Next.js – Server e Client Components

Server Components

Os componentes no Server permitem que você escreva UI que pode ser renderizada e opcionalmente armazenada em cache no servidor. No Next.js, o trabalho de renderização é dividido por segmentos de rota para permitir streaming e renderização parcial, e há três estratégias diferentes de renderização de servidor:

- Renderização Estática
- Renderização Dinâmica
- Streaming

Os components do servidor são o padrão da nossa aplicação, então não precisamos escrever nenhuma linha extra para usar.

Importante lembrar que os Server components não tem suporte aos hooks nativos do React, como o useState.

Server Components

FIAP

“use server”

Já usamos essa diretiva na aula de rotas anteriormente, a diretiva “use server” é uma parte crucial das Ações do Servidor. Ele garante que a função em que é usado seja executada apenas no servidor. Isso é importante por motivos de segurança e desempenho, pois você não gostaria que a lógica confidencial do lado do servidor fosse exposta ou executada no lado do cliente.

Além disso ela é usada para códigos que só podem ser executados do lado do servidor, como o uso do redirect que vimos na aula passada. Para usar devemos só incluir uma string “use server” no começo do nosso arquivo.

```
You, 20 hours ago | 1 author (You)
1  'use server'
2  import { permanentRedirect } from 'next/navigation'
3
4  const RedirectServer = ({ path }: { path: string }) => {
5    permanentRedirect(path)
6  }
7
8  export default RedirectServer
9
```

Server Components

Por padrão podemos escrever components asyncronas no Next.js para buscar dados no servidor e passar esses dados para o componente da página antes de renderizá-lo. Isso significa que a página será renderizada com os dados já disponíveis, resultando em uma melhor experiência de usuário e SEO.

Vamos usar o metodo fetch do Javascript para fazer uma requisição de teste do lado do servidor. Não se preocupe, vamos ver isso com mais profundidade posteriormente.
Para ver mais sobre o fetch acesse a documentação: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Server Components

1. Vamos usar a API do site DummyJson para listarmos produtos de exemplo na nossa aplicação. A rota que vamos usar vai ser a https://dummyjson.com/products?limit=10&skip=10&select=title,price, essa rota vai devolver uma lista de 10 produtos com id, nome e preço em formato de JSON.
2. Vamos abrir a URL da request no browser a avaliar o response. Olhando pelo response já podemos criar uma interface para os nossos produtos.
3. Vamos criar uma pasta chamada interfaces e dentro um arquivo product.ts
4. Também vamos criar uma interface type genérica para a response da nossa api em um arquivo response.ts

```ts
export interface Product {
  id: number
  title: string
  price: number
}
```

```ts
export type GenericListResponse<T extends string, U> = {
  total: number
  skip: number
  limit: number
} & {
  [key in T]: U
}
```


```
{
  "products": [
    {
      "id": 11,
      "title": "Annibale Colombo Bed",
      "price": 1899.99
    },
    {
      "id": 12,
      "title": "Annibale Colombo Sofa",
      "price": 2499.99
    },
    {
      "id": 13,
      "title": "Bedside Table African Cherry",
      "price": 299.99
    },
    {
      "id": 14,
      "title": "Knoll Saarinen Executive Conference Chair",
      "price": 499.99
    },
    {
      "id": 15,
      "title": "Wooden Bathroom Sink With Mirror",
      "price": 799.99
    },
    {
      "id": 16,
      "title": "Apple",
      "price": 1.99
    },
    {
      "id": 17,
      "title": "Beef Steak",
      "price": 12.99
    },
    {
      "id": 18,
      "title": "Cat Food",
      "price": 8.99
    },
    {
      "id": 19,
      "title": "Chicken Meat",
      "price": 9.99
    },
    {
      "id": 20,
      "title": "Cooking Oil",
      "price": 4.99
    }
  ],
  "total": 194,
  "skip": 10,
  "limit": 10
}
```

Server Components

5. No arquivo products/page.tsx vamos montar uma função que irá fazer a requisição. Vamos usar os tipos que criamos anteriormente.

6. Vamos atribuir o resultado da função a uma variavel data

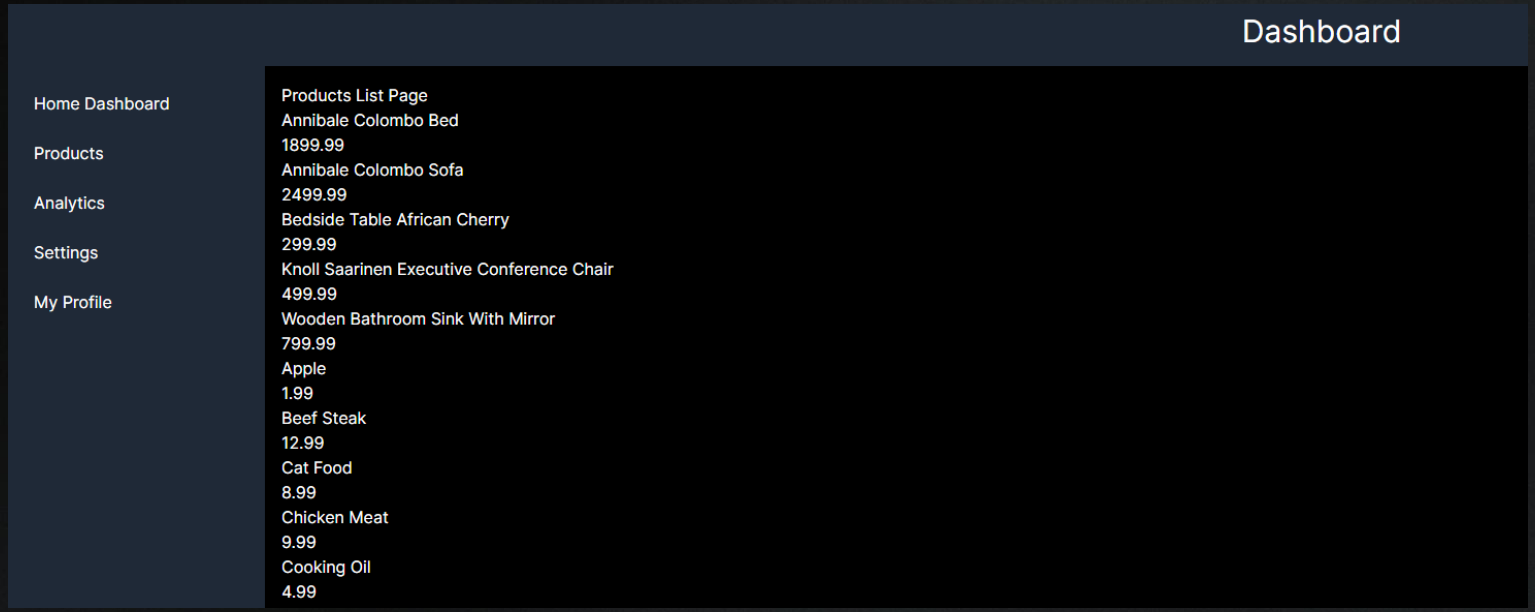
7. Vamos usar o metodo map para listar os produtos na página, assim como vimos nas aulas de React quando trabalhamos com listas

```
You, 1 second ago | 1 author (You)
import { Product } from "@/interfaces/product"
import { GenericListResponse } from "@/interfaces/response"

async function getData() {
  try {
    const request = await fetch('https://dummyjson.com/products?limit=10&skip=10&select=title,price')
    const response: GenericListResponse<'products', Array<Product>> = await request.json()
    return response.products
  } catch (error) {
    console.error(`Error fetching products: ${error}`)
    return []
  }
}

const ProductsListPage = async () => {
  const data = await getData()

  return (
    <div>
      <h1>Products List Page</h1>
      {
        data?.map((product) => (
          <div key={product.id}>
            <h2>{product.title}</h2>
            <p>{product.price}</p>
          </div>
        ))
      }
      {
        data.length === 0 && <p>No products found</p>
      }
    </div>
  )
}

export default ProductsListPage
```

Server Components

FIAP

Resultado



Server Components

6. Vamos criar um componente Product dentro da pasta de components apenas para organizar melhor o nosso código
7. Vamos usar o nosso componente na nossa listagem

```tsx
import { Product } from '../../interfaces/product'

interface ProductComponentProps {
  product: Product
}

const ProductComponent = ({ product }: ProductComponentProps) => {
  return (
    <div key={product.id}>
      <h2>{product.title}</h2>
      <p>{product.price}</p>
    </div>
  )
}

export default ProductComponent
```

```tsx
const ProductsListPage = async () => {
  const data = await getData()

  return (
    <div>
      <h1>Products List Page</h1>
      {
        data?.map((product) => (
          <ProductComponent key={product.id} product={product} />
        ))
      }
      {
        data.length === 0 && <p>No products found</p>
      }
    </div>
  )
}

export default ProductsListPage
```

Server Components

8. Vamos usar o Tailwind para estilizar

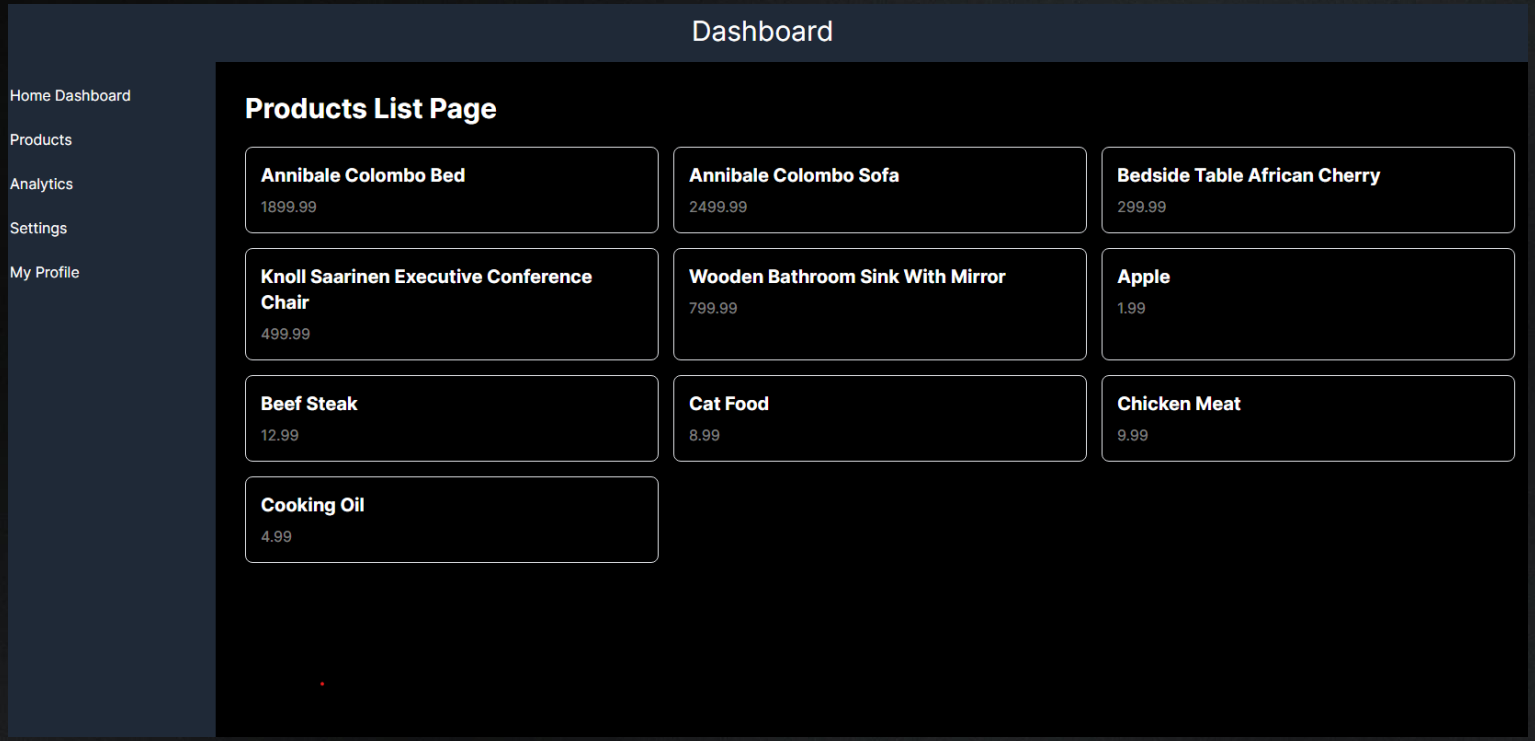
```tsx
const ProductComponent = ({ product }: ProductComponentProps) => {
  return (
    <div key={product.id} className="p-4 border rounded-lg shadow-md">
      <h2 className="text-xl font-bold mb-2">{product.title}</h2>
      <p className="text-700 opacity-50">{product.price}</p>
    </div>
  )
}
```

```tsx
const ProductsListPage = async () => {
  const data = await getData()

  return (
    <div className="container mx-auto p-4">
      <h1 className="text-3xl font-bold mb-6">Products List Page</h1>
      <div className="content grid grid-cols-1 gap-4 sm:grid-cols-2 md:grid-cols-3">
        {data?.map((product) => (
          <ProductComponent key={product.id} product={product} />
        ))}
        {data.length === 0 && <p className="text-gray-600">No products found</p>}
      </div>
    </div>
  )
}
```

Server Components

FIAP

Resultado



Client Components

Client Components são componentes React que são renderizados no cliente. Eles permitem interatividade e atualizações dinâmicas na interface do usuário.

Geralmente usamos um Client Component quando temos alguma interação do usuário ou é uma ação que pode ser carregada posteriormente ao carregamento da página, deixando a página mais leve caso a informação não seja tão relevante.

Aqui é onde podemos usar livremente os hooks do React.

Client Components

“use client”

O use client indica para o Next que é um componente que só vai ser renderizado no Client para uso completo do React.

Usamos anteriormente para testarmos o useRouter do Next na aula sobre roteamento, já que ele não pode ser usado do lado do servidor

```tsx
'use client'

import { useRouter } from 'next/navigation'

export default function Button() {
  const router = useRouter()

  return (
    <button type="button" onClick={() => router.push('/dashboard')}>
      To Dashboard
    </button>
  )
}
```

Client Components

1. Vamos criar um componente para pegar a próxima página de produtos. Dentro da pasta components vamos criar um ProductsListNextPage. Como parâmetro vamos passar a quantidade de produtos por pagina, como uma prop perPage, e lembrar de usar o ‘use client’
2. Vamos por um botão para o usuário carregar mais itens. Também já vamos deixar tudo estilizado com o Tailwind.
3. Vamos chamar nosso botão na página de produtos

```tsx
<div className="container mx-auto p-4">
  <h1 className="text-3xl font-bold mb-6">Products List Page</h1>
  <div className="content grid grid-cols-1 gap-4 sm:grid-cols-2 md:grid-cols-3">
    {data?.map((product) => (
      <ProductComponent key={product.id} product={product} />
    ))}
    {data.length === 0 && <p className="text-gray-600">No products found</p>}
    {data.length > 0 && (
      <ProductsListNextPage />
    )}
  </div>
</div>
```

```tsx
...
'use client'

...
interface ProductsListProps {
  perPage?: number
}

const ProductsListNextPageComponent = ({ perPage = 12 }: ProductsListProps) => {

  return (
    <>
      <footer className='container mx-auto p-4 mt-8 text-center col-span-full'>
        {
          (
            <button className={`bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded`}>
              Load More
            </button>
          )
        }
      </footer>
    </>
  )
}

export default ProductsListNextPageComponent
```

Client Components

4. Vamos criar algumas variaveis de estado: hasMore (para saber quando a paginação acabou), page (contagem de paginas), loading (verificar o carregamento da requisição) e data (nossos produtos)

```
const ProductsListNextPageComponent = ({ perPage = 12 }: ProductsListProps) => {
  const [data, setData] = useState<Product[]>([])
  const [hasMore, setHasMore] = useState(true)
  const [loading, setLoading] = useState(false)
  const [page, setPage] = useState(1)
```

5. Vamos exibir o botão apensar quando hasMore for true

```
<footer className='container mx-auto p-4 mt-8 text-center col-span-full'>
  {
    hasMore && (
      <button className={`bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded`}>
        Load More
      </button>
    )
  }
```

Client Components

6. Vamos exibir uma mensagem para quando não existir mais produtos a serem carregados

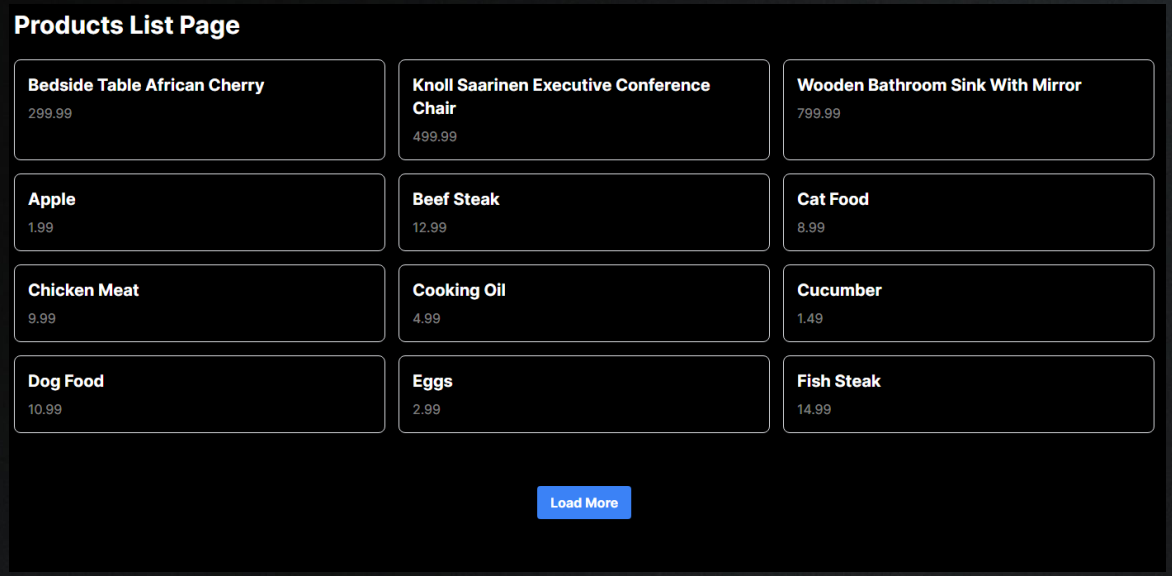
```
<footer className='container mx-auto p-4 mt-8 text-center col-span-full'>
  {
    hasMore && (
      <button className={`bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded`}>
        Load More
      </button>
    )
  }
  {
    !hasMore && <p className='text-gray-600'>No more products to load</p>
  }
</footer>
```

7. Vamos colocar uma mensagem de loading no nosso botão enquanto o client faz a request, junto com classes do tailwind para mudar um pouco a aparência do botão

```
<footer className='container mx-auto p-4 mt-8 text-center col-span-full'>
  {
    hasMore && (
      <button className={`bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded ${loading ? 'opacity-50 cursor-not-allowed' : ''}`}>
        {loading ? 'Loading...' : 'Load More'}
      </button>
    )
  }
```

Client Components

Até agora criamos apenas a base do nosso Client Component. Mas se formos até a nossa página o nosso botão já deve estar lá, mas ao clicar ele ainda não faz nada



Client Components

7. Vamos criar a função fará a chamada das próximas páginas

```
const ProductsListNextPageComponent = ({ perPage = 12 }: ProductsListProps) => {
  const [data, setData] = useState<Product[]>([])
  const [page, setPage] = useState<number>(1)
  const [hasMore, setHasMore] = useState<boolean>(true)
  const [loading, setLoading] = useState<boolean>(false)

  const getData = async () => {
    if (!hasMore || loading) {
      return
    }
    try {
      setLoading(true)
      setPage(page + 1)
      const request = await fetch(`https://dummyjson.com/products?limit=12&skip=${page * perPage}&select=title,price`)
      const response: GenericListResponse<'products', Array<Product>> = await request.json()
      setData([...data, ...response.products])
      if (response.products.length === 0) {
        setHasMore(false)
      }
    } catch (error) {
      setHasMore(false)
      console.error(`Error fetching products: ${error}`)
    } finally {
      setLoading(false)
    }
}
      You, 10 minutes ago • refactor: Add lazy loading for product images o…
  return (
```

Client Components

1. Verificamos se hasMore é false (ou seja, não tem mais produtos) ou se loading é verdadeiro (ou seja, a nossa requisição anterior ainda está em curso). Caso as condições sejam verdadeiras usar um return vazio. Assim a função não vai fazer absolutamente nada

2. Atualizamos o loading para true, indicando o carregamento e depois mudamos a pagina para page + 1 (ou seja, de 1 para 2, de 2 para 3...)

```tsx
const ProductsListNextPageComponent = ({ perPage = 12 }: ProductsListProps) => {
  const [data, setData] = useState<Product[]>([])
  const [page, setPage] = useState<number>(1)
  const [hasMore, setHasMore] = useState<boolean>(true)
  const [loading, setLoading] = useState<boolean>(false)

  const getData = async () => {
    if (!hasMore || loading) {
      return
    }
    try {
      setLoading(true)
      setPage(page + 1)
      const request = await fetch(`https://dummyjson.com/products?limit=12&skip=${page * perPage}&select=title,price`)
      const response: GenericListResponse<'products', Array<Product>> = await request.json()
      setData([...data, ...response.products])
      if (response.products.length === 0) {
        setHasMore(false)
      }
    } catch (error) {
      setHasMore(false)
      console.error(`Error fetching products: ${error}`)
    } finally {
      setLoading(false)
    }
}
      You, 10 minutes ago • refactor: Add lazy loading for product images o…
  return (
```

Client Components

3. Fazemos a request de forma bem semelhante ao que fizemos anteriormente, só que no parâmetro skip, vamos multiplicar o contador da página pelo perPage. Isso fará com que façamos paginação de forma correta (page 1 <-> skip 12, page 2 <-> skip 24, page 3 <-> skip 36 ...)

4. Atualizamos o estado de Data misturando o valor anterior com o novo

5. Se não voltar nenhum produto no array podemos entender que chegamos ao fim da paginação e mudar o hasMore para false

```tsx
const ProductsListNextPageComponent = ({ perPage = 12 }: ProductsListProps) => {
  const [data, setData] = useState<Product[]>([])
  const [page, setPage] = useState<number>(1)
  const [hasMore, setHasMore] = useState<boolean>(true)
  const [loading, setLoading] = useState<boolean>(false)

  const getData = async () => {
    if (!hasMore || loading) {
      return
    }
    try {
      setLoading(true)
      setPage(page + 1)
      const request = await fetch(`https://dummyjson.com/products?limit=12&skip=${page * perPage}&select=title,price`)
      const response: GenericListResponse<'products', Array<Product>> = await request.json()
      setData([...data, ...response.products])
      if (response.products.length === 0) {
        setHasMore(false)
      }
    } catch (error) {
      setHasMore(false)
      console.error(`Error fetching products: ${error}`)
    } finally {
      setLoading(false)
    }
  }

      You, 10 minutes ago • refactor: Add lazy loading for product images o…
  return (
```

Client Components

6. Caso algum erro aconteça durante a requisição, colocamos o hasMore para falso e logamos o erro

7. No final do processo voltamos o loading para false, habilitando o botão novamente

```tsx
const ProductsListNextPageComponent = ({ perPage = 12 }: ProductsListProps) => {
  const [data, setData] = useState<Product[]>([])
  const [page, setPage] = useState<number>(1)
  const [hasMore, setHasMore] = useState<boolean>(true)
  const [loading, setLoading] = useState<boolean>(false)

  const getData = async () => {
    if (!hasMore || loading) {
      return
    }
    try {
      setLoading(true)
      setPage(page + 1)
      const request = await fetch(`https://dummyjson.com/products?limit=12&skip=${page * perPage}&select=title,price`)
      const response: GenericListResponse<'products', Array<Product>> = await request.json()
      setData([...data, ...response.products])
      if (response.products.length === 0) {
        setHasMore(false)
      }
    } catch (error) {
      setHasMore(false)
      console.error(`Error fetching products: ${error}`)
    } finally {
      setLoading(false)
    }
  }

      You, 10 minutes ago • refactor: Add lazy loading for product images o…
  return (
```

1 2 3 4 5 6 7

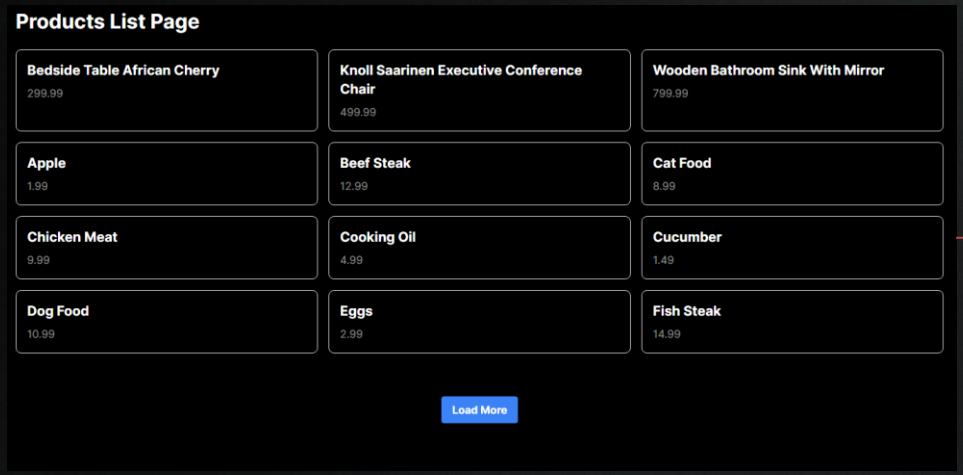
Client Components

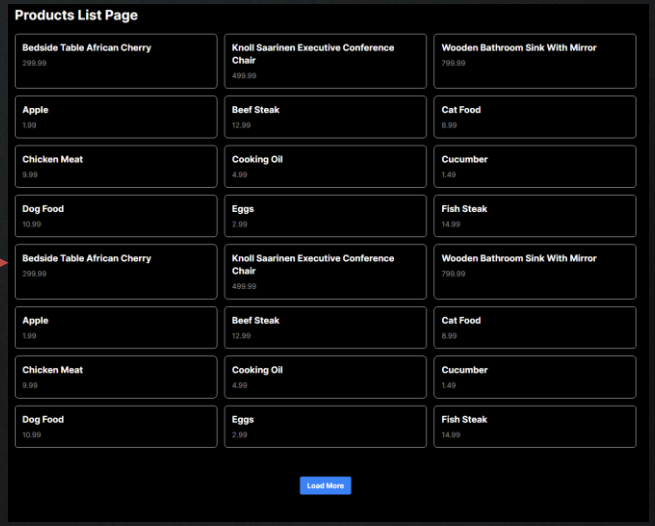
8. Agora basta chamarmos a função no onClick do botão

```
<footer className='container mx-auto p-4 mt-8 text-center col-sp
  {
    hasMore && (
      <button onClick={getData} className={`bg-blue-500 hover:bg
        {loading ? 'Loading...' : 'Load More'}
      </button>
    )
  }
  {
```

Client Components

Resultado





Server x Client Components

FIAP

https://nextjs.org/docs/app/building-your-application/rendering/composition-patterns

What do you need to do?	Server Component	Client Component
Fetch data	✓	×
Access backend resources (directly)	✓	×
Keep sensitive information on the server (access tokens, API keys, etc)	✓	×
Keep large dependencies on the server / Reduce client-side JavaScript	✓	×
Add interactivity and event listeners (`onClick()`, `onChange()`, etc)	×	✓
Use State and Lifecycle Effects (`useState()`, `useReducer()`, `useEffect()`, etc)	×	✓
Use browser-only APIs	×	✓
Use custom hooks that depend on state, effects, or browser-only APIs	×	✓
Use React Class components	×	✓

Exercício

Usando o que vimos hoje crie a página de um produto único no nosso Dashboard

1. Coloque um Link em volta do componente de Produto que leve para a nossa página criada anteriormente (dashboard/products/:id, exemplo: dashboard/products/1)
2. Carregue as informações do Produto através da URL https://dummyjson.com/products/:id (exemplo: https://dummyjson.com/products/1) (Server Side)
3. Crie o layout da página mostrando pelo menos titulo, descrição, preço e a imagem
4. Crie um Client Side Component para exibir uma citação de acordo com o id do produto através da URL https://dummyjson.com/quotes/:id (Exemplo: https://dummyjson.com/quotes/1)
5. Caso não exista uma citação com o ID do produto, não exibir nada
6. Estilize o componente

Dúvidas, críticas ou sugestões?


FIAP