

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б15-мм

# Унификация хранения данных результата задачи в Desbordante

***ЕРМОЛОВИЧ Анна Сергеевна***

Отчёт по учебной практике  
в форме «Производственное задание»

Научный руководитель:  
Ассистент кафедры ИАС, Г.А. Чернышев

Санкт-Петербург  
2024

# Оглавление

|                                                                              |           |
|------------------------------------------------------------------------------|-----------|
| <b>Введение</b>                                                              | <b>3</b>  |
| <b>1. Постановка задачи</b>                                                  | <b>5</b>  |
| <b>2. Обзор</b>                                                              | <b>6</b>  |
| 2.1. Существующее решение . . . . .                                          | 6         |
| 2.2. Используемые термины и технологии . . . . .                             | 7         |
| <b>3. Реализация</b>                                                         | <b>10</b> |
| 3.1. Новая структура файлов . . . . .                                        | 10        |
| 3.2. Алгоритм для минимизации количества запросов к базе<br>данных . . . . . | 10        |
| 3.3. Создание хранилища для FD . . . . .                                     | 11        |
| 3.4. Создание хранилищ для остальных примитивов . . . . .                    | 11        |
| 3.5. Апробация . . . . .                                                     | 12        |
| 3.6. Результаты . . . . .                                                    | 12        |
| <b>Заключение</b>                                                            | <b>13</b> |
| <b>Список литературы</b>                                                     | <b>14</b> |

# Введение

Профилирование данных представляет собой процесс получения метаданных из исходных данных. Очевидно, что в зависимости от выбранного подхода в качестве характеристик могут выступать самые разные показатели, что порождает задачу выделения именно тех признаков, которые важны для исследователя. Одновременно с этим, огромные объемы данных, хранящиеся в различных базах, привели к необходимости разработки алгоритмов и методов профилирования. Одним из примеров является открытая платформа Metanome, реализованная на языке Java — совместный проект Hasso Plattner Institute и Qatar Computing Research Institute, стартовавший в 2015 году. Metanome включает широкий спектр алгоритмов для профилирования, в том числе для поиска функциональных зависимостей. В качестве более современной альтернативы был разработан проект Desbordante [1]. В сравнении с предшественником, Desbordante работает быстрее и эффективнее в использовании памяти, благодаря тому, что написан на языке C++. Основное преимущество Desbordante заключается в способности выявлять сложные математические закономерности, или так называемые примитивы, под которыми подразумеваются правила, описывающие работу с данными или их частями, сформулированные на основе математических методов. Для привлечения пользователей, помимо ядра на C++, в Desbordante был разработан веб-интерфейс, позволяющий протестировать работу примитивов на сайте.

В зависимости от примитива результат может иметь разные представления, для которых нужна различная информация из базы данных. В текущей веб-версии информация о статусе нахождения математических закономерностей получается в нескольких местах кода. Например, часть данных о задаче хранится в TaskContext, и часть этих данных статически SSR-ится на сервере и отображается в одном из табов интерфейса. В то же время другая часть данных подтягивается уже на клиенте с помощью хука useQuery. Дополнительно, информация о MFD хранится в MFDAtom, что создает еще одно место, где происходит рабо-

та с данными задачи. Такая разрозненность усложняет поддержку кода и интеграцию новых данных. В связи с чем появилась необходимость унифицировать хранение данных о задаче. Об этом и пойдет речь в настоящей работе.

# 1. Постановка задачи

Целью работы является унификация хранения данных о задаче в Desbordante. Для её выполнения были поставлены следующие задачи:

1. разработать алгоритм для минимизации количества запросов к базе данных;
2. разделить существующий TaskContext на отдельные jotai-хранилища для примитивов и реализовать логику получения данных с сервера;
3. переработать текущую структуру файлов;

## 2. Обзор

### 2.1. Существующее решение

#### 2.1.1. Текущая реализация

В актуальной версии кода данные о результате задачи хранятся в разрозненных местах, что создает существенные сложности при разработке и поддержке. Часть информации находится в `TaskContext`, выступая в роли глобального состояния на уровне приложения, что приводит к избыточности и проблемам с актуальностью данных. Некоторые данные рендерятся на стороне сервера через SSR и передаются в клиентские компоненты, что усложняет синхронизацию между сервером и клиентом. Ещё одна часть данных загружается динамически на клиенте с использованием библиотек, таких как `React Query`, что, хотя и обеспечивает гибкость, ещё больше разобщает общий процесс работы с данными. Кроме того, информация о примитиве `Metric Dependency Verification` хранится в отдельном атоме `Jotai — MFDAtom`.

Такая разрозненность подходов к хранению данных приводит к созданию перегруженного компонента. Например, в системе существует единый компонент для работы с табами — `TaskContext`, который используется почти во всех примитивах, из-за чего компонент содержит данные сразу для всех возможных примитивов, когда на самом деле требуется информация только для одного конкретного. Это требовало постоянной доработки логики компонента, чтобы он соответствовал нуждам каждого примитива. Чтобы не перегружать уже и без того объемный `TaskContext`, для `Metric Dependency Verification` было решено создать отдельный компонент.

Таким образом, существующий подход к хранению данных вызывал дублирование логики, увеличивал сложность интеграции новых примитивов и усложнял разработку пользовательского интерфейса. Унификация хранения данных результата задачи поможет решить эти проблемы.

### 2.1.2. Выбор библиотеки

При выборе библиотеки для управления состоянием было рассмотрено несколько альтернатив, включая встроенные в React `useState` и `useContext`, а также внешние библиотеки, такие как `Redux`, `Zustand` и `Jotai`. Каждый из этих инструментов имеет свои преимущества и недостатки. `useState` и `useContext` хорошо подходят для локального и простого состояния, однако они усложняют управление разделяемым состоянием и масштабируемость приложения. `Redux` имеет высокую сложность настройки и избыточный объем кода для небольших задач. `Zustand` прост в использовании, но больше ориентирован на глобальное состояние, а не на модульное управление.

В итоге была выбрана библиотека `Jotai`, так как она сочетает простоту и гибкость. `Jotai` позволяет определять независимые атомы состояния, которые можно переиспользовать и передавать между компонентами без необходимости пропускать информацию через контекст или пропсы. Она обеспечивает реактивность, эффективное обновление компонентов и удобную работу с асинхронными данными. Это делает `Jotai` оптимальным выбором для управления полученными данными в проекте, где требуется модульность и возможность масштабирования. А также `Jotai` уже был использован в проекте и существующие атомы можно будет переиспользовать.

### 2.1.3. Вывод

Предлагается перейти на атомарное хранение состояния с использованием `Jotai`, где каждому примитиву будет написано отдельное хранилище с интегрированной логикой получения данных с сервера.

## 2.2. Используемые термины и технологии

### 2.2.1. Jotai

`Jotai` [2] —библиотека для управления состоянием в React-приложениях, основанная на атомарной модели данных. Она позволяет разрабаты-

кам определять минимальные единицы состояния (атомы) и эффективно управлять их взаимодействием. Jotai обеспечивает декларативный и гибкий подход к работе с состоянием, минимизируя сложности, связанные с глобальными состояниями, и повышая производительность за счёт оптимальных перерендеров.

### **2.2.2. Jotai atom**

Jotai atom [3] — базовая единица состояния в Jotai. Атом представляет собой неизменяемую сущность, которая может хранить любое значение (примитивы, объекты, функции и т.д.) и быть подписана на изменения. Атомы позволяют создавать независимые и реактивные данные, которые могут комбинироваться для построения сложной логики приложения.

### **2.2.3. GraphQL**

GraphQL — это язык запросов для API и среда выполнения для обработки запросов, разработанная для обеспечения гибкого и эффективного взаимодействия клиента с сервером. Он позволяет клиенту самостоятельно определять, какие данные ему нужны, минимизируя объем передаваемой информации.

### **2.2.4. Apollo Client**

Apollo Client — библиотека для работы с GraphQL на стороне клиента, обеспечивающая управление состоянием данных, кеширование запросов и упрощение взаимодействия с сервером. Она предоставляет удобный API для отправки GraphQL-запросов и мутаций, а также автоматически обрабатывает обновление локального состояния приложения на основе изменений данных.

### **2.2.5. React**

React [5] — открытая JavaScript-библиотека для создания пользовательских интерфейсов, которая позволяет создавать эффективные и

масштабируемые веб-приложения, используя компонентный подход.

### **2.2.6. Next.js**

Next.js [4] — фреймворк для серверного и статического рендеринга, основанный на базе React, который предлагает возможности построения универсальных приложений, оптимизацию маршрутизации и улучшенное SEO.

### **2.2.7. TypeScript**

TypeScript [6] — это статически типизированный язык программирования, расширяющий синтаксис JavaScript. Он предоставляет возможность определения типов переменных, параметров функций и возвращаемых значений, что позволяет обнаруживать и предотвращать ошибки на этапе разработки.

## 3. Реализация

Реализация была поделена на несколько этапов:

### 3.1. Новая структура файлов

Предлагается создать новую структуру файлов, где каждый примитив будет иметь собственный файл `store.ts`, в котором будут собираться все необходимые для рендеринга этого примитива данные. Также было решено создать отдельное хранилище для вкладок, использующихся сразу несколькими примитивами.

Чтобы не перегружать файлы `store` было решено оставить в них только определение атомов, а хуки для каждой вкладки вынести в директорию `hooks`.

### 3.2. Алгоритм для минимизации количества запросов к базе данных

Теперь каждый раз при переключении на вкладку будет заново вызываться хук с получением данных, из-за чего появляется проблема дублирования запросов. Для решения этой проблемы было решено использовать комбинированный подход для минимизации количества запросов к базе данных, включающий условное выполнение запросов, кэширование, и управление состоянием. Условный пропуск (`skip`) в GraphQL предотвращает отправку запросов, если данные уже загружены или условия для их получения не выполнены. Встроенное кэширование Apollo Client обеспечивает повторное использование ранее полученных данных, исключая избыточные запросы. Состояние управляется через атомы Jotai, которые отслеживают необходимость загрузки данных на основе их наличия или изменений контекста (например, смена `taskID`). Эти механизмы работают совместно, оптимизируя запросы, исключая дублирование и снижая нагрузку на сервер, что критически важно для высокодинамичных приложений.

### 3.3. Создание хранилища для FD

В рамках унификации и оптимизации архитектуры был разработан файл `store.ts`, который заменил ранее используемые контексты и состояния, связанные с примитивами задач.

Были созданы атомы `Jotai` для управления ключевыми аспектами состояния:

1. `taskIDAtom` для хранения идентификатора задачи;
2. `defaultDataAtom` для основного набора данных задачи;
3. `dependenciesFilterAtom` для хранения фильтров зависимостей;
4. `selectedDependencyAtom` и `errorDependencyAtom` для управления выбранными зависимостями и обработкой ошибок;
5. Атомы для работы с графическими представлениями (`datasetAtom`, `pieChartDataAtom`, `pieChartLoadingAtom`) и для общего состояния задачи (`taskInfoAtom`)

В директорию `hooks` были добавлены хуки для выполнения запросов данных с сервера. Использование `useQuery` в связке с `skip` позволило минимизировать количество запросов, отправляемых к серверу, загружая данные только при необходимости. Для управления состоянием загрузки и обновления данных атомы обновляются непосредственно в эффектах `React`, которые отслеживают результат запросов. Логика для `FD` была вынесена в отдельные хуки (`useFDStatistics`, `useFDPrimitiveList`), где каждая задача (например, получение данных, управление графиками) реализуется через отдельные атомы.

### 3.4. Создание хранилищ для остальных примитивов

Для общих табов было создано хранилище путем переиспользования хуков из ранее разработанного `store.ts`. Это упрощает добавление и поддержку новых примитивов, таких как `AR`, `CFD`, так как они используют, уже реализованные в процессе создания хранилища для `FD`, хуки.

После этого были разработаны хуки под уникальные вкладки MFD и ТуроFD.

### **3.5. Апробация**

Чтобы проверить работоспособность хранилищ и корректность получения данных было проведено ручное тестирование. Одинаковые задачи были запущены на официальном сайте Desbordante и локально, получившиеся веб-страницы сравнивались между собой. Таким образом была проверена каждая вкладка каждого примитива.

### **3.6. Результаты**

Для всех пяти примитивов были созданы унифицированные хранилища. Предоставлена новая структура данных. Также было проведено ручное тестирование, в ходе которого не было выявлено проблем.

# Заключение

По итогам данной работы была произведена унификация хранения данных о задаче в Desbordante, а именно:

- разработан алгоритм для минимизации количества запросов к базе данных;
- TaskContext разделен на отдельные jotai-хранилища для примитивов и реализована логика получения данных с сервера;;
- переработана текущая структура файлов;

Ссылка на GitHub-репозиторий<sup>1</sup>, ссылка на pull request<sup>2</sup>.

---

<sup>1</sup><https://github.com/Anna-er/desbordante-web/tree/refactoring-work>

<sup>2</sup><https://github.com/Anna-er/desbordante-web/pull/1>

## Список литературы

- [1] Desbordante. — URL: <https://desbordante.unidata-platform.ru/> (дата обращения: 19 ноября 2024 г.).
- [2] Jotai. — URL: <https://jotai.org/> (дата обращения: 19 ноября 2024 г.).
- [3] Jotai atom. — URL: <https://jotai.org/docs/core/atom> (дата обращения: 19 ноября 2024 г.).
- [4] Next.js by Vercel - The React Framework. — URL: <https://nextjs.org/docs> (дата обращения: 19 ноября 2024 г.).
- [5] React. — URL: <https://react.dev/> (дата обращения: 19 ноября 2024 г.).
- [6] TypeScript Documentation. — URL: <https://www.typescriptlang.org/docs/> (дата обращения: 19 ноября 2024 г.).