Live: Criando um carrossel parallax do Aranhaverso com React, Next.js 13 e Framer Motion

Veja como criar uma interface interativa com a temática do aranhaverso do Spider-Man. Vamos utilizar as principais stacks front-end: React, Next.js 13, a biblioteca Framer Motion, Sass e TypeScript para criar um projeto visual incrível e de alta performance.

Além disso, você vai aprender sobre desenvolvimento de animações de alta performance em projetos front-end, como ter maior controle sobre os componentes dos projetos front-end com Next e boas práticas de desenvolvimento front-end.

Esses são alguns dos projetos que você poderá desenvolver com conhecimento obtido a partir desse evento👇

✅ Páginas web animadas e com alta performance;
✅ Aplicações e-commerce como um menu de seleção de produtos;

✅ Aplicações de streaming como a Twitch.

Materiais

- Link da live no Youtube
- Assets (arquivos de imagens e efeitos sonoros)
- Protótipo no Figma
- Github do projeto
  - Branches do projeto (leia o README.md no Github)
    - `main` projeto finalizado com todas as features implementadas durante a live;
    - `template` estrutura inicial para que você possa iniciar o projeto, contendo todos os assets, bibliotecas, estrutura de pastas e configurações de ESLint;
    - `live` parte do projeto já iniciado para o code-review em live, pronto para implementar as interações e animações do usuário com o projeto.

Protótipo e demonstração do projeto

Clique aqui para ir ao protótipo do projeto no Figma.


SPIDER-MAN
PERSONAGENS
Criando um carrossel parallax do Aranhaverso com React, Next.js 13 e Framer Motion
Desenvolvimento de animações de alta performance em projetos front-end, como ter maior controle sobre os componentes dos projetos front-end com Next e boas práticas de desenvolvimento front-end.
SPIDER-MAN
MULHER-ARANHA (UNIVERSO-65)
INFORMAÇÕES
Nome Completo
Data de Nascimento
Terra Natal
Altura
Peso
Gwen Stacy
01/03/2002
Manhattan, Nova York
1.65m
56.70kg
PRIMEIRA APARIÇÃO
SPIDER-MAN
HOMEM-ARANHA (UNIVERSO-616)
INFORMAÇÕES
Nome Completo
Data de Nascimento
Terra Natal
Altura
Peso
Peter B. Parker
26/08/1980
Queens, Nova Iorque
1.78m
75.75kg
PRIMEIRA APARIÇÃO

Tecnologias utilizadas no projeto

- React.js (v18)
- Next.js (v13)
- TypeScript (v5)
- ESLint
- Framer Motion
- SASS

Estrutura de pastas e arquivos

- ▼ 📁 `public`
  - 📁 `icons`
  - 📁 `songs`
  - 📁 `spiders`
  - 📄 `...`
- ▼ 📁 `src`
  - ▼ 📁 `app`
    - ▼ 📁 `api`
      - ▼ 📁 `heroes`
        - 📄 `heroes.json`
        - 📄 `route.ts`
    - ▼ 📁 `hero`
      - ▼ 📁 `[id]`
        - 📄 `page.tsx`
    - 📄 `favicon.ico`
    - 📄 `globals.scss`
    - 📄 `layout.tsx`

- page.module.scss
- page.tsx
- variables.scss

▼ components

- ▼ Carousel
  - carousel.module.scss
  - index.tsx
- ▼ HeroDetails
  - heroDetails.module.scss
  - index.tsx
- ▼ HeroesList
  - heroesList.module.scss
  - index.tsx
- ▼ HeroPicture
  - index.tsx

▼ fonts

- index.tsx
- spider-man.otf

▼ interfaces

- heroes.ts

- .eslintrc.json
- .gitignore
- next-env.d.ts
- next.config.js
- package-lock.json
- package.json

- `README.md`
- `tsconfig.json`
- `yarn.lock`

Ponto de partida: template para começar do zero

- Dentro do repositório do Github do projeto você terá disponível uma branch denominada `template`, que contém a estrutura inicial para que você possa começar o projeto, contendo todos os assets, bibliotecas, estrutura de pastas e configurações de ESLint;
- Basta realizar um fork para sua conta e iniciar o desenvolvimento à partir dela. Se preferir, também pode começar o projeto do total zero, instalando as dependências descritas na sessão “Tecnologias utilizadas no projeto”.

API com a listagem de heróis

- Um dos recursos do Next.js é o Route Handlers (Manipuladores de Rota), que possibilita manipular requisições para uma dada rota de uma API;
- Suporta todos os métodos HTTP;
- Vamos utilizar este recurso para trazer a listagem de heróis para consumirmos dentro das páginas da nossa aplicação.

Listagem de heróis

- Usaremos um arquivo `.json` para ilustrar o funcionamento desta funcionalidade, mas, poderíamos estar realizando uma requisição para uma API externa para trazermos estes dados;
- Baixe o arquivo abaixo para utilizar como uma base de dados da aplicação:

heroes.json

Arquivo .json com a listagem de todos os heróis do spider-verso

Criando uma rota de listagem de heróis

- Dentro da pasta `app`, adicione a pasta `heroes` e o arquivo `heroes.json` dentro da aplicação para realizar a busca destes dados;
- Dentro de `/app/heroes` adicione o arquivo `route.ts` e iremos criar uma função `get` para trazer a lista de heróis do arquivo JSON. Lembrando que estamos ilustrando o consumo de uma API externa, que é a forma mais aplicada para esse recurso do Next.js.

```
import { NextResponse } from "next/server";

import data from "./heroes.json";

export async function GET() {
  return NextResponse.json({ data });
}
```

- Com isso, temos a API criada. Ao acessar http://localhost:3000/api/heroes poderemos ver a listagem de heróis provinda do JSON que criamos.

Layout e cabeçalho do projeto

- Arquivos de `layout.tsx` são arquivos de uma interface que é compartilhada entre as rotas da aplicação;
- O arquivo `layout.tsx` na raiz da pasta `app` é usada para definir as tags `<html>`, `<head>` e `<body>` (tags que não se repetem na aplicação) e outros elementos da interface que serão compartilhadas em **todas as páginas da aplicação**;
    - Ao mudar de página, o conteúdo do layout não será recarregado, sendo persistido pelo Next;

    - É possível criar mais de um arquivo layout, com estruturas específicas para cada página;
        - Os layouts serão concatenados (efeito cascata).
- Neste arquivo iremos aplicar alguns elementos, entre eles:
    - Metadados: título e descrição da aplicação;
    - Importação dos estilos globais;
    - Cabeçalho da aplicação (ícones hambúrguer e de usuário) e o logotipo do Spiderman.

```
import "./globals.scss";
import Image from "next/image";
import Link from "next/link";

export const metadata = {
  title: "Spider-Verse",
  description:
    "Criando um carrossel parallax do Aranhaverso com React, Next.js 13 e Framer Motion",
};

export default function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <html lang="pt-BR">
      <body className={inter.className}>
        <header>
          <Image
            src="/icons/menu.svg"
            alt="Menu options"
            width={36}
            height={25}
            priority
          />
          <Link href="/">
            <Image
              src="/spider-logo.svg"
              alt="Spiderman"
              width={260}
              height={70}
              priority
            />
```

```
          </Link>
          <Image
            src="/icons/user.svg"
            alt="Login"
            width={36}
            height={36}
            priority
          />
        </header>
        {children}
      </body>
    </html>
  );
}
```

```
$max-width: 1920px;
$text-color: #fff;
$background: linear-gradient(#0069e3, #0056ba);

$spider-man-616-height: 360px;
$mulher-aranha-65-height: 300px;
$spider-man-1610-height: 324px;
$sp-dr-14512-height: 324px;
$spider-ham-8311-height: 146px;
$spider-man-90214-height: 376px;
$spider-man-928-height: 360px;
```

```
@import "./variables.scss";

:root {
  height: 100vh;
  transition: background 1s ease-in-out;

  &::before {
    content: "";
    background: $background;
    height: 100%;
    position: absolute;
    top: 0;
    width: 100%;
  }
}

body {
  color: $text-color;
  height: calc(100vh - 3.5rem);
  margin: auto;
  max-width: $max-width;
  overflow: hidden;
```

```
  position: relative;
}

header {
  align-items: center;
  display: flex;
  justify-content: space-between;
  margin: 3rem 3rem 0;
  position: relative;
  z-index: 4;
}
```

Página inicial

- A próxima etapa é criarmos o conteúdo da página inicial com a lista de heróis do Aranhaverso;
- Seguindo a estrutura do Next.js 13, dentro da pasta `app` temos um arquivo `page.tsx`, que é a página inicial do nosso projeto. Será dentro dela que iremos adicionar o conteúdo da nossa página inicial.

Componente de listagem dos heróis

- Para deixar o nosso código mais organizado, iremos criar um componente denominado `HeroesList`, que irá conter a palavra “Personagens” ao fundo,

juntamente com todos os Spiders do Aranhaverso;

- Para isso, criamos dentro de `/src/components` o componente `HeroesList` que irá conter o `index.tsx` e as estilizações em `heroesList.module.scss`;
- O componente `HeroesList` receberá como propriedade a listagem de heróis originada pela API que criamos anteriormente. Para deixar nosso código mais organizado, criamos também uma interface em `/src/interfaces/heroes.ts`, que será compartilhada em vários pontos do nosso projeto:

```
export interface IHeroData {
  id: string;
  name: string;
  universe: number;
  details: {
    fullName: string;
    birthday: string;
    homeland: string;
    height: number;
    weight: number;
  };
}
```

- Depois disso, iremos criar a estrutura inicial do nosso componente `HeroesList` recebendo, como propriedade, um array da interface `IHeroData`:

```
import { IHeroData } from "@/interfaces/heroes";

interface IProps {
  heroes: IHeroData[];
}

export default function HeroesList({ heroes }: IProps) {
  return (
    <>
      <h1>Personagens</h1>
    </>
  );
}
```

Chamando o componente `HeroesList` na página inicial e enviando a listagem de heróis via propriedade

- Dentro de `/src/app/page.tsx` iremos realizar uma chamada para a API que criamos para trazermos a listagem de heróis;
- Primeiro criamos uma função assíncrona `getData()` e realizamos a chamada para a API que criamos dentro da aplicação:

```
import { IHeroData } from "@/interfaces/heroes";

async function getData(): Promise<{ data: IHeroData[] }> {
  const res = await fetch("http://localhost:3000/api/heroes");

  if (!res.ok) {
    throw new Error("Falha ao buscar heróis");
  }

  return res.json();
}

export default function Home() {
  return <h1>Hello World</h1>;
}
```

- Em seguida, transformamos o nosso componente em uma função assíncrona e realizamos a requisição à API;
- Importamos o componente `HeroesList` e passamos via propriedade a resposta da requisição:

```
...
async function getData(): Promise<{ data: IHeroData[] }> {
  ...
}

export default async function Home() {
```

```
  const res = await getData();

  return <HeroesList heroes={res.data} />;
}
```

- Agora envolvemos o componente `HeroesList` dentro da tag `<main>` para que possamos estilizar e centralizar o conteúdo dentro da página principal:

```
import styles from "./page.module.scss";
...

async function getData(): Promise<{ data: IHeroData[] }> {
  ...
}

export default async function Home() {
  const res = await getData();

  return (
    <main className={styles.main}>
      <HeroesList heroes={res.data} />
    </main>
  );
}
```

```
.main {
  align-items: center;
  display: flex;
  flex-direction: column;
  position: relative;
}
```

Criando o título “Personagens” ao fundo dos heróis

- Dentro do projeto, temos a pasta `/src/fonts` com o arquivo `spider-man.otf`, que é a fonte dos títulos que usaremos em todo o projeto;
- O Next.js oferece uma forma de otimizarmos as fontes da nossa aplicação através do componente Font, incluindo fontes personalizadas (que não são do Google

Fonts), removendo a solicitações de rede externa para melhorar a privacidade e o desempenho;

- Através do `next/font`, é incluído a **auto-hospedagem automática**, integrada para qualquer arquivo de fonte. Isso significa que você pode carregar fontes da Web de maneira otimizada, sem mudanças de layout;
- Este novo sistema de fontes também permite que você use todas as fontes do Google com desempenho e privacidade;
- Os arquivos CSS e de fonte são baixados no momento da compilação e auto-hospedados com o restante de seus recursos estáticos. Nenhuma solicitação é enviada ao Google pelo navegador;
- Neste caso, temos uma fonte customizada em nosso projeto e, para não criarmos uma nova instância toda vez que formos usar esta fonte, criamos um arquivo `/src/fonts/index.ts` e iremos exportar a `spidermanFont` com todas as configurações necessárias:

```
import localFont from "next/font/local";

export const spidermanFont = localFont({
  src: "../fonts/spider-man.otf",
  weight: "400",
  display: "swap",
});
```

- Dentro do componente `HeroesList` iremos criar o título “Personagens” e o estilizaremos:

```
import styles from "./heroesList.module.scss";

import { spidermanFont } from "@/fonts";
import { IHeroData } from "@/interfaces/heroes";

interface IProps {
  heroes: IHeroData[];
```

```
}

export default function HeroesList({ heroes }: IProps) {
  return (
    <>
      <h1 className={`${spidermanFont.className} ${styles.title}`}>
        Personagens
      </h1>
    </>
  );
}
```

```
@import "@/app/variables.scss";

.title {
  font-size: 26vw;
  margin: 0;

  @media screen and (min-width: $max-width) {
    font-size: 30rem;
  }
}
```

Componente `HeroPicture`

- Como iremos usar em vários pontos as imagens dos heróis, iremos criar um componente chamado `HeroPicture`, que terá a responsabilidade de renderizar a foto de um determinado herói passado via propriedade;
- No arquivo `/src/components/HeroPicture/index.tsx` iremos utilizar o componente `Image` no Next.js;
- O componente Image do Next.js extende o elemento `<img>` do HTML com algumas otimizações de imagens automáticas:
  - **Otimização de tamanho:** de forma automática, exibe imagens do tamanho correto para cada dispositivo;
  - **Estabilidade visual:** Evita as mudanças de layout automaticamente quando as imagens estiverem sendo carregadas na tela;
  - **Carregamento de páginas mais rápido:** as imagens são carregadas apenas quando entram na viewport usando o lazy loading nativo do navegador;

    - **Flexibilidade dos Assets:** redimensiona as imagens sob demanda, mesmo em imagens armazenadas em servidores remotos.

- De acordo com o herói passado via parâmetro para o componente, nós iremos mapear qual a imagem que deve ser renderizada através do `id` do spiderman:

```
import Image, { StaticImageData } from "next/image";
import ImageSpiderMan616 from "@public/spiders/spider-man-616.png";
import ImageSpiderMan1610 from "@public/spiders/spider-man-1610.png";
import ImageSpiderWoman65 from "@public/spiders/mulher-aranha-65.png";
import ImageSpDr14512 from "@public/spiders/sp-dr-14512.png";
import ImageSpiderHam8311 from "@public/spiders/spider-ham-8311.png";
import ImageSpiderMan928 from "@public/spiders/spider-man-928.png";
import ImageSpiderMan90214 from "@public/spiders/spider-man-90214.png";

import { IHeroData } from "@/interfaces/heroes";

const heroesImage: Record<string, StaticImageData> = {
  "spider-man-616": ImageSpiderMan616,
  "mulher-aranha-65": ImageSpiderWoman65,
  "spider-man-1610": ImageSpiderMan1610,
  "sp-dr-14512": ImageSpDr14512,
  "spider-ham-8311": ImageSpiderHam8311,
  "spider-man-90214": ImageSpiderMan90214,
  "spider-man-928": ImageSpiderMan928,
};

interface IProps {
  hero: IHeroData;
}

export default function HeroPicture({ hero }: IProps) {
  return (
    <Image
      src={heroesImage[hero.id] || ImageSpiderMan616}
      alt={`${hero.name} (Universo-${hero.universe})` || ""}
      priority
    />
  );
}
```

ℹ️ Adicionando a propriedade **priority** estamos fazendo com que o Next.js priorize especialmente esta imagem para carregá-la com prioridade. Normalmente adicionamos essa propriedade às imagens que possui um destaque maior na página.

Listando as imagens dos heróis

- Dentro do componente `HeroesList` iremos listar todos os heróis, chamando o componente `HeroPicture` que acabamos de criar para mostrar as imagens:

```
import HeroPicture from "../HeroPicture";

import styles from "./heroesList.module.scss";

import { spidermanFont } from "@/fonts";
import { IHeroData } from "@/interfaces/heroes";

interface IProps {
  heroes: IHeroData[];
}

export default function HeroesList({ heroes }: IProps) {
  return (
    <>
      <h1 className={`${spidermanFont.className} ${styles.title}`}>
        Personagens
      </h1>
      <section className={styles.heroes}>
        {heroes.map((hero) => (
          <div
            key={hero.id}
            className={`${styles.imageContainer} ${styles[hero.id]}`}
          >
            <HeroPicture hero={hero} />
          </div>
        ))}
      </section>
    </>
  );
}
```

```scss
@import "@/app/variables.scss";

.heroes {
  align-items: flex-end;
  display: flex;
  justify-content: space-between;
  max-width: calc($max-width - 700px);
  position: absolute;
  top: 40%;
  width: 100%;

  @media screen and (min-width: $max-width) {
    max-width: calc($max-width - 400px);
    top: 50%;
  }
}

.imageContainer {
  cursor: pointer;
  position: relative;

  img {
    height: 100%;
    object-fit: contain;
    object-position: bottom;
    width: 100%;
  }

  &.spider-man-616 {
    height: $spider-man-616-height;
  }

  &.mulher-aranha-65 {
    height: $mulher-aranha-65-height;
  }

  &.spider-man-1610 {
    height: $spider-man-1610-height;
  }

  &.sp-dr-14512 {
    height: $sp-dr-14512-height;
  }

  &.spider-ham-8311 {
    height: $spider-ham-8311-height;
  }

  &.spider-man-90214 {
    height: $spider-man-90214-height;
  }
```

```
  &.spider-man-928 {
    height: $spider-man-928-height;
  }
}

.title {
  font-size: 26vw;
  margin: 0;

  @media screen and (min-width: $max-width) {
    font-size: 30rem;
  }
}
```

[live] Animando a página inicial

Framer Motion

O Framer Motion é uma biblioteca JavaScript popular utilizada para animações e transições de componentes em aplicações React. Ela fornece uma API simples e intuitiva para criar animações fluidas e interativas.

Algumas das vantagens do Framer Motion são:

1. **Fácil de usar**: O Framer Motion possui uma API simples e declarativa, facilitando a criação de animações sem a necessidade de lidar diretamente com propriedades CSS complexas ou manipulação direta do DOM.
2. **Controle total**: Ele oferece controle detalhado sobre as animações, permitindo definir propriedades como duração, atraso, curvas de animação personalizadas, entre outros. Isso permite criar animações precisas e adaptáveis.
3. **Suporte a diversos componentes**: O Framer Motion não está restrito apenas a elementos HTML. Ele pode ser usado com qualquer componente React, incluindo componentes personalizados. Isso permite animar não apenas elementos individuais, mas também a transição entre diferentes estados de componentes complexos.
4. **Gerenciamento de estado**: O Framer Motion facilita o gerenciamento do estado das animações. Você pode controlar o início, pausa, reinício e até mesmo encadeamento de animações em resposta a eventos do componente ou interações do usuário.

5. **Performance otimizada**: O Framer Motion é projetado para ter um desempenho eficiente, utilizando técnicas como animações baseadas em transformações CSS e a API do `requestAnimationFrame` para aproveitar a renderização otimizada do navegador.
6. **Ecossistema ativo**: O Framer Motion possui uma comunidade ativa e uma documentação abrangente, o que facilita encontrar exemplos, tutoriais e suporte em caso de dúvidas.

Essas são apenas algumas das vantagens do Framer Motion. No geral, ele é uma escolha popular para adicionar animações e transições elegantes a aplicações React, oferecendo facilidade de uso, controle preciso e bom desempenho.

Criando a animação do título

- Transforme o elemento `h1` do título em um componente `motion`
  - O `motion` é um componente do Framer Motion que permite animar elementos React de forma fácil e declarativa.
  - Ele adiciona a capacidade de animação a qualquer elemento React, permitindo que você defina propriedades como `animate`, `initial`, `whileHover`, `whileTap`, `transition`, entre outras.

Como estamos usando animações, envolvendo interações do usuário, precisamos declarar que esse componente não é um Server Component, mas sim um componente renderizado no cliente (Client Component). Para isso, adicionamos na primeira linha do arquivo o `"use client"` para podermos usar esses recursos do React.

```
"use client";

...

export default function HeroesList({ heroes }: IProps) {
  return (
```

```
    <>
      <motion.h1
        className={`${spidermanFont.className} ${styles.title}`}
        initial={{ opacity: 0 }}
        animate={{ opacity: 1 }}
        transition={{ delay: 2, duration: 2 }}
      >
        Personagens
      </motion.h1>
      <section className={styles.heroes}>
        ...
      </section>
    </>
  );
}
```

Animando a entrada dos heróis

```
...
export default function HeroesList({ heroes }: IProps) {
  return (
    <>
      <motion.h1
        className={`${spidermanFont.className} ${styles.title}`}
        initial={{ opacity: 0 }}
        animate={{ opacity: 1 }}
        transition={{ delay: 2, duration: 2 }}
      >
        Personagens
      </motion.h1>
      <motion.section
        className={styles.heroes}
        initial={{ y: -100, opacity: 0 }}
        animate={{ y: 0, opacity: 1 }}
        transition={{ duration: 2 }}
      >
        {heroes.map((hero) => (
          <div
            key={hero.id}
            className={`${styles.imageContainer} ${styles[hero.id]}`}
          >
            <HeroPicture hero={hero} />
          </div>
        ))}
      </motion.section>
    </>
  );
}
```

Animando o evento de clicar nos heróis

```
...
export default function HeroesList({ heroes }: IProps) {
  return (
    <>
      <motion.h1
        className={`${spidermanFont.className} ${styles.title}`}
        initial={{ opacity: 0 }}
        animate={{ opacity: 1 }}
        transition={{ delay: 2, duration: 2 }}
      >
        Personagens
      </motion.h1>
      <motion.section
        className={styles.heroes}
        initial={{ y: -100, opacity: 0 }}
        animate={{ y: 0, opacity: 1 }}
        transition={{ duration: 2 }}
      >
        {heroes.map((hero) => (
          <motion.div
            key={hero.id}
            className={`${styles.imageContainer} ${styles[hero.id]}`}
            whileHover={{ scale: 1.3 }}
            whileTap={{ scale: 0.8 }}
            transition={{ duration: 0.8 }}
          >
            <HeroPicture hero={hero} />
          </motion.div>
        ))}
      </motion.section>
    </>
  );
}
```

Rotas e página de heróis

- Para criarmos o carrossel e termos os detalhes de cada herói, iremos criar uma rota separada e receberemos, via parâmetro, qual o id do spider clicado na página inicial para trazermos suas informações logo em seguida;
- Para isso, criamos dentro da pasta `app` o diretório `/app/hero/[id]` e, dentro dessa nova pasta, o arquivo `page.tsx` que irá ter o conteúdo dessa nova página;

```
interface IProps {
  params: {
    id: string;
  };
}

export default function Hero({ params: { id } }: IProps) {
  return <h1>Spider selecionado: {id}</h1>;
}
```

Chamando as rotas de acordo com a listagem de heróis na página inicial

- No componente `HeroesList`, iremos criar um link, para cada uma das imagens dos heróis, que irá redirecionar, ao clicarmos na imagem, para essa nova página que acabamos de criar;
- Colocaremos o id do spider via parâmetro para identificar qual dos heróis do aranhaverso devemos abrir as informações em um primeiro momento:

```
import Link from "next/link";

...

export default function HeroesList({ heroes }: IProps) {
  return (
    <>
      ...
      <section className={styles.heroes}>
        {heroes.map((hero) => (
          <div
            key={hero.id}
            className={`${styles.imageContainer} ${styles[hero.id]}`}
          >
            <Link href={`/hero/${hero.id}`}>
              <HeroPicture hero={hero} />
            </Link>
          </div>
        ))}
      </section>
    </>
  );
}
```

Componente de carrossel

Criando o componente `Carousel`

- Criaremos um componente `Carousel` para criar o efeito de carrossel dos spiders do aranhaverso;
- Esse componente receberá via propriedade:
  - `heroes`, com a listagem de todas as informações dos spiders;
  - `activeId`: id do spider clicado na página anterior.

```
import { IHeroData } from "@/interfaces/heroes";

interface IProps {
  heroes: IHeroData[];
  activeId: string;
}

export default function Carousel({ heroes, activeId }: IProps) {
  return <h1>Componente Carousel</h1>;
}
```

Chamando o componente `Carousel` na nova página e enviando os dados necessários

- O componente `Carousel` precisa que seja enviado, via propriedade, a lista com as informações de todos os spiders. Para isso, iremos criar uma função para chamar a API para obter esses dados (lembre-se de mudar seu componente para uma função assíncrona);
- Enviaremos também via propriedade o parâmetro id que foi recebido pela rota.

```
import Carousel from "@/components/Carousel";
import { IHeroData } from "@/interfaces/heroes";

interface IProps {
  params: {
    id: string;
```

```
  };
}

async function getData(): Promise<{ data: IHeroData[] }> {
  const res = await fetch("http://localhost:3000/api/heroes");

  if (!res.ok) {
    throw new Error("Falha ao buscar heróis");
  }

  return res.json();
}

export default async function Hero({ params: { id } }: IProps) {
  const res = await getData();

  return <Carousel heroes={res.data} activeId={id} />;
}
```

Criando o componente com as informações dos heróis

- Criaremos um novo componente, o `HeroDetails` que será responsável por mostrar todas as informações detalhadas do spider que estiver ativo no carrossel;
- Ele receberá via propriedade `data` os dados do herói no qual ele deve mostrar as informações:

```
import Image from "next/image";
import { Quicksand } from "next/font/google";

import styles from "./heroDetails.module.scss";

import { spidermanFont } from "@/fonts";
import { IHeroData } from "@/interfaces/heroes";

const quicksand = Quicksand({
  subsets: ["latin"],
  weight: ["400", "600", "700"],
});

interface IProps {
  data: IHeroData;
}

export default function HeroDetails({ data }: IProps) {
  const { id, name, universe, details } = data;
```

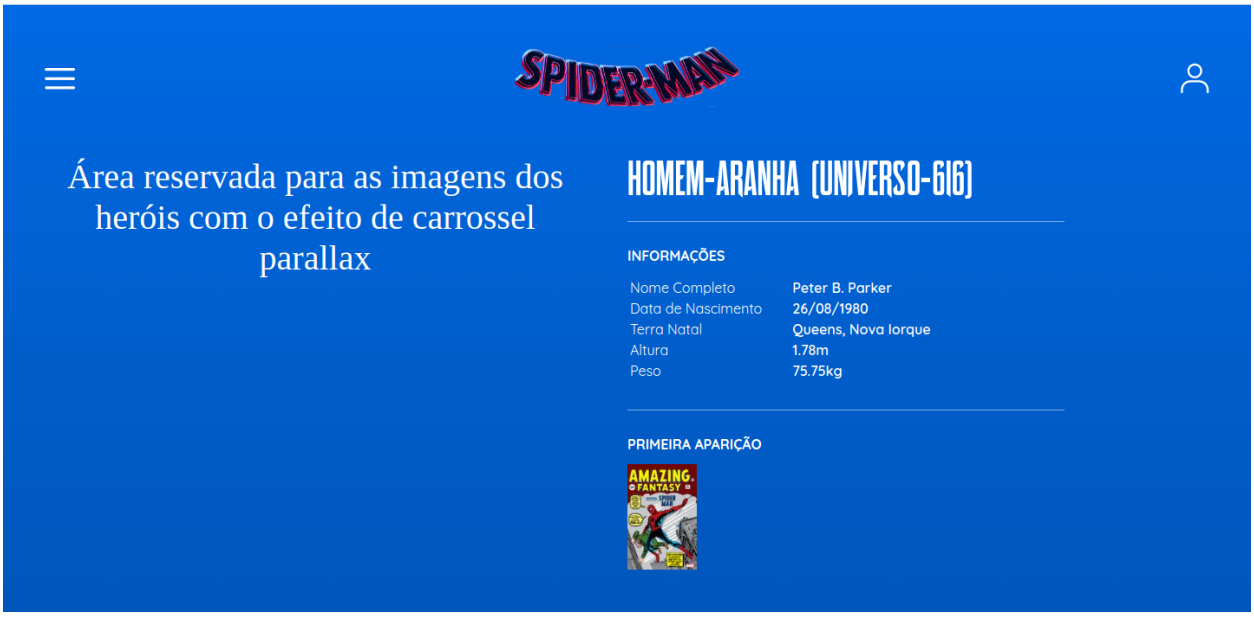
```
  return (
    <div className={quicksand.className}>
      <h1 className={`${spidermanFont.className} ${styles.title}`}>
        {name} (Universo-{universe})
      </h1>
      <div className={styles.details}>
        <h2 className={styles.subtitle}>Informações</h2>
        <table className={styles.table}>
          <tbody>
            <tr>
              <td className={styles.label}>Nome Completo</td>
              <td>{details.fullName}</td>
            </tr>
            <tr>
              <td className={styles.label}>Data de Nascimento</td>
              <td>{new Date(details.birthday).toLocaleDateString("pt-BR")}</td>
            </tr>
            <tr>
              <td className={styles.label}>Terra Natal</td>
              <td>{details.homeland}</td>
            </tr>
            <tr>
              <td className={styles.label}>Altura</td>
              <td>{details.height.toFixed(2)}m</td>
            </tr>
            <tr>
              <td className={styles.label}>Peso</td>
              <td>{details.weight.toFixed(2)}kg</td>
            </tr>
          </tbody>
        </table>
      </div>
      <div className={styles.details}>
        <h2 className={styles.subtitle}>Primeira Aparição</h2>
        <Image
          src={`/spiders/${id}-comic-book.png`}
          alt={`Primeira aparição nos quadrinhos de ${name} no universo ${universe}`}
          width={80}
          height={122}
        />
      </div>
    </div>
  );
}
```

```
.title {
  font-size: 3.5rem;
  margin: 0;
}

.subtitle {
```

```
  font-size: 1rem;
  text-transform: uppercase;
  font-weight: 700;
}

.details {
  border-top: 1px solid rgba(255, 255, 255, 0.5);
  margin: 1rem 0;
  padding: 1rem 0;
  width: 70%;

  .table {
    td {
      font-weight: 600;
      padding-right: 2rem;
    }

    .label {
      font-weight: normal;
    }
  }
}
```

Estruturando as áreas do componente `Carousel`

- À esquerda do componente do carrossel iremos ter as imagens dos heróis e, à direita, o componente `HeroDetails`, com os dados do herói que está ativo no momento.



```
import HeroDetails from "../HeroDetails";

import styles from "./carousel.module.scss";

import { IHeroData } from "@/interfaces/heroes";

interface IProps {
  heroes: IHeroData[];
  activeId: string;
}

export default function Carousel({ heroes, activeId }: IProps) {
  return (
    <div className={styles.container}>
      <div className={styles.carousel}>
        <div className={styles.wrapper}>Lista com os heróis</div>
      </div>
      <div className={styles.details}>
        <HeroDetails data={heroes[0]} />
      </div>
    </div>
  );
}
```

```
.container {
  display: flex;
}

.carousel {
  flex: 1;
  width: 100%;
}

.wrapper {
  cursor: grab;
  height: 130vh;
  position: relative;

  &:active {
    cursor: grabbing;
  }
}

.details {
  margin-top: 3rem;
  position: relative;
```

```
  flex: 1;
}
```

Listando os heróis do aranhaverso no componente `Carousel`

- Dentro do efeito do carrossel parallax, nós temos 3 heróis que ficam visíveis: o spider ativo, que fica no centro da animação, um spider à frente e um spider atrás com um efeito de desfoque (*blur*);
- Para controlar esses heróis que ficam visíveis na interface, iremos criar um `state` chamado `visibleItems`, que será um `array` e será iniciado como `null`;
- Também criaremos um outro `state`, o `activeIndex`, que irá armazenar qual o item ativo do carrossel para que possamos rotacionar e dar o efeito proposto do projeto:
  - Seu valor inicial irá buscar na listagem de heróis a posição no qual o `activeId` possui *match* com o que foi passado no parâmetro da rota.

```
"use client";

import { useState } from "react";
...

export default function Carousel({ heroes, activeId }: IProps) {
  // Controla os itens visíveis do carrossel
  const [visibleItems, setVisibleItems] = useState<IHeroData[] | null>(null);

  // Armazena o item ativo do carrossel
  const [activeIndex, setActiveIndex] = useState(
    heroes.findIndex((hero) => hero.id === activeId)
  );

  return (
    ...
  );
}
```

💡 Como estamos usando o `useState` e usaremos `useEffect` + animações, precisamos declarar que esse componente não é um Server Component, mas sim um componente renderizado no cliente (Client Component). Para isso, adicionamos na primeira linha do arquivo o `"use client"` para podermos usar esses recursos do React.

- Adicionamos um `useEffect` para montar os itens/heróis visíveis no carrossel de acordo com que o `activeIndex` altere seu valor:

```
// Altera o visibleItems sempre que o activeIndex é alterado
useEffect(() => {
  // itens que serão mostrados ao longo do carrossel
  const items = [...heroes];

  // calcula o índice do array de acordo com o item ativo
  // de forma que o número nunca saia do escopo do array
  const indexInArrayScope =
    ((activeIndex % items.length) + items.length) % items.length;

  // itens que estão visíveis neste momento para o usuário
  // duplicamos o array para dar a impressão de um carrossel infinito (360deg)
  const visibleItems = [...items, ...items].slice(
    indexInArrayScope,
    indexInArrayScope + 3
  );

  setVisibleItems(visibleItems);
}, [heroes, activeIndex]);

if (!visibleItems) {
  return null;
}
```

ℹ️ **Cálculo do indexInArrayScope**

O cálculo apresentado é usado para obter um índice válido dentro de um determinado array, chamado `items`, com base em um índice ativo, chamado `activeIndex`.

A fórmula `(activeIndex % items.length)` calcula o resto da divisão entre o `activeIndex` e o número de itens do array `items`. Isso garante que o valor resultante esteja dentro dos limites do array (14 posições).

Em seguida, adiciona-se o `items.length` ao resultado anterior e, em seguida, realiza-se novamente o módulo `%` pelo comprimento do array `items`. O objetivo dessa etapa adicional é garantir que o resultado seja sempre um índice válido dentro do array, mesmo quando o `activeIndex` for um número negativo.

A adição de `items.length` antes de realizar o módulo garante que o resultado seja sempre não negativo, independentemente do valor de `activeIndex`.

Por fim, o valor resultante é atribuído à variável `indexInArrayScope`, que representa o índice válido dentro do array `items`.

Em resumo, o cálculo garante que o índice calculado esteja dentro do intervalo válido de índices do array **`items`**, independentemente do valor de **`activeIndex`** (positivo, negativo ou zero). Isso é útil para manipular arrays circularmente, onde o índice pode "voltar" ao início do array quando excede o tamanho do mesmo.

- Listamos as imagens do carrossel de acordo com o conteúdo do estado do `array` de `visibleItems`. O seu componente ficará da seguinte forma:

```
"use client";

import { useEffect, useState } from "react";

import HeroDetails from "../HeroDetails";
```

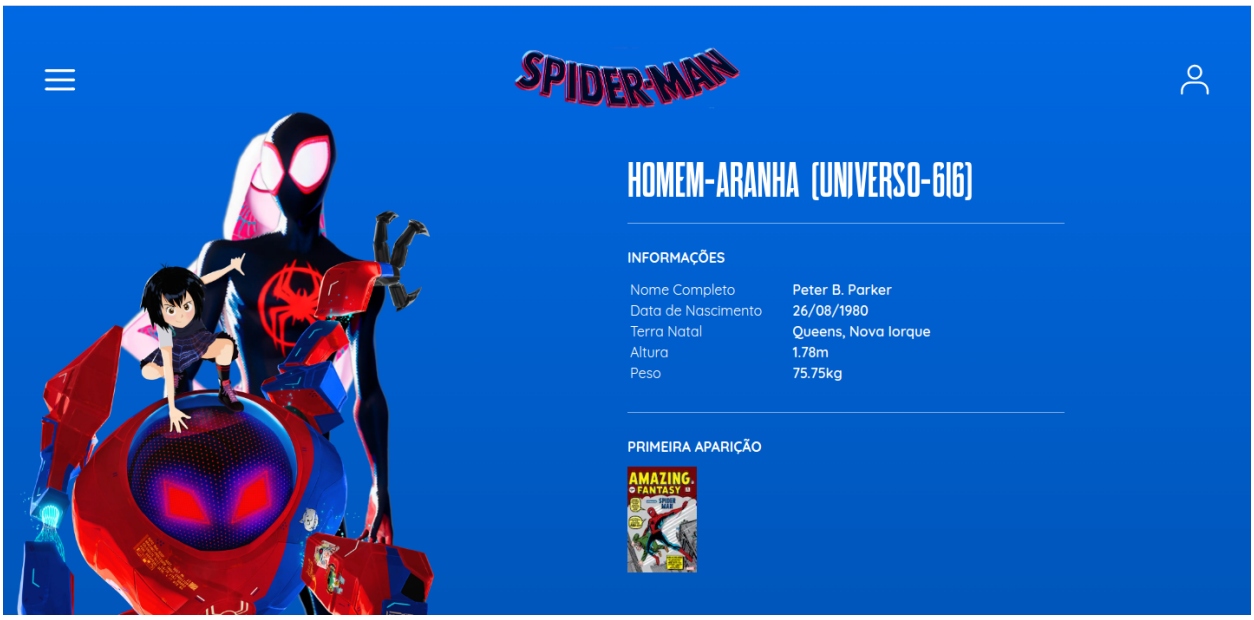
```
import HeroPicture from "../HeroPicture";

import styles from "./carousel.module.scss";

import { IHeroData } from "@/interfaces/heroes";

interface IProps {
  heroes: IHeroData[];
  activeId: string;
}

export default function Carousel({ heroes, activeId }: IProps) {
  // Controla os itens visíveis do carrossel
  const [visibleItems, setVisibleItems] = useState<IHeroData[] | null>(null);

  // Armazena o item ativo do carrossel
  const [activeIndex, setActiveIndex] = useState(
    heroes.findIndex((hero) => hero.id === activeId)
  );

  // Altera o visibleItems sempre que o activeIndex é alterado
  useEffect(() => {
    // itens que serão mostrados ao longo do carrossel
    const items = [...heroes];

    // calcula o índice do array de acordo com o item ativo
    // de forma que o número nunca saia do escopo do array
    const indexInArrayScope =
      ((activeIndex % items.length) + items.length) % items.length;

    // itens que estão visíveis neste momento para o usuário
    // duplicamos o array para dar a impressão de um carrossel infinito (360deg)
    const visibleItems = [...items, ...items].slice(
      indexInArrayScope,
      indexInArrayScope + 3
    );

    setVisibleItems(visibleItems);
  }, [heroes, activeIndex]);

  if (!visibleItems) {
    return null;
  }

  return (
    <div className={styles.container}>
      <div className={styles.carousel}>
        <div className={styles.wrapper}>
          {visibleItems?.map((item) => (
            <div key={item.id} className={styles.hero}>
              <HeroPicture hero={item} />
            </div>
          ))}
        </div>
```

```
      </div>
      <div className={styles.details}>
        <HeroDetails data={heroes[0]} />
      </div>
    </div>
  );
}
```

- Agora precisamos estilizar as imagens, de maneira que um herói fique por cima do outro para dar o efeito que precisamos posteriormente. Para isso, estilizamos a classe CSS `hero` da seguinte maneira:

```
.hero {
  height: 100%;
  left: 0;
  position: absolute;
  width: 500px;

  img {
    height: 100%;
    object-fit: contain;
    object-position: center right;
    width: 100%;
  }
}
```

- Seu projeto, neste momento, ficará assim:



Alterando os itens visíveis do carrossel

- Para alterarmos os itens visíveis do carrossel para dar a impressão de que ele está rotacionando, nós iremos criar uma função que altera o estado do `activeIndex` e com isso, automaticamente, o hook `useEffect` que criamos irá alterar o estado do `visibleItems`;
- Por enquanto chamaremos essa função na interação click do usuário em cima do carrossel para ver o funcionamento.

```
...

export default function Carousel({ heroes, activeId }: IProps) {
 ...

  // Altera herói ativo no carrossel
  // +1 rotaciona no sentido horário
  // -1 rotaciona no sentido anti-horário
  const handleChangeActiveIndex = (newDirection: number) => {
    setActiveIndex((prevActiveIndex) => prevActiveIndex + newDirection);
  };

  if (!visibleItems) {
    return null;
```

```
  }

  return (
    <div className={styles.container}>
      <div className={styles.carousel}>
        <div
          className={styles.wrapper}
          onClick={() => handleChangeActiveIndex(1)}
        >
          {visibleItems?.map((item) => (
            <div key={item.id} className={styles.hero}>
              <HeroPicture hero={item} />
            </div>
          ))}
        </div>
      </div>
      <div className={styles.details}>
        <HeroDetails data={heroes[0]} />
      </div>
    </div>
  );
}
```

- Com isso, já podemos perceber que os itens do carrossel estão sendo alterados. Ao clicar na tela, o item que está atrás some e um novo é adicionado na frente, o que dará a impressão de rotação do carrossel posteriormente.

[live] Animando o carrossel

Aplicando o componente `AnimatePresence`

- Começaremos a animar os itens do carrossel e a adicionar os efeitos de profundidade conforme o que está no protótipo/proposta do projeto;
- Também utilizaremos o Framer Motion para realizar a animação de rotação do carrossel, juntamente com o posicionamento dos heróis em tela;
- A primeira coisa que faremos é encapsular a nossa listagem de imagens de heróis no componente `AnimatePresence` do Framer Motion;

```
...
import { AnimatePresence } from "framer-motion";

export default function Carousel({ heroes, activeId }: IProps) {
```

```
  ...
  return (
    <div className={styles.container}>
      <div className={styles.carousel}>
        <div
          className={styles.wrapper}
          onClick={() => handleChangeActiveIndex(1)}
        >
          <AnimatePresence mode="popLayout">
            {visibleItems?.map((item) => (
              <div key={item.id} className={styles.hero}>
                <HeroPicture hero={item} />
              </div>
            ))}
          </AnimatePresence>
        </div>
      </div>
      <div className={styles.details}>
        <HeroDetails data={heroes[0]} />
      </div>
    </div>
  );
```

- Componente `AnimatePresence`
  - O `AnimatePresence` é um componente do Framer Motion que nos permite animar elementos que entram ou saem de uma lista;
  - Ele é útil em casos em que a lista está sendo atualizada dinamicamente e os elementos estão sendo adicionados ou removidos.
  - A propriedade `mode` desse componente decide como o `AnimatePresence` lida com a entrada e saída dos filhos. O valor `popLayout` faz com que os elementos-filhos do componente, que irão sair ao alterar o array de `visibleItems`, sejam "removidos" do layout da página. Isso permite que os elementos ao redor se movam para sua nova posição imediatamente.

Posicionando e estilizando os elementos

- A próxima etapa é transformarmos a `div` que é o elemento pai do componente `HeroPicture` em um elemento `motion` para adicionarmos animações dentro:

```
...
```

```
import { AnimatePresence, motion } from "framer-motion";

...

export default function Carousel({ heroes, activeId }: IProps) {
  ...

  return (
    ...
      <AnimatePresence mode="popLayout">
        {visibleItems?.map((item) => (
          <motion.div
            key={item.id}
            className={styles.hero}
            transition={{ duration: 0.8 }}
          >
            <HeroPicture hero={item} />
          </motion.div>
        ))}
      </AnimatePresence>
    ...
  );
}
```

- Para os itens que estão visíveis no carrossel, dependendo de sua posição, nós iremos aplicar uma estilização diferente para cada um deles;
- Para deixar o nosso código mais organizado e de fácil manutenção, iremos criar uma função separada que, dependendo da posição do item no array que é passada via parâmetro, irá aplicar um determinado estilo;
- Essa função será chamada de `getItemStyles` e receberá via atributo a posição do item através do `enum` denominado de `enPosition`;
- Depois fazemos uma desestruturação do retorno dessa função na propriedade `animate` da `div` que está em volta da imagem do herói.
  - A propriedade `animate` do Framer Motion é utilizada para definir o estado inicial dos elementos que serão animados;
  - É um objeto que deve conter todas as propriedades CSS que serão animadas.

```
...
```

```
enum enPosition {
  FRONT = 0,
  MIDDLE = 1,
  BACK = 2,
}

export default function Carousel({ heroes, activeId }: IProps) {
  ...
  return (
    ...
      <AnimatePresence mode="popLayout">
        {visibleItems?.map((item, position) => (
          <motion.div
            key={item.id}
            className={styles.hero}
            transition={{ duration: 0.8 }}
            animate={{ ...getItemStyles(position) }}
          >
            <HeroPicture hero={item} />
          </motion.div>
        ))}
      </AnimatePresence>
    ...
  );
}

// estilos para o item que está visível na animação
// dependendo da posição do herói no carrossel
const getItemStyles = (position: enPosition) => {
  if (position === enPosition.FRONT) {
    return {
      filter: "blur(10px)",
      scale: 1.2,
      zIndex: 3,
    };
  }

  if (position === enPosition.MIDDLE) {
    return {
      left: 300,
      scale: 0.8,
      top: "-10%",
      zIndex: 2,
    };
  }

  return {
    filter: "blur(10px)",
    scale: 0.6,
    left: 160,
    opacity: 0.8,
    zIndex: 1,
    top: "-20%",
```

```
  };
};
```

- Com isso, já temos uma base de como a nossa animação ficará e o posicionamento dos heróis de acordo com o índice do array que se encontram em `visibleItems`;
- Para melhorar e polir ainda mais essa transição, podemos definir a estilização das propriedades `initial` e `exit` do componente `motion`:
  - A propriedade `initial` permite definir o estado inicial dos elementos que serão animados;
  - A propriedade `exit` é utilizada para definir como os elementos são animados ao sair da tela.

```
...
<motion.div
  key={item.id}
  className={styles.hero}
  transition={{ duration: 0.8 }}
  initial={{
    x: -1500,
    scale: 0.75,
  }}
  animate={{ x: 0, ...getItemStyles(position) }}
  exit={{
    x: 0,
    left: "-20%",
    opacity: 0,
    scale: 1,
  }}
>
  <HeroPicture hero={item} />
</motion.div>
...
```

- Dessa forma, temos a impressão de que o herói está surgindo de um carrossel 3D, onde uma parte não está visível para o usuário;

- Para deixar o carrossel com os heróis mais a esquerda, como no protótipo, adicionamos nas estilizações da classe `.carousel` as seguintes regras CSS:

```
.carousel {
  flex: 1;
  left: -15%;
  position: relative;
  width: 100%;
}
```

[live] Alterando o fundo da página de acordo com o Spider selecionado no carrossel

- Para alterar o fundo da página de acordo com o herói selecionado no carrossel, iremos criar um hook de efeito (`useEffect`) que ficará observando a alteração do `array` de `visibleItems` e irá aplicar o fundo via style inline no elemento `html`:

```
...
// Altera o fundo da página de acordo com o herói selecionado
useEffect(() => {
  const htmlEl = document.querySelector("html");

  if (!htmlEl || !visibleItems) {
    return;
  }

  const currentHeroId = visibleItems[1].id;
  htmlEl.style.backgroundImage = `url("/spiders/${currentHeroId}-background.png")`;
}, [visibleItems]);
...
```

- Também adicionaremos a classe `hero-page` no elemento `html` para que possamos esconder o fundo azul quando estivermos na página com os detalhes do herói:

```
// Altera o fundo da página de acordo com o herói selecionado
useEffect(() => {
  const htmlEl = document.querySelector("html");

  if (!htmlEl || !visibleItems) {
    return;
  }

  const currentHeroId = visibleItems[1].id;
  htmlEl.style.background = `url("/spiders/${currentHeroId}-background.png")`;
  htmlEl.classList.add("hero-page");

  // remove a classe quando o componente é desmontado
  return () => {
    htmlEl.classList.remove("hero-page");
  };
}, [visibleItems]);
```

```
:root {
  height: 100vh;
  transition: background 1s ease-in-out;
  background-size: cover;

  &:not(.hero-page)::before {
    content: "";
    background: $background;
    height: 100%;
    position: absolute;
    top: 0;
    width: 100%;
  }
}
```

ℹ️ Por conta da propriedade `transition` temos um efeito de animação *fade* quando o fundo da página é alterado.

[live] Animando o bloco de detalhes dos heróis

- Para criarmos um efeito de fade para o bloco de detalhes de heróis, podemos transformar a `div` em volta do componente `HeroDetails` em um elemento `motion` e aplicar as propriedades de animação:

```
...

export default function Carousel({ heroes, activeId }: IProps) {
  ...

  return (
    <div className={styles.container}>
      <div className={styles.carousel}>
        <div
          className={styles.wrapper}
          onClick={() => handleChangeActiveIndex(1)}
        >
          <AnimatePresence mode="popLayout">
            ...
          </AnimatePresence>
        </div>
      </div>
      <motion.div
        className={styles.details}
        initial={{ opacity: 0 }}
        animate={{ opacity: 1 }}
        transition={{ delay: 1, duration: 2 }}
      >
        <HeroDetails data={heroes[0]} />
      </motion.div>
    </div>
  );
}

...
```

[live] Corrigindo bug de herói ativo ao clicar na imagem da página inicial

- Se navegarmos da página inicial para a página dos detalhes do herói ao clicar no spider que queremos ver os detalhes, podemos ver que ele não está em evidência no carrossel.
- Ao acessar, por exemplo, o link http://localhost:3000/hero/mulher-aranha-65, vemos que a Mulher Aranha está a frente e não em destaque no carrossel.
- Para corrigir isso, basta subtraírmos 1 do valor inicial do estado `activeIndex`:

```
// Armazena o item ativo do carrossel
const [activeIndex, setActiveIndex] = useState(
  heroes.findIndex((hero) => hero.id === activeId) - 1
);
```

[live] Altera detalhes do herói de acordo com o spider em destaque

- Para alterar, de acordo com a rotação do carrossel do spider, as informações do herói em evidência na página, basta enviar via parâmetro do componente `HeroDetails` o item do centro (`visibleItems[1]`):

```
<motion.div
  className={styles.details}
  initial={{ opacity: 0 }}
  animate={{ opacity: 1 }}
  transition={{ delay: 1, duration: 2 }}
>
  <HeroDetails data={visibleItems[enPosition.MIDDLE]} />
</motion.div>
```

[live] Aplicando efeitos sonoros ao interagir com o carrossel

- Primeiro, criamos a instância das classes de Audio que iremos utilizar para os efeitos sonoros. Para ajudar na performance da aplicação, podemos utilizar o `useMemo` para isso:

```
"use client";

import { useEffect, useMemo, useState } from "react";
...

export default function Carousel({ heroes, activeId }: IProps) {
  ...

  // Som de transição
```

```
  const transitionAudio = useMemo(() => new Audio("/songs/transition.mp3"), []);

  // Voz de cada personagem
  const voicesAudio: Record<string, HTMLAudioElement> = useMemo(
    () => ({
      "spider-man-616": new Audio("/songs/spider-man-616.mp3"),
      "mulher-aranha-65": new Audio("/songs/mulher-aranha-65.mp3"),
      "spider-man-1610": new Audio("/songs/spider-man-1610.mp3"),
      "sp-dr-14512": new Audio("/songs/sp-dr-14512.mp3"),
      "spider-ham-8311": new Audio("/songs/spider-ham-8311.mp3"),
      "spider-man-90214": new Audio("/songs/spider-man-90214.mp3"),
      "spider-man-928": new Audio("/songs/spider-man-928.mp3"),
    }),
    []
  );

  ...

  return (
    ...
  );
}

...
```

- Por fim, basta criar um `useEffect` que será executado sempre que o estado `activeIndex` for alterado, executando o áudio de transição e da voz do personagem em destaque na página:

```
// Reproduz efeitos sonoros ao rotacionar o carrossel
useEffect(() => {
  if (!visibleItems) {
    return;
  }

  transitionAudio.play();
  const voiceAudio = voicesAudio[visibleItems[1].id];

  if (voiceAudio) {
    voiceAudio.volume = 0.3;
    voiceAudio?.play();
  }
}, [visibleItems, transitionAudio, voicesAudio]);
```

[live] Adicionando interação de clicar e arrastar para rotacionar o carrossel

- Crie um estado para armazenar a posição inicial das interações na tela para que seja possível identificar se o movimento está sendo feito da esquerda para a direita, ou da direita para a esquerda:

```
// Armazena a posição inicial, no eixo x, da interação com o carrossel
const [startInteractionPosition, setStartInteractionPosition] = useState<number>(0);
```

Interação com o ponteiro/mouse

```
...
export default function Carousel({ heroes, activeId }: IProps) {
  ...

  // onDragStart (mouse): armazena a posição inicial da interação
  const handleDragStart = (e: React.DragEvent<HTMLDivElement>) => {
    setStartInteractionPosition(e.clientX);
  };

  // onDragEnd (mouse): armazena a posição final da interação
  // Mexe o carrossel na direção que o usuário fez o evento de interação
  const handleDragEnd = (e: React.DragEvent<HTMLDivElement>) => {
    if (!startInteractionPosition) {
      return null;
    }

    const endInteractionPosition = e.clientX;
    const diffPosition = endInteractionPosition - startInteractionPosition;

    // diffPosition > 0 => direita para esquerda
    // diffPosition < 0 => esquerda para direita
    const newPosition = diffPosition > 0 ? -1 : 1;
    handleChangeActiveIndex(newPosition);
  };

  return (
    <div className={styles.container}>
      <div className={styles.carousel}>
        <div
          className={styles.wrapper}
          onDragStart={handleDragStart}
          onDragEnd={handleDragEnd}
        >
```

```
          <AnimatePresence mode="popLayout">
            ...
          </AnimatePresence>
        </div>
      </div>
      <motion.div
        className={styles.details}
        initial={{ opacity: 0 }}
        animate={{ opacity: 1 }}
        transition={{ delay: 1, duration: 2 }}
      >
        <HeroDetails data={visibleItems[enPosition.MIDDLE]} />
      </motion.div>
    </div>
  );
}

...
```

Interação com touch

```
...

export default function Carousel({ heroes, activeId }: IProps) {
  ...

  // onTouchStart (touch): armazena a posição inicial da interação
  const handleTouchStart = (e: React.TouchEvent<HTMLDivElement>) => {
    setStartInteractionPosition(e.touches[0].clientX);
  };

  // onTouchEnd (touch): armazena a posição final da interação
  // Mexe o carrossel na direção que o usuário fez o evento de interação
  const handleTouchEnd = (e: React.TouchEvent<HTMLDivElement>) => {
    if (!startInteractionPosition) {
      return null;
    }

    const endInteractionPosition = e.changedTouches[0].clientX;

    // diffPosition > 0 => direita para esquerda
    // diffPosition < 0 => esquerda para direita
    const diffPosition = endInteractionPosition - startInteractionPosition;

    const newPosition = diffPosition > 0 ? -1 : 1;
    handleChangeActiveIndex(newPosition);
  };

  if (!visibleItems) {
    return null;
```

```
  }

  return (
    <div className={styles.container}>
      <div className={styles.carousel}>
        <div
          className={styles.wrapper}
          onDragStart={handleDragStart}
          onDragEnd={handleDragEnd}
          onTouchStart={handleTouchStart}
          onTouchEnd={handleTouchEnd}
        >
          <AnimatePresence mode="popLayout">
            ...
          </AnimatePresence>
        </div>
      </div>
      <motion.div
        className={styles.details}
        initial={{ opacity: 0 }}
        animate={{ opacity: 1 }}
        transition={{ delay: 1, duration: 2 }}
      >
        <HeroDetails data={visibleItems[enPosition.MIDDLE]} />
      </motion.div>
    </div>
  );
}
...
```

Desafio final

O projeto ainda não está totalmente responsivo. Então, que tal trabalhar na responsividade para que esse projeto possa ser acessado de qualquer dispositivo móvel também? Finalize o projeto e nos marque nas redes sociais! 😉