Final App

Tutorial

Node Version

The minimum Node.js version has been bumped from 16.14 to 18.17, since 16.x has reached end-of-life.

Create Next App

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

Folder Structure

- app folder - where we will spend most of our time
  - setup routes, layouts, loading states, etc
- node_modules - project dependencies
- public - static assets
- .gitignore - sets which will be ignored by source control
- bunch of config files (will discuss as we go)
- in package.json look scripts
- 'npm run dev' to spin up the project on http://localhost:3000

```
npm run dev
```

Home Page

- page.js in the root of app folder
- represents root of our application '/' local domain or production domain
- react component (server component)
- bunch of css classes (will discuss Tailwind in few lectures)
- export component as default
- file name "page" has a special meaning
- snippets extension

  app/page.js

```
const HomePage = () => {
  return (
```

```
      <div>
        <h1 className='text-7xl'>HomePage</h1>
      </div>
    );
};
export default HomePage;
```

Create Pages in Next.js

- in the app folder create a folder with the page.js file
  - about/page.js
  - contact/page.js
- can have .js .jsx .tsx extension

app/about/page.js

```
const AboutPage = () => {
  return (
    <div>
      <h1 className='text-7xl'>AboutPage</h1>
    </div>
  );
};
export default AboutPage;
```

Link Component

- navigate around the project
- import Link from 'next/link' home page

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

const HomePage = () => {
  return (
    <div>
      <h1 className='text-7xl'>HomePage</h1>
      <Link href='/about' className='text-2xl'>
        about page
      </Link>
    </div>
  );
};
export default HomePage;
```

Nested Routes

- app/about/info/page.js

- if no page.js in a segment will result in 404

```
const AboutInfoPage = () => {
  return <h1 className='text-7xl'>AboutInfoPage</h1>;
};
export default AboutInfoPage;
```

Challenge

- create following pages :
  - client, drinks, prisma-example, query and tasks

Tailwind and DaisyUI

- both videos optional
- remove extra code in globals.css
- install daisyui
- install tailwindcss typography plugin
- configure tailwind

```
npm i -D daisyui@latest
npm i @tailwindcss/typography
```

tailwind.config.js

```
/** @type {import('tailwindcss').Config} */
module.exports = {
  ...
  plugins: [require('@tailwindcss/typography'), require('daisyui')]
};
```

Layouts and Templates

- layout.js
- template.js

  Layout is a component which wraps other pages and layouts. Allow to share UI. Even when the route changes, layout DOES NOT re-render. Can fetch data but can't pass it down to children. Templates are the same but they re-render.

- the top-most layout is called the Root Layout. This required layout is shared across all pages in an application. Root layouts must contain html and body tags.
- any route segment can optionally define its own Layout. These layouts will be shared across all pages in that segment.
- layouts in a route are nested by default. Each parent layout wraps child layouts below it using the React children prop.

```
import { Inter } from 'next/font/google';
import './globals.css';

const inter = Inter({ subsets: ['latin'] });

export const metadata = {
  title: 'Next.js Tutorial',
  description: 'Build awesome stuff with Next.js!',
};

export default function RootLayout({ children }) {
  return (
    <html lang='en'>
      <body className={inter.className}>{children}</body>
    </html>
  );
}
```

Challenge - Navbar

- in the root create components folder
- create Navbar.jsx component
- import Link component
- setup a list of links (to existing pages)
- iterate over the list
- render Navbar in the app/layout.js
- setup the same layout for all pages (hint: wrap children prop)

Setup Navbar

components/Navbar

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

const links = [
  { href: '/client', label: 'client' },
  { href: '/drinks', label: 'drinks' },
  { href: '/tasks', label: 'tasks' },
  { href: '/query', label: 'react-query' },
];
```

```
const Navbar = () => {
  return (
    <nav className='bg-base-300 py-4'>
      <div className='navbar px-8 max-w-6xl mx-auto flex-col sm:flex-row'>
        <li>
          <Link href='/' className='btn btn-primary'>
            Next.js
          </Link>
        </li>
        <ul className='menu menu-horizontal md:ml-8'>
          {links.map((link) => {
            return (
              <li key={link.href}>
                <Link href={link.href} className=' capitalize'>
                  {link.label}
                </Link>
              </li>
            );
          })}
        </ul>
      </div>
    </nav>
  );
};

export default Navbar;
```

app/layout

```
import { Inter } from 'next/font/google';
import './globals.css';

// alias
import Navbar from '@/components/Navbar';

const inter = Inter({ subsets: ['latin'] });

export const metadata = {
  title: 'Next.js Tutorial',
  description: 'Build awesome stuff with Next.js!',
};

export default function RootLayout({ children }) {
  return (
    <html lang='en' className='bg-base-200'>
      <body className={inter.className}>
        <Navbar />
        <main className='px-8 py-20 max-w-6xl mx-auto '>{children}</main>
      </body>
    </html>
```

```
    );
}
```

Server Components VS Client Components

- Server Components
- Client Components
- BY DEFAULT, NEXT.JS USES SERVER COMPONENTS !!!!
- To use Client Components, you can add the React "use client" directive

Server Components

Benefits :

- data fetching
- security
- caching
- bundle size

Data Fetching: Server Components allow you to move data fetching to the server, closer to your data source. This can improve performance by reducing time it takes to fetch data needed for rendering, and the amount of requests the client needs to make. Security: Server Components allow you to keep sensitive data and logic on the server, such as tokens and API keys, without the risk of exposing them to the client. Caching: By rendering on the server, the result can be cached and reused on subsequent requests and across users. This can improve performance and reduce cost by reducing the amount of rendering and data fetching done on each request. Bundle Sizes: Server Components allow you to keep large dependencies that previously would impact the client JavaScript bundle size on the server. This is beneficial for users with slower internet or less powerful devices, as the client does not have to download, parse and execute any JavaScript for Server Components. Initial Page Load and First Contentful Paint (FCP): On the server, we can generate HTML to allow users to view the page immediately, without waiting for the client to download, parse and execute the JavaScript needed to render the page. Search Engine Optimization and Social Network Shareability: The rendered HTML can be used by search engine bots to index your pages and social network bots to generate social card previews for your pages. Streaming: Server Components allow you to split the rendering work into chunks and stream them to the client as they become ready. This allows the user to see parts of the page earlier without having to wait for the entire page to be rendered on the server.

Client Components

Benefits :

- Interactivity: Client Components can use state, effects, and event listeners, meaning they can provide immediate feedback to the user and update the UI.
- Browser APIs: Client Components have access to browser APIs, like geolocation or localStorage, allowing you to build UI for specific use cases.

Challenge - Setup Counter

- setup our home page (no restrictions)
- setup a simple counter in app/client/page.js

Home Page

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

export default function Home() {
  return (
    <div>
      <h1 className='text-5xl mb-8 font-bold'>Next.js Tutorial</h1>
      <Link href='/client' className='btn btn-accent'>
        get started
      </Link>
    </div>
  );
}
```

Counter (Client Component)

```
// try without
'use client';
import { useState } from 'react';
const Client = () => {
  const [count, setCount] = useState(0);
  return (
    <div>
      <h1 className='text-7xl font-bold mb-4 '>{count}</h1>
      <button className='btn btn-primary' onClick={() => setCount(count + 1)}>
        increase
      </button>
    </div>
  );
};
export default Client;
```

Fetch Data in Server Components

- just add async and start using await 🚀🚀🚀
- the same for db
- Next.js extends the native Web fetch() API to allow each request on the server to set its own persistent caching semantics.

```
const url = 'someUrl';

const ServerComponent = async () => {
  const response = await fetch(url);
```

```
  const data = await response.json();
  console.log(data);
  return (
    <div>
      <h1 className='text-7xl'>DrinksPage</h1>
    </div>
  );
};
export default ServerComponent;
```

Challenge - Fetch Data in Drinks Page

- setup logic
- visit the page
- look for log in the terminal 😀

app/drinks/page.js

```
const url = 'https://www.thecocktaildb.com/api/json/v1/1/search.php?f=a';

const DrinksPage = async () => {
  const response = await fetch(url);
  const data = await response.json();
  console.log(data);
  return (
    <div>
      <h1 className='text-7xl'>DrinksPage</h1>
    </div>
  );
};
export default DrinksPage;
```

Loading Component

The special file loading.js helps you create meaningful Loading UI with React Suspense. With this convention, you can show an instant loading state from the server while the content of a route segment loads. The new content is automatically swapped in once rendering is complete.

- drinks/loading.js

```
const loading = () => {
  return <span className='loading'></span>;
};
export default loading;
```

- refactor drinks page

```
const fetchDrinks = async () => {
  // just for demo purposes
  await new Promise((resolve) => setTimeout(resolve, 1000));
  const response = await fetch(url);
  const data = await response.json();
  return data;
};

const DrinksPage = async () => {
  const data = await fetchDrinks();
  console.log(data);
  return (
    <div>
      <h1 className='text-7xl'>DrinksPage</h1>
    </div>
  );
};
export default DrinksPage;
```

Error Component

The error.js file convention allows you to gracefully handle unexpected runtime errors in nested routes.

- drinks/error.js
- 'use client'

```
'use client';
const error = () => {
  return <div>there was an error...</div>;
};
export default error;
```

- refactor drinks (optional)

```
// modify the url
const url = 'https://www.thecocktaildb.com/api/json/v1/1/search.phps?f=a';

const fetchDrinks = async () => {
  // just for demo purposes
  await new Promise((resolve) => setTimeout(resolve, 1000));
  const response = await fetch(url);
  // throw error
  if (!response.ok) {
    throw new Error('Failed to fetch drinks...');
  }
  const data = await response.json();
  return data;
};
```

- DON'T FORGET TO FIX THE URL 😜😜😜

Nested Layouts

- create app/drinks/layout.js
- UI will be applied to app/drinks - segment
- don't forget about the "children"
- we can fetch data in the layout but... at the moment can't pass data down to children (pages) 😣

layout.js

```
export default function DrinksLayout({ children }) {
  return (
    <div className='max-w-xl '>
      <div className='mockup-code mb-8'>
        <pre data-prefix='$'>
          <code>npx create-next-app@latest nextjs-tutorial</code>
        </pre>
      </div>
      {children}
    </div>
  );
}
```

Dynamic Routes

- app/drinks/[id]/page.js

```
const page = ({ params }) => {
  console.log(params);

  return (
    <div>
      <h1 className='text-7xl'>{params.id}</h1>
    </div>
  );
};
export default page;
```

Challenge - Drinks List

- render a list of drinks
- each item is a link to a drink[id] page

Drinks List

- create components/DrinksList.jsx
- render in drinks/page and pass the drinks down
- iterate over the drinks and return links

DrinksList.jsx

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

const DrinksList = ({ drinks }) => {
  return (
    <ul className='menu menu-vertical p1-0'>
      {drinks.map((drink) => (
        <li key={drink.idDrink}>
          <Link
            href={`/drinks/${drink.idDrink}`}
            className='text-xl font-medium'
          >
            {drink.strDrink}
          </Link>
        </li>
      ))}
    </ul>
  );
};
export default DrinksList;
```

app/drinks/page.js

```
const url = 'https://www.thecocktaildb.com/api/json/v1/1/search.php?f=a';
import DrinksList from '@/components/DrinksList';
const fetchDrinks = async () => {
  // just for demo purposes
  await new Promise((resolve) => setTimeout(resolve, 1000));
  const response = await fetch(url);

  if (!response.ok) {
    throw new Error('Failed to fetch drinks...');
  }
  const data = await response.json();
  return data;
};

const DrinksPage = async () => {
  const data = await fetchDrinks();

  return (
    <div>
      <DrinksList drinks={data.drinks} />
```

```
      </div>
    );
};
export default DrinksPage;
```

Challenge - Render Single Drink

- in drinks/[id], fetch and render drink title
- hint:params object

```
const url = 'https://www.thecocktaildb.com/api/json/v1/1/lookup.php?i=';
```

Render Single Drink

drinks/[id]/page.js

```
import Link from 'next/link';
const url = 'https://www.thecocktaildb.com/api/json/v1/1/lookup.php?i=';

const getSingleDrink = async (id) => {
  const res = await fetch(`${url}${id}`);

  if (!res.ok) {
    throw new Error('Failed to fetch drink...');
  }
  return res.json();
};

const SingleDrink = async ({ params }) => {
  const data = await getSingleDrink(params.id);
  const title = data?.drinks[0]?.strDrink;
  const imgSrc = data?.drinks[0]?.strDrinkThumb;
  return (
    <div>
      <Link href='/drinks' className='btn btn-primary mt-8 mb-12'>
        back to drinks
      </Link>
      <h1 className='text-4xl mb-8'>{title}</h1>
    </div>
  );
};
export default SingleDrink;
```

Next Image Component

- get random image from pexels site Random Drink

The Next.js Image component extends the HTML element with features for automatic image optimization:

- Size Optimization: Automatically serve correctly sized images for each device, using modern image formats like WebP and AVIF.
- Visual Stability: Prevent layout shift automatically when images are loading.
- Faster Page Loads: Images are only loaded when they enter the viewport using native browser lazy loading, with optional blur-up placeholders.
- Asset Flexibility: On-demand image resizing, even for images stored on remote servers

drinks[id]/page.js

```
import Link from 'next/link';
const url = 'https://www.thecocktaildb.com/api/json/v1/1/lookup.php?i=';

import drinkImg from './drink.jpg';
import Image from 'next/image';

const SingleDrink = async ({ params }) => {
  const data = await getSingleDrink(params.id);
  const title = data?.drinks[0]?.strDrink;
  const imgSrc = data?.drinks[0]?.strDrinkThumb;
  return (
    <div>
      <Link href='/drinks' className='btn btn-primary mt-8 mb-12'>
        back to drinks
      </Link>
      {/* <img src={imgSrc} alt='' /> */}
      <Image src={drinkImg} className='w-48 h-48 rounded' alt='' />
      <h1 className='text-4xl mb-8'>{title}</h1>
    </div>
  );
};
export default SingleDrink;
```

Remote Images

- To use a remote image, the src property should be a URL string.
- Since Next.js does not have access to remote files during the build process, you'll need to provide the width, height and optional blurDataURL props manually.
- The width and height attributes are used to infer the correct aspect ratio of image and avoid layout shift from the image loading in. The width and height do not determine the rendered size of the image file.
- To safely allow optimizing images, define a list of supported URL patterns in next.config.js. Be as specific as possible to prevent malicious usage.
- restart dev server

- priority property to prioritize the image for loading

```
const SingleDrink = async ({ params }) => {
  const data = await getSingleDrink(params.id);
  const title = data?.drinks[0]?.strDrink;
  const imgSrc = data?.drinks[0]?.strDrinkThumb;
  return (
    <div>
      <Link href='/drinks' className='btn btn-primary mt-8 mb-12'>
        back to drinks
      </Link>
      {/* <img src={imgSrc} alt='' /> */}
      {/* <Image src={drinkImg} className='w-48 h-48 rounded' alt='' /> */}
      <Image
        src={imgSrc}
        width={300}
        height={300}
        className='w-48 h-48 rounded shadow-lg mb-4'
        priority
        alt=''
      />
      <h1 className='text-4xl mb-8'>{title}</h1>
    </div>
  );
};
export default SingleDrink;
```

```
** @type {import('next').NextConfig} */
const nextConfig = {
  images: {
    remotePatterns: [
      {
        protocol: 'https',
        hostname: 'www.thecocktaildb.com',
        port: '',
        pathname: '/images/**',
      },
    ],
  },
};

module.exports = nextConfig;
```

Remote Images - Responsive

- The fill prop allows your image to be sized by its parent element
- sizes property helps the browser select the most appropriate image size to load based on the user's device and screen size, improving website performance and user experience.

DrinksList.jsx

```
import Image from 'next/image';
import Link from 'next/link';

const DrinksList = ({ drinks }) => {
  return (
    <ul className='grid sm:grid-cols-2 gap-6 mt-6'>
      {drinks.map((drink) => (
        <li key={drink.idDrink}>
          <Link
            href={`/drinks/${drink.idDrink}`}
            className='text-xl font-medium'
          >
            <div className='relative h-48 mb-4'>
              <Image
                src={drink.strDrinkThumb}
                fill
                sizes='(max-width: 768px) 100vw, (max-width: 1200px) 50vw'
                alt={drink.strDrink}
                className='rounded-md object-cover'
              />
            </div>
            {drink.strDrink}
          </Link>
        </li>
      ))}
    </ul>
  );
};
export default DrinksList;
```

More Routing

- Private Folders _folder
- Route Groups (dashboard)
- Dynamic Routes
    - [...folder] - Catch-all route segment
    - [[...folder]] Optional catch-all route segment (used by Clerk)
- create test folder app/_css
- create app/(dashboard)/auth
    - the url is just '/auth'
- create app/(dashboard)/auth/[sign-in]

```
const SignInPage = ({ params }) => {
  console.log(params);
  return <div>SignInPage</div>;
};
export default SignInPage;
```

- create app/(dashboard)/auth/[...sign-in]
- create app/(dashboard)/auth/[[...sign-in]]

Prisma

- install prisma vs-code extension

Prisma sqlite

Prisma ORM is a database toolkit that simplifies database access in web applications. It allows developers to interact with databases using a type-safe and auto-generated API, making database operations easier and more secure.

- Prisma server: A standalone infrastructure component sitting on top of your database.
- Prisma client: An auto-generated library that connects to the Prisma server and lets you read, write and stream data in your database. It is used for data access in your applications.

```
npm install prisma --save-dev
npm install @prisma/client
```

```
npx prisma init
```

This creates a new prisma directory with your Prisma schema file and configures SQLite as your database. You're now ready to model your data and create your database with some tables.

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

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

- ADD .ENV TO .GITIGNORE !!!!

.env

```
DATABASE_URL = 'file:./dev.db';
```

Setup Instance

In development, the command next dev clears Node.js cache on run. This in turn initializes a new PrismaClient instance each time due to hot reloading that creates a connection to the database. This can quickly exhaust the database connections as each PrismaClient instance holds its own connection pool.

(Prisma Instance)[https://www.prisma.io/docs/guides/other/troubleshooting-orm/help-articles/nextjs-prisma-client-dev-practices#solution]

- create 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 Model

```
model Task {
  id String @id @default(uuid())
  content String
  createdAt DateTime @default(now())
  completed Boolean @default(false)
}
```

- safely applies and tracks changes to the database structure.

```
npx prisma migrate dev
```

- in a new terminal window
- launch Prisma Studio, which is a visual editor for your database.
- http://localhost:5555

```
npx prisma studio
```

Prisma Model

The Prisma model provided is a representation of a Task entity in the context of a database schema. Here's a detailed description of each component within this model:

model Task { ... }: This is the definition of the Task model. In Prisma, a model represents a table in the database. It serves as a blueprint for the records that will be stored in the corresponding table, defining the structure and behavior of the data.

id String @id @default(uuid()): This line defines a field named id of type String which is marked with @id, signifying that this field is the primary key of the model. The @default(uuid()) directive indicates that the default value for this field will be a UUID (Universally Unique Identifier) generated by Prisma.

content String: This line declares a field named content of type String. This field will store textual data, presumably the details or description of the task.

createdAt DateTime @default(now()): This field is named createdAt and is of type DateTime. It has a default value set to the current timestamp at the time of record creation, indicated by the @default(now()) directive.

completed Boolean @default(false): Lastly, the completed field is of type Boolean and is used to indicate whether the task has been completed. It defaults to false, meaning when a new task record is created, it is considered incomplete by default.

In Prisma, the term "model" refers to an abstraction that maps to a table in your database. Prisma models are defined in the Prisma schema, which is a declarative representation of your database's structure. Each model in the Prisma schema corresponds to a table in the database, and each field within a model corresponds to a column in that table. This schema plays a central role in Prisma's features, such as type-safe database access and migrations.

Prisma Example

- remove query
- fix the links the navbar

```
import prisma from '@/utils/db';

const prismaHandlers = async () => {
  await prisma.task.create({
    data: {
      content: 'wake up',
    },
  });
```

```
  const allTasks = await prisma.task.findMany({
    orderBy: {
      createdAt: 'desc',
    },
  });

  return allTasks;
};

const Prisma = async () => {
  const tasks = await prismaHandlers();
  return (
    <div>
      <h1 className='text-3xl font-medium mb-4'>Prisma Example </h1>
      {tasks.map((task) => {
        return (
          <h2 key={task.id} className='text-xl py-2'>
            😬 {task.content}
          </h2>
        );
      })}
    </div>
  );
};
export default Prisma;
```

Optional - Prisma Crud

Prisma Docs

- Create Single Record

```
const task = await prisma.task.create({
  data: {
    content: 'some task',
  },
});
```

- Get All Records

```
const tasks = await prisma.task.findMany();
```

- Get record by ID or unique identifier

```
// By unique identifier
const user = await prisma.user.findUnique({
  where: {
```

```
    email: 'elsa@prisma.io',
  },
});

// By ID
const task = await prisma.task.findUnique({
  where: {
    id: id,
  },
});
```

- Update Record

```
const updateTask = await prisma.task.update({
  where: {
    id: id,
  },
  data: {
    content: 'updated task',
  },
});
```

- Update or create records

```
const upsertTask = await prisma.task.upsert({
  where: {
    id: id,
  },
  update: {
    content: 'some value',
  },
  create: {
    content: 'some value',
  },
});
```

- Delete a single record

```
const deleteTask = await prisma.task.delete({
  where: {
    id: id,
  },
});
```

Challenge - Display Tasks

- create TaskForm, TaskList, DeleteForm components
- render them in tasks page
- in TaskList render all tasks
- also display editBtn and DeleteForm
- editBtn - link to single task page
- reference the complete project

tasks/page.js

```
import TaskForm from '@/components/TaskForm';
import TaskList from '@/components/TaskList';

const TasksPage = () => {
  return (
    <div className='max-w-lg'>
      <TaskForm />
      <TaskList />
    </div>
  );
};
export default TasksPage;
```

components/TaskList

```
import prisma from '@/utils/db';
import Link from 'next/link';
import DeleteForm from './DeleteForm';

const TaskList = async () => {
  const tasks = await prisma.task.findMany({
    orderBy: {
      createdAt: 'desc',
    },
  });
  if (tasks.length === 0)
    return <h2 className='mt-8 font-medium text-lg'>No tasks to show</h2>;
  return (
    <ul className='mt-8'>
      {tasks.map((task) => (
        <li
          key={task.id}
          className='flex justify-between items-center px-6 py-4 mb-4 border
border-base-300 rounded-lg shadow-lg'
        >
          <h2
            className={`text-lg capitalize ${
              task.completed ? 'line-through' : null
            }`}
          >
            {task.content}
```

```
            </h2>
            <div className='flex gap-6 items-center'>
              <Link href={`/tasks/${task.id}`} className='btn btn-accent btn-xs'>
                edit
              </Link>
              <DeleteForm id={task.id} />
            </div>
          </li>
        ))}
      </ul>
    );
};
export default TaskList;
```

Server Actions

- asynchronous server functions that can be called directly from your components.
- typical setup for server state mutations (create, update, delete)
    - endpoint on the server (api route on Next.js)
    - make request from the front-end
        - setup form, handle submission etc
- Next.js server actions allow you to mutate server state directly from within a React component by defining server-side logic alongside client-side interactions.

Rules :

- must be async
- add 'use server' in function body
    - use only in React Server Component

```
export default function ServerComponent() {
  async function myAction(formData) {
    'use server';
    // access input values with formData
    // formData.get('name')
    // mutate data (server)
    // revalidate cache
  }

  return <form action={myAction}>...</form>;
}
```

- or setup in a separate file ('use server' at the top)
    - can use in both (RSC and RCC)

utils/actions.js

```
'use server';

export async function myAction() {
  // ...
}
```

```
'use client';

import { myAction } from './actions';

export default function ClientComponent() {
  return (
    <form action={myAction}>
      <button type='submit'>Add to Cart</button>
    </form>
  );
}
```

TaskForm

```
import prisma from '@/utils/db';
import { revalidatePath } from 'next/cache';

const createTask = async (formData) => {
  'use server';
  const content = formData.get('content');
  // some validation here

  await prisma.task.create({
    data: {
      content,
    },
  });
  // revalidate path
  revalidatePath('/tasks');
};

const TaskForm = () => {
  return (
    <form action={createTask}>
      <div className='join w-full'>
        <input
          className='input input-bordered join-item w-full'
          placeholder='Type Here'
          type='text'
          name='content'
          required
        />
```

```
          <button type='submit' className='btn join-item btn-primary'>
            create task
          </button>
        </div>
      </form>
    );
};
export default TaskForm;
```

Refactor

- create utils/actions
- move get all tasks and create task functionality
- refactor TaskForm and TaskList

```
'use server';

import prisma from '@/utils/db';
import { revalidatePath } from 'next/cache';

export const getAllTasks = async () => {
  return prisma.task.findMany({
    orderBy: {
      createdAt: 'desc',
    },
  });
};

export const createTask = async (formData) => {
  const content = formData.get('content');
  // some validation here

  await prisma.task.create({
    data: {
      content,
    },
  });
  // revalidate path
  revalidatePath('/tasks');
};
```

DeleteForm

- will use "action" approach since it works without JS
- invoke by using startTransition (useTransition hook)

```
import { deleteTask } from '@/utils/actions';
```

```
const DeleteForm = ({ id }) => {
  return (
    <form action={deleteTask}>
      <input type='hidden' name='id' value={id} />
      <button className='btn btn-error btn-xs'>delete</button>
    </form>
  );
};
export default DeleteForm;
```

utils/actions

```
export const deleteTask = async (formData) => {
  const id = formData.get('id');
  await prisma.task.delete({ where: { id } });
  revalidatePath('/tasks');
};
```

Challenge - Edit Task

- create single task page
- get task info (hint:params)
- create EditForm component
- setup form with all the inputs
- render in single task page
- create server action to update task
- extra - redirect when complete

Edit Task

tasks/[id]/page.js

```
import EditForm from '@/components/EditForm';
import { getTask } from '@/utils/actions';
import Link from 'next/link';
const TaskPage = async ({ params }) => {
  const task = await getTask(params.id);

  return (
    <>
      <div className='mb-16'>
        <Link href='/tasks' className='btn btn-accent'>
          Back to Tasks
        </Link>
      </div>
      <EditForm task={task} />
    </>
  );
```

```
};
export default TaskPage;
```

actions

```
export const getTask = async (id) => {
  return prisma.task.findUnique({
    where: {
      id,
    },
  });
};

export const editTask = async (formData) => {
  const id = formData.get('id');
  const content = formData.get('content');
  const completed = formData.get('completed');

  await prisma.task.update({
    where: {
      id: id,
    },
    data: {
      content: content,
      completed: completed === 'on' ? true : false,
    },
  });
  // redirect won't works unless the component has 'use client'
  // another option, setup the editTask in the component directly
  redirect('/tasks');
};
```

EditForm.js

```
'use client';
import { editTask } from '@/utils/actions';

const EditForm = ({ task }) => {
  const { id, completed, content } = task;
  return (
    <form
      action={editTask}
      className='max-w-sm bg-base-100 p-12 border border-base-300  rounded-lg'
    >
      <input type='hidden' name='id' value={id} />
      {/* content */}

      <input
        type='text'
```

```
          required
          defaultValue={content}
          name='content'
          className='input input-bordered w-full'
        />

        {/* completed */}
        <div className='form-control my-4'>
          <label className='label cursor-pointer'>
            <span className='label-text'>Completed</span>
            <input
              type='checkbox'
              defaultChecked={completed}
              name='completed'
              className='checkbox checkbox-primary checkbox-sm'
            />
          </label>
        </div>
        <button className='btn btn-primary btn-block btn-sm'>edit</button>
      </form>
    );
};
export default EditForm;
```

Server Actions - Loading State, Response, Errors

- clone TaskForm - copy and rename TaskFormCustom
- make it client component
- import and replace in Tasks page
- clone createTask in actions - copy and rename createTaskCustom
- import and replace in TaskFormCustom component

Server Actions - Loading State

TaskFormCustom

```
'use client';

import { createTaskCustom } from '@/utils/actions';
import { useFormStatus } from 'react-dom';
// The useFormStatus Hook provides status information of the last form submission.
// useFormState is a Hook that allows you to update state based on the result of a
form action.

const SubmitButton = () => {
  const { pending } = useFormStatus();

  return (
    <button
      type='submit'
```

```
        className='btn join-item btn-primary'
        disabled={pending}
      >
        {pending ? 'please wait... ' : 'create task'}
      </button>
    );
};

const TaskForm = () => {
  return (
    <form action={createTaskCustom}>
      <div className='join w-full'>
        <input
          className='input input-bordered join-item w-full'
          placeholder='Type Here'
          type='text'
          name='content'
          required
        />
        <SubmitButton />
      </div>
    </form>
  );
};
export default TaskForm;
```

```
export const createTaskCustom = async (formData) => {
  await new Promise((resolve) => setTimeout(resolve, 2000));
  const content = formData.get('content');
  // some validation here

  await prisma.task.create({
    data: {
      content,
    },
  });
  // revalidate path
  revalidatePath('/tasks');
};
```

Error Handling and Response To User

TaskFormCustom.jsx

```
'use client';

import { createTaskCustom } from '@/utils/actions';
import { useFormStatus, useFormState } from 'react-dom';
// The useFormStatus Hook provides status information of the last form submission.
```

```
// useFormState is a Hook that allows you to update state based on the result of a
form action.

const SubmitButton = () => {
  const { pending } = useFormStatus();

  return (
    <button
      type='submit'
      className='btn join-item btn-primary'
      disabled={pending}
    >
      {pending ? 'please wait... ' : 'create task'}
    </button>
  );
};

const initialState = {
  message: null,
};

const TaskForm = () => {
  const [state, formAction] = useFormState(createTaskCustom, initialState);

  return (
    <form action={formAction}>
      {state.message ? <p className='mb-2'>{state.message}</p> : null}
      <div className='join w-full'>
        <input
          className='input input-bordered join-item w-full'
          placeholder='Type Here'
          type='text'
          name='content'
          required
        />
        <SubmitButton />
      </div>
    </form>
  );
};
export default TaskForm;
```

actions.js

```
// fix params
export const createTaskCustom = async (prevState, formData) => {
  await new Promise((resolve) => setTimeout(resolve, 2000));
  const content = formData.get('content');
  // some validation here
  try {
    await prisma.task.create({
      data: {
```

```
        content,
      },
    });
    // revalidate path
    revalidatePath('/tasks');
    return { message: 'success!!!' };
  } catch (error) {
    // can't return error
    return { message: 'error...' };
  }
};
```

Extra - More User Input Validation Options

- required attribute a great start
- zod library

The Zod library is a TypeScript-first schema declaration and validation library that allows developers to create complex type checks with simple syntax.

Zod

```
npm install zod
```

actions.js

```
import { z } from 'zod';

export const createTaskCustom = async (prevState, formData) => {
  await new Promise((resolve) => setTimeout(resolve, 2000));
  const content = formData.get('content');

  const Task = z.object({
    content: z.string().min(5),
  });

  // some validation here
  try {
    Task.parse({
      content,
    });
    await prisma.task.create({
      data: {
        content,
      },
    });
    // revalidate path
    revalidatePath('/tasks');
    return { message: 'success!!!' };
```

```
  } catch (error) {
    console.log(error);
    // can't return error
    return { message: 'error...' };
  }
};
```

Providers

Beautiful Toasts

```
npm install react-hot-toast
```

- create providers.js file in app

```
'use client';
import { Toaster } from 'react-hot-toast';

const Providers = ({ children }) => {
  return (
    <>
      <Toaster />
      {children}
    </>
  );
};
export default Providers;
```

layout.js

```
import Providers from './providers';

export default function RootLayout({ children }) {
  return (
    <html lang='en'>
      <body className={inter.className}>
        <Navbar />
        <main className='px-8 py-20 max-w-6xl mx-auto'>
          <Providers>{children}</Providers>
        </main>
      </body>
    </html>
  );
}
```

- simplify createTaskCustom action

```
try {
  return { message: 'success' };
} catch (error) {
  console.log(error);
  return { message: 'error' };
}
```

- add toast

TaskFormCustom.jsx

```
'use client';
import { createTaskCustom } from '@/utils/actions';
import { useEffect } from 'react';

import { useFormStatus, useFormState } from 'react-dom';
import toast from 'react-hot-toast';
const SubmitBtn = () => {
  const { pending } = useFormStatus();
  return (
    <button
      type='submit'
      className='btn btn-primary join-item'
      disabled={pending}
    >
      {pending ? 'please wait...' : 'create task'}
    </button>
  );
};

const initialState = {
  message: null,
};

const TaskForm = () => {
  const [state, formAction] = useFormState(createTaskCustom, initialState);

  useEffect(() => {
    if (state.message === 'error') {
      toast.error('there was an error');
      return;
    }
    if (state.message) {
      toast.success('task created....');
    }
  }, [state]);

  return (
    <form action={formAction}>
      <div className='join w-full'>
```

```
          <input
            type='text '
            className='input input-bordered join-item w-full'
            placeholder='type here'
            name='content'
            required
          />
          <SubmitBtn />
        </div>
      </form>
    );
};
export default TaskForm;
```

Challenge - Add Functionality

DeleteForm.jsx

```
'use client';
import { useFormStatus } from 'react-dom';
import { deleteTask } from '@/utils/actions';

const SubmitBtn = () => {
  const { pending } = useFormStatus();
  return (
    <button type='submit' className='btn btn-xs btn-error' disabled={pending}>
      {pending ? 'pending...' : 'delete'}
    </button>
  );
};

const DeleteForm = ({ id }) => {
  return (
    <form action={deleteTask}>
      <input type='hidden' name='id' value={id} />
      <SubmitBtn />
    </form>
  );
};
export default DeleteForm;
```

Route Handlers

- install Thunder Client

Route Handlers allow you to create custom request handlers for a given route using the Web Request and Response APIs.

- in app create folder "api"
- in there create folder "tasks" with route.js file

The following HTTP methods are supported: GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS. If an unsupported method is called, Next.js will return a 405 Method Not Allowed response.

In addition to supporting native Request and Response. Next.js extends them with NextRequest and NextResponse to provide convenient helpers for advanced use cases.

app/api/tasks/route.js

```
// the following HTTP methods are supported: GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS. If an unsupported method is called, Next.js will return a 405 Method Not Allowed response.

import { NextResponse } from 'next/server';
import db from '@/utils/db';

export const GET = async (request) => {
  const tasks = await db.task.findMany();
  return Response.json({ data: tasks });
  // return NextResponse.json({ data: tasks });
};

export const POST = async (request) => {
  const data = await request.json();
  const task = await db.task.create({
    data: {
      content: data.content,
    },
  });
  return NextResponse.json({ data: task });
};
```

Middleware

Docs

Middleware in Next.js is a piece of code that allows you to perform actions before a request is completed and modify the response accordingly.

- create middleware.js in the root
- by default will be invoked for every route in your project

```
export function middleware(request) {
  return Response.json({ msg: 'hello there' });
}

export const config = {
  matcher: '/about',
};
```

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

// This function can be marked `async` if using `await` inside
export function middleware(request) {
  return NextResponse.redirect(new URL('/', request.url));
}

// See "Matching Paths" below to learn more
export const config = {
  matcher: ['/about/:path*', '/tasks/:path*'],
};
```

PlanetScale

Host DB

- set DATABASE_URL in .env

```
generator client {
  provider = "prisma-client-js"
}
datasource db {
  provider = "mysql"
  url = env("DATABASE_URL")
  relationMode = "prisma"
}
```

```
npx prisma db push
```

Local Build

Setup App

package.json

```
 "build": "npx prisma generate && next build",
```

- prisma-example

```
import prisma from '@/utils/db';

const prismaHandlers = async () => {
  console.log('prisma example');
  // await prisma.task.create({
```

```
    //   data: {
    //     content: 'wake up',
    //   },
    // });
    return prisma.task.findMany();
  };

  const PrismaExample = async () => {
    const tasks = await prismaHandlers();
    if (tasks.length === 0) {
      return <h2 className='mt-8 font-medium text-lg'>No tasks to show...</h2>;
    }

    return (
      <div>
        <h1 className='text-7xl'>PrismaExample</h1>
        {tasks.map((task) => {
          return (
            <h2 key={task.id} className='text-xl py-2'>
              😬 {task.content}
            </h2>
          );
        })}
      </div>
    );
  };
  export default PrismaExample;
```

- clean out the DB

```
npm run build
```

```
npm start
```

Force Dynamic

- add loading.js to tasks

tasks.js

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

Deploy

Vercel

- sign up for account
- create github repo