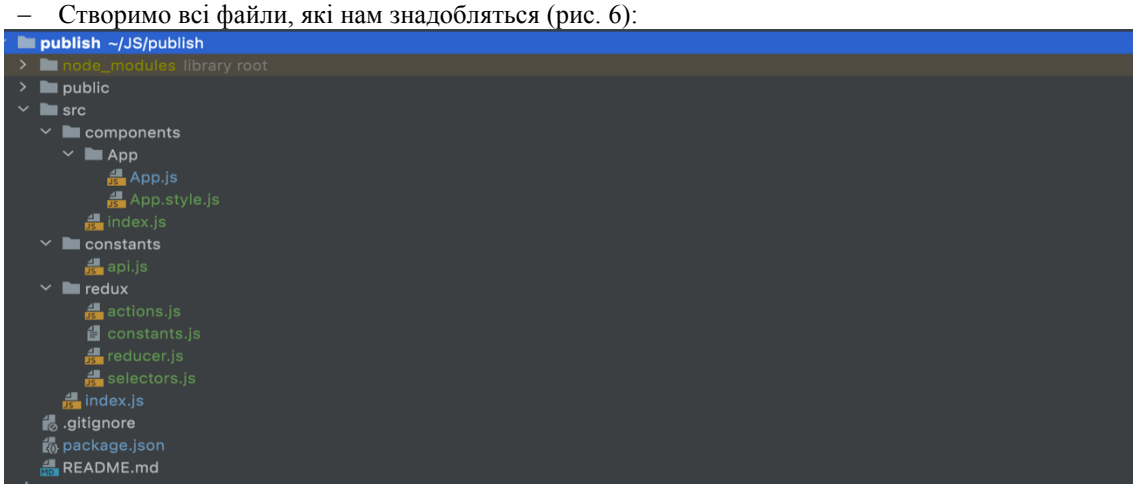

Рис. 6. Структура проекту.

– У файлі *constant.js* (рис. 7) опишемо усі потрібні нам типи дій.

```
export default {
  TODO_FETCH_LIST: '/todo/fetch/list',          // Запит на отримання данних із серверу
  TODO_FETCH_SUCCESS: '/todo/fetch/success',    // Дія у разі вдалого з'єднання
  TODO_FETCH_FAILURE: '/todo/fetch/failure',    // Дія у разі помілки з'єднання
  TODO_CREATE_ITEM: '/todo/create/item',        // Запит на створення данних на сервері
};
```

Рис. 7. constants.js.

– У файлі *reduser.js* (рис. 8) Створимо сховище, де будуть зберігатися дані, помилки та флаг загрузки.

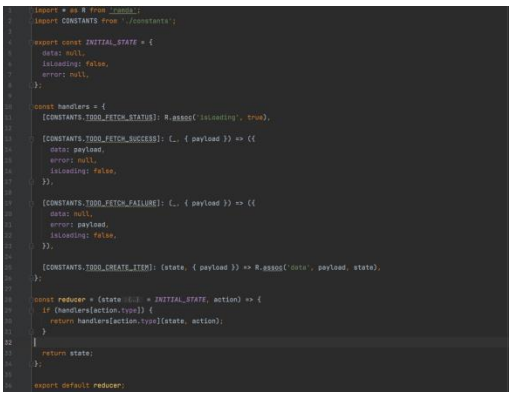

Рис. 8. reducer.js.

Також опишемо маніпуляції згідно з типами. Коли відправлено запит на сервер *isLoading: true*, коли сервер відповів коректно, то у *data* записуємо відповідь серверу, а якщо сталося помилка, тоді у *error* записуємо повідомлення помилки. У разі невідомої дії сховище повертається до первісного стану *(INITIAL_STATE).*

– У файлі actions.js (рис. 9) опишемо дві дії. Запит серверу на отримання, та на зміну.

```
import CONSTANTS from './constants';
import API from '../constants/api'

export const fetchTodoList = () => async dispatch => {
  dispatch({
    type: CONSTANTS.CREATE_FETCH_STATUS,
  });

  try {
    const clients = await API.get("todo/");
    dispatch({
      type: CONSTANTS.TODO_FETCH_SUCCESS,
      payload: clients.data,
    });
  } catch (error) {
    dispatch({
      type: CONSTANTS.TODO_FETCH_FAILURE,
      payload: error,
    });
  }
}

export const createTodo = todo => async dispatch => {
  dispatch({
```

Рис. 9. actions.js.

Висновки. В результаті отримано веб-застосунок із відокремленою графічною частиною та логічною.

За допомогою властивості *__REDUX_DEVTOOLS_EXTENSION_COMPOSE__* (рис. 10) можна спостерігати за історією виклику дій та зміною станів на кожному кроці життєвого циклу додатку. А це означає, що якщо десь розробник допустить помилку, побачити її буде у рази легше. На скріншоті видно, що після запиту на сервер викликається дія *'/todo/fetch/failure'* оскільки зазначеної властивості не існує.

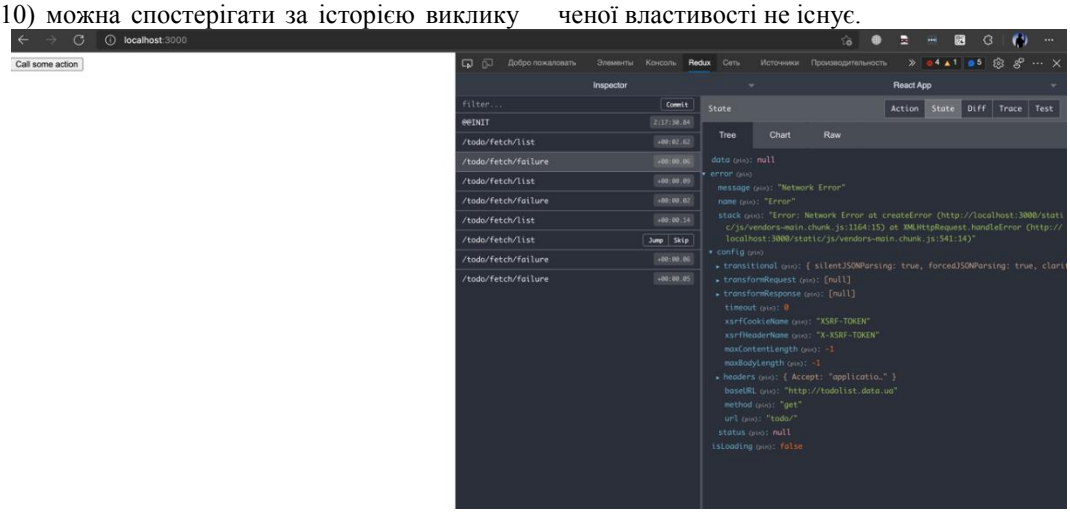

Рис. 9. Середа розробки Redux у браузері.

СПИСОК ЛІТЕРАТУРИ:

1. Abramov D. Getting Started with Redux [Електронний ресурс] / Dan Abramov. – 2021. – Режим доступу до ресурсу: https://redux.js.org/introduction/getting-started.

2. Tilley M. What is Flux [Електронний ресурс] / Michelle Tilley. – 2014. – Режим доступу до ресурсу: http://fluxxor.com/what-is-flux.html.

3. Flux и Redux [Електронний ресурс] // Metanit. – 2021. – Режим доступу до ресурсу: https://metanit.com/web/react/5.1.php.

4. Catalin V. Изучаем архитектуру Flux в React [Електронний ресурс] / Vasile Catalin. – 2017. – Режим доступу до ресурсу: https://code.tutsplus.com/ru/tutorials/getting-started-with-the-flux-architecture-in-react--cms-28906.

5. Yangshun T. In-Depth Overview [Електронний ресурс] / Tay Yangshun. – 2019. – Режим доступу до ресурсу: https://facebook.github.io/flux/docs/in-depth-overview.

6. MobX [Електронний ресурс] – Режим доступу до ресурсу: https://mobx.js.org/README.html.

7. Библиотека MobX [Електронний ресурс] // ТМ «Web Creator» – Режим доступу до ресурсу: https://web-creator.ru/technologies/webdev/mobx.

8. What is Vuex [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://next.vuex.vuejs.org/.

9. What is NgRx [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://ngrx.io/docs.

10. Бодух А. Flux Architecture / Адам Бодух., 2016. – 352 с.

11. Garreau M. Redux in Action / M. Garreau, W. Faurot., 2018. – 312 с.