Содержание

Введение

Организация совместных поездок и мероприятий с друзьями и знакомыми стала обыденным явлением. При этом одной из наиболее трудоёмких задач является справедливое разделение общих расходов между участниками. В процессе путешествия или группового мероприятия возникают различные траты: проживание, питание, транспорт, развлечения. Зачастую один человек оплачивает расходы за всю группу, а затем возникает необходимость вести учёт и производить взаиморасчёты.

Традиционные методы ведения учёта расходов – использование бумажных блокнотов, таблиц в мессенджерах или Google-таблицах – создают ряд проблем:

- Высокая вероятность ошибок при ручном подсчёте
- Сложность отслеживания изменений и истории операций
- Отсутствие автоматического расчёта итоговых долгов
- Необходимость дублирования информации между участниками

Существующие решения для разделения расходов, такие как Splitwise, Tricount и Settle Up, решают эти проблемы, хоть и имеют свои недостатки.

Что хотим получить

Результатом этой курсовой работы должно стать веб-приложение для управления групповыми расходами. Назовём это приложение **TripF** (Trip Finance).

Основные требования к системе:

- Интуитивно понятный интерфейс;
- Регистрация пользователей;
- Создание групп;
- Возможность приглашения других пользователей в группу;
- Добавление расходов;
- Автоматический расчёт долгов;
- Отметка о выполненных платежах;
- Возможность удалять/редактировать расходы, удалять участников из группы, удалять группы.

Средства реализации

3.1 Клиентская часть

Для разработки клиентской части приложения использовались следующие технологии:

React 19 [1] – современная JavaScript-библиотека для создания пользовательских интерфейсов.

Next.js 16 [2] – React-фреймворк с поддержкой серверного рендеринга. Используется для: API Routes для создания backend-функционала, file-based роутинга для удобной навигации.

TypeScript 5 [3] – типизированный JavaScript. Обеспечивает: Статическую проверку типов на этапе разработки, улучшение читаемости и поддерживаемости кода.

Tailwind CSS 4 [4] – utility-first Iinline CSS фреймворк. Позволяет определять стили прямо в HTML-коде, используя предопределённые классы-хелперы.

SWR 2.3 [5] – библиотека для управления состоянием и кэширования данных. Используется для автоматического кэширования ответов API (то есть позволяет избежать повторных запросов к серверу).

3.2 Серверная часть

Node.js – серверная платформа на базе V8 JavaScript-движка. Обеспечивает единый язык для фронта и бэка (TypeScript) и большую экосистему пакетов npm.

Next.js API Routes – встроенная в Next.js возможность создания API-эндпоинтов (файлы в папке `app/api/` автоматически становятся API-роутами).

Better-SQLite3 12.4 [6] – библиотека для работы с SQLite [7]. Выбор обусловлен простым синтаксисом и высокой скоростью работы.

Custom HTTPS Server – собственный сервер на базе Node.js `https` модуля [8] (`server.js`). Так как у меня iPhone он отказывается открывать http сайты из локальной сети. Поэтому были созданы самоподписанные сертификаты и соотвественно для этого нужен такой «костыль».

3.3 Обоснование выбора технологий

Главным критерием выбора такого стека – единая экосистема Next.js. Это позволяет использовать один и тот же язык для фронта и бэка, что упрощает разработку и позволяет переиспользовать код (типы данных, утилиты).

Архитектура системы

4.1 Общая архитектура приложения

Приложение TripF построено по классической клиент-серверной архитектуре с использованием паттерна MVC (Model-View-Controller), адаптированного для современных веб-приложений.

Архитектурные слои:

1. **Слой представления (View)** – React-компоненты, отвечающие за отображение интерфейса и взаимодействие с пользователем (`app/components/`).
2. **Слой маршрутизации (Controller)** – Next.js страницы (`app/[route]/page.tsx`) и API Routes (`app/api/[route]/route.ts`).
3. **Бизнес-логика** – функции для работы с данными, расчётов, валидации (`app/lib/`).
4. **Слой данных (Model)** – взаимодействие с SQLite через библиотеку Better-SQLite3. Схема БД и функции доступа в `app/lib/db.ts`.

Поток данных:

1. Пользователь взаимодействует с React-компонентом
2. Компонент вызывает API-эндпоинт через fetch/SWR
3. API Route проверяет авторизацию, валидирует данные
4. Бизнес-логика обрабатывает запрос, работает с БД
5. Результат возвращается в виде JSON
6. SWR кэширует ответ и обновляет компонент
7. React перерисовывает интерфейс

4.2 Структура базы данных

База данных состоит из 8 таблиц.

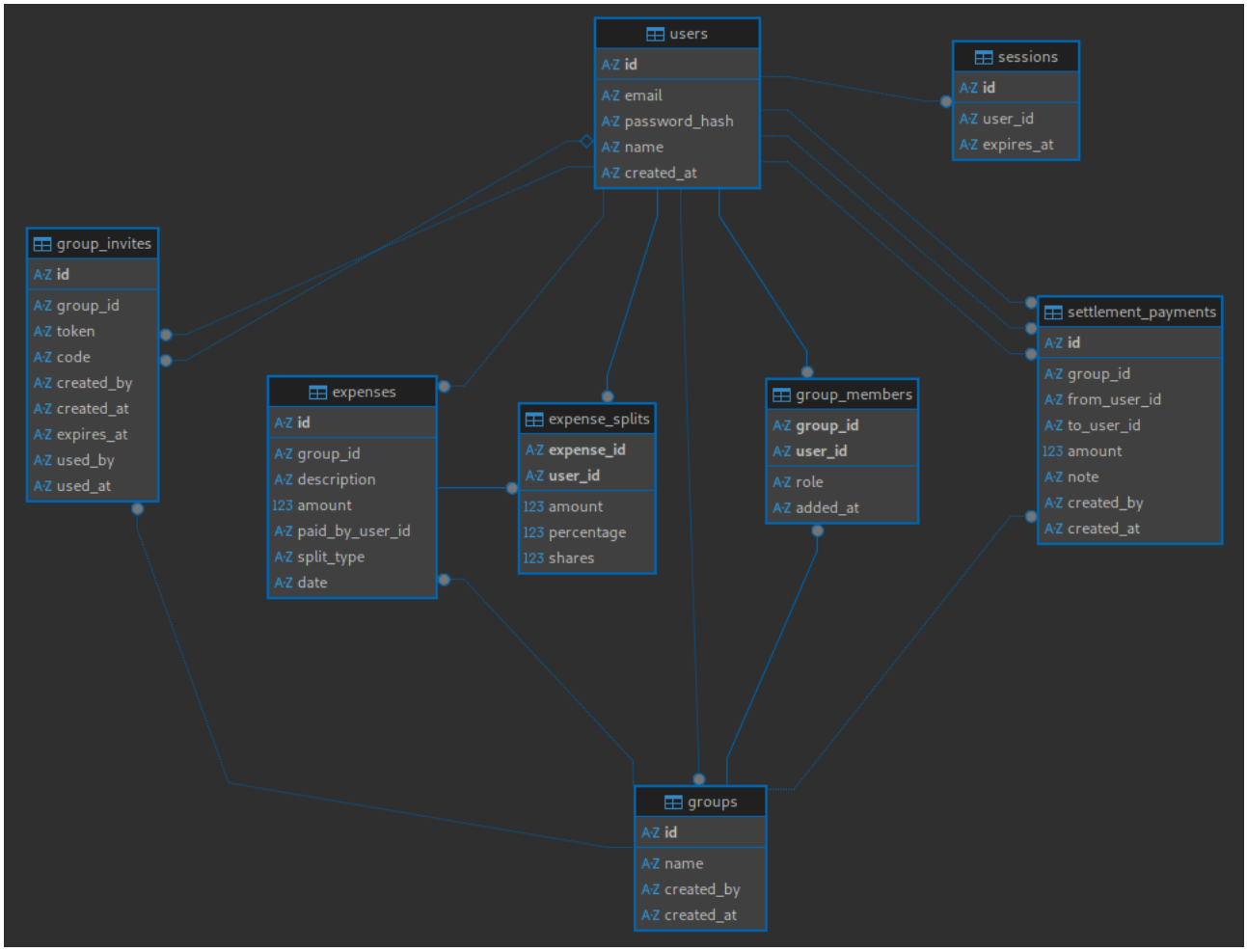

Рис. 1: Схема базы данных

Основные сущности: users – таблица с пользователями, groups – таблица с группами, expenses – таблица с расходами, settlement_payments – таблица с платежами.

Вспомогательные сущности: group_members, expense_splits – для связи многие ко многим, group_invites – таблица с приглашениями в группу, sessions – таблица с сессиями пользователей.

4.3 API-routers

Аутентификация (`/api/auth/*`):

- `POST /api/auth/register` – регистрация нового пользователя. Создаёт пользователя, хеширует пароль, создаёт сессию.
- `POST /api/auth/login` – вход в систему. Проверяет пароль, создаёт сессию, устанавливает cookie.
- `POST /api/auth/logout` – выход из системы. Удаляет сессию и cookie.
- `GET /api/auth/me` – получение информации о текущем пользователе.

Управление группами (`/api/groups/*`):

- `GET /api/groups` – получение списка групп текущего пользователя
- `POST /api/groups` – создание новой группы.
-
- `GET /api/groups/[id]` – получение информации о группе с участниками
- `DELETE /api/groups/[id]` – удаление группы.

Расходы (`/api/groups/[id]/expenses/*`):

- `GET /api/groups/[id]/expenses` – список всех расходов группы с разбивкой
- `POST /api/groups/[id]/expenses` – создание расхода. После создания генерируется событие `expense:created`.
- `GET /api/groups/[id]/expenses/[expenseId]` – получение конкретного расхода

- `PUT /api/groups/[id]/expenses/[expenseId]` – редактирование расхода. Удаляет старые splits и создаёт новые.
- `DELETE /api/groups/[id]/expenses/[expenseId]` – удаление расхода. Каскадно удаляет splits благодаря FOREIGN KEY.

Платежи (`/api/groups/[id]/payments/*`):

- `GET /api/groups/[id]/payments` – список всех платежей группы
- `POST /api/groups/[id]/payments` – регистрация платежа. Вставляет запись в `settlement_payments`. Генерирует событие `payment:created`.
- `DELETE /api/groups/[id]/payments/[paymentId]` – удаление платежа

Приглашения (`/api/groups/[id]/invites/*`):

- `GET /api/groups/[id]/invites` – список активных приглашений группы
- `POST /api/groups/[id]/invites` – создание нового приглашения. Генерирует уникальный 6-символьный код и 48-символьный токен.
- `POST /api/invites/accept` – принятие приглашения. Добавляет пользователя в `group_members`, отмечает приглашение использованным.

Обработка ошибок:

Все API-маршруты используют единообразную обработку ошибок: [9]

- 401 Unauthorized – нет сессии или сессия истекла
- 403 Forbidden – нет доступа к ресурсу
- 404 Not Found – ресурс не найден
- 400 Bad Request – невалидные данные

- 409 Conflict – конфликт (например, email уже существует)
- 500 Internal Server Error – ошибка сервера

Все ошибки возвращаются в формате `{"error": "[Описание ошибки]"}`.

4.4 Клиентские компоненты

React-компоненты организованы в папке `app/components/`

AuthGuard.tsx – компонент-обёртка для проверки наличия сессии, при её отсутствии перенаправляет на страницу входа.

GroupCard.tsx – карточка группы на главной странице. Отображает название, количество участников, общую сумму расходов.

GroupView.tsx – главный компонент страницы группы.

ExpenseList.tsx – список расходов группы. Отображает каждый расход с описанием, суммой, плательщиком, датой. Кнопки редактирования и удаления.

ExpenseForm.tsx – форма добавления/редактирования расхода.

SettlementList.tsx – список расчётов. Отображает кто кому и сколько должен. Кнопки для регистрации платежа. Показывает историю выполненных платежей.

ParticipantList.tsx – список участников группы с отображением их балансов (переплата/долг).

GroupInvitePanel.tsx – панель управления приглашениями. Показывает активные приглашения, позволяет создать новое, копировать код/ссылку в буфер обмена.

JoinGroupForm.tsx – форма для присоединения к группе по коду приглашения.

UserMenu.tsx – header приложения с отображением логотипа, имени и

кнопкой выхода.

Рис. 2: Диаграмма классов

Реализация

5.1 Система аутентификации

Система аутентификации реализована в файле `app/lib/auth.ts`. Обрабатывает регистрацию, вход, управление сессиями и защиту паролей.

Управление сессиями

Сессии хранятся в таблице `sessions` БД. При входе создаётся новая сессия: реализация функции `createSession` приведена в листинге 1.

Время жизни сессии – 30 дней. При каждом запросе проверяется актуальность сессии.

Идентификатор сессии хранится в HTTP-only cookie, откуда он и извлекается при необходимости валидировать сессию: пример функции установки cookie приведён в листинге 3.

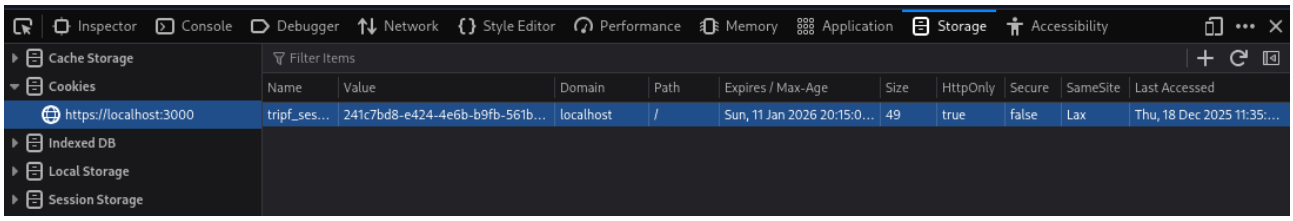

Рис. 3: Cookie с идентификатором сессии

5.2 Управление группами

Функции для работы с группами находятся в `app/lib/groups.ts` и `app/lib/invites.ts`.

Система приглашений

Для присоединения к группе используются два типа идентификаторов:

- **Токен** – для ссылок-приглашений вида `/invite/[token]`

- **Код** – короткий код типа «A3K9M2» для ручного ввода

Генерация уникального кода (`app/lib/invites.ts`): реализация генерации уникального кода показана в листинге 4.

Реализация генерации приглашения приведена в листинге 5.

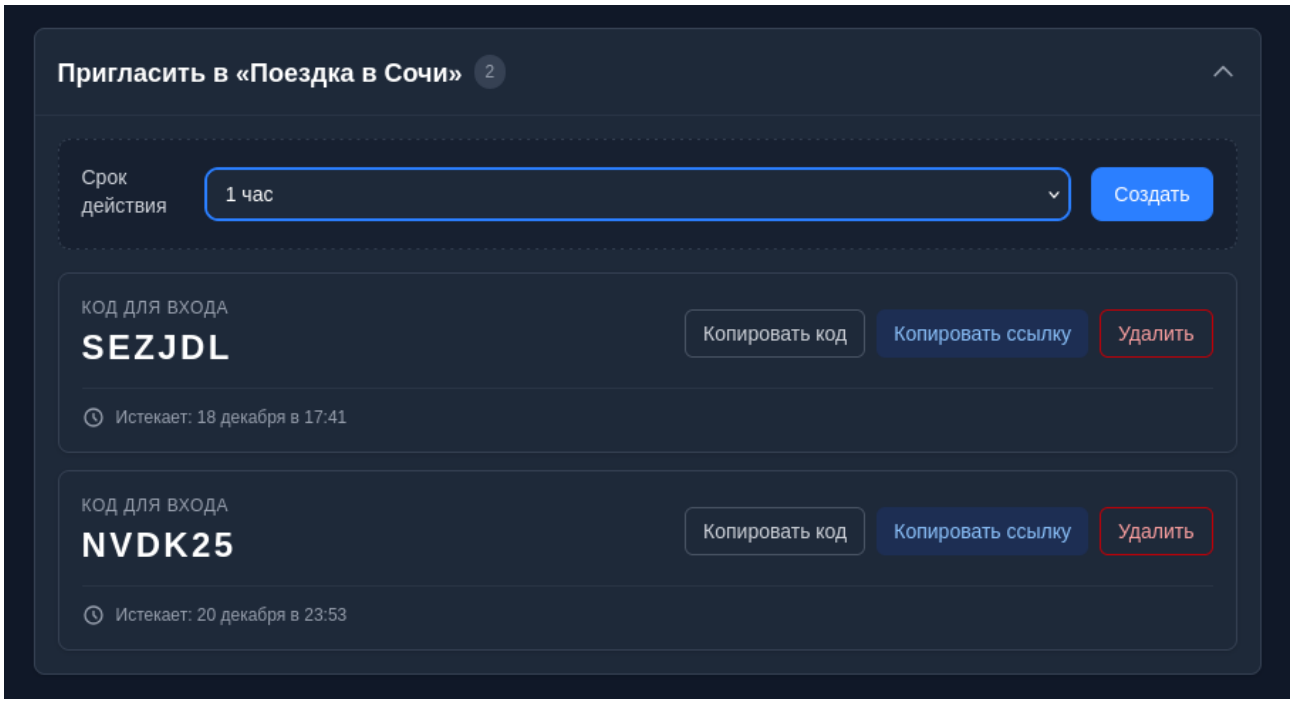

Рис. 4: Примеры приглашений

Примеры ссылки приглашения:

`localhost:3000/invite/558130fc7f7d34e9b4650578bc1fa6cb5dce7823c0e8d6c7`

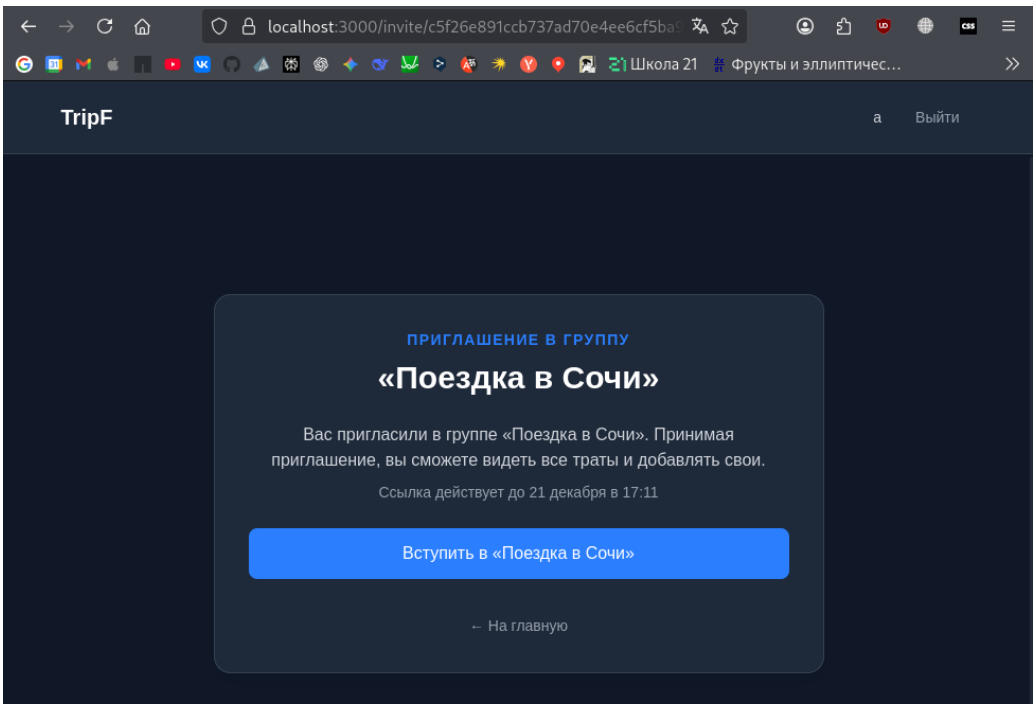

Рис. 5: Пример перехода по ссылке приглашения

При повторном переходе по ссылке приглашения будет отображаться ошибка:

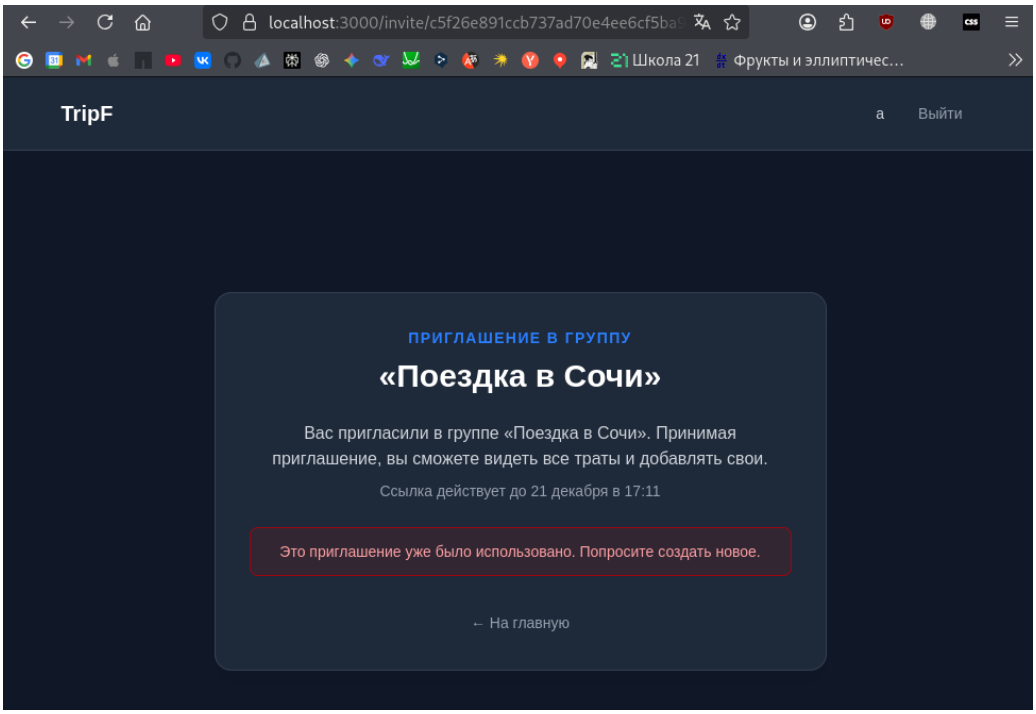

Рис. 6: Ошибка перехода по ссылке приглашения

5.3 Учёт расходов

Создание расхода

API-маршрут `api/groups/[id]/expenses/route.ts` принимает JSON формата указанного в листинге 6.

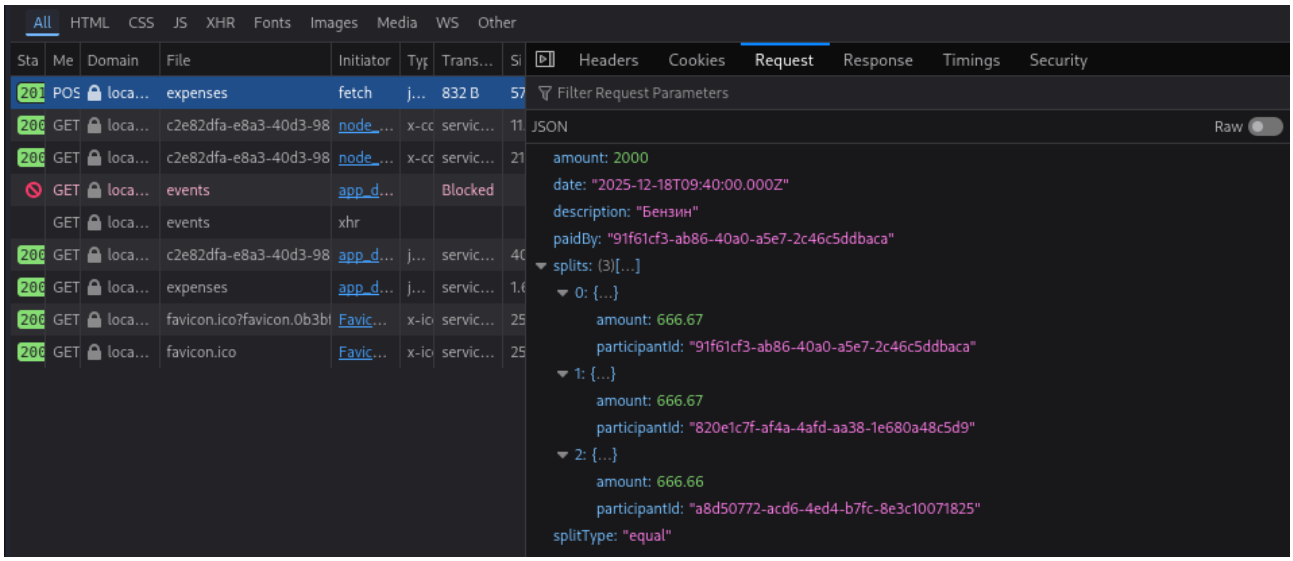

Рис. 7: Запрос на создание расхода

Реализация в `api/groups/[id]/expenses/route.ts`: фрагмент кода вставки расхода и разбивки приведён в листинге 7.

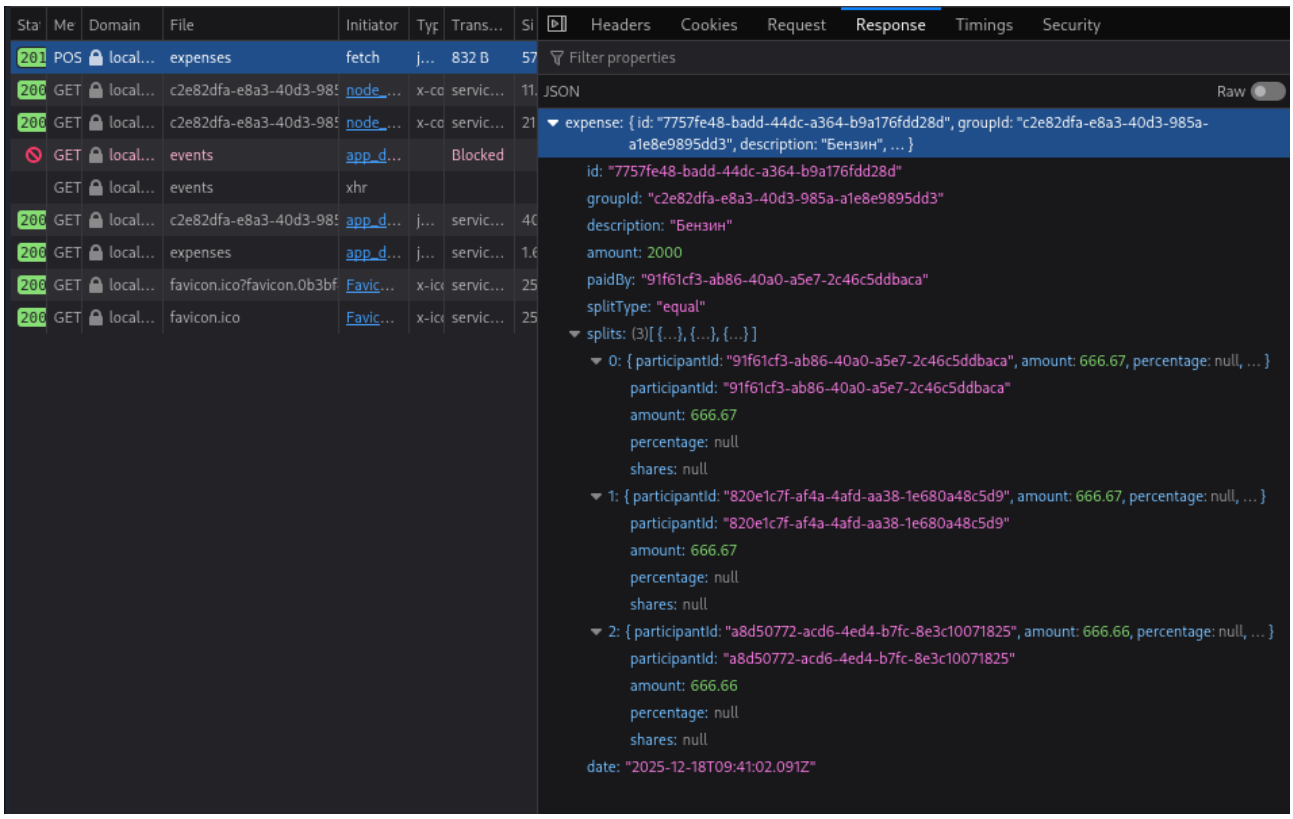

Рис. 8: Ответ о создании расхода

Затем генерируем событие для обновления UI: пример генерации события об изменении данных показан в листинге 8.

5.4 Регистрация платежей

Когда участники фактически производят переводы друг другу (через банк, наличными), они регистрируют это в приложении для пересчёта балансов.

Создание платежа

API-маршрут `POST api/groups/[id]/payments/route.ts`: полный обработчик создания платежа приведён в листинге 9.

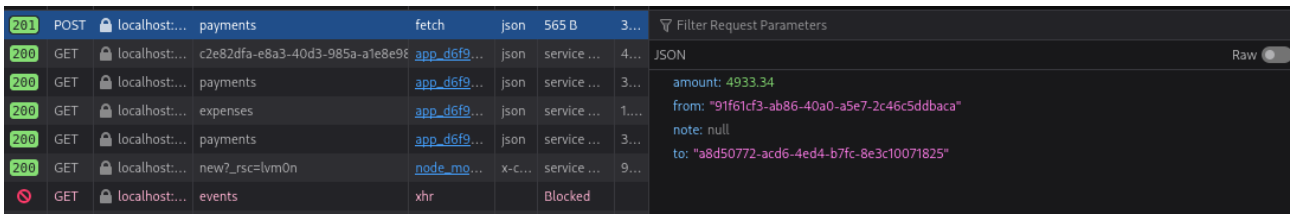

Рис. 9: Запрос на создание платежа

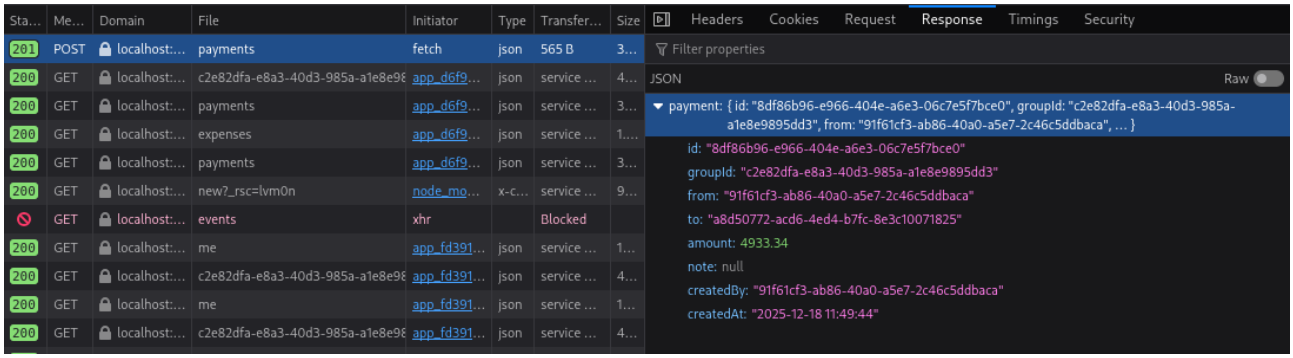

Рис. 10: Ответ на создание платежа

5.5 Система событий

Для синхронизации интерфейса при изменении данных используется простая реализация паттерна Observer в `app/lib/eventBus.ts`.

Рреализация `EventBus` вынесены в листинг 10.

Использование в компоненте:

В `GroupView.tsx` компонент подписывается на события группы: пример использования `EventBus` на клиенте в компоненте `GroupView` приведён в листинге 11.

Функция `mutate` из SWR инвалидирует кэш и перезапрашивает данные с сервера. Это позволяет обновлять данные в UI при изменении данных на сервере, у всех пользователей в группе. Но такая система не сохранит события при перезагрузке сервера.

Для production-версии в идеале переписать на Redis Pub/Sub или WebSocket.

Тестирование

Допустим есть 4 человека, которые хотять поехать в путешествие. Они регистрируются в приложении. Один из них создаёт группу и приглашает трёх других. Пользователи вводят коды приглашений и попадают в группу. Далее каждый вносит расходы в группу. Затем происходит автоматический расчёт долгов между участниками. После этого пользователи отмечают платежи как оплаченные и система автоматически рассчитывает балансы между участниками.

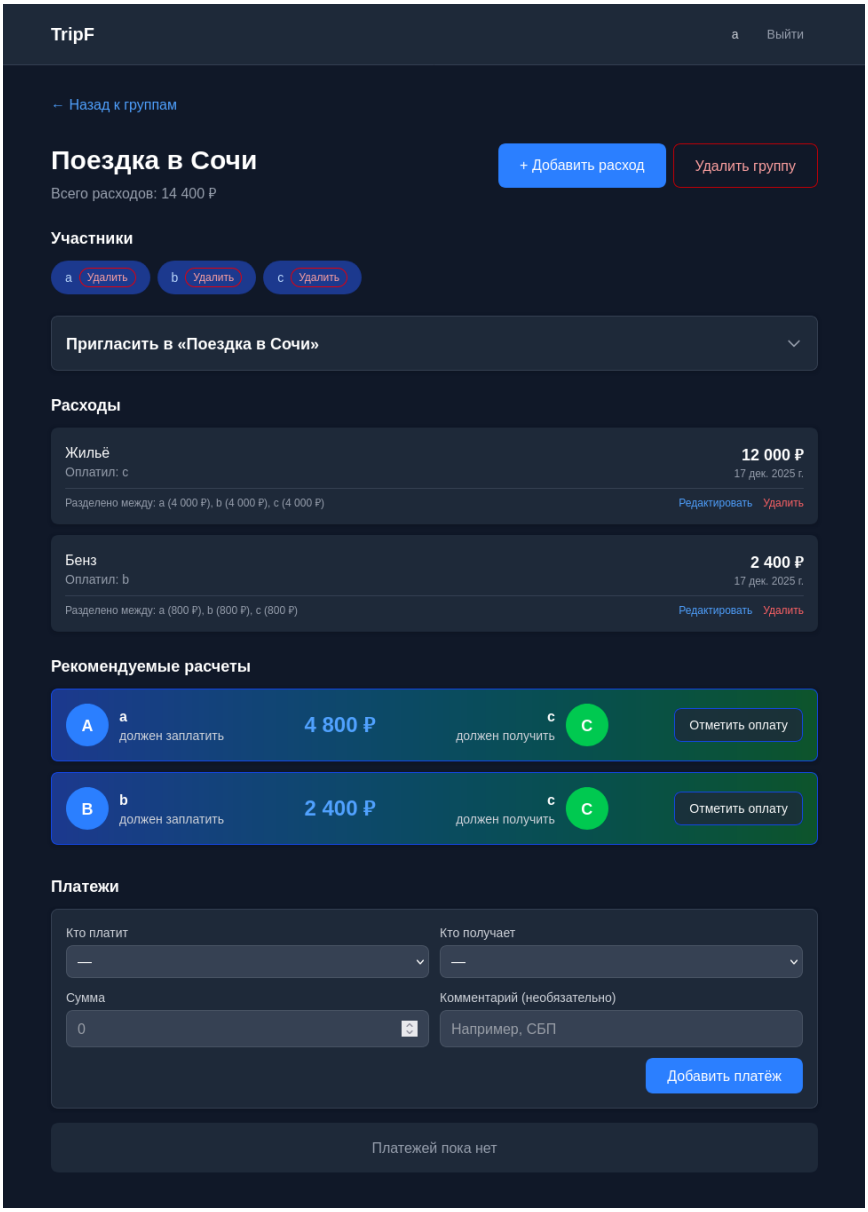

Рис. 11: Интерфейс приложения

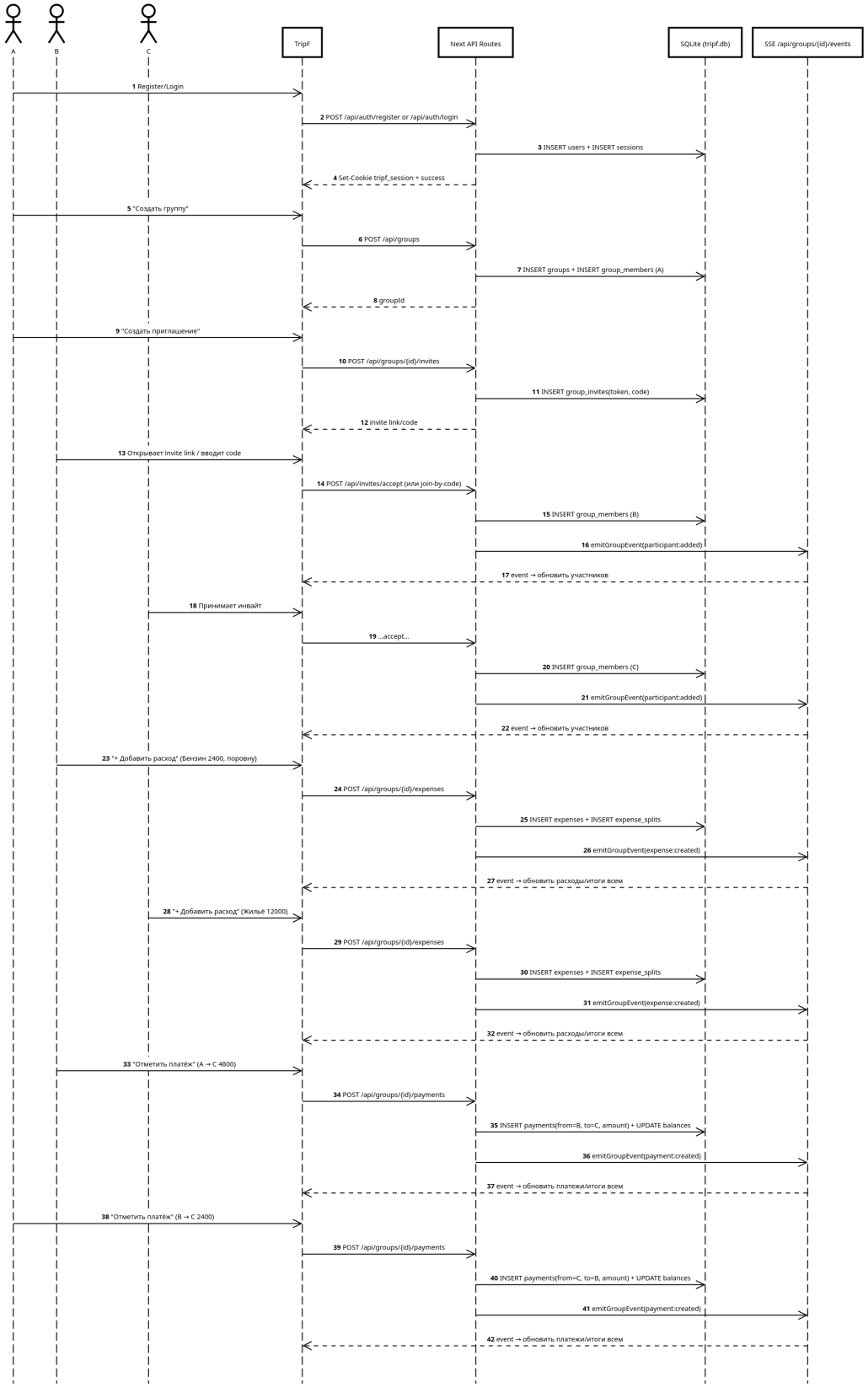

Рис. 12: Сценарий использования

Дальнейшее развитие системы

В текущей версии реализованы базовые функции для учёта групповых расходов. Для расширения возможностей приложения можно добавить возможность разделения расхода не только поровну, но и по процентам, суммам, долям.

Добавить мультивалютность, то есть возможность использования нескольких валют в одной группе. Сделать интеграцию с API курсов валют и пересчитывать суммы расходов в основную валюту группы.

Сделать категоризацию расходов (гостиницы, такси, питание, развлечения, ...).

Сделать дашборд с графиком расходов по категориям, участникам. Средний расход на человека и тому подобное.

Реализовать интеграцию с платёжными системами, например генерировать QR-код для переводов по СБП.

Возможность загружать фотографии чеков к расходам.

Заключение

В ходе выполнения курсовой работы было разработано функциональное веб-приложение TripF для управления групповыми расходами.

Разработанное приложение решает реальную проблему справедливого разделения расходов в группах. Приложение готово к использованию в реальных поездках и групповых мероприятиях.

TripF имеет потенциал для дальнейшего развития: добавление мультивалютности, категорий расходов, статистики, интеграции с платёжными системами. Открытый исходный код позволяет сообществу участвовать в улучшении проекта.

Исходный код приложения доступен по ссылке: `https://github.com/arpefly/tripf`.

Список литературы

[1] React Documentation. https://react.dev. Дата обращения: 14.11.2025.

[2] Next.js Documentation. https://nextjs.org/docs. Дата обращения: 14.11.2025.

[3] TypeScript Handbook. https://www.typescriptlang.org/docs/. Дата обращения: 15.11.2025.

[4] Tailwind CSS Documentation. https://tailwindcss.com/docs. Дата обращения: 17.11.2025.

[5] SWR Documentation. https://swr.vercel.app. Дата обращения: 10.12.2025.

[6] Better-SQLite3 GitHub Repository. https://github.com/WiseLibs/better-sqlite3. Дата обращения: 16.11.2025.

[7] SQLite Documentation. https://www.sqlite.org/docs.html. Дата обращения: 16.11.2025.

[8] Node.js HTTPS Module Documentation. https://nodejs.org/api/https.html. Дата обращения: 14.11.2025.

[9] Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, RFC Editor, June 2014. Obsoletes RFC 2616.

Приложение

Листинги кода

Листинг 1: Создание сессии пользователя

```
export function createSession(userId: string): string {
  const sessionId = randomUUID();
  const expiresAt = new Date(
    Date.now() + 30 * 24 * 60 * 60 * 1000
  ).toISOString();
  db.prepare(
    `INSERT INTO sessions (id, user_id, expires_at)
     VALUES (?, ?, ?)`
  ).run(sessionId, userId, expiresAt);

  return sessionId;
}
```

Листинг 2: Проверка актуальности сессии

```
export function getSession(sessionId: string):
  Session | null {
  const session = db.prepare(
    `SELECT * FROM sessions
     WHERE id = ? AND expires_at > datetime('now')`
  ).get(sessionId);

  return session || null;
}
```

Листинг 3: Установка cookie с идентификатором сессии

```
export async function setSessionCookie(
  sessionId: string
): Promise<void> {
```

```
  const cookieStore = await cookies();
  cookieStore.set('tripf_session', sessionId, {
    httpOnly: true,
    sameSite: 'lax',
    maxAge: 30 * 24 * 60 * 60,
    path: '/',
  });
}
```

Листинг 4: Генерация уникального кода приглашения

```
const CODE_ALPHABET =
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890';
const CODE_LENGTH = 6;

function generateUniqueCode() {
  let code: string;
  do {
    code = '';
    for (let i = 0; i < CODE_LENGTH; i++) {
      const index = Math.floor(
        Math.random() * CODE_ALPHABET.length
      );
      code += CODE_ALPHABET[index];
    }
  } while (
    db.prepare(
      `SELECT 1 FROM group_invites WHERE code = ?`
    ).get(code)
  );
  return code;
}
```

Листинг 5: Создание приглашения в группу

```
export function createGroupInvite(
  groupId: string,
```

```
  createdBy: string,
  expiresInHours: number = 72
): GroupInvite {
  const inviteId = randomUUID();
  const token = randomBytes(24).toString('hex');
  const code = generateUniqueCode();
  const expiresAt = new Date(
    Date.now() + expiresInHours * 60 * 60 * 1000
  );

  db.prepare(
    `INSERT INTO group_invites
     (id, group_id, token, code, created_by,
      expires_at)
     VALUES (?, ?, ?, ?, ?, ?)`
  ).run(inviteId, groupId, token, code, createdBy,
        expiresAt);

  return getInviteById(inviteId);
}
```

Листинг 6: Структура запроса на создание расхода

```
{
  "description": "desc",
  "amount": 0000,
  "paidBy": "user-uuid",
  "splitType": "equal",
  "splits": [
    { "participantId": "user1-uuid", "amount": 0000 },
    { "participantId": "user2-uuid", "amount": 0000 },
    ...
  ]
}
```

Листинг 7: Вставка расхода и разбивки по участникам

```
const expenseId = randomUUID();
const expenseDate = new Date().toISOString();

const insertExpense = db.transaction(() => {
  //  Создаём расход
  db.prepare(
    `INSERT INTO expenses
     (id, group_id, description, amount,
      paid_by_user_id, split_type, date)
     VALUES (?, ?, ?, ?, ?, ?, ?)`
  ).run(expenseId, groupId, description, amount,
        paidBy, splitType, expenseDate);

  //  Создаём разбивку
  const insertSplit = db.prepare(
    `INSERT INTO expense_splits
     (expense_id, user_id, amount)
     VALUES (?, ?, ?)`
  );

  for (const split of splits) {
    insertSplit.run(
      expenseId,
      split.participantId,
      split.amount
    );
  }
});

insertExpense();
```

Листинг 8: Генерация события о создании расхода

```
emitGroupEvent(groupId, {
  type: 'expense:created',
  expenseId
```

```
});
```

Листинг 9: Обработчик создания платежа

```
export async function POST(request, { params }) {
  const userId = await getCurrentUserId();
  const { id: groupId } = await params;
  const { from, to, amount, note } =
    await request.json();

  // Валидация
  if (!from || !to || !amount || amount <= 0) {
    return NextResponse.json(
      { error: 'Неверные данные' },
      { status: 400 }
    );
  }

  if (from === to) {
    return NextResponse.json(
      { error: 'Нельзя платить самому себе' },
      { status: 400 }
    );
  }

  const paymentId = randomUUID();
  const createdAt = new Date().toISOString();

  db.prepare(
    `INSERT INTO settlement_payments
     (id, group_id, from_user_id, to_user_id,
      amount, note, created_by, created_at)
     VALUES (?, ?, ?, ?, ?, ?, ?, ?)`
  ).run(paymentId, groupId, from, to, amount,
        note, userId, createdAt);
```

```
  //  Генерируем событие
  emitGroupEvent(groupId, {
    type: 'payment:created',
    paymentId
  });

  return NextResponse.json({
    payment: { id: paymentId, from, to, amount,
               note, createdBy: userId, createdAt }
  }, { status: 201 });
}
```

Листинг 10: Простая реализация EventBus

```
type Listener = (payload: unknown) => void;

const listeners =
  new Map<string, Set<Listener>>();

export function subscribeToGroup(
  groupId: string,
  listener: Listener
): () => void {
  const groupListeners =
    listeners.get(groupId) ?? new Set<Listener>();
  groupListeners.add(listener);
  listeners.set(groupId, groupListeners);

  //  Возвращаем функцию отписки
  return () => {
    const current = listeners.get(groupId);
    if (!current) return;
    current.delete(listener);
    if (current.size === 0) {
      listeners.delete(groupId);
    }
```

```
  };
}

export function emitGroupEvent(
  groupId: string,
  payload: unknown
): void {
  const groupListeners = listeners.get(groupId);
  if (!groupListeners) return;

  for (const listener of groupListeners) {
    try {
      listener(payload);
    } catch (error) {
      console.error('Event listener error:', error);
    }
  }
}
```

Листинг 11: Подписка компонента GroupView на события группы

```
useEffect(() => {
  const unsubscribe = subscribeToGroup(
    groupId,
    (payload) => {
      //  При любом изменении перезагружаем данные
      mutate(`/api/groups/${groupId}`);
      mutate(`/api/groups/${groupId}/expenses`);
      mutate(`/api/groups/${groupId}/payments`);
    }
  );

  return unsubscribe; //  Отписка при размонтировании
}, [groupId]);
```

Скриншоты интерфейса

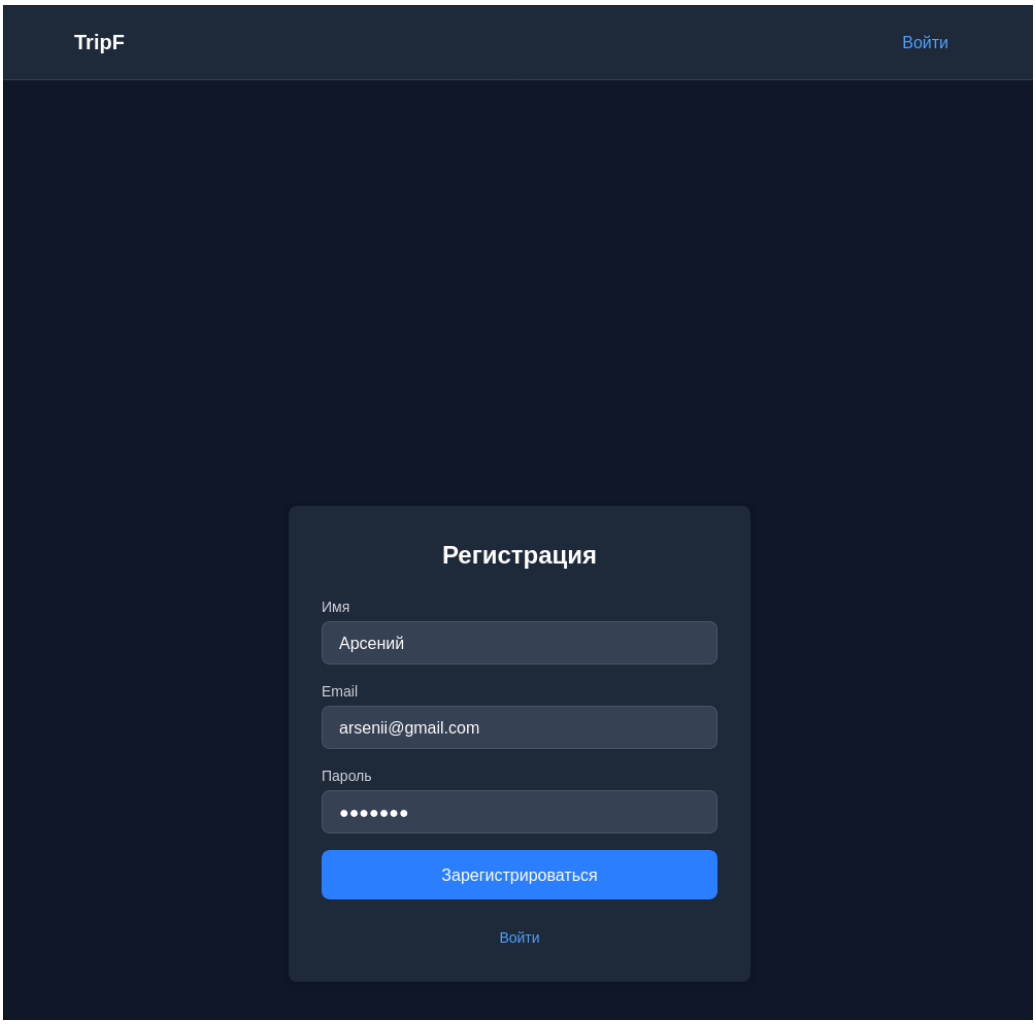

Рис. 13: Форма регистрации

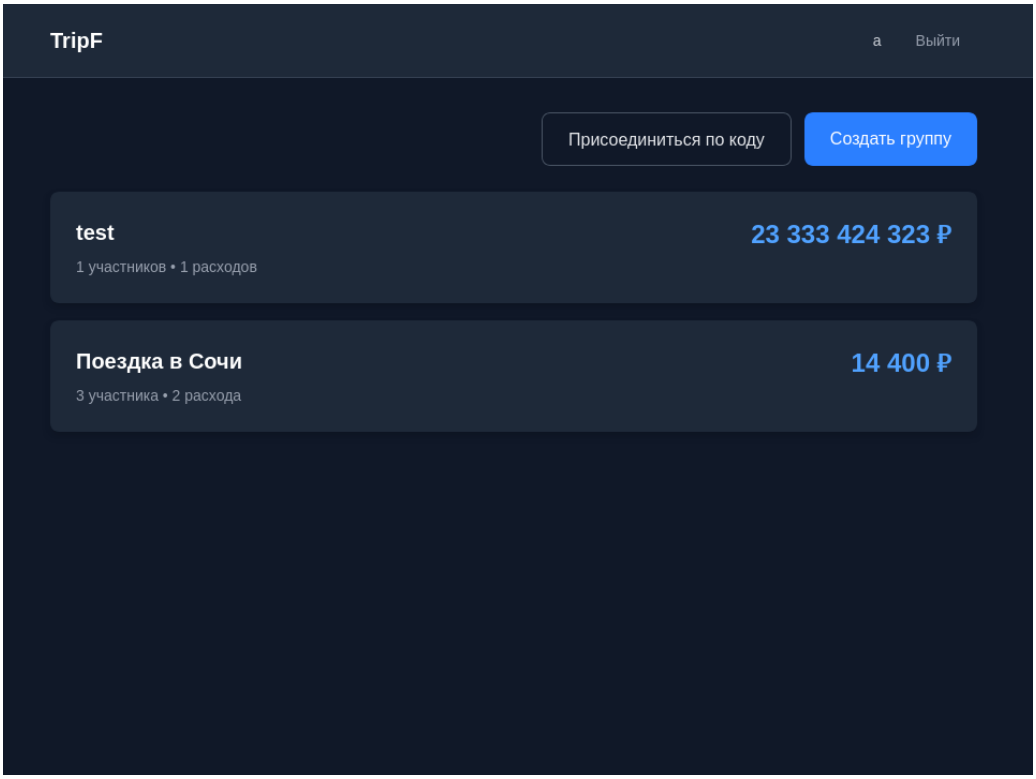

Рис. 14: Главная страница

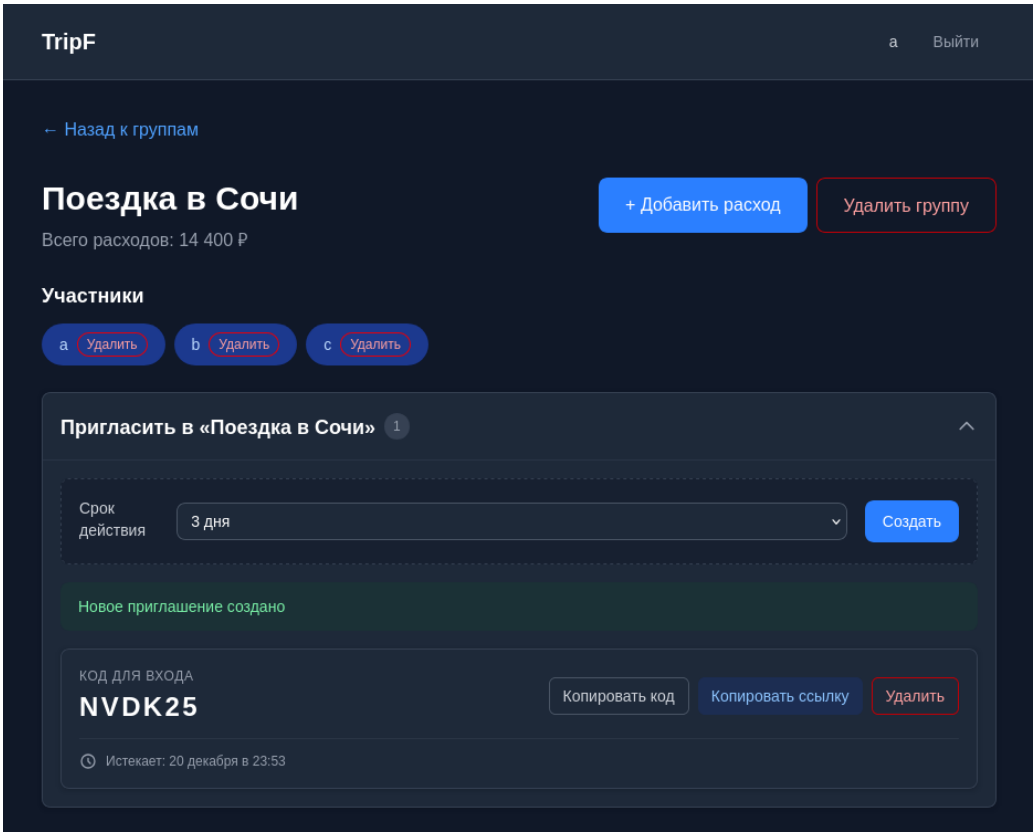

Рис. 15: Панель создания приглашения

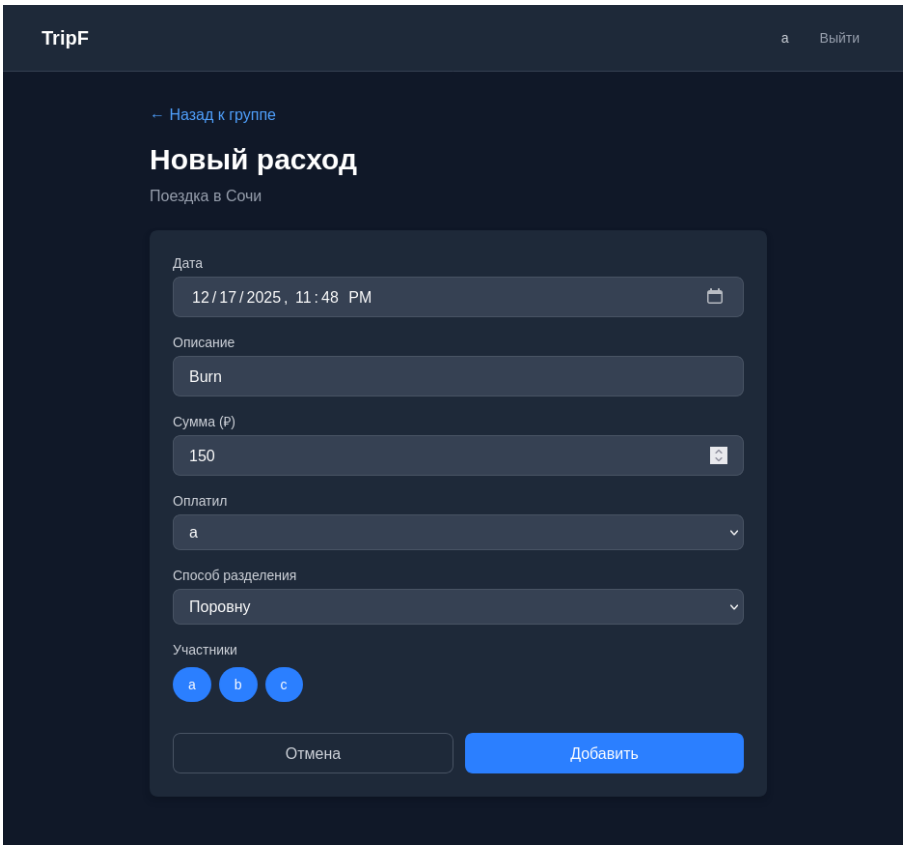

Рис. 16: Форма добавления расхода

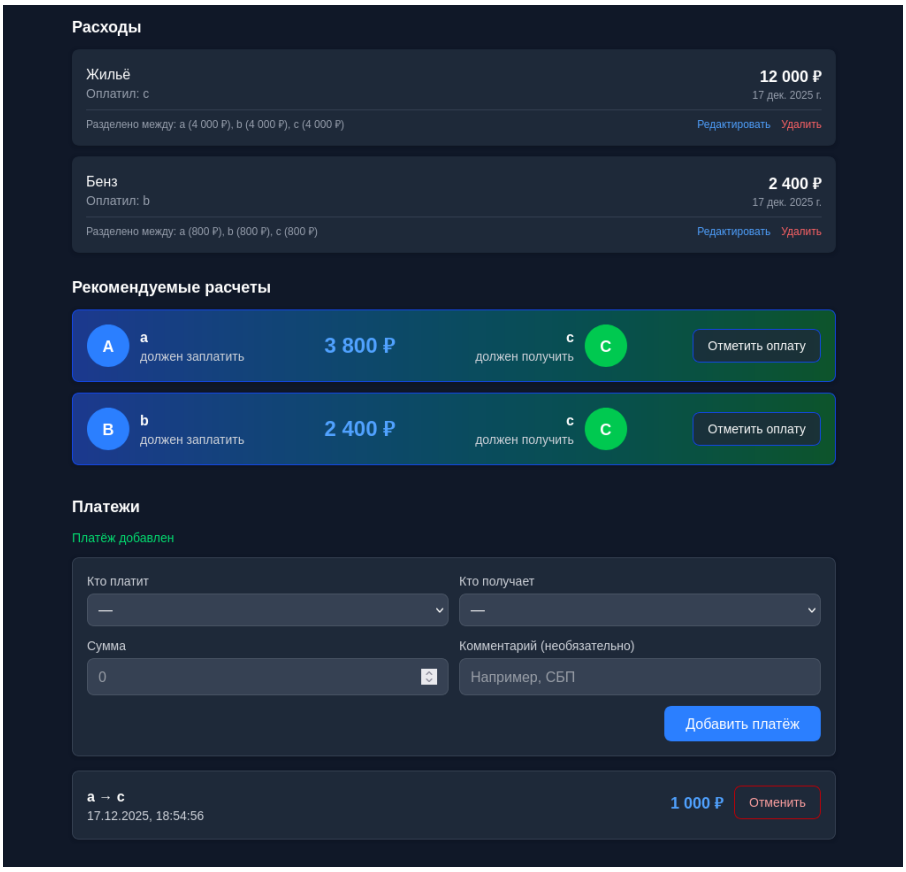

Рис. 17: Список расчётов