УДК 004. 8:004.9 **doi: 10.26906/SUNZ.2025.4.78-81**

О. Ю. Заковоротний, О. А. Сапальський

Національний технічний університет “Харківський політехнічний інститут”, Харків, Україна

ПРОБЛЕМИ DATA-DRIVEN ПІДХОДУ У ФРОНТЕНД-РОЗРОБЦІ

Анотація. Data-Driven-підхід став домінуючою практикою у фронтенд-розробці. Попри зручність декларативного підходу, ускладнення систем призвело до виявлення ряду архітектурних недоліків. Основними з них є втрата прозорості джерел змін, асинхронні конфлікти при оновленні стану, а також відсутність семантичного контексту подій. У статті порівнюються Data-Driven та Event-Driven підходи, досліджуються ключові проблеми першого і пропонуються практичні рішення, здатні покращити контроль над логікою застосунків. **Мета** цієї роботи є виявлення типових архітектурних недоліків, властивих Data-Driven моделі в клієнтській розробці, аналіз причин їх виникнення та розробка концептуальних і технічних шляхів мінімізації відповідних ризиків. Особливу увагу приділено проблемам неявної мутації стану, втрати контексту змін, синхронізації запитів і залежності від життєвого циклу компонентів у фреймворк-орієнтованих системах. **Отримані наступні результати:** У результаті проведеного дослідження було встановлено, що Data-Driven підхід у складних застосунках не забезпечує достатньої контрольованості над джерелами змін. Також було доведено, що навіть при використанні інструментів типу Redux DevTools або React Developer Tools розробник часто не має повної картини змін стану, оскільки вони відбуваються в різних точках системи без єдиного шляху контролю. Встановлено, що найбільш ефективними компенсаторними підходами є створення шару семантичних подій, централізація мутацій, а також комбінування реактивного моделювання з декларативним представленням. **Висновки.** Data-Driven архітектура значно спрощує побудову UI в умовах простих або середньої складності проектів. Проте при зростанні кількості джерел стану, складності взаємозв’язків між компонентами і високому рівні асинхронності така модель демонструє структурні обмеження. У таких умовах доцільним є перехід до гібридних рішень, які поєднують Data-Driven рендеринг з Event-Driven семантикою та контролем через єдині точки мутації.

Ключові слова: Data-Driven, Event-Driven, frontend, стан, мутація, асинхронність, реактивність.

Вступ

У сучасній фронтенд-розробці поширений Data-Driven підхід, за яким інтерфейс користувача визначається внутрішнім станом даних. Цю ідею часто формулюють рівнянням UI = f(state) – інтерфейс є функцією від стану програми [1]. Зміни в стані автоматично відображаються у UI (реактивність), що звільняє розробника від ручного оновлення DOM-елементів. Такий декларативний підхід лежить в основі бібліотек на кшталт React.

Натомість Event-Driven підхід побудовано навколо подій: логіка додатка реагує на дискретні події користувача або системи.

У Event-Driven архітектурі стан визначається послідовністю подій, а не навпаки.

Іншими словами, у Data-Driven сценарії події виступають лише наслідком зміни даних, тоді як у Event-Driven сценарії вони самі керують змінами стану і поведінкою UI [2].

Кожен підхід має свої переваги, але й породжує унікальні архітектурні проблеми. У статті розглянуто три ключові проблеми Data-Driven підходу у фронтенд-розробці та можливі шляхи їх вирішення.

Постановка проблеми

У Data-Driven системі оновлення стану можуть відбуватися неявно з різних місць, що ускладнює відстеження джерела змін. Якщо кожен компонент або модуль «вільно» модифікує спільний стан, потік даних стає непередбачуваним. Без єдиного центру керування важко відповісти на запитання: що саме спричинило зміну стану? Як зазначають розробники, якщо всі компоненти будуть змінювати state як заманеться, в додатку може виникнути хаотична ситуація. JavaScript виконує код переважно асинхронно, тож різні частини програми теоретично можуть намагатися змінити одну і ту ж змінну одночасно, що призводить до умов гонки та непередбачених багів. Навіть коли конкурентні зміни трапляються рідко, сама можливість неявних мутацій ускладнює підтримку: розробник не впевнений, який код і коли саме змінив стан.

У реактивному інтерфейсі часто паралельно відбуваються кілька асинхронних операцій, які змінюють стан [3]. Гонки асинхронних оновлень (race conditions) виникають, коли результати таких операцій приходять у невизначеному порядку і накладаються один на одного. У Data-Driven підході UI автоматично відображає останній стан, але “останній” не завжди означає правильний [4]. Класичний сценарій – користувач ініціює два запити до сервера майже одночасно (наприклад, швидко вводить різні пошукові запити). Перший запит може повернутися пізніше другого і перезаписати дані, незважаючи на те, що вже є новіший результат. Як наслідок, інтерфейс може показати неактуальну інформацію, не відповідаючи поточним діям користувача. Розробники зазначають, що якщо існує навіть невелика ймовірність отримати такий стан (невідповідність між намірами користувача і відображеним результатом), додаток вже схильний до race condition-багів [5]. Причини цього явища криються в непередбачуваності мережевих затримок, різному часу виконання промісів та інших асинхронних процесів. Порядок завершення запитів не гарантується: останній запит може виконуватися раніше попереднього або взагалі зазнати помилки. У кращому разі, гонки призводять до мерехтіння застарілих даних в UI; у гіршому – до логічних помилок [6]. В окремих доменах це може мати критичні наслідки: наприклад, користувач може випадково купити не той товар, або лікар – призначити не

той препарат, якщо інтерфейс відобразив несвоєчасну інформацію.

Data-Driven підхід зосереджений на стані, але не фіксує сенс зміни цього стану. Іншими словами, він відповідає на питання «що змінилося?», але не завжди «чому змінилося і що це означає?». Коли оновлення UI відбувається автоматично від зміни даних, втрачається контекст: чи була ця зміна спричинена дією користувача, чи приходом нових даних із сервера, чи внутрішньою подією? Відсутність семантичних міток для змін утруднює як відлагодження, так і розширення функціоналу. Пов'язана проблема – брак централізованого каналу для реакцій на події. У Event-Driven підході зазвичай є шина подій або диспетчер, куди можна підписати різні частини системи на отримання повідомлень про певні події.

У Data-Driven UI такої єдиної шини немає: компоненти безпосередньо реагують на зміну стану (наприклад, через двобічне зв'язування), але якщо потрібно виконати якусь глобальну дію при конкретній зміні – це складно організувати. Іншими словами, бракує центральної точки контролю, де можна було б перехопити та осмислити зміни перед тим, як вони розійдуться по UI.

Метою цієї роботи є виявлення типових архітектурних недоліків, властивих Data-Driven моделі у фронтенд-розробці, аналіз причин їх виникнення та розробка концептуальних і технічних шляхів мінімізації відповідних ризиків.

Особливу увагу приділено проблемам неявної мутації стану, втрати контексту змін, синхронізації запитів і залежності від життєвого циклу компонентів у фреймворк-орієнтованих системах.

Основна частина роботи

Головна причина – відсутність єдиного джерела правди та явних точок входу для змін.

У традиційній Event-Driven моделі кожна значуща зміна ініціюється подією (наприклад, обробник кліку кнопки), і розробник знає, що саме цей обробник змінює стан. В Data-Driven моделі зміна стану може бути спричинена будь-де: HTTP-викликом, результатом промісу, побічним ефектом хуку або сторонньою бібліотекою. Якщо такі зміни не централізовані, з часом код стає важчим для підтримки та тестування.

Один із способів розв'язати проблему – запровадити сувору дисципліну оновлення стану через визначені інтерфейси (рис. 1). У централізованих сховищах стану типу Redux або Vuex всі зміни проходять через спеціальні методи (екшени/мутації), що робить їх явними.

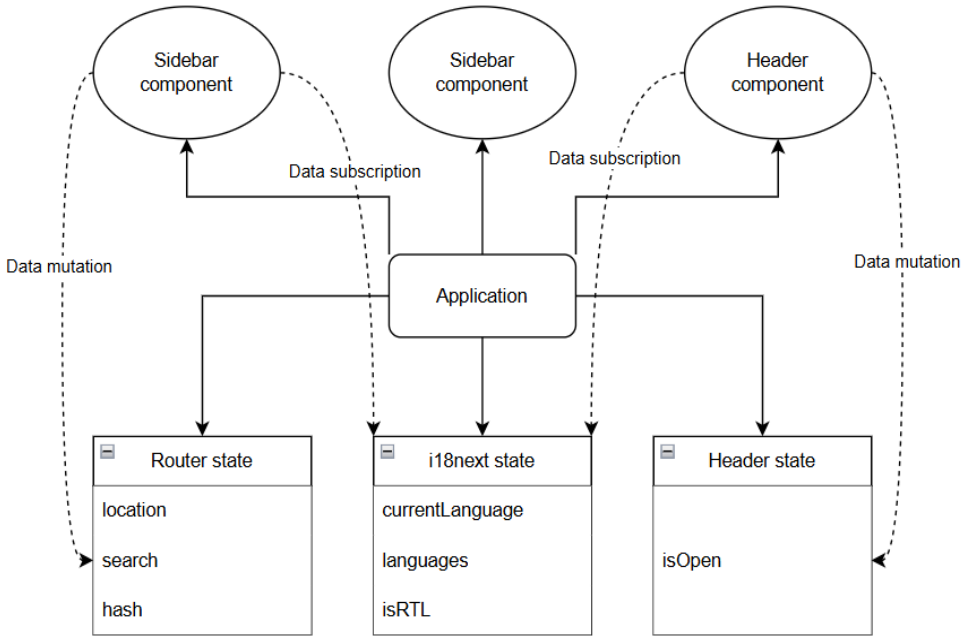

Рис. 1. Приклад схеми використання даних у додатку

Наприклад, у Redux прийнято ніколи не змінювати стан напряму – натомість, слід відправляти action з описом змін, який обробляється reducer-функцією. Такий уніфікований потік (state → action → reducer → new state) гарантує, що кожна зміна здійснюється під контролем розробника і може бути відстежена у коді [7]. Аналогічно, у Vuex рекомендовано змінювати стан лише через коміти мутацій – дотримання цього правила означає, що Ви завжди знатимете, де саме у застосунку змінюються дані. Іншими словами, потрібно уніфікувати джерело змін. Практично це реалізується через централізований стор або керування станом, куди всі компоненти відправляють запити на зміну (події або екшени) замість прямої мутації. Таким чином усувається "невидимість" змін: кожна модифікація проходить через явний канал, що спрощує відлагодження і виключає побічні ефекти.

Основна причина гонок – необмежений паралелізм оновлень. Data-Driven підхід передбачає, що UI завжди відповідає актуальному стану, але не визначає, як вирішувати конфлікти, коли "актуальність" спірна. В браузері JavaScript виконується в одному потоці, але асинхронні дії (запити, таймери) плануються у черзі і можуть завершуватися у будь-якому порядку. Якщо

на один і той самий фрагмент стану одночасно претендують кілька результатів, стандартний механізм просто відображає останній записаний результат – який може виявитися застарілим. Race condition-помилки важко виявити під час розробки, бо на локальному швидкому з'єднанні і послідовних діях користувача вони майже не проявляються. Проблема проявляється у реальних умовах: при повільній чи нестабільній мережі, високому навантаженні серверу або просто при дуже швидких діях користувача. Отже, потрібно спеціально проєктувати фронтенд-логіку, щоб запобігати таким конкурентним ситуаціям. Таким чином, на практиці можна ігнорувати застарілі запити за допомогою спеціальних міток timestamp. Як альтернатива можливе гнучке управління запитами за допомогою класу AbortControler, який дозволяє скасовувати запит або навіть цілі групи у потрібний момент [8].

До Data-Driven архітектури часто додають елементи подієво-орієнтованого керування. Зокрема, впроваджується поняття семантичних дій (action) як доповнення до простої мутації стану.

Замість безпосереднього оновлення даних, компоненти генерують події з бізнес-сенсом (напр., USER_DELETED_ITEM), а система вже вирішує, як змінити стан і які побічні ефекти виконати у відповідь. Такий підхід реалізований у Redux: кожна дія (action) – це простий об’єкт з полем type (семантика змін) та додатковими даними. Важливо, що всі дії проходять через централізований диспетчер (store), де можуть бути зафіксовані та оброблені. Це дає одразу дві переваги: явний журнал змін і можливість розширених реакцій. У Redux/Flux кожна дія може бути записана і відтворена пізніше, що робить систему детермінованою та піддатливою до дебагу – маючи послідовність action-ів, можна завжди відновити певний стан UI. Застосунок стає прозорішим: розробник точно знає, яка подія що змінює, і може прослідкувати історію (наприклад, через Redux DevTools, де всі екшени логуються з можливістю покрокового перегляду). Подібний механізм існує і у Vuex: всі коміти мутацій реєструються, і можна «прокрутити» історію змін стану покроково, оцінюючи, які дії до них призвели [9]. Таким чином досягається семантична трасованість: кожна зміна має мітку і може бути проаналізована централізовано. Подібні методи можуть значно підвищити якість коду та його надійність. Наприклад, нижче можна побачити приклад використання Zustand у зв'язці з Immer (рис. 2). Така комбінація дозволяє писати простий та зрозумілий код, який неможливо мутувати безпосередньо ззовні.

Стан неможливо змінити ззовні. Усі зміни прозорі та централізовані. Налагодження та відтворюваність стають простими. Будь-яка спроба змінити цей стан буде призводити до помилки (рис. 3).

```
const useCounter = create<State & Actions>()(
  immer((set) => ({
    count: 0,
    nested: [{ name: "Alex" }],

    increment() {
      set((state) => {
        debugger; // Tracks any state change
        state.count++;
      });
    },
  }))
);
```

Рис. 2. Захищений стан

```
function App() {
  const counterSlice = useCounter();

  counterSlice.nested[0].name = "Direct mutation";
  console.log(counterSlice.nested[0].name);
```

```
▶Uncaught TypeError: Cannot assign to read only property 'name' of object '#<Object>'    App.tsx:36
    at App (App.tsx:36:26)
    at renderWithHooks (react-dom.development.js:15486:18)
    at updateFunctionComponent (react-dom.development.js:19617:20)
```

Рис. 3. Помилка при оновленні стану

Висновки та перспективи подальших досліджень

Data-Driven підхід у фронтенді значно спрощує побудову інтерфейсів і усуває потребу в імперативній маніпуляції DOM. Однак його впровадження висвітлює низку архітектурних проблем. По-перше, неявні джерела змін стану ускладнюють підтримку – вирішити це можна впровадженням чітких правил оновлення через єдиний канал (екшени, мутації), що робить потік даних прозорим. По-друге, гонки асинхронних оновлень можуть призводити до неконсистентного UI – для їх запобігання застосовують техніки фільтрації та скасування паралельних запитів, гарантуючи обробку тільки актуальних даних. По-третє, брак семантики змін та централізованої системи подій

позбавляє систему контексту і гнучкості – ця проблема розв'язується доповненням Data-Driven моделі шаром подій, де кожна зміна маркується і обробляється узгоджено (як у Flux/Redux архітектурі). Загальний висновок: реактивні інтерфейси доцільно поєднувати з принципами подієвоорієнтованого проектування, щоб отримати найкраще з обох світів. UI = f(state) залишається потужною концепцією, що забезпечує передбачуваний рендеринг , проте навколо неї варто побудувати інфраструктуру для явного керування змінами. Академічні і практичні джерела сходяться на тому, що правильна організація потоків даних і подій підвищує передбачуваність та керованість фронтенд-систем [10]. Дотримуючись описаних підходів – єдиний потік змін, контроль асинхронності, семантичні екшени – розробники можуть знизити складність, мінімізувати баги та полегшити розвиток додатків, заснованих на Data-Driven парадигмі.

Список літератури

1. Flutter. Flutter Architectural Overview. Flutter Documentation. URL: https://docs.flutter.dev/resources/architectural-overview
2. Beyer D., Ghanbari H., Hasselbring W. An empirical characterization of event-sourced systems and their schema evolution—Lessons from industry. Journal of Systems and Software. 2021. Vol. 182:110931. DOI: 10.1016/j.jss.2021.110931. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000674
3. Wikipedia. Reactive programming. URL: https://en.wikipedia.org/wiki/Reactive_programming
4. Yee M.-H., Badouraly A., Lhoták O., Tip F., Vitek J. Precise Dataflow Analysis of Event-Driven Applications. arXiv. DOI: 10.48550/arXiv.1910.12935. URL: https://arxiv.org/abs/1910.12935
5. E. Mutlu et al. Detecting JavaScript Races that Matter. FSE. URL: https://www.doc.ic.ac.uk/~livshits/papers/pdf/fse15.pdf
6. MDN. JavaScript execution model. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Execution_model
7. Redux. Thinking in Redux: Three Principles. Redux Documentation. URL: https://redux.js.org/understanding/thinking-in-redux/three-principles
8. WHATWG. DOM Standard — Aborting ongoing activities (AbortController). WHATWG. URL: https://dom.spec.whatwg.org/
9. Vuex. Committing Mutations in Components. Vuex Guide – Mutations. URL: https://vuex.vuejs.org/guide/mutations.html#committing-mutations-in-components
10. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2003. Розділ 5, с. 63, available at: https://martinfowler.com/books/eaa.html

Received (Надійшла) 12.08.2025
Accepted for publication (Прийнята до друку) 22.10.2025

Відомості про авторів / About the Authors

Заковоротний Олександр Юрійович – доктор технічних наук, професор, завідувач кафедри комп'ютерної інженерії та програмування, Національний технічний університет «Харківський політехнічний інститут», Харків, Україна;
Oleksandr Zakovorotnyi – Doctor of Technical Sciences, Professor, Head of the Computer Engineering and Programming Department, National Technical University «Kharkiv Polytechnic Institute», Kharkiv, Ukraine;
e-mail: Oleksandr.Zakovorotnyi@khpi.edu.ua; ORCID Author ID: https://orcid.org/0000-0003-4415-838X;
Scopus Author ID: https://www.scopus.com/authid/detail.uri?authorId=57201613700.

Сапальський Олександр Андрійович – аспірант кафедри "Комп'ютерна інженерія та програмування", Національний технічний університет "Харківський політехнічний інститут", Харків, Україна;
Oleksandr Sapalskyi – PhD student, Department of Computer Engineering and Programming, National Technical University 'Kharkiv Polytechnic Institute', Kharkiv, Ukraine;
e-mail: Oleksandr.Sapalskyi@cs.khpi.edu.ua; ORCID Author ID: https://orcid.org/0009-0002-5749-8527.

Solving productivity problems using asynchronous methods in JavaScript

Oleksandr Zakovorotnyi, Oleskandr Sapalskyi

Abstract. The Data-Driven approach, in which the interface is defined as a function of the current state (UI = f(state)), has become the dominant practice in front-end development. Despite the convenience of the declarative approach, the complexity of systems has led to the identification of a number of architectural shortcomings. The main ones are the loss of transparency of the sources of changes, asynchronous conflicts when updating the state, and the lack of semantic context of events. The article compares the Data-Driven and Event-Driven approaches, explores the key problems of the former, and offers practical solutions that can improve control over application logic. The **purpose** of this work is to identify typical architectural shortcomings inherent in the Data-Driven model in client-side development, analyze the causes of their occurrence, and develop conceptual and technical ways to minimize the corresponding risks. Special attention is paid to the problems of implicit state mutation, loss of change context, query synchronization, and dependence on the component life cycle in framework-oriented systems. The following **results were obtained**: As a result of the study, it was found that the Data-Driven approach in complex applications does not provide sufficient control over the sources of changes. It was also proven that even when using tools such as Redux DevTools or React Developer Tools, the developer often does not have a complete picture of state changes, since they occur at different points in the system without a single control path. It was found that the most effective compensatory approaches are the creation of a semantic event layer, centralization of mutations, and the combination of reactive modeling with declarative representation. **Conclusions**. Data-Driven architecture significantly simplifies UI construction in the conditions of simple or medium-complexity projects. However, with an increase in the number of state sources, the complexity of the relationships between components, and a high level of asynchrony, such a model demonstrates structural limitations. In such conditions, it is advisable to switch to hybrid solutions that combine Data-Driven rendering with Event-Driven semantics and control through single mutation points.

Keywords: Data-Driven, Event-Driven, frontend, state, mutation, asynchrony, reactivity.