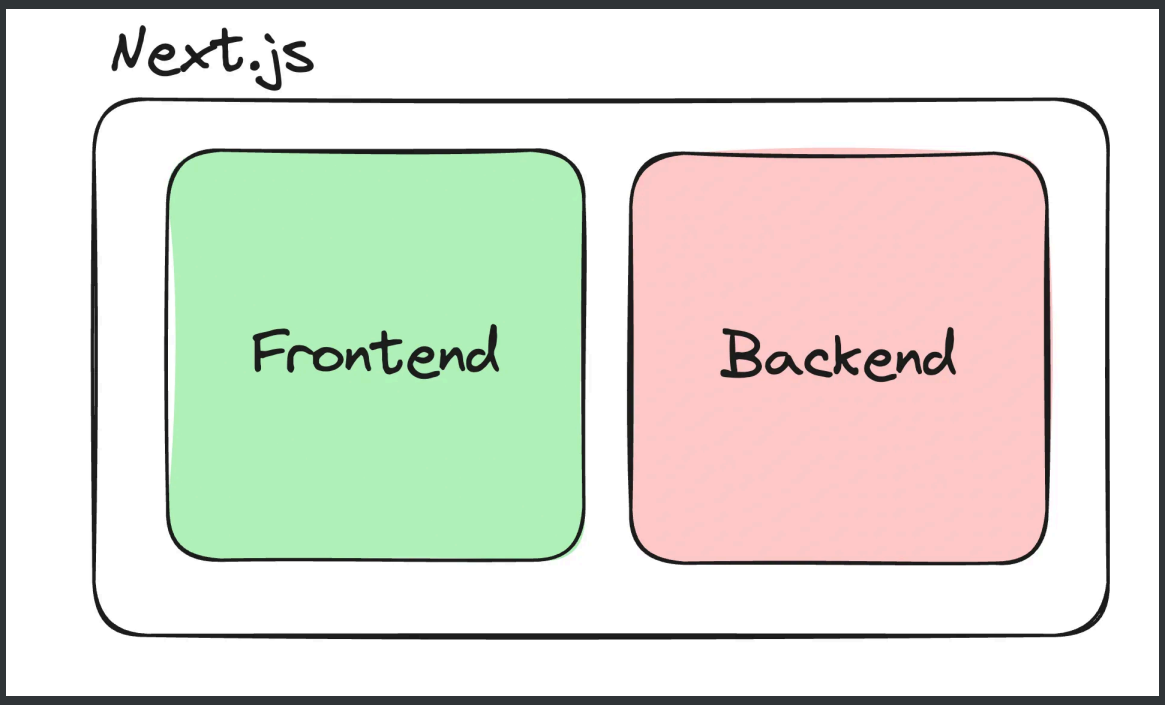
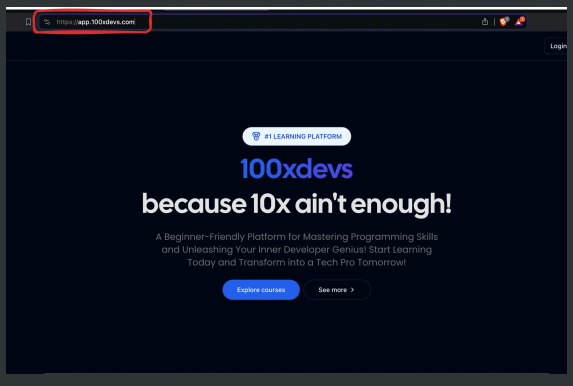
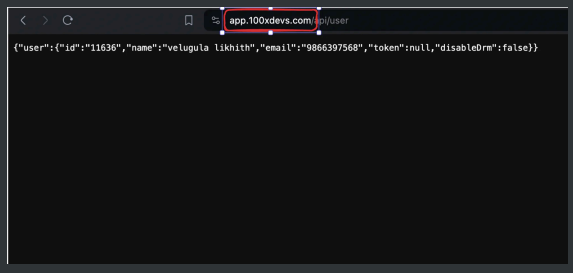
Step 1 - Backends in Next.js

Next.js is a full stack framework



This means the same process can handle frontend and backend code.





1. Single codebase for all your codebase
2. No cors issues, single domain name for your FE and BE
3. Ease of deployment, deploy a single codebase

Step 2 - Recap of Data fetching in React

Let's do a quick recap of how data fetching works in React

> 💡 We're not building backend yet
> Assume you already have this backend route - https://week-13-offline.kirattechnologies.workers.dev/api/v1/user/details

Code - https://github.com/100xdevs-cohort-2/week-14-2.1

Website - https://week-14-2-1.vercel.app/

User card website

Build a website that let's a user see their name and email from the given endpoint

UserCard component

Data fetching happens on the client

Step 3 - Data fetching in Next

Ref - https://nextjs.org/docs/app/building-your-application/data-fetching/fetching-caching-and-revalidating

> 💡 You can do the same thing as the last slide in Next.js, but then you lose the benefits of `server side rendering`

You should fetch the user details on the server side and `pre-render` the page before returning it to the user.

Let's try to build this

1. Initialise an empty next project

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

1. Install axios

```
npm i axios
```

1. Clean up `page.tsx` , `global.css`
2. In the root `page.tsx` , write a function to fetch the users details

```
async function getUserDetails() {
  const response = await axios.get("https://week-13-offline.kirattechnologies.w
    return response.data;
}
```

1. Convert the default export to be an async function (yes, nextjs now supports `async` components)

```
import axios from "axios";

async function getUserDetails() {
  const response = await axios.get("https://week-13-offline.kirattechnologies.w
    return response.data;
}

export default async function Home() {
  const userData = await getUserDetails();

  return (
```

```
    <div>
      {userData.email}
      {userData.name}
    </div>
  );
}
```

1. Check the network tab, make sure there is no waterfalling

1. Prettify the UI

```
import axios from "axios";

async function getUserDetails() {
  const response = await axios.get("https://week-13-offline.kirattechnologies.w
   return response.data;
}

export default async function Home() {
  const userData = await getUserDetails();

  return (
    <div className="flex flex-col justify-center h-screen">
      <div className="flex justify-center">
        <div className="border p-8 rounded">
          <div>
            Name: {userData?.name}
          </div>

          {userData?.email}
        </div>
      </div>
    </div>
```

```
  );
}
```


Good question to ask at this point - Where is the `loader` ?

Do we even need a `loader` ?

Step 4 - Loaders in Next

What if the `getUserDetails` call takes 5s to finish (lets say the backend is slow). You should show the user a `loader` during this time

loading.tsx file

Just like `page.tsx` and `layout.tsx` , you can define a `skeleton.tsx` file that will render until all the async operations finish

1. Create a `loading.tsx` file in the root folder
2. Add a custom loader inside

```
export default function Loading() {
  return <div className="flex flex-col justify-center h-screen">
    <div className="flex justify-center">
      Loading...
    </div>
```

Step 5 - Introducing api routes in Next.js

NextJS lets you write backend routes, just like express does.

This is why Next is considered to be a `full stack` framework.

The benefits of using NextJS for backend includes -

1. Code in a single repo
2. All standard things you get in a backend framework like express
3. Server components can directly talk to the backend

Step 6 - Let's move the backend into our own app

We want to introduce a route that returns `hardcoded` values for a user's details (email, name, id)

1. Introduce a new folder called `api`
2. Add a folder inside called `user`
3. Add a file inside called `route.ts`
4. Initialize a `GET` route inside it

```
export async function GET() {
  return Response.json({ username: "harkirat", email: "harkirat@gmail.com" })
}
```

1. Try replacing the api call in `page.tsx` to hit this URL

```
async function getUserDetails() {
  try {
    const response = await axios.get("http://localhost:3000/api/user")
     return response.data;
  } catch(e) {
    console.log(e);
```

```
}
}
```


This isn't the best way to fetch data from the backend. We'll make this better as time goes by

Step 7 - Frontend for Signing up

1. Create `app/signup/page.tsx`
2. Create a simple Page

```
import { Signup } from "@/components/Signup"

export default function() {
  return <Signup />
}
```

1. Create `components/Signup.tsx`

▼ Code

```
import axios from "axios";
import { ChangeEventHandler, useState } from "react";

export function Signup() {
    const [username, setUsername] = useState("");
    const [password, setPassword] = useState("");
```

```
    return <div className="h-screen flex justify-center flex-col">
      <div className="flex justify-center">
      <a href="#" className="block max-w-sm p-6 bg-white border border
          <div>
            <div className="px-10">
              <div className="text-3xl font-extrabold">
                Sign up
              </div>
            </div>
            <div className="pt-2">
              <LabelledInput onChange={(e) => {
                setUsername(e.target.value);
              }} label="Username" placeholder="harkirat@gmail.com" />
              <LabelledInput onChange={(e) => {
                setPassword(e.target.value)
              }} label="Password" type={"password"} placeholder="123456" /
              <button type="button" className="mt-8 w-full text-white bg-
            </div>
          </div>
        </a>
      </div>
    </div>

  }

  function LabelledInput({ label, placeholder, type, onChange }: LabelledInput
    return <div>
      <label className="block mb-2 text-sm text-black font-semibold pt-4'
      <input onChange={onChange} type={type || "text"} id="first_name" cla
    </div>
  }

  interface LabelledInputType {
    label: string;
    placeholder: string;
    type?: string;
    onChange: ChangeEventHandler<HTMLInputElement>
  }
```

1. Convert `components/Signup.tsx` to a client component

```
"use client"
```

1. Add a `onclick handler` that sends a `POST request` to `/user`

```
<button onClick={async () => {
  const response = await axios.post("http://localhost:3000/api/user", {
    username,
    password
  });

}} type="button" className="mt-8 w-full text-white bg-gray-800 focus:ring-4
```

1. Route the user to landing page if the signup succeeded
   Ref useRouter hook - https://nextjs.org/docs/app/building-your-application/routing/linking-and-navigating#userouter-hook

▼ Final signup.tsx

```
import axios from "axios";
import { useRouter } from "next/router";
import { ChangeEventHandler, useState } from "react";

export function Signup() {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");
  const router = useRouter();

  return <div className="h-screen flex justify-center flex-col">
    <div className="flex justify-center">
    <a href="#" className="block max-w-sm p-6 bg-white border border
        <div>
          <div className="px-10">
            <div className="text-3xl font-extrabold">
              Sign up
            </div>
          </div>
          <div className="pt-2">
            <LabelledInput onChange={(e) => {
              setUsername(e.target.value);
            }} label="Username" placeholder="harkirat@gmail.com" />
            <LabelledInput onChange={(e) => {
              setPassword(e.target.value)
            }} label="Password" type={"password"} placeholder="123456" />
            <button onClick={async () => {
```

```
                const response = await axios.post("http://localhost:3000/ap
                    username,
                    password
                });
                router.push("/")
            }} type="button" className="mt-8 w-full text-white bg-gray-
        </div>
    </div>
</a>
</div>
</div>

}

function LabelledInput({ label, placeholder, type, onChange }: LabelledInput
    return <div>
        <label className="block mb-2 text-sm text-black font-semibold pt-4"
        <input onChange={onChange} type={type || "text"} id="first_name" cla
    </div>
}

interface LabelledInputType {
    label: string;
    placeholder: string;
    type?: string;
    onChange: ChangeEventHandler<HTMLInputElement>
}
```

We still have to implement the backend route, lets do that in the next slide

Step 8 - Backend for signing up

Add a `POST` route that takes the users email and password and for now just returns them back

1. Navigate to `app/api/user/route.ts`
2. Initialize a POST endpoint inside it

```
import { NextRequest, NextResponse } from 'next/server';

export async function POST(req: NextRequest) {
    const body = await req.json();

    return NextResponse.json({ username: body.username, password: body.pas
}
```

Ref - https://nextjs.org/docs/app/api-reference/functions/next-response

Step 9 - Databases!

We have a bunch of dummy routes, we need to add a database layer to persist data.

Adding prisma to a Next.js project is straightforward.

> 💡 Please get a free Postgres DB from either neon or aiven

1. Install prisma

```
npm install prisma
```

1. Initialize prisma schema

```
npx prisma init
```

1. Create a simple user schema

```
model User {
  id       Int    @id @default(autoincrement())
  username String @unique
  password String
}
```

1. Replace `.env` with your own Postgres URL

```
DATABASE_URL="postgresql://johndoe:randompassword@localhost:5432/myc
```

1. Migrate the database

```
npx prisma migrate dev --name init_schema
```

1. Generate the client

```
npx prisma generate
```

1. Finish the `signup` route

```
export async function POST(req: NextRequest) {
    const body = await req.json();
    // should add zod validation here
    const user = await client.user.create({
        data: {
            username: body.username,
            password: body.password
        }
    });

    console.log(user.id);

    return NextResponse.json({ message: "Signed up" });
}
```

1. Update the `GET` endpoint

```
export async function GET() {
    const user = await client.user.findFirst({});
    return Response.json({ name: user?.username, email: user?.username })
}
```

💡 We're not doing any authentication yet. Which is why we're not returning a jwt (or setting a cookie) here

Step 10 - Better fetches

For the root page, we are fetching the details of the user by hitting an HTTP endpoint in `getUserDetails`

Current solution

```
import axios from "axios";

async function getUserDetails() {
 try {
  const response = await axios.get("http://localhost:3000/api/user")
   return response.data;
 } catch(e) {
  console.log(e);
 }
}

export default async function Home() {
 const userData = await getUserDetails();

 return (
  <div className="flex flex-col justify-center h-screen">
    <div className="flex justify-center">
      <div className="border p-8 rounded">
        <div>
          Name: {userData?.name}
        </div>

        {userData?.email}
      </div>
    </div>
  </div>
 );
}
```

`getUserDetails` runs on the server. This means you're sending a request from a server back to the server

Better solution

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

const client = new PrismaClient();

async function getUserDetails() {
 try {
  const user = await client.user.findFirst({});
   return {
   name: user?.username,
   email: user?.username
  }
 } catch(e) {
  console.log(e);
 }
}

export default async function Home() {
 const userData = await getUserDetails();

 return (
  <div className="flex flex-col justify-center h-screen">
    <div className="flex justify-center">
      <div className="border p-8 rounded">
        <div>
          Name: {userData?.name}
        </div>

        {userData?.email}
      </div>
    </div>
  </div>
```

```
  );
}
```


Step 11 - Singleton prisma client

Ref - https://www.prisma.io/docs/orm/more/help-and-troubleshooting/help-articles/nextjs-prisma-client-dev-practices

1. Create `db/index.ts`
2. Add a prisma client singleton inside it

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

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

declare global {
  var prisma: undefined | ReturnType<typeof prismaClientSingleton>
}

const prisma = globalThis.prisma ?? prismaClientSingleton()

export default prisma

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

1. Update imports of prisma everywhere

```
import client from "@/db"
```

Step 12 - Server Actions

Ref - https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions-and-mutations

Right now, we wrote an `API endpoint` that let's the user sign up

```
export async function POST(req: NextRequest) {
    const body = await req.json();
    // should add zod validation here
    const user = await client.user.create({
        data: {
            username: body.username,
            password: body.password
        }
    });

    console.log(user.id);

    return NextResponse.json({ message: "Signed up" });
}
```

What if you could do a simple function call (even on a `client component` that would run on the server?) (similar to `RPC`)

Under the hood, still an HTTP request would go out. But you would feel like you're making a function call

Steps to follow

1. Create `actions/user.ts` file (you can create it in a different folder)
2. Write a function that takes `username` and `password` as input and stores it in the DB

```
"use server"

import client from "@/db"

export async function signup(username: string, password: string) {
    // should add zod validation here
    const user = await client.user.create({
        data: {
            username: username,
            password: password
        }
    });

    console.log(user.id);

    return "Signed up!"
}
```

1. Update the `Signup.tsx` file to do the function call

```
import { signup } from "@/actions/user";;

...

<button onClick={async () => {
    const response = await signup(username, password);
    localStorage.setItem("token", response);
    router.push("/")
}} type="button" className="mt-8 w-full text-white bg-gray-800 focus:ring-4
```

Check the network tab

Benefits of server actions

1. Single function can be used in both client and server components
2. Gives you types of the function response on the frontend (very similar to trpc)
3. Can be integrated seamlessly with forms (ref https://www.youtube.com/watch?v=dDpZfOQBMaU)