Курс NextJS. Поиск и постраничный вывод

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 / Agenda

1	Search & Pagination in Next.js. Statement of the problem	Поиск и постраничный вывод в Next.js. Постановка задачи
2	Benefits of using URL search parameters	Преимущества использования параметров поиска URL
3	Invoices page starting code	Заготовка страницы Invoices
4	Next.js client-side API hooks	Клиентские хуки Next.js
5	Search. Implementation steps	Поиск. Последовательность реализации
6	Search. Capturing the user's input	Поиск. Захватываем введенные данные
7	Search. Add the query URL param	Поиск. Добавляем параметр query в URL

Официальная версия от NextJS: https://nextjs.org/learn/dashboard-app/adding-search-and-pagination

План работы / Agenda (2)

8	Search. Synchronizing URL and <input>	Поиск. Синхронизируем URL и <input>
9	Search. Updating the invoice table data	Поиск. Обновляем таблицу инвойсов
10	Search. Excessive bouncing issue	Поиск. Проблема излишних запросов
11	Implementing pagination	Реализуем постраничный вывод
12	Component interaction diagram	Схема взаимодействия компонентов
13	Dashboard static compilation issue	Проблема статической компиляции Dashboard
14	Static, dynamic rendering, and (PPR) partial pre-rendering	Статический, динамический рендеринг и PPR
15	Summary & Closing comments	Итоги и заключительные замечания

План работы / Agenda (3)

A	**Launching a project testbed**	**Запуск тестовой среды** **Slide 47: Приложение A. Запуск рабочей среды проекта**
B	**Stopping & Removing the testbed**	**Останов и удаление тестовой среды** **Slide 52: Приложение B. Останов и очистка рабочей среды проекта**

Реализация поиска и постраничного вывода в NextJS. Постановка задачи

В предыдущей главе мы улучшили начальную производительность загрузки панели мониторинга (dashboard) с помощью потоковой передачи. Теперь создадим страницу инвойсов (invoices) и узнаем, как добавить поиск и пагинацию (постраничный вывод)

Задача стандартная, посмотрим, как ее эффективно реализовать средствами NextJS.

Напомним, что в NextJS существуют как клиентские (client-side), так и серверные (server-side, default) компоненты, в отличие от React, в котором все содержимое - “клиентское”

За взаимодействие с пользователем (например, ввод данных) отвечают клиентские компоненты, за обращения в БД - серверные. Нам необходимо организовать взаимодействие между этими компонентами в рамках одной страницы invoices

Есть разные варианты реализации подобного (frontend-backend) взаимодействия, например, с использованием API-сервера на backend, взаимодействующего с БД. Мы же решим задачу иначе - через параметры поиска запроса (GET) URL на странице инвойсов

Преимущества использования параметров поиска URL

Подход может показаться довольно экзотичным, но он обладает рядом преимуществ:

1. Поисковые URL-адреса можно добавлять в закладки: поскольку параметры поиска находятся в URL-адресе, пользователи могут добавлять в закладки текущее состояние приложения, включая поисковые запросы и фильтры, для дальнейшего (совместного) использования
2. Рендеринг на стороне сервера: параметры URL-адреса напрямую используются серверным компонентом для формирования запроса в БД и последующей отрисовки полученных данных
3. Аналитика и логирование: наличие поисковых запросов и фильтров непосредственно в URL-адресе упрощает отслеживание действий пользователя

Заготовка страницы Invoices

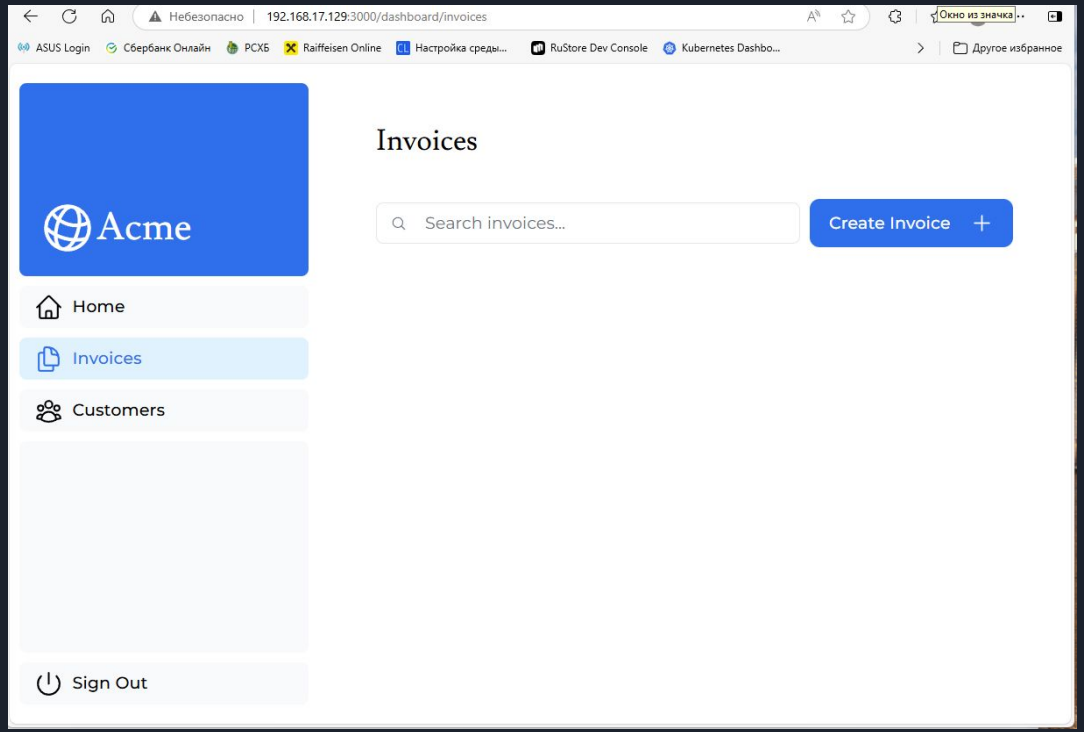
0	**Launch the project environment**	**Склонируйте и запустите проект:** **Slide 47: Приложение A. Запуск рабочей среды проекта**
1	**Examine the page.tsx file**	**Проанализируйте исходную структуру страницы app/dashboard/invoices/page.tsx:**

Заготовка страницы Invoices (2)

<Search />	**Блок ввода строки поиска (и создания) инвойсов, клиентский (client-side) компонент, формирует параметр URL *query***
<Table />	**Таблица инвойсов, изменяется в зависимости от результатов выполнения запроса с параметрами {*query*, *page*} в БД, серверный (server-side) компонент**
<Pagination />	**Блок навигации по страницам таблицы инвойсов, клиентский (client-side) компонент, формирует параметр URL *page***

Заготовка страницы Invoices (3)

3	**Open and refresh localhost:3000/dashboard/invoices**	**Откройте и обновите страницу localhost:3000/dashboard/invoices**



Используем клиентские хуки Next.js

Клиентские компоненты (<Search / > и <Pagination />) работают со строкой URL, используя следующие хуки

useSearchParams	**Позволяет получить доступ к параметрам текущего URL. Например, параметры поиска для этого URL /dashboard/invoices?page=1&query=pending будут выглядеть так: {page: '1', query: 'pending'}**
usePathname	**Позволяет прочитать текущий путь URL. Например, для маршрута /dashboard/invoices usePathname вернет '/dashboard/invoices'**
useRouter	**Позволяет осуществлять навигацию между маршрутами в клиентских компонентах программным способом. Существует несколько методов, которые вы можете использовать (в нашем случае это replace)**

Поиск. Последовательность реализации

Реализуем поиск следующим образом:

1. Захватываем вводимые пользователем данные.
2. Добавляем в URL (запрос GET) параметры поиска (query)
3. Устанавливаем синхронизацию строки URL с полем ввода <input />
4. Забираем параметры поиска (query) из строки URL в компонент <Table>
5. Обновляем таблицу, чтобы отразить результаты поиска (query) в БД

Поиск. Захватываем пользовательские данные

1	**Update the search.tsx file**	**Обновите файл app/ui/search.tsx:** **cp search.tsx.01.CaptureUserInput search.tsx**
2	**Examine the search.tsx file**	**Проанализируйте изменения в файле app/ui/search.tsx:**

1. "use client" — это клиентский компонент: вы можете использовать event listeners и хуки
2. <input> — здесь пользователь вводит свои данные
3. К элементу <input> добавлен обработчик события onChange, который будет вызывать функцию handleSearch всякий раз, когда изменяется входное значение в поле <input>
4. handleSearch выводит введенную пользовательскую строку в консоль

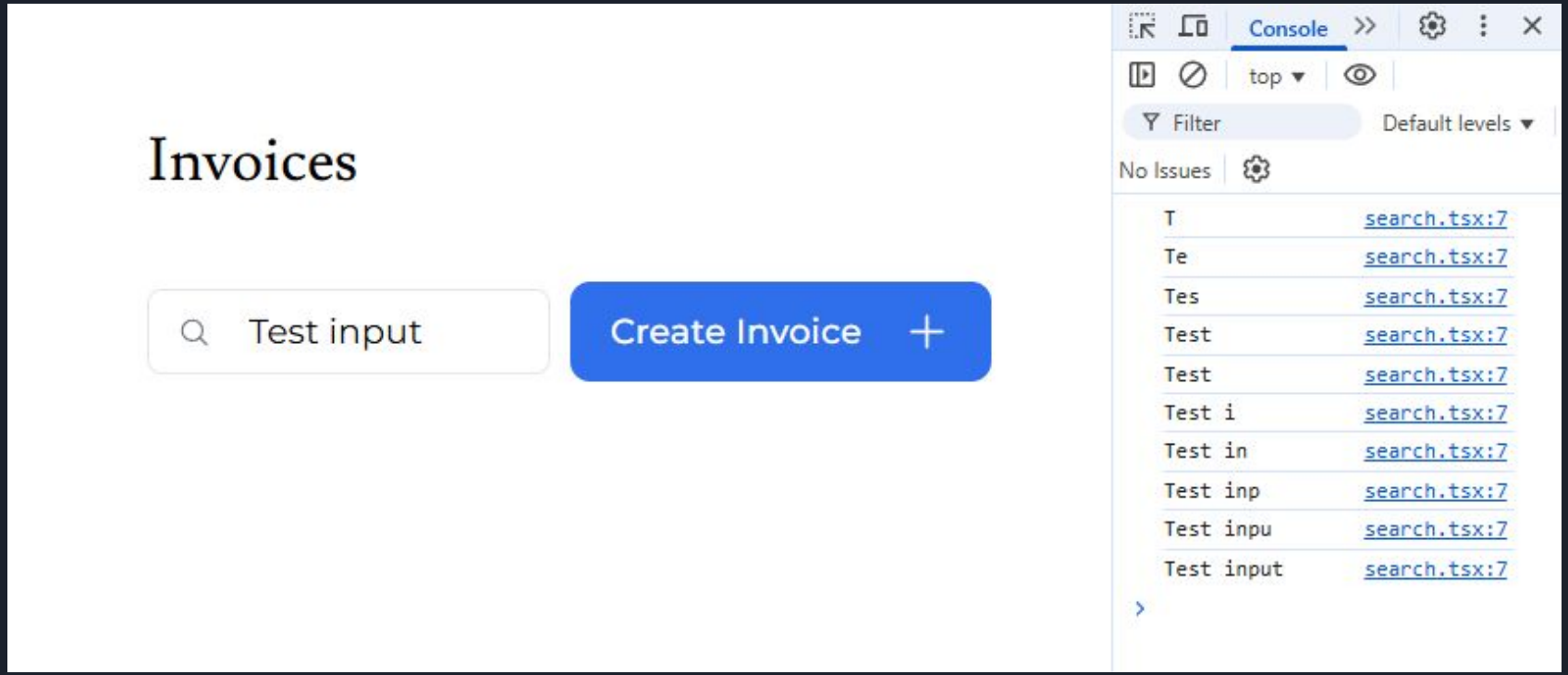
Поиск. Захватываем пользовательские данные (2)

/app/ui/search.tsx:

```
'use client';  # This is a client-side component
…
export default function Search({ placeholder }: { placeholder: string }) {
  function handleSearch(term: string) {
    console.log(term);    #Output the user data to the console
  }

  return (
 …
      <input
        className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm
outline-2 placeholder:text-gray-500"
        placeholder={placeholder}
        onChange={(e) => {
          handleSearch(e.target.value);
        }}
      />
…
}
```

Поиск. Захватываем пользовательские данные (3)

3	**Reload localhost:3000/dashboard/invoices, open DevTools and test the input**	**Перезагрузите страницу localhost:3000/dashboard/invoices Откройте DevTools и протестируйте ввод**



Поиск. Добавляем параметр поиска (query) в URL

1	**Update the search.tsx file**	**Обновите файл app/ui/search.tsx:** **cp search.tsx.02.UpdateURLWithSearchParams.tsx**
2	**Examine the search.tsx file**	**Проанализируйте изменения в файле app/ui/search.tsx:**

Импортируем и далее используем для передачи параметра в строку браузера три клиентских хука: ***useSearchParams, usePathname, useRouter*** внутри функции ***handleSearch(),*** которая срабатывает каждый раз при изменении строки пользовательского ввода <input>. В результате формируется новая строка с “хвостом” в виде параметра ?query=<String>, где String - строка, введенная пользователем

NB URL-адрес обновляется плавно, без перезагрузки страницы благодаря клиентской (client-side) навигации Next.js (которую уже обсуждали в лекции о навигации между страницами)

Тем не менее, этот запрос GET вполне рабочий, он может быть скопирован и использован повторно, например, другим пользователем и в другом браузере

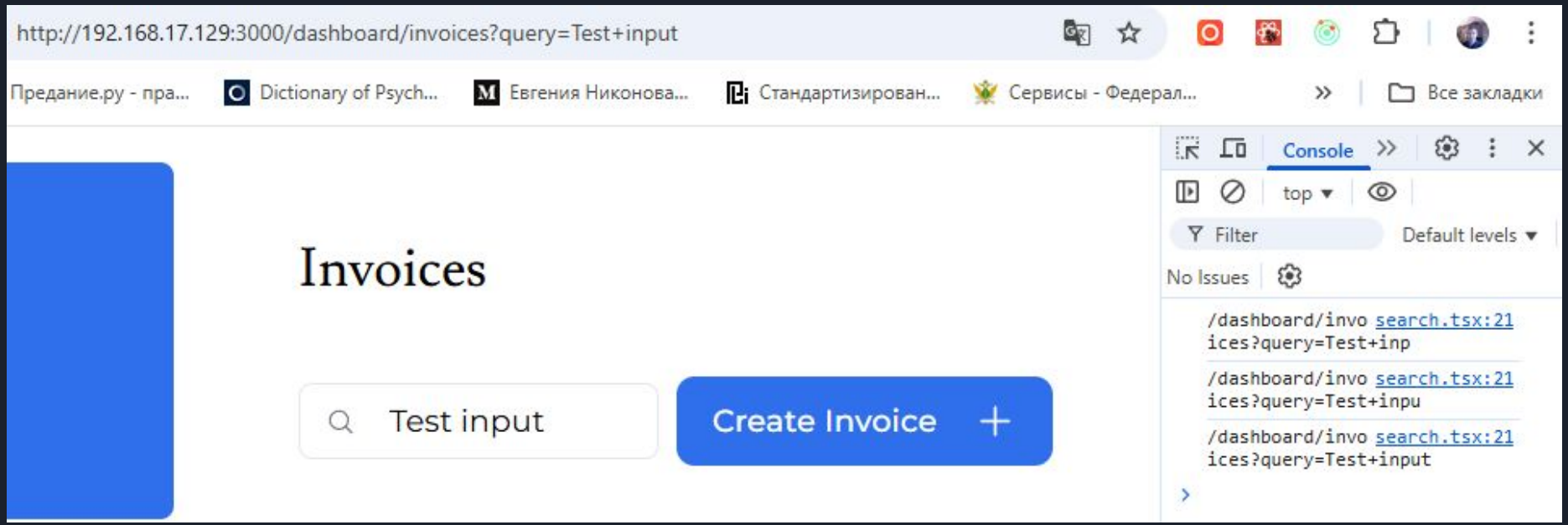
Поиск. Добавляем параметр поиска (query) в URL (2)

/app/ui/search.tsx:

```
'use client';
…
import { useSearchParams, usePathname, useRouter } from 'next/navigation';

export default function Search({ placeholder }: { placeholder: string }) {
  const searchParams = useSearchParams();
  const pathname = usePathname();
  const { replace } = useRouter();

  function handleSearch(term: string) {
    const params = new URLSearchParams(searchParams); //For manipulating URL query
    if (term) {
      params.set('query', term);
    } else {
      params.delete('query');
    }
    replace(`${pathname}?${params.toString()}`);  //final URL string replaces the current one
    }
…
  return (...);
}
```

Поиск. Добавляем параметр поиска (query) в URL (3)

3	**Reload localhost:3000/dashboard/invoices, open DevTools and test the input**	**Перезагрузите страницу localhost:3000/dashboard/invoices Откройте DevTools и протестируйте ввод Параметр ?query=<...> появляется в строке запроса браузера**



Поиск. Синхронизируем URL и <input>

1	**Update the search.tsx file**	**Обновите файл app/ui/search.tsx:** **cp search.tsx.03.KeepURLandInputInSync search.tsx**
2	**Examine the search.tsx file**	**Проанализируйте изменения в файле app/ui/search.tsx:**

Синхронизируем поле ввода **<input>** со значением параметра **query** из URL: читаем его из ***searchParameters*** и копируем в ***defaultValue*** элемента **<input>**. При перезагрузке страницы актуальное значение query появляется в строке **<input>**

NB defaultValue или value ?

Если вы используете **state** для управления значением ввода, вы бы использовали атрибут **value**, чтобы сделать его контролируемым компонентом. В этом случае React будет управлять состоянием ввода. Однако, мы не используем **state**, воспользовавшись взамен **defaultValue**. Это означает, что **<input>** будет управлять своим собственным состоянием. Это корректно, поскольку мы сохраняем поисковый запрос в **URL**, а не в **state**.

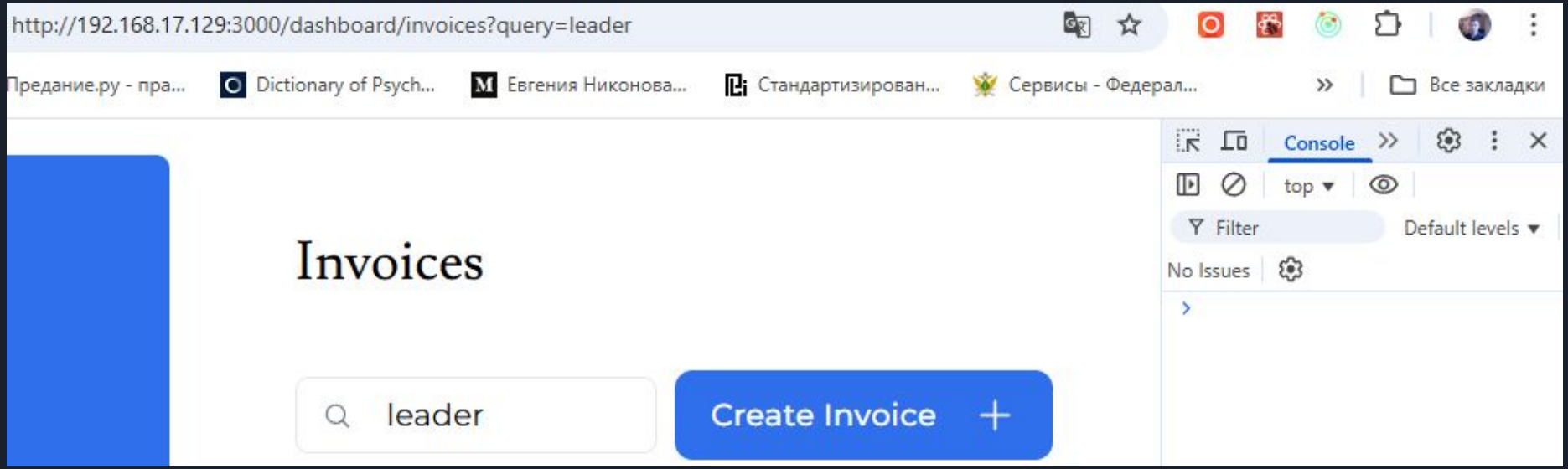
Поиск. Синхронизируем URL и <input> (2)

/app/ui/search.tsx:

```
…
<input
        className="peer block w-full rounded-md border border-gray-200 py-[9px] pl-10 text-sm
outline-2 placeholder:text-gray-500"
        placeholder={placeholder}
        onChange={(e) => {
          handleSearch(e.target.value);
        }}
        defaultValue={searchParams.get('query')?.toString()}
/>
…
# we populate the <input> box with the query parameter value of the GET request
# just performed on reloading the page:
#GET /dashboard/invoices?query=leader 200 in 1674m
```

Поиск. Синхронизируем URL и <input> (3)

3	**Reload localhost:3000/dashboard/invoices, open DevTools and test the synch**	**Перезагрузите страницу localhost:3000/dashboard/invoices Откройте DevTools и протестируйте синхронизацию Значение, введенное в ?query=<...>, отображается в строке <input> после обновления страницы**



Поиск. Обновляем таблицу инвойсов

1	**Update the page.tsx file**	**Обновите файл app/dashboard/invoices/page.tsx:** **cp page.tsx.01.UpdatingTheTable page.tsx**
2	**Examine the page.tsx file**	**Проанализируйте изменения в файле app/dashboard/invoices/page.tsx:**

Страницы **page.tsx** обладают параметрами (props) **searchParams**, с информацией о текущем URL страницы. Серверный компонент <Table /> воспользуется **searchParams.?query** в качестве своего параметра **query** и вызовет серверную функцию **fetchFilteredInvoices(query,...)**, которая создаст запрос (select) в БД и вернет инвойсы для обновленной таблицы. В результате, таблица будет обновлена в соответствии с результатами запроса

Поиск. Обновляем таблицу инвойсов (2)

/app/dashboard/invoices/page.tsx:

```
…
export default async function Page(props: {
  searchParams?: Promise<{
    query?: string;
    page?: string;
  }>;
}) {
  const searchParams = await props.searchParams;
  const query = searchParams?.query || '';
  const currentPage = Number(searchParams?.page) || 1;

  return (
…
  <Suspense key={query + currentPage} fallback={<InvoicesTableSkeleton />}>
        <Table query={query} currentPage={currentPage} />
  </Suspense>
…
            );
}
```

Поиск. Обновляем таблицу инвойсов (3)

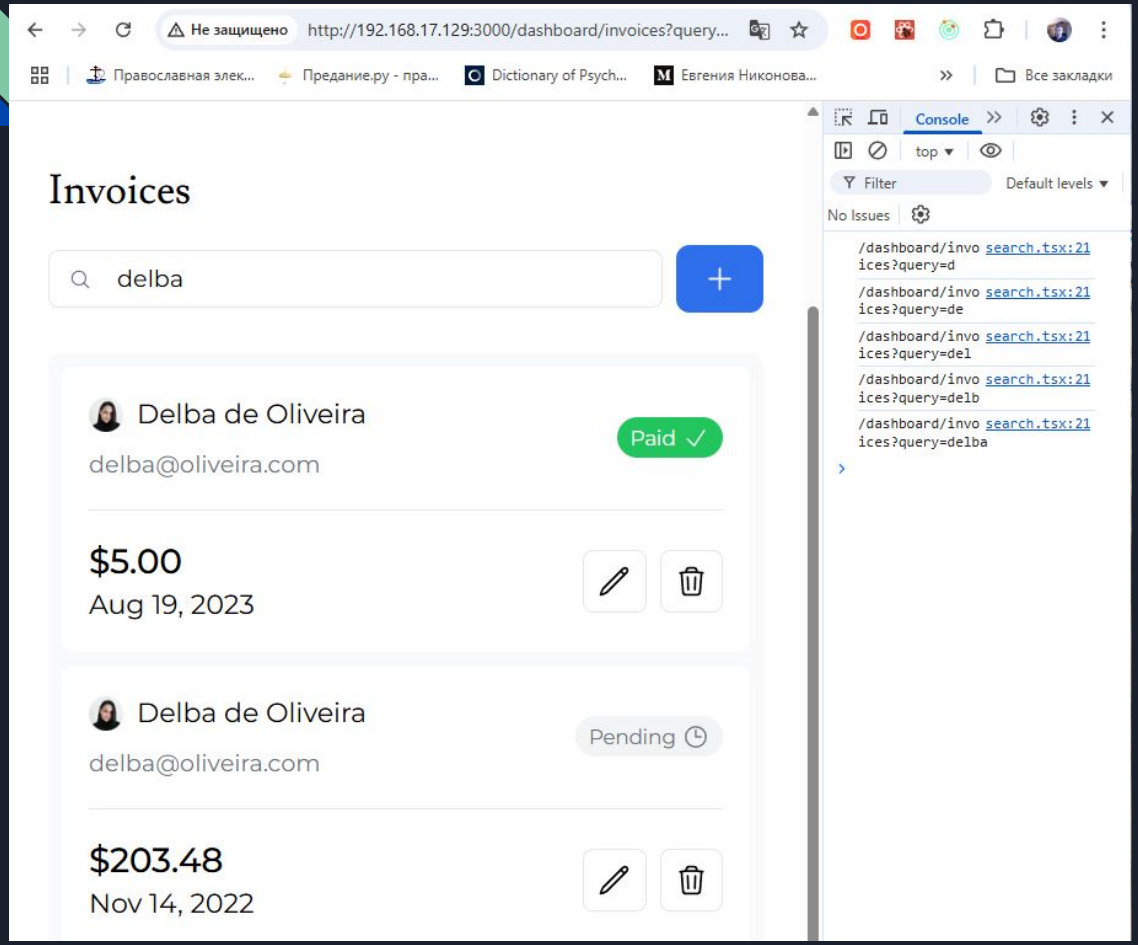
/app/ui/invoices/table.tsx:

```
…
import { fetchFilteredInvoices } from '@/app/lib/data'; #import the server-side function

export default async function InvoicesTable({
  query,
  currentPage,
}: {
  query: string;
  currentPage: number;
}) {
  const invoices = await fetchFilteredInvoices(query, currentPage); #Select invoices from DB
…
```

NB Когда используем хук useSearchParams() и searchParams prop?

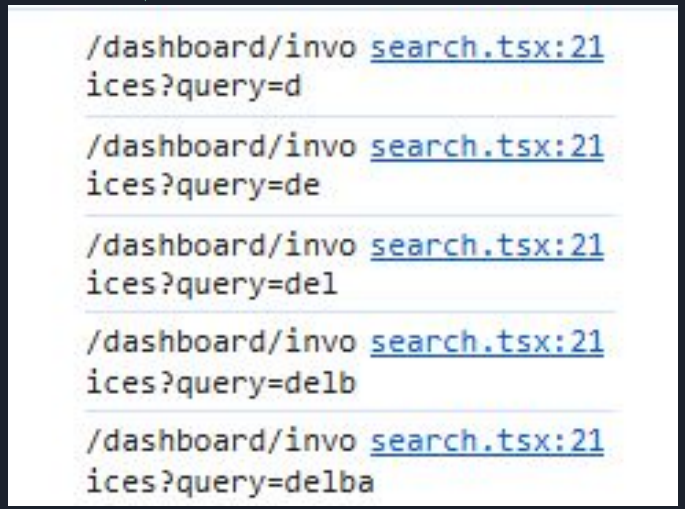
- В Клиентских компонентах (‘use client’), таких как <Search />, используем useSearchParams()
- В Серверных компонентах, например <Table />, пользуемся searchParams страницы page.tsx

Поиск. Обновляем таблицу инвойсов (4)



3	Reload localhost:3000/dashboard/invoices, and perform a test query	Перезагрузите страницу localhost:3000/dashboard/invoices и сделайте тестовый запрос

Поиск. Решаем проблему излишних запросов к БД (Excessive Bouncing)



Проблема:

Параметр query при вводе пользователя в поле <input> формируется посимвольно, и каждое добавление символа сопровождается запросом в БД, что потенциально перегружает сервер

Решение (Debouncing):

Необходимо включать таймер после ввода символа пользователем, и отсылать запрос только по его истечении (скажем, если нет никаких пользовательских действий спустя 500 ms после последнего ввода символа)

Для реализации de-bouncing необходимо установить библиотеку use-debounce:

```
pnpm i use-debounce   #это уже сделано в проекте:

cat package.json | grep use-

"use-debounce": "^10.0.1",
```

Поиск. Решаем проблему излишних запросов (2)

1	**Update the search.tsx file**	**Обновите файл app/ui/search.tsx:** **cp search.tsx.04.Debouncing search.tsx**
2	**Examine the search.tsx file**	**Проанализируйте изменения в файле app/ui/search.tsx:**

Формально, Debouncing — это практика программирования, которая ограничивает частоту срабатывания функции (снижает количество ее вызовов). В нашем случае, мы хотим запрашивать БД только в том случае, когда пользователь прекратил печать. Процедура сводится к трем шагам:

1. Событие-триггер: таймер запускается, когда происходит отслеживаемое событие (например, нажатие клавиши в поле поиска)
2. Ожидание: если новое событие произойдет до истечения времени таймера, таймер сбрасывается и перезапускается
3. Выполнение: если время таймера истекло, ограничиваемая функция выполняется (handleSearch, в нашем случае)

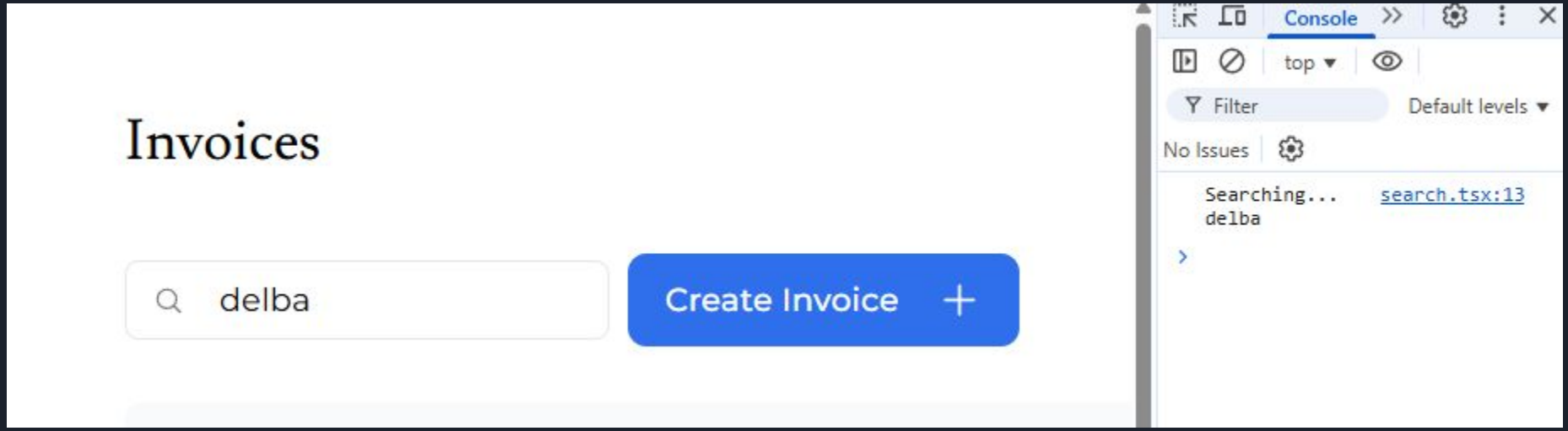
Поиск. Решаем проблему излишних запросов (3)

/app/ui/search.tsx:

```
'use client';
…
import { useDebouncedCallback } from 'use-debounce'; #we will use the debounce library

export default function Search({ placeholder }: { placeholder: string }) {
  const searchParams = useSearchParams();
  const pathname = usePathname();
  const { replace } = useRouter();

  const handleSearch = useDebouncedCallback((term) => {
    console.log(`Searching... ${term}`);

    const params = new URLSearchParams(searchParams);
    if (term) {
      params.set('query', term);
    } else {
      params.delete('query');
    }
    replace(`${pathname}?${params.toString()}`);
  }, 500);  #wrapping up the handleSearch function to delay its start by 500 ms since the last keystorke
```

Поиск. Решаем проблему излишних запросов (4)

3	**Reload localhost:3000/dashboard/invoices, open DevTools and test the debouncing**	**Перезагрузите страницу localhost:3000/dashboard/invoices Откройте DevTools и убедитесь в том, что тестовая строка порождает *единственный* запрос в БД**



Постраничный вывод

1	**Update the page.tsx file**	**Обновите файл app/dashboard/invoices/page.tsx:** **cp page.tsx.02.Pagination page.tsx**
2	**Examine page.tsx**	**Проанализируйте изменения в файле app/dashboard/invoices/page.tsx**

В настоящий момент, поиск ограничен одной страницей и шестью инвойсами. Добавим функционал постраничного вывода (пагинацию) по следующей схеме:

1. Из searchParams.?query page.tsx получаем текущий запрос и (на серверной стороне) запускаем функцию fetchInvoicesPages (query), которая выдает количество страниц totalPages
2. Основной (клиентский) компонент <Pagination /> принимает totalPages как props и определяет, таким образом, интервал страниц, из которых пользователь выбирает нужную (currentPage = page)
3. Указанный параметр (page) дополняет строку URL, и оба параметра (query, page) отправляются из searchParams в серверный компонент <Table />, где и происходит обновление таблицы инвойсов по результатам запроса в БД

NB Для нового или скорректированного пользователем запроса значение page устанавливается в 1 в handleSearch компонента <Search />, т.е. отображается первая страница таблицы инвойсов

Постраничный вывод (2)

/app/dashboard/invoices/page.tsx:

```
…
import { fetchInvoicesPages } from '@/app/lib/data';

export default async function Page(props: {
  searchParams?: Promise<{
    query?: string;
    page?: string;
  }>;
}) {
  const searchParams = await props.searchParams;
  const query = searchParams?.query || '';
  const currentPage = Number(searchParams?.page) || 1;
  const totalPages = await fetchInvoicesPages(query); #calculate number of pages based on query results

  return (
…
      <div className="mt-5 flex w-full justify-center">
       <Pagination totalPages={totalPages} />  #pass total number of pages to Pagination
      </div>
    </div>
  );
}
```

Постраничный вывод (3)

3	**Update the pagination.tsx file**	**Обновите файл app/ui/invoices/pagination.tsx:** **cp pagination.tsx.01.Pagination pagination.tsx**
4	**Examine pagination.tsx**	**Проанализируйте изменения в файле app/ui/invoices/pagination.tsx**

Клиентский компонент <Pagination {totalPages}/> отображает доступный интервал страниц для текущего запроса query и дает возможность пользователю переключиться на желаемую страницу:

1. Количество страниц приходит в totalPages
2. Текущую страницу (currentPage), и URL страницы (pathname) получаем из хуков useSearchParams, usePathname
3. При нажатии пользователем кнопки с номером страницы, левой или правой стрелкой, запускается функция-обработчик createPageURL(pageNumber), которая обновляет строку URL, изменяя в ней параметр page=<новый номер страницы>
4. В соответствии с изменениями в строке URL, серверный элемент <Table {query} {page}/> делает запрос в БД и перерисовывает таблицу инвойсов

Постраничный вывод (4)

/app/ui/invoices/pagination.tsx:

```
…
import { usePathname, useSearchParams } from 'next/navigation';

export default function Pagination({ totalPages }: { totalPages: number }) {  #receive total pages
  const pathname = usePathname();    #get current URL path
  const searchParams = useSearchParams();
  const currentPage = Number(searchParams.get('page')) || 1; #get current page number

  const createPageURL = (pageNumber: number | string) => {  #set  page number in the URL request
    const params = new URLSearchParams(searchParams);  #Get current search parameters
    params.set('page', pageNumber.toString()); #Update the 'page' to the provided page number
    return `${pathname}?${params.toString()}`; #Construct URL with pathname and updated page number
  };
…
return (
…
<PaginationArrow
          direction="left"
          href={createPageURL(currentPage - 1)}    #The similar call is for right arrow and page No buttons
          isDisabled={currentPage <= 1}
        />
…  ) ; }
```

Постраничный вывод (5)

5	**Update the search.tsx file**	**Обновите файл app/ui/search.tsx:** **cp search.tsx.05.Pagination search.tsx**
6	**Examine search.tsx**	**Проанализируйте изменения в файле app/ui/search.tsx**

При изменении пользователем запроса в поле <input> компонента <Search /> должен также сбрасываться и номер страницы в единицу, иначе возможна ситуация, когда произойдет попытка вывести несуществующую для данного условия query страницу инвойсов

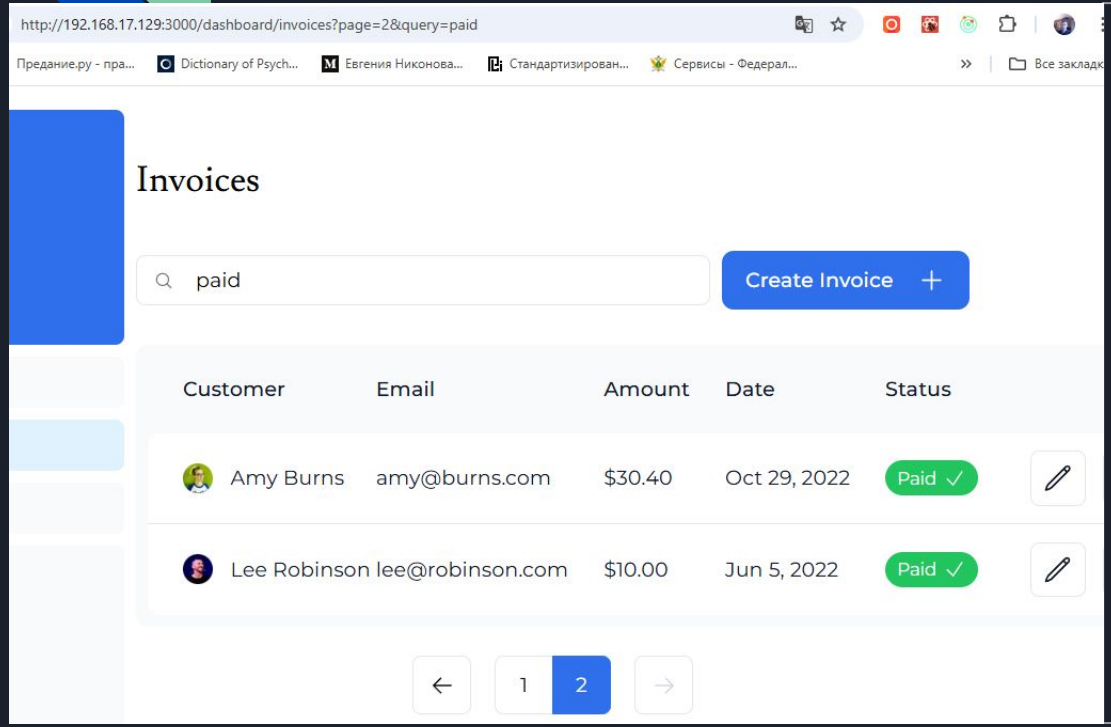
Авто-возврат на первую страницу реализован в функции handleSearch()

Постраничный вывод (6)

/app/ui/search.tsx:

```
...
const handleSearch = useDebouncedCallback((term) => {
  console.log(`Searching... ${term}`);

  const params = new URLSearchParams(searchParams);
  params.set('page', '1');  # we will first display page=1 of invoices for a new or changed search query=term
  if (term) {
    params.set('query', term);
  } else {
    params.delete('query');
  }
  replace(`${pathname}?${params.toString()}`);
}, 500);
```

Постраничный вывод (7)



7	**Reload localhost:3000/dashboard/invoices, open DevTools and test the pagination**	**Перезагрузите страницу localhost:3000/dashboard/invoices Откройте DevTools и протестируйте постраничный вывод**

Схема взаимодействия компонентов страницы инвойсов

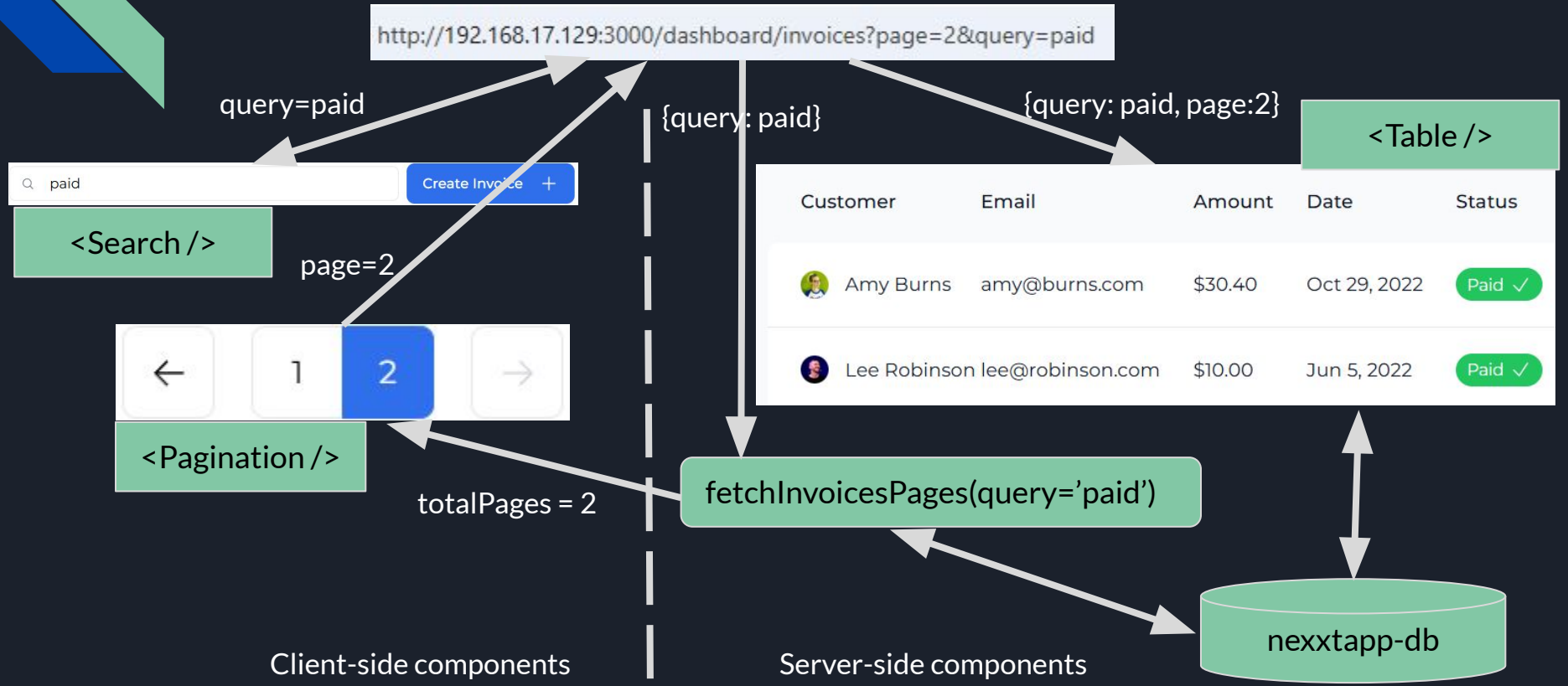


Схема взаимодействия компонентов страницы инвойсов (2)

Как видим, page.tsx в Next.js представляет собой весьма мозаичную картинку, состоящую из клиентских (client-side, ‘use client’) и серверных (server-side, по умолчанию), первые из которых размещаются в браузере и отвечают за взаимодействие с пользователем, а вторые - на сервере, в большинстве своем ориентированы на взаимодействие с БД

В этом - коренное отличие от React, где все решение - “клиентское”, фронтендное, а бэкенд поддерживается серверными средствами Node.js. Взаимодействие, по большей части, идет по REST API

В Next.js, как мы увидели, для передачи параметров между клиентскими и серверными компонентами можно использовать даже строку URL браузера, причем запрос GET при этом впрямую не выполняется (Вы, конечно, можете выполнить этот запрос и непосредственно из строки браузера или переслать его своему коллеге, но в нашем случае этого не требуется). При этом лишь серверные компоненты взаимодействуют с БД, что повышает защищенность приложения Next.js

Также обратим внимание на то, что пересылка параметров или результатов функций между клиентской и серверной частью совсем не ограничивается использованием URL. Так, например, клиентский компонент <Pagination {totalPages} / > принимает в props результат выполнения серверной функции fetchInvoicesPages(query) напрямую из соответствующей константы

Все сказанное относится, в основном, к извлечению (Read, fetching, retrieving) данных из БД. При необходимости преобразовать (Create, Update, Delete) данные, Next.js предоставляет отдельный набор Server Actions, о котором пойдет речь в следующей лекции

Решаем проблему статической компиляции Dashboard

```
> 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                           294 B          92.4 kB
├ ○ /dashboard/customers                 138 B          87.2 kB
├ ƒ /dashboard/invoices                  2.57 kB         101 kB
└ ○ /seed                                0 B               0 B
+ First Load JS shared by all            87.1 kB
  ├ chunks/842-5f5918a06d3967bc.js       31.5 kB
  ├ chunks/94c12b52-b8f2fec12ddd9df7.js  53.6 kB
  └ other shared chunks (total)          1.9 kB


○  (Static)   prerendered as static content
ƒ  (Dynamic)  server-rendered on demand
```

Проблема:

При сборке в production (pnpm run build) маршрут /dashboard считается компилятором статическим, все поля, в том числе, данные из БД намертво встраиваются в страницу page.tsx при сборке и не отражают последующих изменений в БД, что легко выявляется при запуске (pnpm run start)

Решение (Dynamic rendering):

Необходимо сделать маршрут /dashboard динамическим, чтобы страница перерисовывалась при каждом своем обновлении

Решаем проблему статической компиляции Dashboard (2)

1	**Update the page.tsx file**	**Обновите файл app/dashboard/’(overview)’/page.tsx:** **cp page.tsx.01.ForceDynamic page.tsx**
2	**Examine page.tsx**	**Проанализируйте изменения в файле app/dashboard/’(overview)’/page.tsx**

Для того, чтобы сделать данную страницу page.tsx “динамической”, необходимо добавить следующую константу:

```
export const dynamic = 'force-dynamic';
```

В результате, страница будет обновляться при каждой своей (пере)загрузке (dynamic rendereing)

NB. Константа применима на уровне layout.tsx и page.tsx. При этом следует иметь в виду, что все страницы (ветки) данного маршрута вниз по иерархии также станут трактоваться как динамические. В частности, поэтому мы применяем константу именно к странице, а не к layout.tsx уровнем выше, который распространяется и на /customers, и на /invoices

Решаем проблему статической компиляции Dashboard (3)

/app/dashboard/’(overview)’/page.tsx

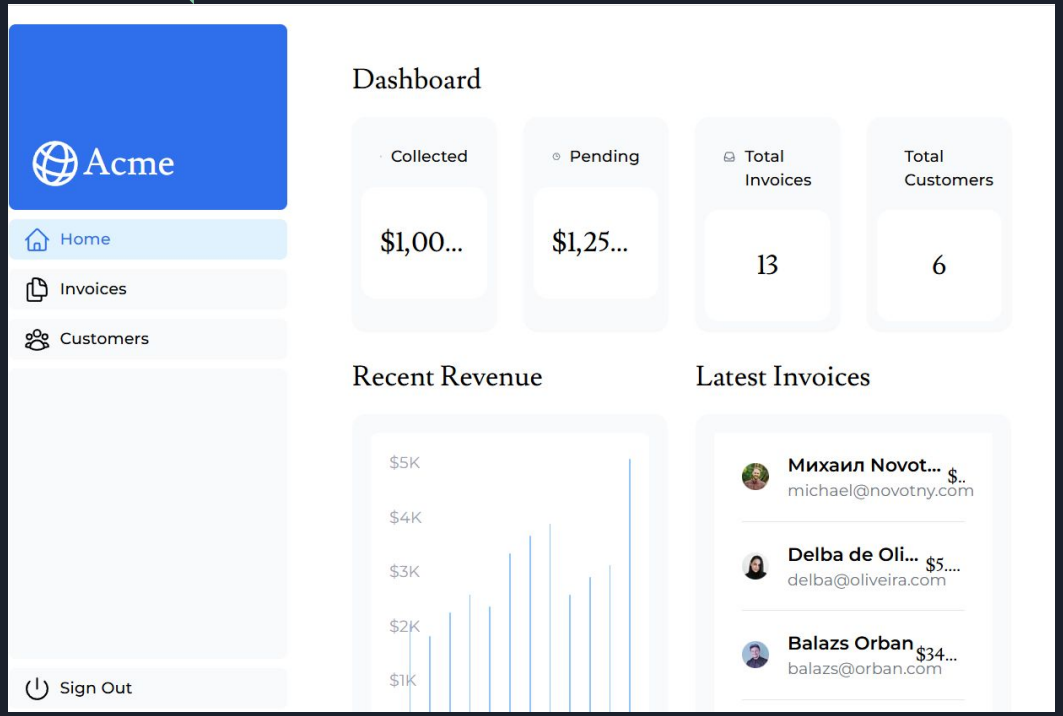
```
import CardWrapper from '@/app/ui/dashboard/cards';        //DM <CardWrapper> is used instead of <Card>s
import RevenueChart from '@/app/ui/dashboard/revenue-chart';
import LatestInvoices from '@/app/ui/dashboard/latest-invoices';
import { lusitana } from '@/app/ui/fonts';
import { Suspense } from 'react';
import {
  RevenueChartSkeleton,
  LatestInvoicesSkeleton,
  CardsSkeleton,
} from '@/app/ui/skeletons';   //DM <CardsSkeleton> added

export const dynamic = 'force-dynamic';  # force the compiler to treat this route as dynamically rendered

export default async function Page() {

  return ( …);
}
```

Решаем проблему статической компиляции Dashboard (4)

```
> 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
├ ƒ /dashboard                           294 B          92.4 kB
├ ○ /dashboard/customers                 138 B          87.2 kB
├ ƒ /dashboard/invoices                  2.57 kB         101 kB
└ ○ /seed                                0 B               0 B
+ First Load JS shared by all            87.1 kB
  ├ chunks/842-5f5918a06d3967bc.js       31.5 kB
  ├ chunks/94c12b52-b8f2fec12ddd9df7.js  53.6 kB
  └ other shared chunks (total)          1.9 kB


○  (Static)   prerendered as static content
ƒ  (Dynamic)  server-rendered on demand
```

3	**Rebuild the production**	**Пересоберите optimized production build:** **pnpm run build** **и убедитесь, что маршрут /dashboard стал динамическим**
4	**Launch the production**	**Запустите optimized production build** **pnpm run start**

Решаем проблему статической компиляции Dashboard (5)



5	**Reload localhost:3000/dashboard/, and test dynamic rendering**	**Перезагрузите страницу localhost:3000/dashboard/ И протестируйте динамическую отрисовку: измените имя клиента в таблице БД customers и отследите это изменение на странице /dashboard после ее перезагрузки**

Статический, динамический рендеринг и PPR

В настоящий момент, в Next.js реализованы два вида маршрутов: статический (static rendering) и динамический (dynamic rendering).

По умолчанию, маршрут считается статическим, если нет некоторых дополнительных условий: например, если страница использует searchParams (как в нашем случае с /invoices), маршрут автоматически становится динамическим

Использование константы force-dynamic (и некоторые другие методы) позволяют сделать желаемый маршрут (наш случай - /dashboard) и его ветви динамическими

При этом минимальная степень “гранулярности” этой динамики - одна страница page.tsx: она целиком может быть либо статической, либо динамической

Next.js разрабатывает и внедряет функциональность PPR (partial pre-rendering), которая позволит сочетать динамические и статические компоненты в пределах одной страницы page.tsx. При этом граница между динамическим и статическим контентом будет устанавливаться по элементу <Suspense / >: внутри него - динамика, вне - статика. Функциональность пока обкатывается в релизах next@canary, поэтому лекция 10 о PPR (https://nextjs.org/learn/dashboard-app/partial-prerendering) пока не включена в курс.

Подробней о пре-рендеринге:
https://nextjs.org/docs/app/getting-started/partial-prerendering

Подведем итоги

1	**Научились использовать API Next.js для редактирования строки URL в клиентских (client-side) компонентах: хуки useSearchParams, usePathname и useRouter**
2	**Реализовали поиск и постраничный вывод (pagination) с пересылкой параметров поиска в (запросе GET) URL в строке браузера**
3	**Научились извлекать параметры URL из SearchParams и обновлять (server-side) таблицу инвойсов по результатам соответствующего запроса в БД**
4	**Решили проблему отсутствия обновления данных в статических страницах: применили динамический рендеринг к маршруту /dashboard**

В следующей лекции научимся ***изменять*** данные в БД с использованием Server Actions от Next.js

Следующая лекция - 12. Mutating Data / Изменение данных в БД

Презентация доступна для скачивания здесь:

https://dmpsy.club/references/NextJS/lesson_011_search_and_pagination_rus.pdf

Загрузить приложение с GitHub:
https://github.com/DmitryMakar/nextjs-lesson-011.git

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

Приложение А. Запуск рабочей среды проекта

Prerequisite	**Docker installed on your host**
To verify	**docker –v**
Example output	**Docker version 28.1.1, build 4eba377**
Postgres User	**nexxtapp-db**
Postgres Password	**NextJS**
Postgres DB*	**nexxtapp-db**

1	**Launch the project testbed**	**Клонируйте проект и перейдите в папку scripts:** **git clone https://github.com/DmitryMakar/nextjs-lesson-011.git** **cd nextjs-lesson-011/scripts** **mkdir pgadmin && sudo chown -R 5050:80 pgadmin # set permissions** **docker compose up -d # start the environment** **docker compose logs # inspect logs** **docker container ls -a # check that pgAdmin and DB containers started ok**

* Либо используйте свою собственную локальную БД с настройками, указанными выше, в этом случае docker разворачивать не надо, соответствующие пункты пропустите

Приложение А. Запуск рабочей среды проекта (2)

2	Configure pgAdmin 4	Откройте http://localhost:5050 # или по IP: http://<Your_(VM)_IP_address>:5050 Установите master password: <Your_password> Нажмите Add a new server и укажите: Name: test Host name/address: postgres_container Port: 5432 Maintenance DB: nexxtapp-db Username: nexxtapp-db Password: NextJS
3	Install node modules	Установите необходимые пакеты (node modules): cd nextjs-lesson-011 pnpm install Если потребуется, подтвердите установку доп. пакетов: pnpm approve-builds

Приложение А. Запуск рабочей среды проекта (3)

4	**Copy the PostgreSQL credentials to the .env file**	**Скопируйте учетные данные PostgreSQL в файл .env:**
		`cp .env.example .env`
		`cat .env  # verify it:`
		`POSTGRES_HOST=localhost`
		`POSTGRES_USER=nexxtapp-db`
		`POSTGRES_PASSWORD=NextJS`
		`POSTGRES_DATABASE=nexxtapp-db`
		`POSTGRES_PORT=5437`
		NB Соединяемся с БД “извне” контейнеров, поэтому localhost:5437, как это указано в docker-compose.yaml (5437:5432). Это сделано во избежание конфликта с локальной БД Postgres, если она уже установлена на хосте и слушает стандартный порт 5432
		Если же предполагаете использовать свою локальную БД, укажите здесь ее порт 5432 и остановите контейнеры PgAdmin и Postgres, т.к. тогда docker не нужен
		`cd scripts`
		`docker compose down`

Приложение А. Запуск рабочей среды проекта (4)

5	**Seed the PostgreSQL with initial data**	**Заполните схему БД исходными данными:** pnpm seed > @ seed /home/dmitry-makarenkov/NextJSTraining/nextjs-lesson-009 > node -r dotenv/config ./scripts/seed.mjs Created "users" table Seeded 1 users Created "customers" table Seeded 6 customers Created "invoices" table Seeded 13 invoices Created "revenue" table Seeded 12 revenue
6	**Check the data seeded via pgAdmin**	**Проверьте, что данные поступили в БД,** **Откройте http://localhost:5050/browser/ # или по IP:** **http://<Your_(VM)_IP_address>:5050/browser/** Tools | Query tool select * from book; select * from users; select * from customers; select * from invoices; select * from revenue;

Приложение А. Запуск рабочей среды проекта (5)

7	**Start the dev server**	**Запустите сервер разработки:** **pnpm run dev** **> next dev** **▲ Next.js 14.2.5** **- Local: http://localhost:3000** **- Environments: .env** **✓ Starting...** **✓ Ready in 14.8s**
8	**Verify it works**	**Проверьте, что приложение открывается;** **Откройте http://localhost:3000/dashboard # или по IP:** **http://<Your_(VM)_IP_address>:3000/dashboard/** **В логах отобразится, примерно, следующее:** **○ Compiling /dashboard ...** **✓ Compiled /dashboard in 1970ms (618 modules)** **GET /dashboard 200 in 6420ms**

Вернуться к Slide 8: Заготовка страницы Invoices

Приложение B. Останов и очистка рабочей среды проекта

1	**Clean the project when done**	**Закройте web-страницы приложения и pgAdmin’a** **Остановите сервер разработки (Ctrl-C) и очистите среду:** **cd scripts** **docker compose down** **sudo rm -rf pgadmin pgdata** **docker image rm dpage/pgadmin4:latest postgres:15-alpine**

Вернуться к Slide 5: План работы / Agenda (3)

Приложение C. Задерживаем запуск запроса в ***fetchRevenue()***

0	**Prepare fetchRevenue() for app/lib/data.ts**	**fetchRevenue() отредактирована в файле app/lib/data.ts.slowedFetchRevenue:**
		…
		await client.connect();
		
		//Delay a response for demo purposes
		console.log("Fetching revenue data...");
		await new Promise((resolve) => setTimeout(resolve, 10000));
		
		const result = await client.query('select * from revenue');
		console.log(result.rows);
		console.log('Data fetch completed after 10 seconds.');
		await client.end();
		
		return result.rows;
		…

Вернуться к Slide 19: Потоковый вывод компонента <RevenueChart> (5)

Slide 24: Потоковый вывод компонента <LatestInvoices> (5)

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

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