ORES
Научно-практический электронный журнал
Оригинальные исследования (ОРИС), *ISSN 2222-0402*
www.ores.su
original-research.ru

ОДИН РАЗРАБОТЧИК — ВЕСЬ ФРОНТЕНД: ПРОЕКТИРОВАНИЕ, РЕАЛИЗАЦИЯ И ОПТИМИЗАЦИЯ ИНТЕРФЕЙСА

Ноздрин Георгий Денисович,
React frontend-разработчик, студент 4 курса направления «Информационные системы и технологии» университета НИТУ «МИСИС»

Аннотация

Разработка масштабного клиентского приложения (SPA) силами одного специалиста предоставляет широкую архитектурную свободу, но кратно повышает риск накопления технического долга. В настоящей статье представлен практический опыт проектирования, реализации и ввода в эксплуатацию клиентской части HR-платформы (соискатели, работодатели, вакансии, чаты). Рассматриваются вопросы выбора технологического стека, адаптации методологии Feature-Sliced Design, управления модальными окнами, централизованной работы с API и глобального перехвата ошибок. В статье представлены готовые архитектурные паттерны, которые могут быть полезны при создании аналогичных SPA-решений. В финале приводятся результаты успешной эксплуатации системы в условиях пиковых нагрузок.

Ключевые слова: frontend, SPA, React, TypeScript, Vite, Feature-Sliced Design, Redux, Cypress, оптимизация, развертывание веб-приложений.

ONE DEVELOPER — THE ENTIRE FRONTEND: DESIGN, IMPLEMENTATION AND OPTIMIZATION OF THE INTERFACE

Nozdrin George Denisovich,
React frontend developer, 4th year student in Information Systems and Technologies at NUST MISIS University
george.nozdrin@yandex.ru

ABSTRACT

The development of a large-scale client application (SPA) by a single specialist provides wide architectural freedom, but significantly increases the risk of accumulation of technical debt. This article presents practical experience in the design, implementation and commissioning of the client part of the HR platform (job seekers, employers, vacancies, chats). The issues of choosing a technology stack, adapting the Feature-Sliced Design methodology, managing modal windows, centralized API work, and global error interception are considered. The article presents ready-made architectural patterns that can be useful when creating similar SPA solutions. In the final, the results of successful operation of the system under peak load conditions are presented.

Keywords: frontend, SPA, React, TypeScript, Vite, Feature-Sliced Design, Redux, Cypress, optimization, web application deployment.

Введение

Традиционный подход к созданию сложных веб-приложений подразумевает наличие кросс-функциональной команды, где задачи распределяются между системным архитектором, разработчиками бизнес-логики и инженерами по тестированию (QA). Однако в условиях стартапов или специфики бизнес-процессов разработка всего клиентского слоя может быть делегирована одному специалисту.

В рамках описываемого проекта стояла задача разработать клиентскую часть HR-платформы. Продукт включал в себя:

- Личные кабинеты двух категорий пользователей (кандидатов и работодателей).
- Систему поиска, публикации и импорта вакансий.
- Внутренний чат реального времени.
- Информационные разделы (публикации, инструкции).

Индивидуальная разработка лишает проект многоэтапного код-ревью и дискуссий о стандартах кодирования, что повышает ответственность за каждое техническое решение. Ошибки на этапе проектирования файловой структуры или выбора паттернов управления состоянием быстро приводят к неконтролируемому росту технического долга. В данной статье обобщен опыт и описаны инженерные подходы, позволившие избежать классических архитектурных проблем и успешно довести проект до релиза.

1. Базовые архитектурные принципы и ограничения

До начала написания исходного кода был сформирован набор жестких архитектурных ограничений, призванных компенсировать отсутствие команды [1]:

1. Использование строгой модульной структуры проекта и недопущение перекрестных зависимостей.
2. Создание единого API-слоя для всех HTTP-вызовов (полный отказ от использования fetch или axios непосредственно внутри UI-компонентов).
3. Глобальный перехват и обработка сетевых ошибок для исключения дублирования блоков try/catch на страницах.
4. Фокус на E2E-тестировании критических путей приложения.
5. Максимальная автоматизация процессов сборки и развертывания (CI/CD).

2. Обоснование технологического стека

В условиях ограниченных ресурсов инструментарий должен обеспечивать высокую скорость разработки и предсказуемость результата:

- React + TypeScript. Использование TypeScript в строгом режиме (strict: true) выступило в роли первичного инструмента статического анализа [2]. Строгая типизация контрактов между клиентской и серверной частями позволила исключить значительный процент типовых ошибок еще на этапе компиляции.
- Vite. В отличие от классических сборщиков (например, Webpack), Vite использует esbuild, что обеспечивает практически мгновенный запуск сервера разработки и Hot Module Replacement (HMR) [3]. Это существенно сократило время на ожидание сборки.

- Redux Toolkit (RTK). Встроенного Context API оказалось недостаточно для управления сложным состоянием (данные авторизации, активные чаты, ролевые модели), так как он провоцирует избыточные циклы рендеринга. RTK обеспечил предсказуемый контейнер состояния приложения [4, 5].
- Cypress. Выбран в качестве основного инструмента для автоматизированного E2E-тестирования бизнес-критичных сценариев [6].
- Docker + Nginx. Стандартизированный подход для контейнеризации и доставки статических файлов в production-среду [7].

3. Модульная структура: Feature-Sliced Design

Одной из проблем растущих React-приложений является неструктурированное разрастание директории components. Для решения этой задачи была адаптирована методология Feature-Sliced Design (FSD) [8, 9].

Проект был разделен на следующие иерархические слои (рис. 1):

- app/ — инициализация приложения, настройка маршрутизации, конфигурация глобального хранилища (store) и провайдеров.
- pages/ — компоненты-страницы, отвечающие исключительно за композицию виджетов и фичей для конкретного маршрута.
- widgets/ — крупные самостоятельные UI-блоки (Header, Footer, SideBar, ModalManager).
- features/ — компоненты, реализующие конкретные пользовательские сценарии (форма авторизации, логика импорта вакансий).
- entities/ — предметные области приложения (vacancy, candidate, employer). Содержат внутренние директории api, model, ui, service.
- shared/ — переиспользуемый код, не привязанный к бизнес-логике: UI-библиотека (базовые компоненты), глобальный ApiClient, утилиты.

Данный подход обеспечивает четкие границы ответственности. Изменение презентационного слоя карточки вакансии в entitiesгарантированно не затрагивает логику авторизации в слое features.

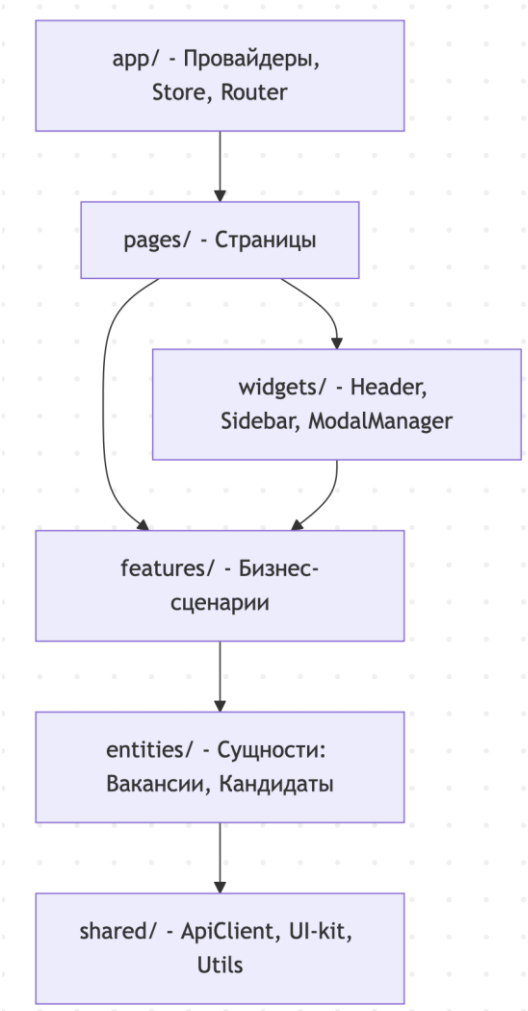

Рисунок 1. Архитектура слоев приложения. Источник: разработано автором

4. Сетевой слой: ApiClient и обработка состояний гонки

Децентрализованное выполнение HTTP-запросов усложняет поддержку кода [10]. В связи с этим в слое shared был реализован единый модуль ApiClient.ts, через который проходят все сетевые вызовы.

На этапе разработки была выявлена проблема состояния гонки (race condition) при обновлении токена доступа. В случае истечения срока действия access_token, приложение при монтировании страницы могло отправить несколько параллельных запросов (например, профиль, список вакансий, уведомления). В результате сервер возвращал серию ошибок 401, что приводило к многократным попыткам выполнить запрос refresh_token и сбросу сессии.

Решение: В ApiClient был интегрирован механизм очереди запросов. Если один из запросов возвращает ошибку 401, клиент переходит в состояние обновления токена, а все последующие запросы приостанавливаются и помещаются в массив-очередь. После успешного обновления токена накопленные запросы выполняются повторно с новыми учетными данными. Это исключило сбои в сессиях пользователей.

5. Глобальный перехват исключений

Дублирование логики обработки сетевых ошибок (блоки try/catch) в каждом компоненте является антипаттерном. Для централизации данного процесса было реализовано Redux-middleware — errorListenerMiddleware.ts.

Данный модуль отслеживает все асинхронные действия в Redux, завершающиеся со статусом rejected. При возникновении ошибки middleware анализирует ответ сервера:

- Ошибка 403 (Доступ запрещен) — инициируется глобальное всплывающее уведомление (Toast).
- Ошибка 500 (Внутренняя ошибка сервера) — выводится системное уведомление о недоступности сервиса.
- Ошибка 401 (Необходима авторизация) — происходит очистка локального состояния и корректное перенаправление на страницу входа.

Управление сетевыми исключениями было полностью делегировано errorListenerMiddleware, что позволило очистить UI-компоненты от избыточной логики (рис. 2).

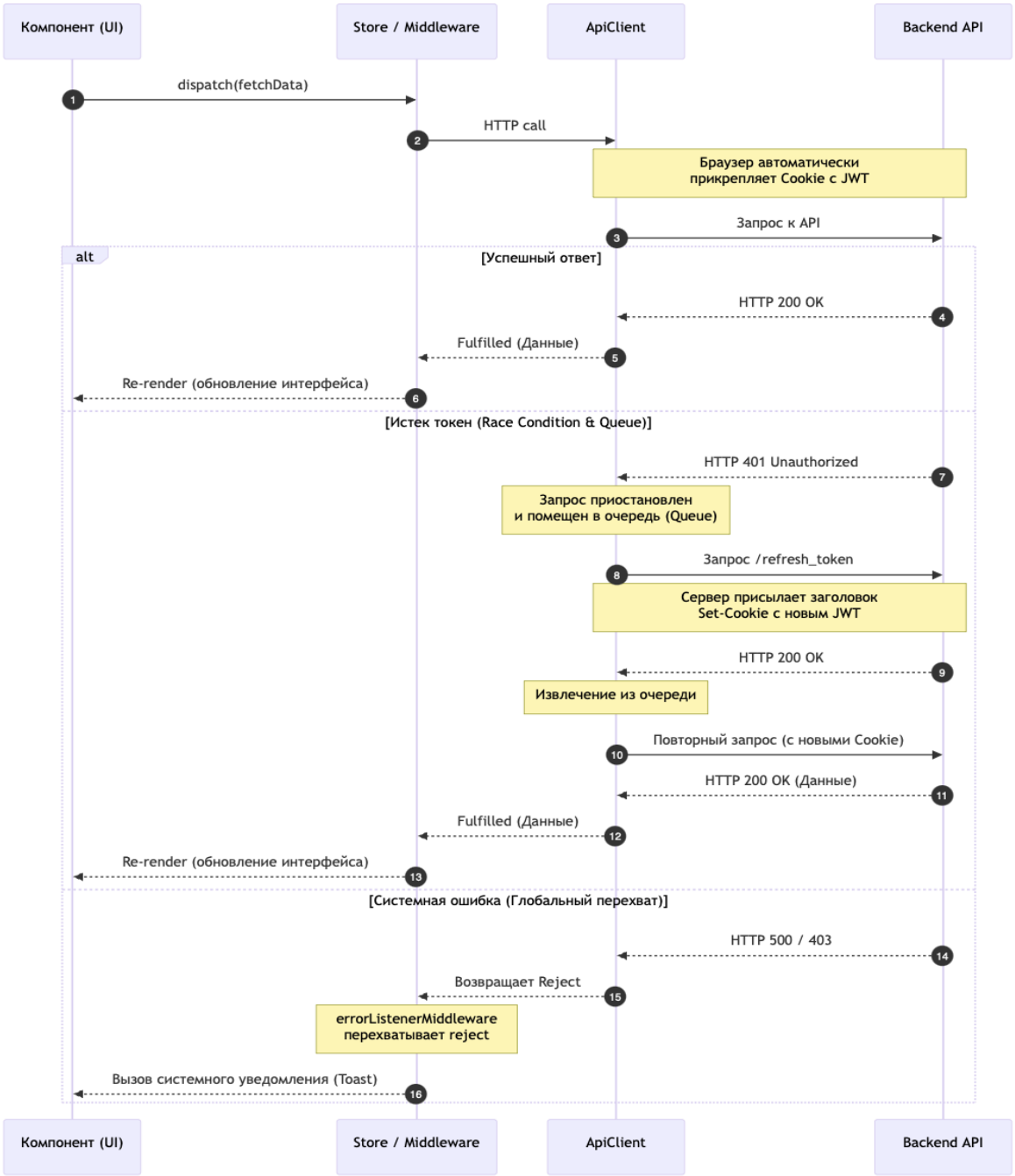

Рисунок 2. Типичный поток данных и глобальный отлов ошибок. Источник: разработано автором

6. Ролевая маршрутизация и оптимизация загрузки

Интерфейсы для соискателей и работодателей кардинально различаются. Разграничение доступа реализовано с помощью компонента высшего порядка [11] (HOC) — RoleBasedRoute.tsx.

Компонент оборачивает защищенные маршруты и проверяет роль пользователя, сохраненную в глобальном хранилище. При попытке несанкционированного доступа (например, переход кандидата по прямой ссылке в панель работодателя) маршрутизатор блокирует рендеринг страницы и выполняет перенаправление, предотвращая отправку невалидных запросов к API.

Кроме того, для оптимизации производительности применен механизм разделения кода (Code-Splitting) через React.lazy. Модули, предназначенные для работодателей, физически не загружаются в браузер соискателя, что значительно снижает объем первоначального бандла.

7. Управление модальными окнами и стилизация

Одной из классических проблем проектирования интерфейсов является управление множеством модальных окон, что часто приводит к конфликтам z-index и перегрузке DOM-дерева.

Данная задача была решена созданием централизованного компонента ModalManager в слое widgets. Глобальное состояние (какое окно должно быть открыто в данный момент) хранится в Redux. Вызов окна редактирования профиля осуществляется диспетчеризацией единственного события: dispatch(openModal('EDIT_PROFILE')). ModalManager отслеживает изменения и рендерит необходимый компонент поверх приложения через React Portals.

Стилизация проекта разделена на два уровня: глобальные дизайн-токены (цвета, типографика) вынесены в файлы CSS-переменных, а стили конкретных компонентов изолированы с использованием CSS Modules (например, contestInstructionsPage.module.css), что исключает коллизии имен селекторов.

8. Автоматизация тестирования (E2E)

В условиях индивидуальной разработки полное покрытие кода unit-тестами зачастую нецелесообразно. Оптимальным компромиссом стал фокус на End-to-End (E2E) тестировании.

С использованием фреймворка Cypress были написаны сценарии для проверки критичных пользовательских маршрутов: процессов регистрации, поиска вакансий, формирования отклика и создания вакансии работодателем [6]. Стабильное выполнение данных сценариев перед сборкой релиза гарантирует работоспособность бизнес-ядра продукта. Для поддержания качества кодовой базы также интегрированы строгие правила ESLint и линтер истории коммитов commitlint.

9. Развертывание и результаты нагрузочного тестирования

Для минимизации человеческого фактора процесс развертывания был полностью автоматизирован (рис. 3). В корне проекта описан многоэтапный Dockerfile. На первом этапе среда Node.js собирает оптимизированный бандл с помощью Vite [3], после чего полученная директория dist передается в легковесный образ веб-сервера Nginx [7]. Файл nginx.conf сконфигурирован для корректной маршрутизации SPA (перенаправление запросов на index.html) и сжатия статического контента.

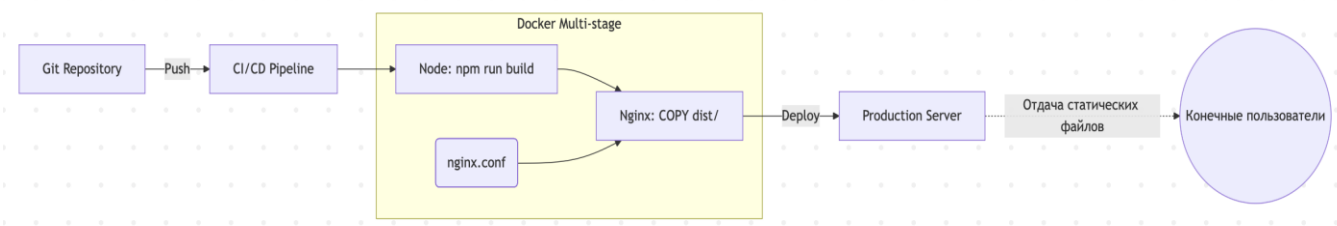

Рисунок 3. Процесс сборки и доставки приложения. Источник: разработано автором

Результаты эксплуатации

К моменту публикации платформа успешно эксплуатируется в production-среде. В системе зарегистрировано более 2 000 пользователей.

Практическая валидация выбранной архитектуры произошла в рамках проведения масштабного профильного конкурса. В пиковый период было зафиксировано около 600 одновременных активных сессий. Пользователи проходили регистрацию, просматривали инструкции, заполняли профили и выполняли конкурсные задания.

Благодаря статической модели доставки контента (SPA) генерация страниц на сервере не требовалась — Nginx успешно обрабатывал запросы на статические файлы. Редкие отклонения во времени ответа бэкенда из-за пиковых нагрузок корректно обрабатывались механизмом errorListenerMiddleware на стороне клиента. Клиентское приложение продемонстрировало высокую отказоустойчивость, выдержав пиковую нагрузку без сбоев.

Заключение

Опыт разработки клиентской части HR-платформы подтверждает, что реализация и успешная поддержка масштабного SPA-приложения силами одного разработчика является достижимой задачей при условии соблюдения инженерной дисциплины.

Сформулированы следующие выводы, применимые при проектировании аналогичных систем:

1. Строгая архитектурная изоляция. Внедрение Feature-Sliced Design предотвращает неконтролируемое связывание компонентов и замедляет рост технического долга.
2. Централизация сетевого взаимодействия. Инкапсуляция HTTP-запросов и обработки ошибок в единые сущности (ApiClient, errorListenerMiddleware) существенно повышает читаемость UI-компонентов.
3. Централизация UI-компонентов. Использование паттернов наподобие ModalManager исключает конфликты в DOM-дереве.
4. Автоматизация QA и развертывания. E2E-тестирование и контейнеризация посредством Docker критически важны для обеспечения стабильности релизов при отсутствии выделенных специалистов.

Предложенная архитектура продемонстрировала высокую надежность в условиях пиковых нагрузок и обладает достаточным потенциалом для дальнейшего масштабирования и подключения к разработке новых специалистов.

Список литературы:

1. Мартин, Роберт С. Чистая архитектура. Искусство разработки программного обеспечения / Роберт Мартин; перевод с английского А. Макаровой. – Санкт-Петербург: Питер, 2018. – 352 с. – (Библиотека программиста). – ISBN 978-5-4461-0772-8.
2. React. The library for web and native user interfaces. – URL: https://react.dev/ (дата обращения: 20.04.2026). – Текст: электронный.
3. Vite: Next Generation Frontend Tooling. — URL: https://vitejs.dev/ (дата обращения: 20.04.2026). – Текст: электронный.

4. Redux Toolkit. The official, opinionated, batteries-included toolset for efficient Redux development. — URL: https://redux-toolkit.js.org/ (дата обращения: 20.04.2026). – Текст: электронный.
5. Krutika P. Redux State Management System-A Comprehensive Review // International Journal of Trend in Scientific Research and Development. – 2022. – V. 6. – №. 7. – Pp. 1021-1027.
6. Cypress End-to-End Testing. — URL: https://www.cypress.io/ (дата обращения: 20.04.2026). – Текст: электронный.
7. Nginx Documentation. — URL: https://nginx.org/en/docs/ (дата обращения: 20.04.2026). – Текст: электронный.
8. Feature-Sliced Design (FSD) Architectural Methodology. — URL: https://feature-sliced.design/ (дата обращения: 20.04.2026). – Текст: электронный.
9. Alekhin, S. S., Building an Application Architecture Using the FSD Concept / A. I. Kanev, D. S. Danilov // 2024 Conference of Young Researchers in Electrical and Electronic Engineering (ElCon). – IEEE, 2024. – Pp. 96-99.
10. Алпатов, А. Н. Аутентификация с использованием блокчейн в децентрализованных приложениях концепции WEB 3.0 // Технологии, модели и алгоритмы модернизации науки в современных геополитических условиях. – 2022. – С. 27-35.
11. Arora, C., Higher-order relationship-based access control: A temporal instantiation with iot applications / S. Z. R. Rizvi, P. W. L. Fong // Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies. – 2022. – Pp. 223-234.

References:

1. Martin, Robert S. Pure Architecture. The Art of Software Development / Robert Martin; translated from English by A. Makarova. Saint Petersburg: Peter, 2018. 352 p. (Programmer's Library). – ISBN 978-5-4461-0772-8.
2. React. The library for web and native user interfaces. – URL: https://react.dev/ (accessed: 20.04.2026). – Text: electronic.
3. Vite: Next Generation Frontend Tooling. — URL: https://vitejs.dev/ (accessed: 20.04.2026). – Text: electronic.
4. Redux Toolkit. The official, opinionated, batteries-included toolset for efficient Redux development. — URL: https://redux-toolkit.js.org/ (accessed: 20.04.2026). – Text: electronic.
5. Krutika P. Redux State Management System-A Comprehensive Review // International Journal of Trend in Scientific Research and Development. – 2022. – V. 6. – №. 7. – Pp. 1021-1027.
6. Cypress End-to-End Testing. — URL: https://www.cypress.io/ (accessed: 20.04.2026). – Text: electronic.
7. Nginx Documentation. — URL: https://nginx.org/en/docs/ (дата обращения: 20.04.2026). – Text: electronic.
8. Feature-Sliced Design (FSD) Architectural Methodology. — URL: https://feature-sliced.design/ (accessed: 20.04.2026). – Text: electronic.

9. Alekhin, S. S., Building an Application Architecture Using the FSD Concept / A. I. Kanev, D. S. Danilov // 2024 Conference of Young Researchers in Electrical and Electronic Engineering (ElCon). – IEEE, 2024. – Pp. 96-99.
10. Alpatov, A. N. Authentication using blockchain in decentralized applications of the WEB 3.0 concept // Technologies, models and algorithms of modernization of science in modern geopolitical conditions. – 2022. – Pp. 27-35.
11. Arora, C., Higher-order relationship-based access control: A temporal instantiation with iot applications / S. Z. R. Rizvi, P. W. L. Fong // Proceedings of the 27th ACM on Symposium on Access Control Models and Technologies. – 2022. – Pp. 223-234.