Jobify

Create New Next.js Project

```
npx create-next-app@latest projectName
```

- choose typescript and eslint

Assets

- project repo
    - 03-jobify/assets

Libraries

```
   npm install @clerk/nextjs@^4.27.7 @prisma/client@^5.7.0
@tanstack/react-query@^5.14.0 @tanstack/react-query-devtools@^5.14.0
dayjs@^1.11.10 next-themes@^0.2.1 recharts@^2.10.3
   npm install prisma@^5.7.0 -D
```

shadcn/ui

Docs

- follow Next.js install steps (starting with 2)
- open another terminal window (optional)

```
npx shadcn-ui@latest init
```

- setup Button

```
npx shadcn-ui@latest add button
```

Icons

page.tsx

```
import { Button } from '@/components/ui/button';
import { Camera } from 'lucide-react';

export default function Home() {
```

```
  return (
    <div className='h-screen flex items-center justify-center'>
      <Button>default button</Button>
      <Button variant='outline' size='icon'>
        <Camera />
      </Button>
    </div>
  );
}
```

Challenge - Build the Home Page (app/page.tsx)

1. **Import necessary modules and components:**
    - Import the `Image` component from 'next/image' for displaying images.
    - Import the `Logo` and `LandingImg` SVG files from the assets directory.
    - Import the `Button` component from the UI components directory.
    - Import the `Link` component from 'next/link' for navigation.
2. **Define the `Home` component:**
    - This component doesn't receive any props.
3. **Inside the `Home` component, return the JSX:**
    - The main wrapper is a `main` HTML element.
    - Inside `main`, there are two main sections: `header` and `section`.
    - The `header` contains the `Image` component that displays the `Logo`.
    - The `section` contains a `div` and an `Image` component.
    - The `div` contains a `h1` heading, a `p` paragraph, and a `Button` component.
    - The `Button` component wraps a `Link` component that navigates to the '/add-job' route when clicked.
    - The `Image` component displays the `LandingImg`.
4. **Apply CSS classes for styling:**
    - CSS classes are applied to the elements for styling. These classes are from Tailwind CSS, a utility-first CSS framework.
5. **Export the `Home` component as the default export of the module.**

Layout and Home Page

- setup title and description
- add favicon
- setup home page

layout.tsx

```
export const metadata: Metadata = {
  title: 'Jobify Dev',
  description: 'Job application tracking system for job hunters',
};
```

page.tsx

```
import Image from 'next/image';
import Logo from '../assets/logo.svg';
import LandingImg from '../assets/main.svg';
import { Button } from '@/components/ui/button';
import Link from 'next/link';
export default function Home() {
  return (
    <main>
      <header className='max-w-6xl mx-auto px-4 sm:px-8 py-6 '>
        <Image src={Logo} alt='logo' />
      </header>
      <section className='max-w-6xl mx-auto px-4 sm:px-8 h-screen -mt-20
grid lg:grid-cols-[1fr,400px] items-center'>
        <div>
          <h1 className='capitalize text-4xl md:text-7xl font-bold'>
            job <span className='text-primary'>tracking</span> app
          </h1>
          <p className='leading-loose max-w-md mt-4 '>
            I am baby wayfarers hoodie next level taiyaki brooklyn cliche
blue
            bottle single-origin coffee chia. Aesthetic post-ironic venmo,
            quinoa lo-fi tote bag adaptogen everyday carry meggings +1
brunch
            narwhal.
          </p>
          <Button asChild className='mt-4'>
            <Link href='/add-job'>Get Started</Link>
          </Button>
        </div>
        <Image src={LandingImg} alt='landing' className='hidden lg:block '
/>
      </section>
    </main>
  );
}
```

Favicon and Logo (optional)

- Favicon
- Undraw
- Logo - Figma File

Challenge - Setup Dashboard Pages

- create add-job, jobs and stats pages
- group them in (dashboard)
- setup a layout file (for now just pass children)

Dashboard Pages

- create add-job, jobs and stats pages
- group them in (dashboard)
- setup a layout file (just pass children)

(dashboard)/layout.tsx

```
function layout({ children }: { children: React.ReactNode }) {
  return <div>{children}</div>;
}
export default layout;
```

Challenge - Add Clerk Auth

- setup new app, configure fields - (or use existing)
- add ENV Vars
- wrap layout in Clerk Provider
- add middleware
- set only home page public
- restart dev server

Clerk Auth

- setup new app, configure fields - (or use existing)
- add ENV Vars
- wrap layout
- add middleware
- make '/' public
- restart dev server

layout.tsx

```
import { ClerkProvider } from '@clerk/nextjs';

export default function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <ClerkProvider>
```

```
      <html lang='en'>
        <body className={inter.className}>{children}</body>
      </html>
    </ClerkProvider>
  );
}
```

middleware.tsx

```
import { authMiddleware } from '@clerk/nextjs';

// This example protects all routes including api/trpc routes
// Please edit this to allow other routes to be public as needed.
// See https://clerk.com/docs/references/nextjs/auth-middleware for more
information about configuring your Middleware
export default authMiddleware({
  publicRoutes: ['/'],
});

export const config = {
  matcher: ['/((?!.+\\.[\\w]+$|_next).*)', '/', '/(api|trpc)(.*)'],
};
```

Challenge - Build the links.tsx Component

1. **Create File and Import necessary modules and components:**
    - create utils/links.tsx
    - Import the `AreaChart`, `Layers`, and `AppWindow` components from 'lucide-react' for displaying icons.
2. **Define the `NavLink` type:**
    - This type has three properties: `href` (a string), `label` (a string), and `icon` (a React Node).
3. **Define the `links` constant:**
    - This constant is an array of `NavLink` objects.
    - Each object represents a navigation link with a `href`, `label`, and `icon`.
4. **Define the navigation links:**
    - The first link has a `href` of '/add-job', a `label` of 'add job', and an `icon` of `<Layers />`.
    - The second link has a `href` of '/jobs', a `label` of 'all jobs', and an `icon` of `<AppWindow />`.
    - The third link has a `href` of '/stats', a `label` of 'stats', and an `icon` is not defined yet.
5. **Export the `links` constant:**
    - This constant can be imported in other components to create navigation menus.

Links Data

- create utils/links.tsx

utils/links.tsx

```
import { AreaChart, Layers, AppWindow } from 'lucide-react';

type NavLink = {
  href: string;
  label: string;
  icon: React.ReactNode;
};

const links: NavLink[] = [
  {
    href: '/add-job',
    label: 'add job',
    icon: <Layers />,
  },
  {
    href: '/jobs',
    label: 'all jobs',
    icon: <AppWindow />,
  },
  {
    href: '/stats',
    label: 'stats',
    icon: <AreaChart />,
  },
];

export default links;
```

Challenge - Dashboard Layout

- create following components :
  - Sidebar
  - Navbar
  - LinksDropdown
  - ThemeToggle
- setup (dashboard/layout.tsx)

1. **Import necessary modules and components:**
   - Import `Navbar` and `Sidebar` components.
   - Import `PropsWithChildren` from 'react'.
2. **Define the `layout` component:**

- This component receives `children` as props.

3. **Return the JSX:**
   - The main wrapper is a `main` element with a grid layout.
   - The first `div` contains the `Sidebar` component and is hidden on small screens.
   - The second `div` spans 4 columns on large screens and contains the `Navbar` component and the `children`.

4. **Export the `layout` component.** dashboard/layout.tsx

Dashboard Layout

- create following components :
  - Sidebar
  - Navbar
  - LinksDropdown
  - ThemeToggle

(dashboard/layout.tsx)

```
import Navbar from '@/components/Navbar';
import Sidebar from '@/components/Sidebar';

import { PropsWithChildren } from 'react';

function layout({ children }: PropsWithChildren) {
  return (
    <main className='grid lg:grid-cols-5'>
      {/* first-col hide on small screen */}
      <div className='hidden lg:block lg:col-span-1 lg:min-h-screen'>
        <Sidebar />
      </div>
      {/* second-col hide dropdown on big screen */}

      <div className='lg:col-span-4'>
        <Navbar />
        <div className='py-16 px-4 sm:px-8 lg:px-16'>{children}</div>
      </div>
    </main>
  );
}
export default layout;
```

Challenge - Build Sidebar Component

1. **Import necessary modules and components:**
   - Import `Logo`, `links`, `Image`, `Link`, `Button`, and `usePathname`.

2. **Define the `Sidebar` component:**
    - Use `usePathname` to get the current route.
3. **Return the JSX:**
    - The main wrapper is an `aside` element.
    - Inside `aside`, display the `Logo` using `Image`.
    - Map over `links` to create `Button` components for each link.
    - Each `Button` wraps a `Link` that navigates to the link's `href`.
4. **Export the `Sidebar` component.**

Sidebar

- render links and logo
- check the path, if active use different variant Sidebar.tsx

```
'use client';
import Logo from '@/assets/logo.svg';
import links from '@/utils/links';
import Image from 'next/image';
import Link from 'next/link';
import { Button } from './ui/button';
import { usePathname } from 'next/navigation';
function Sidebar() {
  const pathname = usePathname();

  return (
    <aside className='py-4 px-8 bg-muted h-full'>
      <Image src={Logo} alt='logo' className='mx-auto' />
      <div className='flex flex-col mt-20 gap-y-4'>
        {links.map((link) => {
          return (
            <Button
              asChild
              key={link.href}
              variant={pathname === link.href ? 'default' : 'link'}
            >
              <Link href={link.href} className='flex items-center gap-x-2
'>
                {link.icon} <span className='capitalize'>{link.label}
</span>
              </Link>
            </Button>
          );
        })}
      </div>
    </aside>
  );
}
export default Sidebar;
```

Challenge - Build Navbar Component

1. **Import necessary modules and components:**
    - Import `LinksDropdown`, `UserButton` from '@clerk/nextjs', and `ThemeToggle`.
2. **Define the `Navbar` component:**
    - This component doesn't receive any props.
3. **Return the JSX:**
    - The main wrapper is a `nav` element with Tailwind CSS classes for styling.
    - Inside `nav`, there are two `div` elements.
    - The first `div` contains the `LinksDropdown` component.
    - The second `div` contains the `ThemeToggle` and `UserButton` components.
4. **Export the `Navbar` component.**

Navbar

Navbar.tsx

```
import LinksDropdown from './LinksDropdown';
import { UserButton } from '@clerk/nextjs';
import ThemeToggle from './ThemeToggle';

function Navbar() {
  return (
    <nav className='bg-muted py-4 sm:px-16 lg:px-24 px-4 flex items-center
justify-between'>
      <div>
        <LinksDropdown />
      </div>
      <div className='flex items-center gap-x-4'>
        <ThemeToggle />
        <UserButton afterSignOutUrl='/' />
      </div>
    </nav>
  );
}
export default Navbar;
```

Challenge - Build LinksDropdown Component

1. **Explore the Dropdown-Menu Component:**
    - Explore the dropdown-menu component in the shadcn library.
2. **Install the Dropdown-Menu Component:**

- Install it using `npx shadcn-ui@latest add dropdown-menu`

3. **Import necessary modules and components:**

   - Import `DropdownMenu`, `DropdownMenuContent`, `DropdownMenuItem`, `DropdownMenuTrigger` from the dropdown-menu component.
   - Import `AlignLeft` from 'lucide-react' for the menu icon.
   - Import `Button` from the local UI components.
   - Import `links` from the local utilities.
   - Import `Link` from 'next/link' for navigation.

4. **Define the `DropdownLinks` function component:**

   - This component doesn't receive any props.

5. **Inside the `DropdownLinks` component, return the JSX:**

   - The main wrapper is the `DropdownMenu` component.
   - Inside `DropdownMenu`, there is a `DropdownMenuTrigger` component that triggers the dropdown menu. It has a `Button` component with an `AlignLeft` icon. This button is hidden on large screens.
   - The `DropdownMenuContent` component contains the dropdown menu items. Each item is a `DropdownMenuItem` component that wraps a `Link` component. The `Link` component navigates to the link's `href` when clicked.

6. **Export the `DropdownLinks` component:**

   - The `DropdownLinks` component is exported as the default export of the module. This allows it to be imported in other files using the file path.

LinksDropdown

- docs

```
npx shadcn-ui@latest add dropdown-menu
```

LinksDropdown.tsx

```
import {
  DropdownMenu,
  DropdownMenuContent,
  DropdownMenuItem,
  DropdownMenuTrigger,
} from '@/components/ui/dropdown-menu';
import { AlignLeft } from 'lucide-react';
import { Button } from './ui/button';
import links from '@/utils/links';
import Link from 'next/link';
function DropdownLinks() {
```

```
    return (
      <DropdownMenu>
        <DropdownMenuTrigger asChild className='lg:hidden'>
          <Button variant='outline' size='icon'>
            <AlignLeft />

            <span className='sr-only'>Toggle links</span>
          </Button>
        </DropdownMenuTrigger>
        <DropdownMenuContent
          className='w-52 lg:hidden '
          align='start'
          sideOffset={25}
        >
          {links.map((link) => {
            return (
              <DropdownMenuItem key={link.href}>
                <Link href={link.href} className='flex items-center gap-x-2
'>
                  {link.icon} <span className='capitalize'>{link.label}
</span>
                </Link>
              </DropdownMenuItem>
            );
          })}
        </DropdownMenuContent>
      </DropdownMenu>
    );
}
export default DropdownLinks;
```

Challenge - Add New Theme

- reference shadcn docs

Theming Themes

- setup theme in globals.css

Challenge - Setup providers.tsx

- create providers.tsx
- wrap children in layout

Providers

- create providers.tsx
- wrap children in layout
- add suppressHydrationWarning prop

app/providers.tsx

```
'use client';

const Providers = ({ children }: { children: React.ReactNode }) => {
  return <>{children}</>;
};
export default Providers;
```

app/layout

```
<html lang='en' suppressHydrationWarning>
  <body className={inter.className}>
    <Providers>{children}</Providers>
  </body>
</html>
```

Challenge - Add Dark Mode

- reference shadcn docs and setup dark theme Dark Mode

Dark Mode

Dark Mode

```
npm install next-themes
```

components/theme-provider.tsx

```
'use client';

import * as React from 'react';
import { ThemeProvider as NextThemesProvider } from 'next-themes';
import { type ThemeProviderProps } from 'next-themes/dist/types';

export function ThemeProvider({ children, ...props }: ThemeProviderProps)
{
  return <NextThemesProvider {...props}>{children}</NextThemesProvider>;
}
```

app/providers.tsx

```
'use client';
import { ThemeProvider } from '@/components/theme-provider';
const Providers = ({ children }: { children: React.ReactNode }) => {
```

```
  return (
    <>
      <ThemeProvider
        attribute='class'
        defaultTheme='system'
        enableSystem
        disableTransitionOnChange
      >
        {children}
      </ThemeProvider>
    </>
  );
};
export default Providers;
```

ThemeToggle.tsx

```
'use client';

import * as React from 'react';
import { Moon, Sun } from 'lucide-react';
import { useTheme } from 'next-themes';

import { Button } from '@/components/ui/button';
import {
  DropdownMenu,
  DropdownMenuContent,
  DropdownMenuItem,
  DropdownMenuTrigger,
} from '@/components/ui/dropdown-menu';

export default function ModeToggle() {
  const { setTheme } = useTheme();

  return (
    <DropdownMenu>
      <DropdownMenuTrigger asChild>
        <Button variant='outline' size='icon'>
          <Sun className='h-[1.2rem] w-[1.2rem] rotate-0 scale-100
transition-all dark:-rotate-90 dark:scale-0' />
          <Moon className='absolute h-[1.2rem] w-[1.2rem] rotate-90 scale-
0 transition-all dark:rotate-0 dark:scale-100' />
          <span className='sr-only'>Toggle theme</span>
        </Button>
      </DropdownMenuTrigger>
      <DropdownMenuContent align='end'>
        <DropdownMenuItem onClick={() => setTheme('light')}>
          Light
        </DropdownMenuItem>
        <DropdownMenuItem onClick={() => setTheme('dark')}>
          Dark
        </DropdownMenuItem>
```

```
        <DropdownMenuItem onClick={() => setTheme('system')}>
          System
        </DropdownMenuItem>
      </DropdownMenuContent>
    </DropdownMenu>
  );
}
```

Shadcn/ui Forms

- install

```
npx shadcn-ui@latest add form input
```

CreateJobForm Setup

- components/CreateJobForm
- render in add-job/page.tsx

```
'use client';

import * as z from 'zod';
import { zodResolver } from '@hookform/resolvers/zod';
import { useForm } from 'react-hook-form';

import { Button } from '@/components/ui/button';
import {
  Form,
  FormControl,
  FormField,
  FormItem,
  FormLabel,
  FormMessage,
} from '@/components/ui/form';
import { Input } from '@/components/ui/input';

const formSchema = z.object({
  username: z.string().min(2, {
    message: 'Username must be at least 2 characters.',
  }),
});

function CreateJobForm() {
  // 1. Define your form.
  const form = useForm<z.infer<typeof formSchema>>({
    resolver: zodResolver(formSchema),
    defaultValues: {
      username: '',
    },
```

```
  });

  // 2. Define a submit handler.
  function onSubmit(values: z.infer<typeof formSchema>) {
    // Do something with the form values.
    // ✅ This will be type-safe and validated.
    console.log(values);
  }

  return (
    <Form {...form}>
      <form onSubmit={form.handleSubmit(onSubmit)} className='space-y-8'>
        <FormField
          control={form.control}
          name='username'
          render={({ field }) => (
            <FormItem>
              <FormLabel>Username</FormLabel>
              <FormControl>
                <Input placeholder='shadcn' {...field} />
              </FormControl>
              <FormMessage />
            </FormItem>
          )}
        />
        <Button type='submit'>Submit</Button>
      </form>
    </Form>
  );
}
export default CreateJobForm;
```

CreateJobForm - Details

1. **Imports:** Necessary modules and components are imported. This includes form handling and validation libraries, UI components, and the zod schema validation library.
2. **Form Schema:** A `formSchema` is defined using zod. This schema specifies that the `username` field is a string and must be at least 2 characters long.
3. **CreateJobForm Component:** This is the main component. It uses the `useForm` hook from `react-hook-form` to create a form instance which can be used to manage form state, handle form submission, and perform form validation. The form instance is configured with the zod schema as its resolver and a default value for the `username` field.
4. **Submit Handler:** A `onSubmit` function is defined. This function logs the form values when the form is submitted. The form values are type-checked and validated against the zod schema.
5. **Render:** The component returns a form with a single `username` field and a submit button. The `username` field is rendered using the `FormField` component, which is passed the form control and

the field name. The `render` prop of `FormField` is used to render the actual input field and its associated label and message.

6. **Export:** The `CreateJobForm` component is exported as the default export of the module. This allows it to be imported in other files using the file path.

Challenge - Create Types

1. **Create utils/types.ts:**
    - Create a new file named `types.ts` inside the `utils` directory.
2. **Define the `JobStatus` and `JobMode` enums:**
    - Define the `JobStatus` enum with the values 'applied', 'interview', 'offer', and 'rejected'.
    - Define the `JobMode` enum with the values 'fullTime', 'partTime', and 'internship'.
3. **Define the `createAndEditJobSchema` object:**
    - Use `z.object()` from the zod library to define a schema for creating and editing jobs.
    - The schema includes `position`, `company`, `location`, `status`, and `mode`. Each of these fields is a string with a minimum length of 2 characters, except for `status` and `mode` which are enums.
4. **Export the `createAndEditJobSchema` object:**
    - Export the `createAndEditJobSchema` object so it can be used in other files.
5. **Define and export the `CreateAndEditJobType` type:**
    - Use `z.infer<typeof createAndEditJobSchema>` to infer the type of the `createAndEditJobSchema` object.
    - Export the `CreateAndEditJobType` type so it can be used in other files.

Enums in TypeScript are a special type that allows you to define a set of named constants. They can be numeric or string-based.

Types

- utils/types.ts

```
import * as z from 'zod';

export type JobType = {
  id: string;
  createdAt: Date;
  updatedAt: Date;
  clerkId: string;
  position: string;
  company: string;
  location: string;
  status: string;
```

```
  mode: string;
};

export enum JobStatus {
  Pending = 'pending',
  Interview = 'interview',
  Declined = 'declined',
}

export enum JobMode {
  FullTime = 'full-time',
  PartTime = 'part-time',
  Internship = 'internship',
}
// Enums in TypeScript are a special type that allows you to define a set
of named constants. They can be numeric or string-based.

export const createAndEditJobSchema = z.object({
  position: z.string().min(2, {
    message: 'position must be at least 2 characters.',
  }),
  company: z.string().min(2, {
    message: 'company must be at least 2 characters.',
  }),
  location: z.string().min(2, {
    message: 'location must be at least 2 characters.',
  }),
  status: z.nativeEnum(JobStatus),
  mode: z.nativeEnum(JobMode),
});

export type CreateAndEditJobType = z.infer<typeof createAndEditJobSchema>;
```

Explore Select Component

- install

```
npx shadcn-ui@latest add select
```

```
import {
  Select,
  SelectContent,
  SelectItem,
  SelectTrigger,
  SelectValue,
} from '@/components/ui/select';

<Select>
  <SelectTrigger className='w-[180px]'>
```

```
    <SelectValue placeholder='Theme' />
  </SelectTrigger>
  <SelectContent>
    <SelectItem value='light'>Light</SelectItem>
    <SelectItem value='dark'>Dark</SelectItem>
    <SelectItem value='system'>System</SelectItem>
  </SelectContent>
</Select>;
```

- docs

Challenge - FormComponents

1. **Import necessary libraries and components**
    - Import the `Control` type from `react-hook-form`.
    - Import the `Select`, `SelectContent`, `SelectItem`, `SelectTrigger`, and `SelectValue` components from your UI library.
    - Import the `FormControl`, `FormField`, `FormItem`, `FormLabel`, and `FormMessage` components from your UI library.
    - Import the `Input` component from your local UI components.
2. **Define the types for CustomFormField and CustomFormSelect components**
    - Define a type `CustomFormFieldProps` that includes `name` and `control` properties.
    - Define a type `CustomFormSelectProps` that includes `name`, `control`, `items`, and `labelText` properties.
3. **Define the CustomFormField component**
    - Define a new function component named `CustomFormField` that takes `CustomFormFieldProps` as props.
4. **Create the CustomFormField UI**
    - Inside the `CustomFormField` component, return a `FormField` component.
    - Pass `control` and `name` to the `FormField` component.
    - Inside the `FormField` component, render a `FormItem` that contains a `FormLabel`, a `FormControl` with an `Input`, and a `FormMessage`.
5. **Define the CustomFormSelect component**
    - Define a new function component named `CustomFormSelect` that takes `CustomFormSelectProps` as props.
6. **Create the CustomFormSelect UI**
    - Inside the `CustomFormSelect` component, return a `FormField` component.
    - Pass `control` and `name` to the `FormField` component.
    - Inside the `FormField` component, render a `FormItem` that contains a `FormLabel`, a `Select` with a `SelectTrigger` and `SelectContent`, and a `FormMessage`.

- Inside the `SelectContent`, map over the `items` and return a `SelectItem` for each item.

7. **Export the components**
    - Export `CustomFormField` and `CustomFormSelect` so they can be used in other parts of your application.

FormComponents

- components/FormComponents

```
import { Control } from 'react-hook-form';
import {
  Select,
  SelectContent,
  SelectItem,
  SelectTrigger,
  SelectValue,
} from '@/components/ui/select';
import {
  FormControl,
  FormField,
  FormItem,
  FormLabel,
  FormMessage,
} from '@/components/ui/form';
import { Input } from './ui/input';

type CustomFormFieldProps = {
  name: string;
  control: Control<any>;
};

export function CustomFormField({ name, control }: CustomFormFieldProps) {
  return (
    <FormField
      control={control}
      name={name}
      render={({ field }) => (
        <FormItem>
          <FormLabel className='capitalize'>{name}</FormLabel>
          <FormControl>
            <Input {...field} />
          </FormControl>
          <FormMessage />
        </FormItem>
      )}
    />
  );
}

type CustomFormSelectProps = {
```

```
  name: string;
  control: Control<any>;
  items: string[];
  labelText?: string;
};

export function CustomFormSelect({
  name,
  control,
  items,
  labelText,
}: CustomFormSelectProps) {
  return (
    <FormField
      control={control}
      name={name}
      render={({ field }) => (
        <FormItem>
          <FormLabel className='capitalize'>{labelText || name}
</FormLabel>
          <Select onValueChange={field.onChange} defaultValue=
{field.value}>
            <FormControl>
              <SelectTrigger>
                <SelectValue />
              </SelectTrigger>
            </FormControl>
            <SelectContent>
              {items.map((item) => {
                return (
                  <SelectItem key={item} value={item}>
                    {item}
                  </SelectItem>
                );
              })}
            </SelectContent>
          </Select>

          <FormMessage />
        </FormItem>
      )}
    />
  );
}
export default CustomFormSelect;
```

Challenge - CreateJobForm

1. **Import necessary libraries and components**
    - Import the `zodResolver` from `@hookform/resolvers/zod` for form validation.
    - Import the `useForm` hook from `react-hook-form` for form handling.

- Import the necessary types and schemas for your form from `@/utils/types`.
- Import the `Button` and `Form` components from `@/components/ui`.
- Import the `CustomFormField` and `CustomFormSelect` components from `./FormComponents`.

2. **Define the CreateJobForm component**
   - Define a new function component named `CreateJobForm`.
3. **Initialize the form with useForm**
   - Inside the `CreateJobForm` component, use the `useForm` hook to initialize your form.
   - Pass the `CreateAndEditJobType` for your form data to `useForm`.
   - Use `zodResolver` with your `createAndEditJobSchema` for form validation.
4. **Define default values for the form**
   - Define default values for your form fields in the `useForm` hook.
5. **Define the form submission handler**
   - Inside the `CreateJobForm` component, define a function for handling form submission.
   - This function should take the form data as its parameter.
6. **Create the form UI**
   - In the component's return statement, create the form UI using the `Form` component.
   - Use your custom form field components to create the form fields.
   - Add a submit button to the form.
7. **Export the CreateJobForm component**
   - After defining the `CreateJobForm` component, export it so it can be used in other parts of your application.

CreateJobForm

```
'use client';

import { zodResolver } from '@hookform/resolvers/zod';
import { useForm } from 'react-hook-form';

import {
  JobStatus,
  JobMode,
  createAndEditJobSchema,
  CreateAndEditJobType,
} from '@/utils/types';

import { Button } from '@/components/ui/button';
import { Form } from '@/components/ui/form';
```

```
import { CustomFormField, CustomFormSelect } from './FormComponents';

function CreateJobForm() {
  // 1. Define your form.
  const form = useForm<CreateAndEditJobType>({
    resolver: zodResolver(createAndEditJobSchema),
    defaultValues: {
      position: '',
      company: '',
      location: '',
      status: JobStatus.Pending,
      mode: JobMode.FullTime,
    },
  });

  function onSubmit(values: CreateAndEditJobType) {
    // Do something with the form values.
    // ✅ This will be type-safe and validated.
    console.log(values);
  }

  return (
    <Form {...form}>
      <form
        onSubmit={form.handleSubmit(onSubmit)}
        className='bg-muted p-8 rounded'
      >
        <h2 className='capitalize font-semibold text-4xl mb-6'>add
job</h2>
        <div className='grid gap-4 md:grid-cols-2 lg:grid-cols-3 items-
start'>
          {/* position */}
          <CustomFormField name='position' control={form.control} />
          {/* company */}
          <CustomFormField name='company' control={form.control} />
          {/* location */}
          <CustomFormField name='location' control={form.control} />

          {/* job status */}
          <CustomFormSelect
            name='status'
            control={form.control}
            labelText='job status'
            items={Object.values(JobStatus)}
          />
          {/* job  type */}
          <CustomFormSelect
            name='mode'
            control={form.control}
            labelText='job mode'
            items={Object.values(JobMode)}
          />

          <Button type='submit' className='self-end capitalize'>
```

```
            create job
          </Button>
        </div>
      </form>
    </Form>
  );
}
export default CreateJobForm;
```

Create DB in Render

- create .env
- add to .gitignore
- copy external URL DATABASE_URL =

Challenge - Setup Prisma

- setup new prisma instance
- setup connection file
- create Job model

```
model Job {
  id        String      @id @default(uuid())
  clerkId   String
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  position    String
  company   String
  location  String
  status      String
  mode     String
}
```

- push changes to render

Setup Prisma

- setup new prisma instance

```
npx prisma init
```

- setup connection file

utils/db.ts

```

```

```
import { PrismaClient } from '@prisma/client';

const prismaClientSingleton = () => {
  return new PrismaClient();
};

type PrismaClientSingleton = ReturnType<typeof prismaClientSingleton>;

const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClientSingleton | undefined;
};

const prisma = globalForPrisma.prisma ?? prismaClientSingleton();

export default prisma;

if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma =
prisma;
```

- create Job model

schema.prisma

```
/ This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Job {
  id        String      @id @default(uuid())
  clerkId   String
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  position    String
  company   String
  location  String
  status      String
  mode     String
}
```

- push changes to render

```
npx prisma db push
```

Challenge - CreateJobAction

1. **Import necessary libraries and modules**
    - Create utils/action.ts file
    - Import the prisma instance from your database configuration file.
    - Import the auth function from `@clerk/nextjs` for user authentication.
    - Import the necessary types and schemas from your types file.
    - Import the redirect function from `next/navigation` for redirection.
    - Import the `Prisma` namespace from `@prisma/client` for database operations.
    - Import `dayjs` for date and time manipulation.
2. **Define the authenticateAndRedirect function**
    - Define a function named `authenticateAndRedirect` that doesn't take any parameters.
    - Inside this function, call the auth function and destructure `userId` from its return value.
    - If `userId` is not defined, call the redirect function with `'/'` as the argument to redirect the user to the home page.
    - Return `userId`.
3. **Define the createJobAction function**
    - Define an asynchronous function named `createJobAction` that takes values of type `CreateAndEditJobType` as a parameter.
    - This function should return a Promise that resolves to `JobType` or null.
4. **Authenticate the user and validate the form values**
    - Inside the `createJobAction` function, call `authenticateAndRedirect` and store its return value in `userId`.
    - Call `createAndEditJobSchema.parse` with `values` as the argument to validate the form values.
5. **Create a new job in the database**
    - Use the `prisma.job.create` method to create a new job in the database.
    - Pass an object to this method with a `data` property.
    - The `data` property should be an object that spreads the `values` and adds a `clerkId` property with `userId` as its value.
    - Store the return value of this method in `job`.
6. **Handle errors**
    - Wrap the validation and database operation in a try-catch block.
    - If an error occurs, log the error to the console and return null.
7. **Return the new job**

- After the try-catch block, return `job`.

8. **Export the createJobAction function**
    - Export `createJobAction` so it can be used in other parts of your application.

CreateJobAction

- utils/actions.ts

```
'use server';

import prisma from './db';
import { auth } from '@clerk/nextjs';
import { JobType, CreateAndEditJobType, createAndEditJobSchema } from
'./types';
import { redirect } from 'next/navigation';
import { Prisma } from '@prisma/client';
import dayjs from 'dayjs';

function authenticateAndRedirect(): string {
  const { userId } = auth();
  if (!userId) {
    redirect('/');
  }
  return userId;
}

export async function createJobAction(
  values: CreateAndEditJobType
): Promise<JobType | null> {
  // await new Promise((resolve) => setTimeout(resolve, 3000));
  const userId = authenticateAndRedirect();
  try {
    createAndEditJobSchema.parse(values);
    const job: JobType = await prisma.job.create({
      data: {
        ...values,

        clerkId: userId,
      },
    });
    return job;
  } catch (error) {
    console.error(error);
    return null;
  }
}
```

Explore Toast Component

- install

```
npx shadcn-ui@latest add toast
```

docs

Challenge - Add React Query and Toaster

- add React Query and Toaster to providers.tsx
- wrap Home Page in React Query

Add React Query and Toaster

- app/providers.tsx

```
'use client';

import { ThemeProvider } from '@/components/theme-provider';
import { useState } from 'react';
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
import { ReactQueryDevtools } from '@tanstack/react-query-devtools';
import { Toaster } from '@/components/ui/toaster';

const Providers = ({ children }: { children: React.ReactNode }) => {
  const [queryClient] = useState(
    () =>
      new QueryClient({
        defaultOptions: {
          queries: {
            // With SSR, we usually want to set some default staleTime
            // above 0 to avoid refetching immediately on the client
            staleTime: 60 * 1000 * 5,
          },
        },
      })
  );

  return (
    <ThemeProvider
      attribute='class'
      defaultTheme='system'
      enableSystem
      disableTransitionOnChange
    >
      <Toaster />
      <QueryClientProvider client={queryClient}>
        {children}
        <ReactQueryDevtools initialIsOpen={false} />
      </QueryClientProvider>
```

```
    </ThemeProvider>
  );
};
export default Providers;
```

- add-job/page

```
import CreateJobForm from '@/components/CreateJobForm';
import {
  dehydrate,
  HydrationBoundary,
  QueryClient,
} from '@tanstack/react-query';

function AddJobPage() {
  const queryClient = new QueryClient();
  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <CreateJobForm />
    </HydrationBoundary>
  );
}
export default AddJobPage;
```

CreateJobForm Complete

```
// imports
import { useMutation, useQueryClient } from '@tanstack/react-query';
import { createJobAction } from '@/utils/actions';
import { useToast } from '@/components/ui/use-toast';
import { useRouter } from 'next/navigation';

// logic
const queryClient = useQueryClient();
const { toast } = useToast();
const router = useRouter();
const { mutate, isPending } = useMutation({
  mutationFn: (values: CreateAndEditJobType) => createJobAction(values),
  onSuccess: (data) => {
    if (!data) {
      toast({
        description: 'there was an error',
      });
      return;
    }
    toast({ description: 'job created' });
    queryClient.invalidateQueries({ queryKey: ['jobs'] });
    queryClient.invalidateQueries({ queryKey: ['stats'] });
    queryClient.invalidateQueries({ queryKey: ['charts'] });
```

```
    router.push('/jobs');
    // form.reset();
  },
});

function onSubmit(values: CreateAndEditJobType) {
  mutate(values);
}
// return
<Button type='submit' className='self-end capitalize' disabled=
{isPending}>
  {isPending ? 'loading...' : 'create job'}
</Button>;
```

Challenge - GetAllJobsAction

1. **Define the getAllJobsAction function**
    - Define an asynchronous function named `getAllJobsAction` that takes an object as a parameter.
    - This object should have `search`, `jobStatus`, `page`, and `limit` properties.
    - The `page` and `limit` properties should have default values of 1 and 10, respectively.
    - This function should return a Promise that resolves to an object with `jobs`, `count`, `page`, and `totalPages` properties.
2. **Authenticate the user**
    - Inside the `getAllJobsAction` function, call `authenticateAndRedirect` and store its return value in `userId`.
3. **Define the whereClause object**
    - Define a `whereClause` object with a `clerkId` property that has `userId` as its value.
4. **Modify the whereClause object based on search and jobStatus**
    - If `search` is defined, add an `OR` property to `whereClause` that is an array of objects.
    - Each object in the `OR` array should represent a condition where a field contains the search string.
    - If `jobStatus` is defined and not equal to 'all', add a `status` property to `whereClause` that has `jobStatus` as its value.
5. **Fetch jobs from the database**
    - Use the `prisma.job.findMany` method to fetch jobs from the database.
    - Pass an object to this method with `where` and `orderBy` properties.
    - The `where` property should have `whereClause` as its value.
    - The `orderBy` property should be an object with a `createdAt` property that has 'desc' as its value.
    - Store the return value of this method in `jobs`.

6. **Handle errors**
    - Wrap the database operation in a try-catch block.
    - If an error occurs, log the error to the console and return an object with `jobs`, `count`, `page`, and `totalPages` properties, all of which have 0 or [] as their values.
7. **Return the jobs**
    - After the try-catch block, return an object with `jobs`, `count`, `page`, and `totalPages` properties.
8. **Export the getAllJobsAction function**
    - Export `getAllJobsAction` so it can be used in other parts of your application.

GetAllJobsAction

- actions

```
type GetAllJobsActionTypes = {
  search?: string;
  jobStatus?: string;
  page?: number;
  limit?: number;
};

export async function getAllJobsAction({
  search,
  jobStatus,
  page = 1,
  limit = 10,
}: GetAllJobsActionTypes): Promise<{
  jobs: JobType[];
  count: number;
  page: number;
  totalPages: number;
}> {
  const userId = authenticateAndRedirect();

  try {
    let whereClause: Prisma.JobWhereInput = {
      clerkId: userId,
    };
    if (search) {
      whereClause = {
        ...whereClause,
        OR: [
          {
            position: {
              contains: search,
            },
          },
```

```
            {
              company: {
                contains: search,
              },
            },
          ],
        };
      }
      if (jobStatus && jobStatus !== 'all') {
        whereClause = {
          ...whereClause,
          status: jobStatus,
        };
      }

      const jobs: JobType[] = await prisma.job.findMany({
        where: whereClause,
        orderBy: {
          createdAt: 'desc',
        },
      });

      return { jobs, count: 0, page: 1, totalPages: 0 };
    } catch (error) {
      console.error(error);
      return { jobs: [], count: 0, page: 1, totalPages: 0 };
    }
  }
```

Challenge - Jobs Page

- create SearchForm, JobsList, JobCard, JobInfo, DeleteJobBtn components
- setup jobs/loading.tsx
- wrap jobs/page in React Query and pre-fetch getAllJobsAction

Jobs Page

- create SearchForm, JobsList, JobCard, JobInfo, DeleteJobBtn
- setup jobs/loading.tsx

```
function loading() {
  return <h2 className='text-xl font-medium capitalize'>loading...</h2>;
}
export default loading;
```

JobCard.tsx

```
import { JobType } from '@/utils/types';

function JobCard({ job }: { job: JobType }) {
  return <h1 className='text-3xl'>JobCard</h1>;
}
export default JobCard;
```

jobs/page.tsx

```
import JobsList from '@/components/JobsList';
import SearchForm from '@/components/SearchForm';
import {
  dehydrate,
  HydrationBoundary,
  QueryClient,
} from '@tanstack/react-query';
import { getAllJobsAction } from '@/utils/actions';

async function AllJobsPage() {
  const queryClient = new QueryClient();

  await queryClient.prefetchQuery({
    queryKey: ['jobs', '', 'all', 1],
    queryFn: () => getAllJobsAction({}),
  });
  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <SearchForm />
      <JobsList />
    </HydrationBoundary>
  );
}

export default AllJobsPage;
```

Challenge - SearchForm

1. **Import necessary libraries and components**
    - Import the `Input` and `Button` components from your UI library.
    - Import the `usePathname`, `useRouter`, and `useSearchParams` hooks from `next/navigation`.
    - Import the `Select`, `SelectContent`, `SelectItem`, `SelectTrigger`, and `SelectValue` components from your UI library.
    - Import the `JobStatus` type from your types file.
2. **Define the SearchContainer component**
    - Define a function component named `SearchContainer`.

3. **Use hooks to get necessary data**
    - Inside `SearchContainer`, use the `useSearchParams` hook to get the current search parameters.
    - Use the `get` method of the `searchParams` object to get the `search` and `jobStatus` parameters.
    - Use the `useRouter` hook to get the router object.
    - Use the `usePathname` hook to get the current pathname.
4. **Define the form submission handler**
    - Inside `SearchContainer`, define a function named `handleSubmit` for handling form submission.
    - This function should take an event object as its parameter.
    - Inside this function, prevent the default form submission behavior.
    - Create a new `URLSearchParams` object and a new `FormData` object.
    - Use the `get` method of the `formData` object to get the `search` and `jobStatus` form values.
    - Use the `set` method of the `params` object to set the `search` and `jobStatus` parameters.
    - Use the `push` method of the router object to navigate to the current pathname with the new search parameters.
5. **Create the form UI**
    - In the component's return statement, create the form UI using the form element.
    - Use the `Input` and `Select` components to create the form fields.
    - Use the `Button` component to create the submit button.
    - Pass the `handleSubmit` function as the `onSubmit` prop to the form element.
6. **Export the SearchContainer component**
    - After defining the `SearchContainer` component, export it so it can be used in other parts of your application.

SearchForm

```
'use client';
import { Input } from './ui/input';
import { usePathname, useRouter, useSearchParams } from 'next/navigation';
import { Button } from './ui/button';

import {
  Select,
  SelectContent,
  SelectItem,
  SelectTrigger,
  SelectValue,
} from '@/components/ui/select';
import { JobStatus } from '@/utils/types';

function SearchContainer() {
```

```
  // set default values
  const searchParams = useSearchParams();
  const search = searchParams.get('search') || '';
  const jobStatus = searchParams.get('jobStatus') || 'all';

  const router = useRouter();
  const pathname = usePathname();
  const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();

    const formData = new FormData(e.currentTarget);
    const search = formData.get('search') as string;
    const jobStatus = formData.get('jobStatus') as string;
    let params = new URLSearchParams();
    params.set('search', search);
    params.set('jobStatus', jobStatus);

    router.push(`${pathname}?${params.toString()}`);
  };

  return (
    <form
      className='bg-muted mb-16 p-8 grid sm:grid-cols-2 md:grid-cols-3
gap-4 rounded-lg'
      onSubmit={handleSubmit}
    >
      <Input
        type='text'
        placeholder='Search Jobs'
        name='search'
        defaultValue={search}
      />
      <Select defaultValue={jobStatus} name='jobStatus'>
        <SelectTrigger>
          <SelectValue />
        </SelectTrigger>
        <SelectContent>
          {['all', ...Object.values(JobStatus)].map((jobStatus) => {
            return (
              <SelectItem key={jobStatus} value={jobStatus}>
                {jobStatus}
              </SelectItem>
            );
          })}
        </SelectContent>
      </Select>
      <Button type='submit'>Search</Button>
    </form>
  );
}
export default SearchContainer;
```

Challenge - JobsList

1. **Import necessary libraries and modules**
    - Import the `useSearchParams` hook from `next/navigation`.
    - Import the `getAllJobsAction` function from your actions file.
    - Import the `useQuery` hook from `@tanstack/react-query`.
2. **Define the JobsList component**
    - Define a function component named `JobsList`.
3. **Use hooks to get necessary data**
    - Inside `JobsList`, use the `useSearchParams` hook to get the current search parameters.
    - Use the `get` method of the `searchParams` object to get the `search` and `jobStatus` parameters.
    - If `search` or `jobStatus` is null, default them to an empty string and 'all', respectively.
    - Use the `get` method of the `searchParams` object to get the `page` parameter.
    - If `page` is null, default it to 1.
4. **Fetch the jobs from the server**
    - Use the `useQuery` hook to fetch the jobs from the server.
    - Pass an object to this hook with `queryKey` and `queryFn` properties.
    - The `queryKey` property should be an array with 'jobs', `search`, `jobStatus`, and `pageNumber`.
    - The `queryFn` property should be a function that calls `getAllJobsAction` with an object that has `search`, `jobStatus`, and `page` properties.
    - Store the return value of this hook in `data` and `isPending`.
5. **Handle loading and empty states**
    - If `isPending` is true, return a `h2` element with 'Please Wait...' as its child.
    - If `jobs` is an empty array, return a `h2` element with 'No Jobs Found...' as its child.
6. **Export the JobsList component**
    - After defining the `JobsList` component, export it so it can be used in other parts of your application.

JobsList

```
'use client';
import JobCard from './JobCard';
import { useSearchParams } from 'next/navigation';
import { getAllJobsAction } from '@/utils/actions';
import { useQuery } from '@tanstack/react-query';

function JobsList() {
  const searchParams = useSearchParams();
```

```
  const search = searchParams.get('search') || '';
  const jobStatus = searchParams.get('jobStatus') || 'all';

  const pageNumber = Number(searchParams.get('page')) || 1;

  const { data, isPending } = useQuery({
    queryKey: ['jobs', search ?? '', jobStatus, pageNumber],
    queryFn: () => getAllJobsAction({ search, jobStatus, page: pageNumber
}),
  });
  const jobs = data?.jobs || [];

  if (isPending) return <h2 className='text-xl'>Please Wait...</h2>;

  if (jobs.length < 1) return <h2 className='text-xl'>No Jobs Found...
</h2>;
  return (
    <>
      {/*button container  */}
      <div className='grid md:grid-cols-2  gap-8'>
        {jobs.map((job) => {
          return <JobCard key={job.id} job={job} />;
        })}
      </div>
    </>
  );
}
export default JobsList;
```

Explore - shadcn/ui badge separator and card components

- install

```
npx shadcn-ui@latest add badge separator card
```

badge separator card

Challenge - JobCard

1. **Import necessary libraries and components**
    - Import the `JobType` type from your types file.
    - Import the `MapPin`, `Briefcase`, `CalendarDays`, and `RadioTower` components from `lucide-react`.
    - Import the `Link` component from `next/link`.
    - Import the `Card`, `CardContent`, `CardDescription`, `CardFooter`, `CardHeader`, and `CardTitle` components from your UI library.

- Import the `Separator`, `Button`, `Badge`, `JobInfo`, and `DeleteJobButton` components from your components directory.

2. **Define the JobCard component**

   - Define a function component named `JobCard` that takes an object as a prop.
   - This object should have a `job` property of type `JobType`.

3. **Convert the job's creation date to a locale string**

   - Inside `JobCard`, create a new Date object with `job.createdAt` as its argument.
   - Call the `toLocaleDateString` method on this object and store its return value in `date`.

4. **Create the component UI**

   - In the component's return statement, create the component UI using the `Card`, `CardHeader`, `CardTitle`, `CardDescription`, `Separator`, `CardContent`, `CardFooter`, `Button`, `Link`, and `DeleteJobButton` components.
   - Pass the `job.position` and `job.company` as the children of the `CardTitle` and `CardDescription` components, respectively.
   - Pass the `job.id` as the `href` prop to the `Link` component.
   - Pass the `date` as the child of the `CalendarDays` component.

5. **Export the JobCard component**

   - After defining the `JobCard` component, export it so it can be used in other parts of your application.

JobCard

JobCard

```
import { JobType } from '@/utils/types';
import { MapPin, Briefcase, CalendarDays, RadioTower } from 'lucide-
react';

import Link from 'next/link';
import {
  Card,
  CardContent,
  CardDescription,
  CardFooter,
  CardHeader,
  CardTitle,
} from '@/components/ui/card';
import { Separator } from './ui/separator';
import { Button } from './ui/button';
import { Badge } from './ui/badge';
import JobInfo from './JobInfo';
import DeleteJobButton from './DeleteJobButton';

function JobCard({ job }: { job: JobType }) {
```

```
  const date = new Date(job.createdAt).toLocaleDateString();
  return (
    <Card className='bg-muted'>
      <CardHeader>
        <CardTitle>{job.position}</CardTitle>
        <CardDescription>{job.company}</CardDescription>
      </CardHeader>
      <Separator />
      <CardContent>{/* card info */}</CardContent>
      <CardFooter className='flex gap-4'>
        <Button asChild size='sm'>
          <Link href={`/jobs/${job.id}`}>edit</Link>
        </Button>
        <DeleteJobButton />
      </CardFooter>
    </Card>
  );
}
export default JobCard;
```

Challenge - JobInfo

1. **Define the JobInfo component**
   - Define a function component named `JobInfo` that takes an object as a prop.
   - This object should have `icon` and `text` properties.
   - The `icon` property should be of type `React.ReactNode` and the `text` property should be of type `string`.
2. **Create the component UI**
   - In the component's return statement, create a `div` element with a `className` of 'flex gap-x-2 items-center'.
   - Inside this `div`, render the `icon` and `text` props.
3. **Export the JobInfo component**
   - After defining the `JobInfo` component, export it so it can be used in other parts of your application.
4. **Use the JobInfo component**
   - In the `CardContent` component, use the `JobInfo` component four times.
   - For each `JobInfo` component, pass an `icon` prop and a `text` prop.
   - The `icon` prop should be a `Briefcase`, `MapPin`, `CalendarDays`, or `RadioTower` component.
   - The `text` prop should be `job.mode`, `job.location`, `date`, or `job.status`.
   - Wrap the last `JobInfo` component in a `Badge` component with a `className` of 'w-32 justify-center'.

JobInfo

JobInfo.tsx

```
function JobInfo({ icon, text }: { icon: React.ReactNode; text: string })
{
  return (
    <div className='flex gap-x-2 items-center'>
      {icon}
      {text}
    </div>
  );
}
export default JobInfo;
```

JobCard.tsx

```
<CardContent className='mt-4 grid grid-cols-2 gap-4'>
  <JobInfo icon={<Briefcase />} text={job.mode} />
  <JobInfo icon={<MapPin />} text={job.location} />
  <JobInfo icon={<CalendarDays />} text={date} />
  <Badge className='w-32  justify-center'>
    <JobInfo icon={<RadioTower className='w-4 h-4' />} text={job.status}
/>
  </Badge>
</CardContent>
```

Challenge - DeleteJobAction

1. **Define the deleteJobAction function**
    - Define an asynchronous function named `deleteJobAction` that takes a string `id` as a parameter.
    - This function should return a Promise that resolves to a `JobType` object or null.
2. **Authenticate the user**
    - Inside the `deleteJobAction` function, call `authenticateAndRedirect` and store its return value in `userId`.
3. **Delete the job from the database**
    - Use the `prisma.job.delete` method to delete the job from the database.
    - Pass an object to this method with a `where` property.
    - The `where` property should be an object with `id` and `clerkId` properties.
    - The `id` property should have `id` as its value and the `clerkId` property should have `userId` as its value.
    - Store the return value of this method in `job`.
4. **Handle errors**

- Wrap the database operation in a try-catch block.
- If an error occurs, return null.

5. **Return the deleted job**
   - After the try-catch block, return `job`.
6. **Export the deleteJobAction function**
   - Export `deleteJobAction` so it can be used in other parts of your application.

DeleteJobAction

actions

```
export async function deleteJobAction(id: string): Promise<JobType | null>
{
  const userId = authenticateAndRedirect();

  try {
    const job: JobType = await prisma.job.delete({
      where: {
        id,
        clerkId: userId,
      },
    });
    return job;
  } catch (error) {
    return null;
  }
}
```

Challenge - DeleteJobButton

1. **Import necessary libraries and components**
   - Import the `Button`, `Badge`, `JobInfo`, and `useToast` components from your components directory.
   - Import the `useMutation` and `useQueryClient` hooks from `@tanstack/react-query`.
   - Import the `deleteJobAction` function from your actions file.
2. **Define the DeleteJobBtn component**
   - Define a function component named `DeleteJobBtn` that takes an object as a prop.
   - This object should have an `id` property of type string.
3. **Use hooks to get necessary data and functions**
   - Inside `DeleteJobBtn`, use the `useToast` hook to get the `toast` function.
   - Use the `useQueryClient` hook to get the `queryClient` object.
   - Use the `useMutation` hook to get the `mutate` function and `isPending` state.

- Pass an object to the `useMutation` hook with `mutationFn` and `onSuccess` properties.
- The `mutationFn` property should be a function that takes `id` as a parameter and calls `deleteJobAction` with `id`.
- The `onSuccess` property should be a function that takes `data` as a parameter and invalidates the `jobs`, `stats`, and `charts` queries if data is truthy. If data is falsy, it should call `toast` with an object that has a `description` property of 'there was an error'.

4. **Create the component UI**

- In the component's return statement, create the component UI using the `Button` component.
- Pass the `mutate` function as the `onClick` prop to the `Button` component.
- Pass `isPending` as the `loading` prop to the `Button` component.

5. **Export the DeleteJobBtn component**

- After defining the `DeleteJobBtn` component, export it so it can be used in other parts of your application.

DeleteJobButton

```
import { Button } from './ui/button';
import { useMutation, useQueryClient } from '@tanstack/react-query';
import { deleteJobAction } from '@/utils/actions';
import { useToast } from '@/components/ui/use-toast';

function DeleteJobBtn({ id }: { id: string }) {
  const { toast } = useToast();
  const queryClient = useQueryClient();
  const { mutate, isPending } = useMutation({
    mutationFn: (id: string) => deleteJobAction(id),
    onSuccess: (data) => {
      if (!data) {
        toast({
          description: 'there was an error',
        });
        return;
      }
      queryClient.invalidateQueries({ queryKey: ['jobs'] });
      queryClient.invalidateQueries({ queryKey: ['stats'] });
      queryClient.invalidateQueries({ queryKey: ['charts'] });

      toast({ description: 'job removed' });
    },
  });
  return (
    <Button
      size='sm'
      disabled={isPending}
      onClick={() => {
        mutate(id);
      }}
```

```
      >
        {isPending ? 'deleting...' : 'delete'}
      </Button>
    );
}
export default DeleteJobBtn;
```

Challenge - GetSingleJobAction

1. **Define the getSingleJobAction function**
    - Define an asynchronous function named `getSingleJobAction` that takes a string `id` as a parameter.
    - This function should return a Promise that resolves to a `JobType` object or null.
2. **Authenticate the user**
    - Inside the `getSingleJobAction` function, call `authenticateAndRedirect` and store its return value in `userId`.
3. **Fetch the job from the database**
    - Use the `prisma.job.findUnique` method to fetch the job from the database.
    - Pass an object to this method with a `where` property.
    - The `where` property should be an object with `id` and `clerkId` properties.
    - The `id` property should have `id` as its value and the `clerkId` property should have `userId` as its value.
    - Store the return value of this method in `job`.
4. **Handle errors**
    - Wrap the database operation in a try-catch block.
    - If an error occurs, set `job` to null.
5. **Redirect if the job is not found**
    - After the try-catch block, check if `job` is falsy.
    - If `job` is falsy, call `redirect` with '/jobs' as its argument.
6. **Return the fetched job**
    - After the if statement, return `job`.
7. **Export the getSingleJobAction function**
    - Export `getSingleJobAction` so it can be used in other parts of your application.

GetSingleJobAction

```
export async function getSingleJobAction(id: string): Promise<JobType |
null> {
```

```
  let job: JobType | null = null;
  const userId = authenticateAndRedirect();

  try {
    job = await prisma.job.findUnique({
      where: {
        id,
        clerkId: userId,
      },
    });
  } catch (error) {
    job = null;
  }
  if (!job) {
    redirect('/jobs');
  }
  return job;
}
```

Challenge - SingleJob Page

- create single job page (dynamic)
- create EditJobForm which accepts jobId props (string)

1. **Import necessary libraries and components**
    - Import the `EditJobForm` component from your components directory.
    - Import the `getSingleJobAction` function from your actions file.
    - Import the `dehydrate`, `HydrationBoundary`, and `QueryClient` components from `@tanstack/react-query`.
2. **Define the JobDetailPage component**
    - Define an asynchronous function component named `JobDetailPage` that takes an object as a prop.
    - This object should have a `params` property, which is also an object with an `id` property of type string.
3. **Create a new query client**
    - Inside `JobDetailPage`, create a new `QueryClient` instance and store it in `queryClient`.
4. **Prefetch the job data**
    - Use the `prefetchQuery` method of `queryClient` to prefetch the job data.
    - Pass an object to this method with `queryKey` and `queryFn` properties.
    - The `queryKey` property should be an array with 'job' and `params.id`.
    - The `queryFn` property should be a function that calls `getSingleJobAction` with `params.id`.
5. **Create the component UI**

- In the component's return statement, create the component UI using the `HydrationBoundary` and `EditJobForm` components.
- Pass the result of calling `dehydrate` with `queryClient` as the `state` prop to `HydrationBoundary`.
- Pass `params.id` as the `jobId` prop to `EditJobForm`.

6. **Export the JobDetailPage component**

- After defining the `JobDetailPage` component, export it so it can be used in other parts of your application.

SingleJob Page

jobs/[id]/page.tsx

```
import EditJobForm from '@/components/EditJobForm';
import { getSingleJobAction } from '@/utils/actions';

import {
  dehydrate,
  HydrationBoundary,
  QueryClient,
} from '@tanstack/react-query';

async function JobDetailPage({ params }: { params: { id: string } }) {
  const queryClient = new QueryClient();

  await queryClient.prefetchQuery({
    queryKey: ['job', params.id],
    queryFn: () => getSingleJobAction(params.id),
  });

  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <EditJobForm jobId={params.id} />
    </HydrationBoundary>
  );
}
export default JobDetailPage;
```

Challenge - UpdateJobAction

1. **Define the updateJobAction function**

- Define an asynchronous function named `updateJobAction` that takes a string `id` and an object `values` as parameters.
- The `values` parameter should be of type `CreateAndEditJobType`.
- This function should return a Promise that resolves to a `JobType` object or null.

2. **Authenticate the user**

- Inside the `updateJobAction` function, call `authenticateAndRedirect` and store its return value in `userId`.

3. **Update the job in the database**

   - Use the `prisma.job.update` method to update the job in the database.
   - Pass an object to this method with `where` and `data` properties.
   - The `where` property should be an object with `id` and `clerkId` properties.
   - The `id` property should have `id` as its value and the `clerkId` property should have `userId` as its value.
   - The `data` property should be an object that spreads `values`.
   - Store the return value of this method in `job`.

4. **Handle errors**

   - Wrap the database operation in a try-catch block.
   - If an error occurs, return null.

5. **Return the updated job**

   - After the try-catch block, return `job`.

6. **Export the updateJobAction function**

   - Export `updateJobAction` so it can be used in other parts of your application.

UpdateJobAction

```
export async function updateJobAction(
  id: string,
  values: CreateAndEditJobType
): Promise<JobType | null> {
  const userId = authenticateAndRedirect();

  try {
    const job: JobType = await prisma.job.update({
      where: {
        id,
        clerkId: userId,
      },
      data: {
        ...values,
      },
    });
    return job;
  } catch (error) {
    return null;
  }
}
```

Challenge - EditJobForm

1. **Import necessary libraries and components**
    - Import `zodResolver` from `@hookform/resolvers/zod`.
    - Import `useForm` from `react-hook-form`.
    - Import `JobStatus`, `JobMode`, `createAndEditJobSchema`, and `CreateAndEditJobType` from your types file.
    - Import `Button` from your UI components directory.
    - Import `Form` from your UI components directory.
    - Import `CustomFormField` and `CustomFormSelect` from your local `FormComponents` file.
    - Import `useMutation`, `useQueryClient`, and `useQuery` from `react-query`.
    - Import `createJobAction`, `getSingleJobAction`, and `updateJobAction` from your actions file.
    - Import `useToast` from your UI components directory.
    - Import `useRouter` from `next/router`.
2. **Define the EditJobForm component**
    - Define a function component named `EditJobForm` that takes an object as a prop.
    - This object should have a `jobId` property of type string.
3. **Use hooks to get necessary data and functions**
    - Inside `EditJobForm`, use the `useQueryClient` hook to get the `queryClient` object.
    - Use the `useToast` hook to get the `toast` function.
    - Use the `useRouter` hook to get the router object.
    - Use the `useQuery` hook to fetch the job data.
    - Use the `useMutation` hook to get the `mutate` function and `isPending` state.
4. **Use the useForm hook to get form functions**
    - Use the `useForm` hook to get the form object.
    - Pass an object to this hook with `resolver` and `defaultValues` properties.
5. **Define the submit handler**
    - Define a function `onSubmit` that calls `mutate` with values.
6. **Create the component UI**
    - In the component's return statement, create the component UI using the `Form`, `CustomFormField`, `CustomFormSelect`, and `Button` components.
7. **Export the EditJobForm component**
    - After defining the `EditJobForm` component, export it so it can be used in other parts of your application.

EditJobForm

```
'use client';

import { zodResolver } from '@hookform/resolvers/zod';
import { useForm } from 'react-hook-form';

import {
  JobStatus,
  JobMode,
  createAndEditJobSchema,
  CreateAndEditJobType,
} from '@/utils/types';

import { Button } from '@/components/ui/button';
import { Form } from '@/components/ui/form';

import { CustomFormField, CustomFormSelect } from './FormComponents';
import { useMutation, useQueryClient, useQuery } from '@tanstack/react-
query';
import { getSingleJobAction, updateJobAction } from '@/utils/actions';
import { useToast } from '@/components/ui/use-toast';
import { useRouter } from 'next/navigation';
function EditJobForm({ jobId }: { jobId: string }) {
  const queryClient = useQueryClient();
  const { toast } = useToast();
  const router = useRouter();

  const { data } = useQuery({
    queryKey: ['job', jobId],
    queryFn: () => getSingleJobAction(jobId),
  });

  const { mutate, isPending } = useMutation({
    mutationFn: (values: CreateAndEditJobType) =>
      updateJobAction(jobId, values),
    onSuccess: (data) => {
      if (!data) {
        toast({
          description: 'there was an error',
        });
        return;
      }
      toast({ description: 'job updated' });
      queryClient.invalidateQueries({ queryKey: ['jobs'] });
      queryClient.invalidateQueries({ queryKey: ['job', jobId] });
      queryClient.invalidateQueries({ queryKey: ['stats'] });
      router.push('/jobs');
      // form.reset();
    },
  });

  // 1. Define your form.
  const form = useForm<CreateAndEditJobType>({
    resolver: zodResolver(createAndEditJobSchema),
```

```
    defaultValues: {
      position: data?.position || '',
      company: data?.company || '',
      location: data?.location || '',
      status: (data?.status as JobStatus) || JobStatus.Pending,
      mode: (data?.mode as JobMode) || JobMode.FullTime,
    },
  });

  // 2. Define a submit handler.
  function onSubmit(values: CreateAndEditJobType) {
    // Do something with the form values.
    // ✅ This will be type-safe and validated.
    mutate(values);
  }

  return (
    <Form {...form}>
      <form
        onSubmit={form.handleSubmit(onSubmit)}
        className='bg-muted p-8 rounded'
      >
        <h2 className='capitalize font-semibold text-4xl mb-6'>edit
job</h2>
        <div className='grid gap-4 md:grid-cols-2 lg:grid-cols-3 items-
start'>
          {/* position */}
          <CustomFormField name='position' control={form.control} />
          {/* company */}
          <CustomFormField name='company' control={form.control} />
          {/* location */}
          <CustomFormField name='location' control={form.control} />

          {/* job status */}
          <CustomFormSelect
            name='status'
            control={form.control}
            labelText='job status'
            items={Object.values(JobStatus)}
          />
          {/* job  type */}
          <CustomFormSelect
            name='mode'
            control={form.control}
            labelText='job mode'
            items={Object.values(JobMode)}
          />

          <Button
            type='submit'
            className='self-end capitalize'
            disabled={isPending}
          >
            {isPending ? 'updating...' : 'edit job'}
```

```
            </Button>
          </div>
        </form>
      </Form>
    );
}
export default EditJobForm;
```

Seed Database

- create fake data in Mockaroo docs
- copy from assets or final project
- log user id
- create seed.js
- run "node prisma/seed"

```
const { PrismaClient } = require('@prisma/client');
const data = require('./mock-data.json');
const prisma = new PrismaClient();

async function main() {
  const clerkId = 'clerkUserId';
  const jobs = data.map((job) => {
    return {
      ...job,
      clerkId,
    };
  });
  for (const job of jobs) {
    await prisma.job.create({
      data: job,
    });
  }
}
main()
  .then(async () => {
    await prisma.$disconnect();
  })
  .catch(async (e) => {
    console.error(e);
    await prisma.$disconnect();
    process.exit(1);
  });
```

Challenge - GetStatsAction

1. **Define the getStatsAction function**
    - Define an asynchronous function named `getStatsAction`.

- This function should return a Promise that resolves to an object with `pending`, `interview`, and `declined` properties, all of type number.

2. **Authenticate the user**
   - Inside the `getStatsAction` function, call `authenticateAndRedirect` and store its return value in `userId`.
3. **Fetch the job stats from the database**
   - Use the `prisma.job.groupBy` method to fetch the job stats from the database.
   - Pass an object to this method with `by`, `_count`, and `where` properties.
   - The `by` property should be an array with 'status'.
   - The `_count` property should be an object with `status` set to true.
   - The `where` property should be an object with `clerkId` set to `userId`.
   - Store the return value of this method in `stats`.
4. **Convert the stats array to an object**
   - Use the `Array.prototype.reduce` method to convert `stats` to an object and store it in `statsObject`.
   - The initial value of the accumulator should be an empty object.
   - In each iteration, set the property of the accumulator object with the key of `curr.status` to `curr._count.status`.
5. **Create the default stats object**
   - Create an object `defaultStats` with `pending`, `declined`, and `interview` properties all set to 0.
   - Use the spread operator to add the properties of `statsObject` to `defaultStats`.
6. **Handle errors**
   - Wrap the database operation and the stats conversion in a try-catch block.
   - If an error occurs, call `redirect` with '/jobs'.
7. **Return the stats object**
   - After the try-catch block, return `defaultStats`.
8. **Export the getStatsAction function**
   - Export `getStatsAction` so it can be used in other parts of your application.

GetStatsAction

```
export async function getStatsAction(): Promise<{
  pending: number;
  interview: number;
  declined: number;
}> {
  const userId = authenticateAndRedirect();
```

```
  try {
    const stats = await prisma.job.groupBy({
      where: {
        clerkId: userId,
      },
      by: ['status'],
      _count: {
        status: true,
      },
    });
    const statsObject = stats.reduce((acc, curr) => {
      acc[curr.status] = curr._count.status;
      return acc;
    }, {} as Record<string, number>);

    const defaultStats = {
      pending: 0,
      declined: 0,
      interview: 0,
      ...statsObject,
    };
    return defaultStats;
  } catch (error) {
    redirect('/jobs');
  }
}
```

Challenge - GetChartsAction

1. **Define the getChartsDataAction function**
    - Define an asynchronous function named `getChartsDataAction`.
    - This function should return a Promise that resolves to an array of objects, each with `date` and `count` properties.
2. **Authenticate the user**
    - Inside the `getChartsDataAction` function, call `authenticateAndRedirect` and store its return value in `userId`.
3. **Calculate the date six months ago**
    - Use `dayjs` to get the current date, subtract 6 months from it, and convert it to a JavaScript Date object. Store this value in `sixMonthsAgo`.
4. **Fetch the jobs from the database**
    - Use the `prisma.job.findMany` method to fetch the jobs from the database.
    - Pass an object to this method with `where` and `orderBy` properties.
    - The `where` property should be an object with `clerkId` and `createdAt` properties.
    - The `clerkId` property should have `userId` as its value.

- The `createdAt` property should be an object with `gte` set to `sixMonthsAgo`.
- The `orderBy` property should be an object with `createdAt` set to 'asc'.
- Store the return value of this method in `jobs`.

5. **Calculate the number of applications per month**

   - Use the `Array.prototype.reduce` method to calculate the number of applications per month and store it in `applicationsPerMonth`.
   - In each iteration, format the `createdAt` property of the current job to 'MMM YY' and store it in `date`.
   - Find an entry in the accumulator with `date` equal to `date` and store it in `existingEntry`.
   - If `existingEntry` exists, increment its `count` property by 1.
   - If `existingEntry` does not exist, push a new object to the accumulator with `date` and `count` properties.

6. **Handle errors**

   - Wrap the database operation and the applications per month calculation in a try-catch block.
   - If an error occurs, call `redirect` with '/jobs'.

7. **Return the applications per month**

   - After the try-catch block, return `applicationsPerMonth`.

8. **Export the getChartsDataAction function**

   - Export `getChartsDataAction` so it can be used in other parts of your application.

GetChartsAction

```
export async function getChartsDataAction(): Promise<
  Array<{ date: string; count: number }>
> {
  const userId = authenticateAndRedirect();
  const sixMonthsAgo = dayjs().subtract(6, 'month').toDate();
  try {
    const jobs = await prisma.job.findMany({
      where: {
        clerkId: userId,
        createdAt: {
          gte: sixMonthsAgo,
        },
      },
      orderBy: {
        createdAt: 'asc',
      },
    });

    let applicationsPerMonth = jobs.reduce((acc, job) => {
      const date = dayjs(job.createdAt).format('MMM YY');

      const existingEntry = acc.find((entry) => entry.date === date);
```

```
      if (existingEntry) {
        existingEntry.count += 1;
      } else {
        acc.push({ date, count: 1 });
      }

      return acc;
    }, [] as Array<{ date: string; count: number }>);

    return applicationsPerMonth;
  } catch (error) {
    redirect('/jobs');
  }
}
```

Challenge - Stats Page

- create StatsContainer and ChartsContainer components
- create loading in stats
- wrap stats page in React Query and pre-fetch

1. **Import necessary libraries and components**
    - Import `ChartsContainer` and `StatsContainer` from your components directory.
    - Import `getChartsDataAction` and `getStatsAction` from your actions file.
    - Import `dehydrate`, `HydrationBoundary`, and `QueryClient` from `@tanstack/react-query`.
2. **Define the StatsPage component**
    - Define an asynchronous function component named `StatsPage`.
3. **Initialize the query client**
    - Inside `StatsPage`, create a new instance of `QueryClient` and store it in `queryClient`.
4. **Prefetch the stats and charts data**
    - Use the `queryClient.prefetchQuery` method to prefetch the stats and charts data.
    - Pass an object to this method with `queryKey` and `queryFn` properties.
    - The `queryKey` property should be an array with 'stats' or 'charts'.
    - The `queryFn` property should be a function that calls `getStatsAction` or `getChartsDataAction`.
5. **Create the component UI**
    - In the component's return statement, create the component UI using the `HydrationBoundary`, `StatsContainer`, and `ChartsContainer` components.
    - Pass the result of calling `dehydrate` with `queryClient` as the `state` prop to `HydrationBoundary`.

6. **Export the StatsPage component**
    - After defining the `StatsPage` component, export it so it can be used in other parts of your application.

Stats Page

- create StatsContainer and ChartsContainer components

```
import ChartsContainer from '@/components/ChartsContainer';
import StatsContainer from '@/components/StatsContainer';
import { getChartsDataAction, getStatsAction } from '@/utils/actions';
import {
  dehydrate,
  HydrationBoundary,
  QueryClient,
} from '@tanstack/react-query';

async function StatsPage() {
  const queryClient = new QueryClient();

  await queryClient.prefetchQuery({
    queryKey: ['stats'],
    queryFn: () => getStatsAction(),
  });
  await queryClient.prefetchQuery({
    queryKey: ['charts'],
    queryFn: () => getChartsDataAction(),
  });
  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <StatsContainer />
      <ChartsContainer />
    </HydrationBoundary>
  );
}
export default StatsPage;
```

Challenge - StatsCard

- create StatsCard component

1. **Import necessary libraries and components for StatsCards**
    - Import `Card`, `CardDescription`, `CardHeader`, and `CardTitle` from your UI components directory.
2. **Define the StatsCards component**
    - Define a function component named `StatsCards` that takes `title` and `value` as props.

- In the component's return statement, create the component UI using the `Card`, `CardHeader`, `CardTitle`, and `CardDescription` components.
- The `Card` component should have a `CardHeader` child.
- The `CardHeader` component should have `CardTitle` and `CardDescription` children.
- The `CardTitle` component should display the `title` prop.
- The `CardDescription` component should display the `value` prop.

3. **Export the StatsCards component**

- After defining the `StatsCards` component, export it so it can be used in other parts of your application.

StatsCard

```
import {
  Card,
  CardDescription,
  CardHeader,
  CardTitle,
} from '@/components/ui/card';

type StatsCardsProps = {
  title: string;
  value: number;
};

function StatsCards({ title, value }: StatsCardsProps) {
  return (
    <Card className='bg-muted'>
      <CardHeader className='flex flex-row justify-between items-center'>
        <CardTitle className='capitalize'>{title}</CardTitle>
        <CardDescription className='text-4xl font-extrabold text-primary
mt-[0px!important]'>
          {value}
        </CardDescription>
      </CardHeader>
    </Card>
  );
}

export default StatsCards;
```

Challenge - StatsContainer

1. **Import necessary libraries and components**

- Import `useQuery` from the `@tanstack/react-query` library.
- Import `getStatsAction` from your actions file.
- Import `StatsCard` and `StatsLoadingCard` from your components directory.

2. **Define the StatsContainer component**
    - Define a function component named `StatsContainer`.
3. **Use the useQuery hook**
    - Inside `StatsContainer`, call the `useQuery` hook and destructure `data` and `isPending` from its return value.
    - Pass an object to `useQuery` with `queryKey` and `queryFn` properties.
    - The `queryKey` property should be an array with 'stats'.
    - The `queryFn` property should be a function that calls `getStatsAction`.
4. **Handle the data state**
    - After the loading state check, return a `div` element with three `StatsCard` children.
    - Each `StatsCard` should have `title` and `value` props.
    - The `title` prop should be a string that describes the data.
    - The `value` prop should be a value from the data object or 0 if the value is undefined.
5. **Export the StatsContainer component**
    - After defining the `StatsContainer` component, export it so it can be used in other parts of your application.

StatsContainer

```
'use client';
import { useQuery } from '@tanstack/react-query';
import { getStatsAction } from '@/utils/actions';
import StatsCard from './StatsCard';

function StatsContainer() {
  const { data } = useQuery({
    queryKey: ['stats'],
    queryFn: () => getStatsAction(),
  });

  return (
    <div className='grid md:grid-cols-2 gap-4 lg:grid-cols-3'>
      <StatsCard title='pending jobs' value={data?.pending || 0} />
      <StatsCard title='interviews set' value={data?.interview || 0} />
      <StatsCard title='jobs declined' value={data?.declined || 0} />
    </div>
  );
}
export default StatsContainer;
```

Explore - Shadcn/ui Skeleton component

- install

```
npx shadcn-ui@latest add skeleton
```

docs

StatsLoadingCard

StatsCard.tsx

```
export function StatsLoadingCard() {
  return (
    <Card className='w-[330px] h-[88px]'>
      <CardHeader className='flex flex-row justify-between items-center'>
        <div className='flex items-center space-x-4'>
          <Skeleton className='h-12 w-12 rounded-full' />
          <div className='space-y-2'>
            <Skeleton className='h-4 w-[150px]' />
            <Skeleton className='h-4 w-[100px]' />
          </div>
        </div>
      </CardHeader>
    </Card>
  );
}
```

Loading

stats/loading.tsx

```
import { StatsLoadingCard } from '@/components/StatsCard';
function loading() {
  return (
    <div className='grid md:grid-cols-2 gap-4 lg:grid-cols-3'>
      <StatsLoadingCard />
      <StatsLoadingCard />
      <StatsLoadingCard />
    </div>
  );
}
export default loading;
```

jobs/loading.tsx

```
import { Skeleton } from '@/components/ui/skeleton';

function loading() {
```

```
  return (
    <div className='p-8 grid sm:grid-cols-2 md:grid-cols-3  gap-4 rounded-
lg border'>
      <Skeleton className='h-10' />
      <Skeleton className='h-10 ' />
      <Skeleton className='h-10 ' />
    </div>
  );
}
export default loading;
```

Explore Re-charts Library

docs

Challenge - ChartsContainer

1. **Import necessary libraries and components**
    - Import `useQuery` from the react-query library.
    - Import `ResponsiveContainer`, `BarChart`, `CartesianGrid`, `XAxis`, `YAxis`, `Tooltip`, and `Bar` from recharts, a composable charting library built on React components.
2. **Define the ChartsContainer component**
    - Define a function component named `ChartsContainer`.
3. **Use the useQuery hook**
    - Inside `ChartsContainer`, call the `useQuery` hook and destructure `data`, `isPending` from its return value.
    - Pass an object to `useQuery` with `queryKey` and `queryFn` properties.
    - The `queryKey` property should be an array with a unique key.
    - The `queryFn` property should be a function that fetches the data you want to display in the chart.
4. **Handle the empty data state**
    - After the loading state check, add a conditional return statement that checks if `data` is null or `data.length` is less than 1.
    - If the condition is true, return null.
5. **Render the chart**
    - After the empty data state check, return a `section` element.
    - Inside the `section` element, render a `h1` element with a title for the chart.
    - After the `h1` element, render a `ResponsiveContainer` component.
    - Inside the `ResponsiveContainer` component, render a `BarChart` component.
    - Pass the `data` to the `BarChart` component.
    - Inside the `BarChart` component, render `CartesianGrid`, `XAxis`, `YAxis`, `Tooltip`, and `Bar` components.

- Pass appropriate props to each component.

6. **Export the ChartsContainer component**
   - After defining the `ChartsContainer` component, export it so it can be used in other parts of your application.

ChartsContainer

```
'use client';
import {
  BarChart,
  Bar,
  XAxis,
  YAxis,
  CartesianGrid,
  Tooltip,
  ResponsiveContainer,
} from 'recharts';

import { useQuery } from '@tanstack/react-query';
import { getChartsDataAction } from '@/utils/actions';
function ChartsContainer() {
  const { data } = useQuery({
    queryKey: ['charts'],
    queryFn: () => getChartsDataAction(),
  });

  if (!data || data.length < 1) return null;
  return (
    <section className='mt-16'>
      <h1 className='text-4xl font-semibold text-center'>
        Monthly Applications
      </h1>
      <ResponsiveContainer width='100%' height={300}>
        <BarChart data={data} margin={{ top: 50 }}>
          <CartesianGrid strokeDasharray='3 3' />
          <XAxis dataKey='date' />
          <YAxis allowDecimals={false} />
          <Tooltip />
          <Bar dataKey='count' fill='#2563eb' barSize={75} />
        </BarChart>
      </ResponsiveContainer>
    </section>
  );
}
export default ChartsContainer;
```

Refactor

- create ButtonContainer.tsx

```
export async function getAllJobsAction({
  search,
  jobStatus,
  page = 1,
  limit = 10,
}: GetAllJobsActionTypes): Promise<{
  jobs: JobType[];
  count: number;
  page: number;
  totalPages: number;
}> {
  const userId = authenticateAndRedirect();

  try {
    let whereClause: Prisma.JobWhereInput = {
      clerkId: userId,
    };
    if (search) {
      whereClause = {
        ...whereClause,
        OR: [
          {
            position: {
              contains: search,
            },
          },
          {
            company: {
              contains: search,
            },
          },
        ],
      };
    }
    if (jobStatus && jobStatus !== 'all') {
      whereClause = {
        ...whereClause,
        status: jobStatus,
      };
    }
    const skip = (page - 1) * limit;

    const jobs: JobType[] = await prisma.job.findMany({
      where: whereClause,
      skip,
      take: limit,
      orderBy: {
        createdAt: 'desc',
      },
    });
    const count: number = await prisma.job.count({
      where: whereClause,
```

```
    });
    const totalPages = Math.ceil(count / limit);
    return { jobs, count, page, totalPages };
  } catch (error) {
    console.error(error);
    return { jobs: [], count: 0, page: 1, totalPages: 0 };
  }
}
```

Refactor JobsList

```
const jobs = data?.jobs || [];
// add
  const count = data?.count || 0;
  const page = data?.page || 0;
  const totalPages = data?.totalPages || 0;

  if (isPending) return <h2 className='text-xl'>Please Wait...</h2>;

  if (jobs.length < 1) return <h2 className='text-xl'>No Jobs Found...
</h2>;
  return (
    <>
      <div className='flex items-center justify-between mb-8'>
        <h2 className='text-xl font-semibold capitalize '>
          {count} jobs found
        </h2>
        {totalPages < 2 ? null : (
          <ButtonContainer currentPage={page} totalPages={totalPages} />
        )}
      </div>
    <>
```

ButtonContainer

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

type ButtonContainerProps = {
  currentPage: number;
  totalPages: number;
};
import { Button } from './ui/button';
function ButtonContainer({ currentPage, totalPages }:
ButtonContainerProps) {
  const searchParams = useSearchParams();
  const router = useRouter();
  const pathname = usePathname();
```

```
  const pageButtons = Array.from({ length: totalPages }, (_, i) => i + 1);

  const handlePageChange = (page: number) => {
    const defaultParams = {
      search: searchParams.get('search') || '',
      jobStatus: searchParams.get('jobStatus') || '',
      page: String(page),
    };

    let params = new URLSearchParams(defaultParams);

    router.push(`${pathname}?${params.toString()}`);
  };
  return (
    <div className='flex  gap-x-2'>
      {pageButtons.map((page) => {
        return (
          <Button
            key={page}
            size='icon'
            variant={currentPage === page ? 'default' : 'outline'}
            onClick={() => handlePageChange(page)}
          >
            {page}
          </Button>
        );
      })}
    </div>
  );
}
export default ButtonContainer;
```

ComplexButtonContainer

```
'use client';
import { usePathname, useRouter, useSearchParams } from 'next/navigation';
import { ChevronLeft, ChevronRight } from 'lucide-react';

type ButtonContainerProps = {
  currentPage: number;
  totalPages: number;
};

type ButtonProps = {
  page: number;
  activeClass: boolean;
};

import { Button } from './ui/button';
function ButtonContainer({ currentPage, totalPages }:
ButtonContainerProps) {
```

```
  const searchParams = useSearchParams();
  const router = useRouter();
  const pathname = usePathname();

  const handlePageChange = (page: number) => {
    const defaultParams = {
      search: searchParams.get('search') || '',
      jobStatus: searchParams.get('jobStatus') || '',
      page: String(page),
    };

    let params = new URLSearchParams(defaultParams);

    router.push(`${pathname}?${params.toString()}`);
  };

  const addPageButton = ({ page, activeClass }: ButtonProps) => {
    return (
      <Button
        key={page}
        size='icon'
        variant={activeClass ? 'default' : 'outline'}
        onClick={() => handlePageChange(page)}
      >
        {page}
      </Button>
    );
  };

  const renderPageButtons = () => {
    const pageButtons = [];
    // first page
    pageButtons.push(
      addPageButton({ page: 1, activeClass: currentPage === 1 })
    );
    // dots

    if (currentPage > 3) {
      pageButtons.push(
        <Button size='icon' variant='outline' key='dots-1'>
          ...
        </Button>
      );
    }
    // one before current page
    if (currentPage !== 1 && currentPage !== 2) {
      pageButtons.push(
        addPageButton({
          page: currentPage - 1,
          activeClass: false,
        })
      );
    }
    // current page
```

```
      if (currentPage !== 1 && currentPage !== totalPages) {
        pageButtons.push(
          addPageButton({
            page: currentPage,
            activeClass: true,
          })
        );
      }
      // one after current page

      if (currentPage !== totalPages && currentPage !== totalPages - 1) {
        pageButtons.push(
          addPageButton({
            page: currentPage + 1,
            activeClass: false,
          })
        );
      }
      if (currentPage < totalPages - 2) {
        pageButtons.push(
          <Button size='icon' variant='outline' key='dots-2'>
            ...
          </Button>
        );
      }
      pageButtons.push(
        addPageButton({
          page: totalPages,
          activeClass: currentPage === totalPages,
        })
      );
      return pageButtons;
    };

    return (
      <div className='flex  gap-x-2'>
        {/* prev */}
        <Button
          className='flex items-center gap-x-2 '
          variant='outline'
          onClick={() => {
            let prevPage = currentPage - 1;
            if (prevPage < 1) prevPage = totalPages;
            handlePageChange(prevPage);
          }}
        >
          <ChevronLeft />
          prev
        </Button>
        {renderPageButtons()}
        {/* next */}
        <Button
          className='flex items-center gap-x-2 '
          onClick={() => {
```

```
            let nextPage = currentPage + 1;
            if (nextPage > totalPages) nextPage = 1;
            handlePageChange(nextPage);
          }}
          variant='outline'
        >
          next
          <ChevronRight />
        </Button>
      </div>
    );
}
export default ButtonContainer;
```

THE END