Курс NextJS. Извлечение данных из БД PostgreSQL

Д. Макаренков, к.т.н.

https://dmpsy.club

NextJS >= ReactJS + NodeJS

Целевая аудитория

Энтузиасты программирования на NextJS, желающие следовать современным принципам разработки full-stack Web-приложений

Магическая формула:

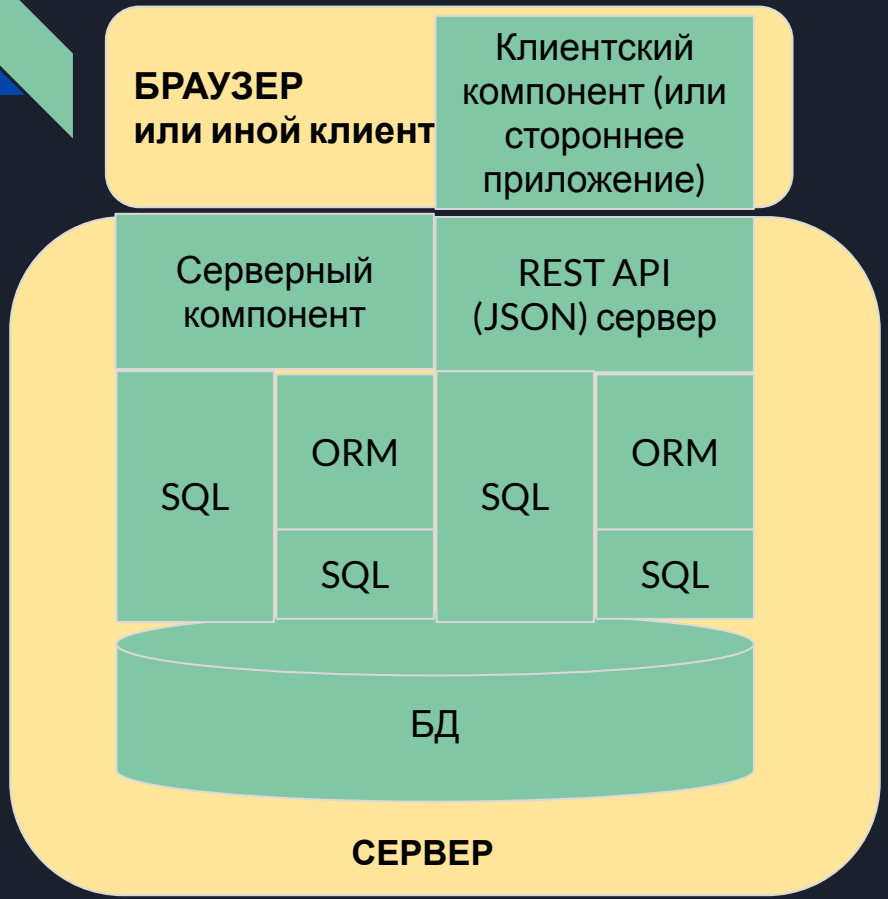
NextJS > = ReactJS (front-end) + NodeJS (back-end)

План работы

1	Basic ways of DB data fetching	Основные способы чтения из БД
2	Why using React Server Components + SQL	Зачем использовать React Server Components + SQL
3	Preliminary steps	Подготовительные шаги
4	Fetching into <RevenueChart>	Загрузка данных в <RevenueChart>
5	Fetching into <LatestInvoices>	Вывод последних счетов-фактур
6	Fetching data into <Cards>	Вывод агрегированных данных
7	Waterfall data fetching	“Каскадное” извлечение данных
8	Parallel data fetching	Параллельное извлечение данных
9	Actual app architecture	Текущая архитектура приложения

Официальная версия от NextJS: https://nextjs.org/learn/dashboard-app/fetching-data

Основные способы чтения из БД



Серверный компонент либо использует функции SQL напрямую, либо через слой ORM (Object-relational mapping, объектно-реляционного отображения) для чтения из БД

Клиентский компонент (или стороннее приложение) использует API для запросов к серверу, который уже транслирует их в SQL или методы ORM

Естественно, защищенность и надежность при использовании серверного компонента выше, но есть и свои минусы: информация, отправляемая в браузер, статична, требует перезагрузки для актуализации

Почему используем React Server Components + SQL?

Серверные компоненты (React Server Components) - относительно новый способ извлечения данных из БД, их преимущества:

- поддерживают promises, что важно для асинхронного чтения: используйте синтаксис ***async/await***, не обращаясь к хукам useEffect, useState и т.д.
- выполняются на сервере, поэтому вы можете хранить ресурсоемкие операции чтения данных и их логику на сервере, отсылая клиенту лишь конечный результат
- располагаясь на сервере, могут напрямую запрашивать БД, без привлечения дополнительного слоя API-функций.

Достоинства ***SQL (Structured Query Language)*** давно известны:

- SQL — это отраслевой стандарт для запросов к реляционным базам данных (например, ORM генерируют SQL “под капотом”)
- Базовое знание SQL может помочь вам понять основы всех реляционных баз данных (ORACLE, mysql, postgres и т.д.)
- SQL универсален, позволяет извлекать и обрабатывать весьма специфические данные
- Грамотно составленный, оптимизированный SQL-запрос позволяет добиться максимальной производительности БД

Образно говоря, SQL - это “ассемблер” для реляционных БД :)

Поэтому для чтения из БД будем использовать комбинацию серверных компонентов и вызовов функций SQL

Подготовительные шаги

1. Применяем скорректированный для использования пакета ***pg*** файл ***data.ts*** (это функции, содержащие SQL-запросы):

```
cd app/lib

mv data.ts data.ts.old

touch data.ts

nano data.ts
```

Скопируйте ***data.ts*** отсюда:

https://dmpsy.club/references/NextJS/lesson_007_data.ts

Подготовительные шаги (2)

2. Копируем страницу ***page.tsx*** - заготовку для панели мониторинга (dashboard):

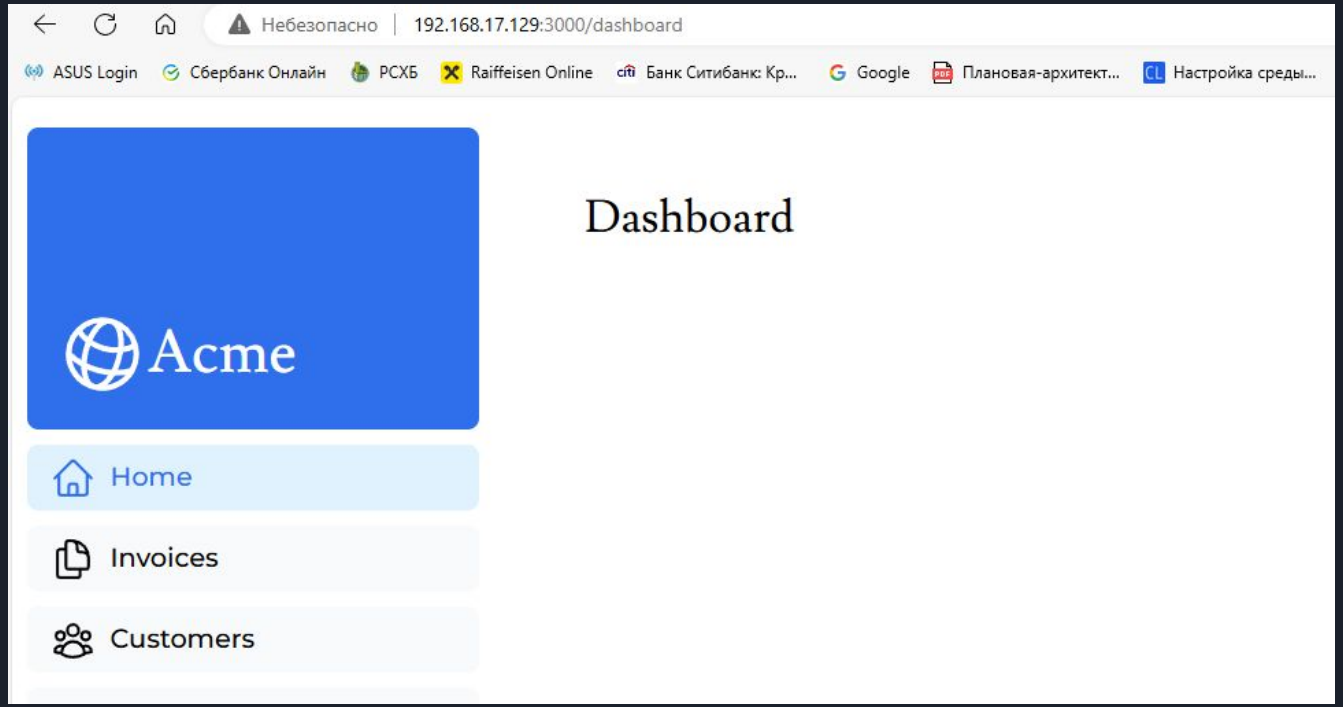
```
cd app/dashboard

mv page.tsx page.old

touch page.tsx

nano page.tsx
```

Скопируйте ***page.tsx*** отсюда:

https://dmpsy.club/references/NextJS/lesson_007_page.tsx

Подготовительные шаги (3)

3	**Launch nexxt-app and open localhost:3000/dashboard**	**Запустите приложение:** **cd nexxt-app** **pnpm dev** **Откройте http://localhost:3000\dashboard**



Загрузка данных в <RevenueChart/>

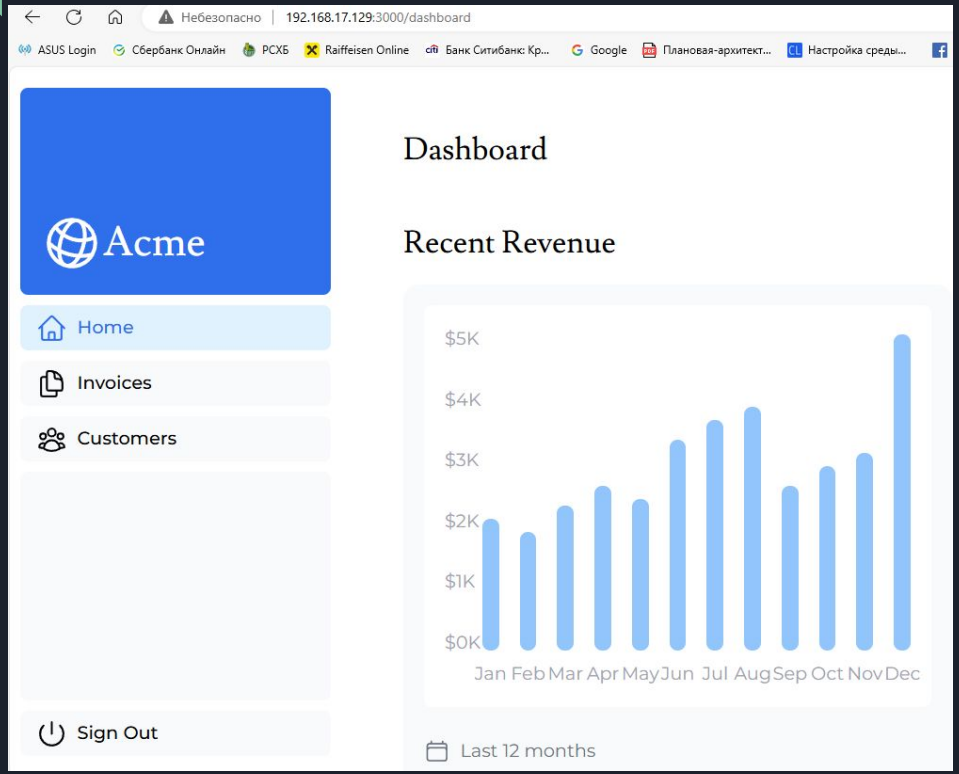
1	Import and call fetchRevenue() from data.ts	Импортируйте и вызовите fetchRevenue() из data.ts /app/dashboard/page.tsx: `import { Card } from '@/app/ui/dashboard/cards';` `import RevenueChart from '@/app/ui/dashboard/revenue-chart';` `import LatestInvoices from '@/app/ui/dashboard/latest-invoices';` `import { lusitana } from '@/app/ui/fonts';` `import { fetchRevenue } from '@/app/lib/data';` `export default async function Page() {` `  const revenue = await fetchRevenue();` `  // ...` `}`
2	Uncomment <RevenueChart/>	Раскомментируйте компонент <RevenueChart/>: `// ...` `<div className="mt-6 grid grid-cols-1 gap-6 md:grid-cols-4 lg:grid-cols-8">` `<RevenueChart revenue={revenue} />` `{/* <LatestInvoices latestInvoices={latestInvoices} /> */}` `</div>` `</main>` `);` `}`

Загрузка данных в <RevenueChart/> (2)

2	**Uncomment the code inside the <RevenueChart/> component**	**Раскомментируйте код внутри <RevenueChart/> /app/ui/dashboard/revenue-chart.tsx:** // … {/* NOTE: Uncomment this code in Chapter 7 */} {/* // … */}); }

Загрузка данных в <RevenueChart/> (3)

3	**Save changes and open localhost:3000/dashboard**	**Сохраните изменения и откройте** http://localhost:3000\dashboard



Вывод пяти последних счетов-фактур

Для компонента ***<LatestInvoices />*** нам нужно вывести последние 5 счетов-фактур, отсортированных по дате. Счета можно сортировать и с помощью JavaScript, но по мере роста БД значительно увеличится объем данных, передаваемых по каждому запросу, поэтому вместо сортировки счетов в памяти (JavaScript) используем SQL-запрос для получения только последних 5 счетов-фактур:

/app/lib/data.ts:

```
// …
export async function fetchLatestInvoices() {
//…
await client.connect();
    const result = await client.query(`
    select i.amount, c.name, c.image_url, c.email, i.id
      from invoices i, customers c
      where i.customer_id = c.id
      order by i.date desc
      limit 5
      `);
    await client.end();
//…
}
```

Вывод пяти последних счетов-фактур (2)

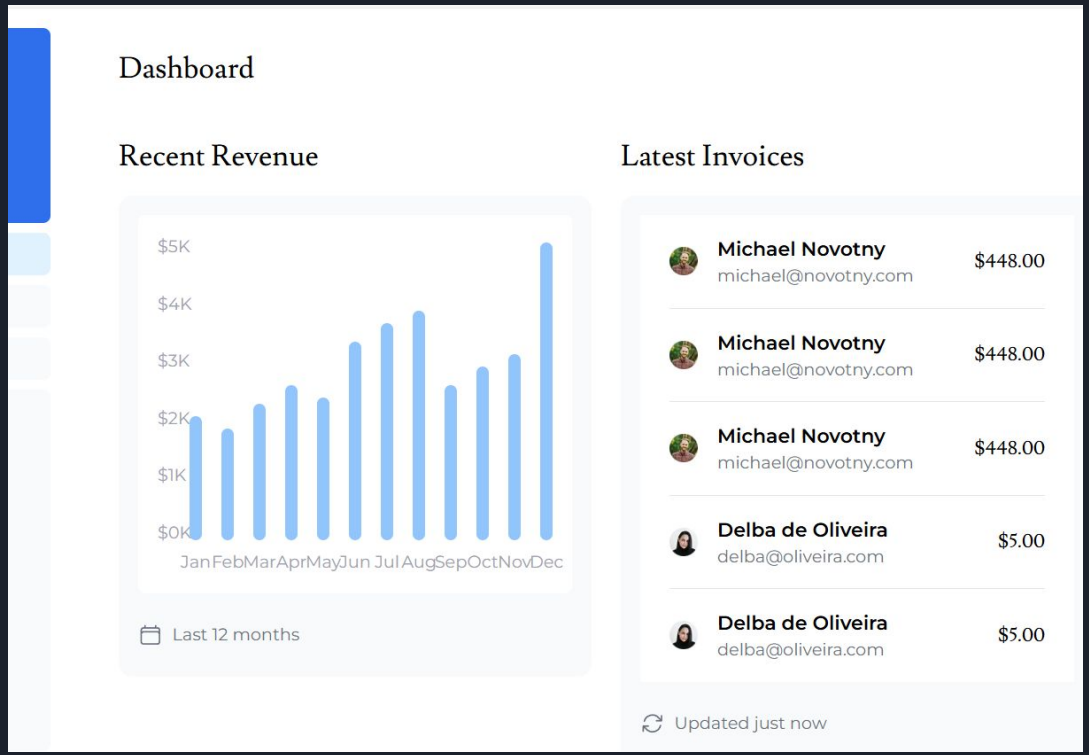
1	Import and call fetchLatestInvoices() from data.ts	**Импортируйте и вызовите fetchLatestInvoices() из data.ts** **/app/dashboard/page.tsx:** `import { Card } from '@/app/ui/dashboard/cards';` `import RevenueChart from '@/app/ui/dashboard/revenue-chart';` `import LatestInvoices from '@/app/ui/dashboard/latest-invoices';` `import { lusitana } from '@/app/ui/fonts';` `import { fetchRevenue, fetchLatestInvoices } from '@/app/lib/data';` `export default async function Page() {` `  const revenue = await fetchRevenue();` `  const latestInvoices = await fetchLatestInvoices();` `  // ...` `}`
2	Uncomment <LatestInvoices/>	**Раскомментируйте компонент <LatestInvoices/>:** `// ...` `<div className="mt-6 grid grid-cols-1 gap-6 md:grid-cols-4 lg:grid-cols-8">` `<RevenueChart revenue={revenue}  />` `<LatestInvoices latestInvoices={latestInvoices} />` `</div>` `</main>` `);` `}`

Вывод пяти последних счетов-фактур (3)

3	Uncomment the code inside the <LatestInvoices/> component	Раскомментируйте код внутри <LatestInvoices/> /app/ui/dashboard/revenue-chart.tsx: // … {/* NOTE: Uncomment this code in Chapter 7 */} {/* // … */} <ArrowPathIcon className="h-5 w-5 text-gray-500" /> <h3 className="ml-2 text-sm text-gray-500 ">Updated just now</h3>); }

Вывод пяти последних счетов-фактур (4)

4	**Save changes and open localhost:3000/dashboard**	**Сохраните изменения и откройте** http://localhost:3000\dashboard



Вывод агрегированных данных в компоненты <Card />

В компоненты ***<Card />*** выведем агрегированные данные по клиентам (“всего”) и счетам-фактурам(“всего”,”оплачено”,“ожидают оплаты”). Снова можно использовать JavaScript, но намного более эффективно использовать SQL, предоставляя в браузер лишь итоговые цифры:

/app/lib/data.ts:

```
// …
export async function fetchCardData() {
//…
await client.connect();
    const invoiceCountPromise = await client.query(`select count(*) from invoices`);
    const customerCountPromise = await client.query(`select count(*) from customers`);
    const invoiceStatusPromise = await client.query(`
     select
       sum(case when status = 'paid' then amount else 0 end) as "paid",
       sum(case when status = 'pending' then amount else 0 end) as "pending"
       from invoices`);
   await client.end();
//…
}
```

Вывод агрегированных данных в компоненты <Card/> (2)

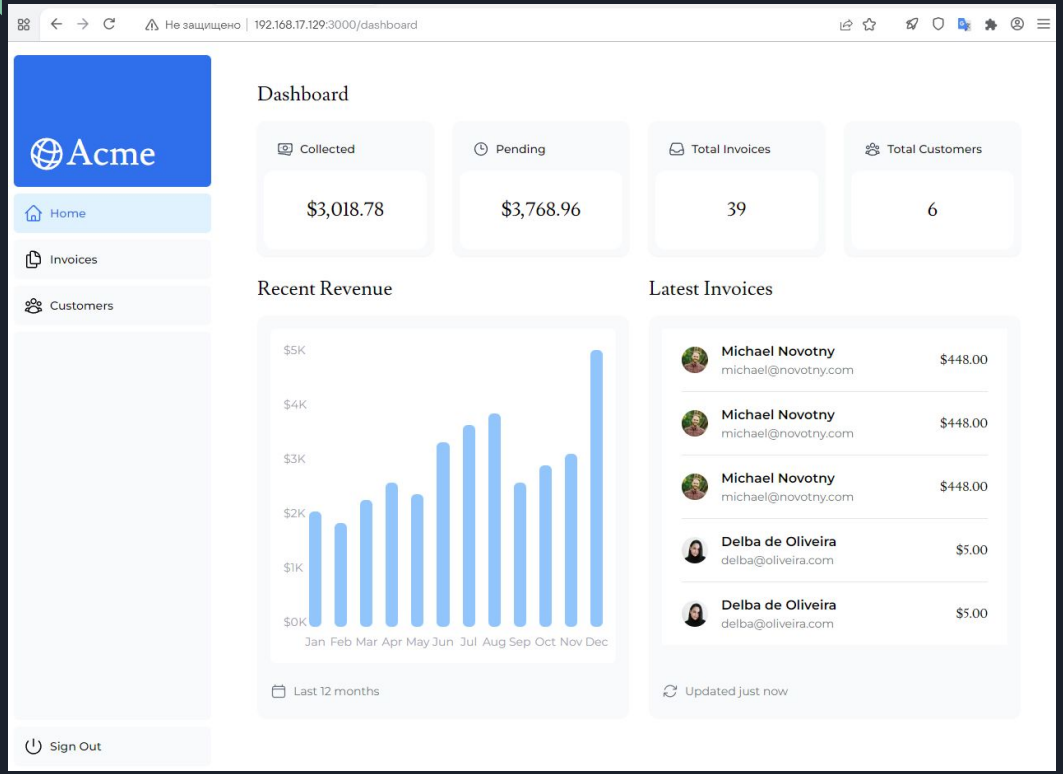
1	Finalize page.tsx	**Приведите код page.tsx к окончательному виду /app/dashboard/page.tsx:** `import { Card } from '@/app/ui/dashboard/cards';` `import RevenueChart from '@/app/ui/dashboard/revenue-chart';` `import LatestInvoices from '@/app/ui/dashboard/latest-invoices';` `import { lusitana } from '@/app/ui/fonts';` `import { fetchRevenue, fetchLatestInvoices, fetchCardData, } from '@/app/lib/data';` `export default async function Page() {` `  const revenue = await fetchRevenue();` `  const latestInvoices = await fetchLatestInvoices();` `  const {` `  numberOfInvoices,` `  numberOfCustomers,` `  totalPaidInvoices,` `  totalPendingInvoices,` `  } = await fetchCardData();` `//…см. след.страницу`

Вывод агрегированных данных в компоненты <Card/> (2a)

1	Finalize page.tsx	// …окончание return (<main> <h1 className={`${lusitana.className} mb-4 text-xl md:text-2xl`}> Dashboard </h1> <Card title="Collected" value={totalPaidInvoices} type="collected" /> <Card title="Pending" value={totalPendingInvoices} type="pending" /> <Card title="Total Invoices" value={numberOfInvoices} type="invoices" /> <Card title="Total Customers" value={numberOfCustomers} type="customers" /> <div className="mt-6 grid grid-cols-1 gap-6 md:grid-cols-4 lg:grid-cols-8"> <RevenueChart revenue={revenue} /> <LatestInvoices latestInvoices={latestInvoices} /> </main>); }

Вывод агрегированных данных в компоненты <Card/> (3)

2	**Save changes and open localhost:3000/dashboard**	**Сохраните изменения и откройте** http://localhost:3000\dashboard



Каскадное (waterfall) извлечение данных

«***Каскад» (waterfall)*** - такая последовательность запросов, в которой каждый следующий запрос может начаться только после того, как предыдущий вернул данные:

/app/dashboard/page.tsx:

```
// …
export default async function Page() {
  const revenue = await fetchRevenue();
  const latestInvoices = await fetchLatestInvoices(); // ждем окончания fetchRevenue()
  const {
    numberOfInvoices,
    numberOfCustomers,
    totalPaidInvoices,
    totalPendingInvoices,
  } = await fetchCardData(); // ждем завершения fetchLatestInvoices()
// …
}
```

Подобный вариант необходим, когда последующий запрос использует данные предыдущего. Если же запросы независимы друг от друга, можно запустить их параллельно и добиться, таким образом, повышения производительности

Параллельное извлечение данных

Используйте параллельное извлечение данных (parallel data fetching) с помощью ***Promise*** от JavaScript:

- Повышаете производительность за счет одновременного старта группы запросов
- Задействуете стандартный шаблон JavaScript, применимый к любой библиотеке или фреймворку.

NB. Остается вопрос: что, если один из группы запросов выполняется намного медленней остальных? Об этом - в следующей лекции о Static and Dynamic Rendering

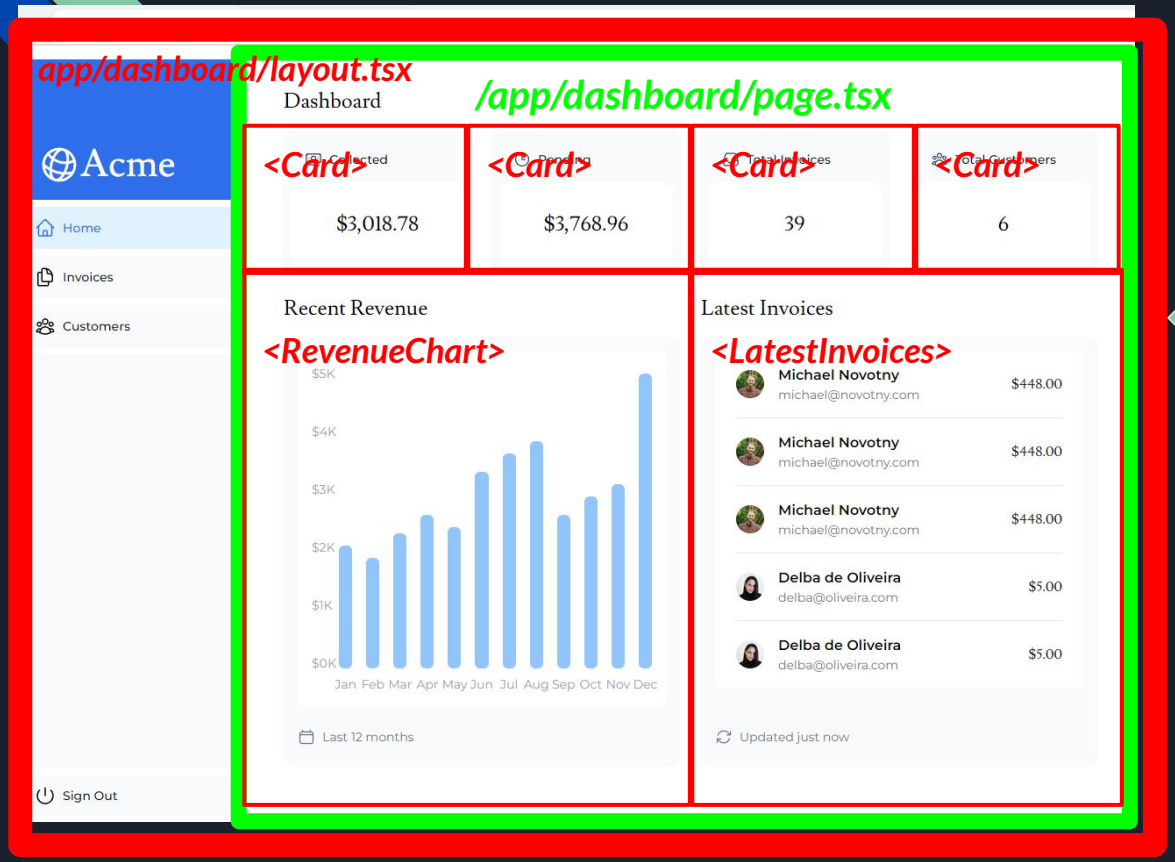
Параллельное извлечение данных (2)

Все запросы инициируются одновременно с помощью ***Promise.all()*** или ***Promise.allSettled()*** стандартного JavaScript, например:

/app/lib/data.ts:

```
export async function fetchCardData() {
  //…
  await client.connect();
  const invoiceCountPromise = await client.query(`select count(*) from invoices`);
  const customerCountPromise = await client.query(`select count(*) from customers`);
  const invoiceStatusPromise = await client.query(`
    select
      sum(case when status = 'paid' then amount else 0 end) as "paid",
      sum(case when status = 'pending' then amount else 0 end) as "pending"
      from invoices`);
  await client.end();
  const data = await Promise.all([
    invoiceCountPromise,
    customerCountPromise,
    invoiceStatusPromise,
  ]);
//…
```

Текущая архитектура приложения



/app/lib/data.ts
fetchRevenue()
fetchLatestInvoices()
fetchCardData()

SQL

nexxtapp-db

Подведем итоги

1	**Обсудили основные способы извлечения данных: API, ORM, SQL и т. д.**
2	**Поняли, каким образом серверные компоненты обеспечивают более безопасный доступ к ресурсам БД.**
3	**Узнали о “каскадном” (waterfall) способе извлечения данных и границах его применимости**
4	**Реализовали параллельное извлечение данных (parallel data fetching) с использованием шаблона Promise JavaScript с целью повышения производительности кода**

Полезные ссылки

1	**The SQL Standard – ISO/IEC 9075:2023 (ANSI X3.135)**	https://blog.ansi.org/sql-standard-iso-iec-9075-2023-ansi-x3-135/
2	**Примеры использования пакета pg для работы с PostreSQL DB**	https://node-postgres.com/apis/client

Замечания / Исправления

1	**pnpm build failed with**	**Type error: Could not find a declaration file for module 'pg'.**
2	**Solution:**	**pnpm install –save @types/pg**

Замечания / Исправления (2)

3	**Result:**	**pnpm build**

```
dmitry-makarenkov@dmitry-makarenkov-VMware-Virtual-Platform:~/NextJSTraining/nexxt-app$
pnpm build

> @ build /home/dmitry-makarenkov/NextJSTraining/nexxt-app
> next build

  ▲ Next.js 14.2.5
  - Environments: .env

   Creating an optimized production build ...
 ✓ Compiled successfully
 ✓ Linting and checking validity of types
 ✓ Collecting page data
 ✓ Generating static pages (8/8)
 ✓ Collecting build traces
 ✓ Finalizing page optimization

Route (app)                              Size     First Load JS
┌ ○ /                                    230 B          99.1 kB
├ ○ /_not-found                          872 B          87.9 kB
├ ○ /dashboard                           293 B          92.4 kB
├ ○ /dashboard/customers                 142 B          87.2 kB
├ ○ /dashboard/invoices                  142 B          87.2 kB
└ ○ /seed                                0 B                0 B
+ First Load JS shared by all            87 kB
  ├ chunks/842-1430a516d2880397.js       31.5 kB
  ├ chunks/94c12b52-5340f2512ce19773.js  53.6 kB
  └ other shared chunks (total)          1.9 kB
```

Следующая лекция -
8. Static and Dynamic Rendering / Статическая и динамическая отрисовка

Презентация доступна для скачивания здесь:

https://dmpsy.club/references/NextJS/lesson_007_fetching_data_rus.pdf

Загрузить приложение с GitHub:
https://github.com/DmitryMakar/nextjs-lesson-007

Поддержать автора: https://www.donationalerts.com/r/dmitrymak

https://dmpsy.club
postmaster@dmspy.club

https://www.donationalerts.com/r/dmitrymak