Next JS #94

Layout.js

Layouts act as wrapper around pages
and
we can have subset of pages where they could also use another specialized layout

eg
so if we have a root layout and another route's folder with root layout

no matter which page you visit the root layout page will always be active

we can use this prop and the content that is made available through the prop will be the content wrapped by those component tags ....

c/

```
export default function MealsLayout({children})
{
  return (
    <>
    <p> Meals Layout </p>
    {children}
    <>
    )
}
```

this renders all the nested content

Next will essentially wrap that layout around pages or nested layouts that are covered by this layout.
hence children will give you access to any nested layout or pages

Importing image

c/

```
import logoImg from '@/assets/logo.png'
:
<img src={logoImg.src} alt='logo' />
```

Optimising images with <Image> component

helps to output images in optimised way eg lazy loading

c/

```
import logoImg from '@/assets/logo.png'
:
<Image src={logoImg} alt='logo' />
```

↑ dont need to add .src

Image Slideshow - not related to next

c/

```
useEffect(() => {

    const interval = setInterval(() => {

        setCurrentImageIndex((prev) =>

            prev < images.length - 1 ?
            prevIndex + 1
                :
            0
        )

    }, 5000)

    return () => clearInterval(interval)

}, [])
```

* this statement cleans up the interval when component unmounts to prevent memory leaks

Server vs Client components - when to use which

By default all the React Components in nextjs are only rendered on server

because of this

we have potentially less client side Javascript code that must be downloaded

Nonetheless
we can still build client components that are still technically prerendered on the server but also potentially rendered on the client

eg
hooks like useState, useEffect are not available on server side components
also event handlers eg onClick function

so if we want we can make a component a client side component
just by writing:

c/ 'use client'

[write this at top of the component

→ Typically we should use client components as far down the component tree as possible
so that
majority of components can stay server components

Highlight active link with usePathName hook

```
import { usePathname } from 'next/navigation'
.
.
.

export default function Header() {

    const path = usePathname()

    return (
        .
        .
        .

        <Link
            className={
                path.startsWith('/meals') ? styles.active :
                undefined
            }
        >
```

How to output Image with unknown dimension

c|

```
<Image src={image}
       alt={title}
       fill
/>
```

simply fill the available space with that image as defined by parent components

Creating custom components

→ we can use client components inside server components

c/

```
export default function NavLink ({ href, children }) {

    return (

        <Link href={href}>

            {children}

        </Link>

        )
```

Using SQL lite in next js

→ Installation

```
npm install better-sqlite3
```

→ Creating file in root directory

c/

```
const sql = require('better-sqlite3')
const db = sql('meals.db')
```

↑ establishes a connection to meals.db SQLlite database file

```
const dummyMeals = [
  ... contains all the dummy data
]
```

```
db.prepare(`
    CREATE TABLE IF NOT EXISTS meals(
        id INTEGER PRIMARY KEY AUTOINCREMENT
        slug TEXT NOT NULL UNIQUE
        title TEXT NOT NULL
        creator_email TEXT NOT NULL
    )
`).run()

async function initData() {
    const stmt = db.prepare(`
        INSERT INTO meals VALUES (
            null,
            @slug,
            @title,
            @creator_email
        )`)
    for (const meal of dummyMeals) {
        stmt.run(meal)
    }
}

initData()
```

→ Running the file

```
node initdb.js
```

↑ run the file we just created

after running this
a seperate file is generated (meals.db)

that is the SQL light database that we'll use

fetching data by leveraging next.js capabilities

because next.js apps have server components we dont need useEffect or fetch function

Note: server component functions can be directly converted to async function

```
export default async function Page() {
  ...
}
```

→ creating seperate file for fetching data

```
import sql from 'better-sqlite3'

const db = sql('meals.db')

export function getMeals() {
  return db.prepare('SELECT * FROM meals').all()
}
```

→ using this function in our component to fetch data

c/ `const meals = getMeals()`

if it is a promise then convert component into async function

c/
```
export default async function Page() {
    const meals = await getMeals()
}
```

Adding a Loading Page

- Next JS performs aggressive caching under the hood
  so
  it caches the page we have visited including data of the page
  so
  if we go to another page and come back it loads that page from cache and hence the page is loaded very quickly

- we can place `loading.js` file
  ↑ reserved name
  with some loading spinner or anything in the directory of page.js which is loading data

Using suspense and streamed responses for granular loading state management

- The problem with the initial approach is the whole page.js is not shown and we only want to show loading state instead of the data which is being fetched and not the other elements such as eg header

so instead of using loading.js nextjs gives us other way of handling loading states a more granular way

c/ import {Suspense} from 'react'

```
async function Meals(){

    const meals = await getMeals()
    return <MealsGrid meals={meals}/>

}
```

↳ seperate component which holds async code and html where data has to be loaded.

This component can be a seperate file or can be created directly inside where we want to use

```
export default function MealsPage() {
    .
    .
    .
    <Suspense fallback={<p> Loading ... </p>}>
        <Meals/>
    </Suspense>
```

Handling Errors

we can also add a error.js file
which handles errors genrated by pages and compnents

c/ client component

```
'use client'

export default function Error ({error}) {

    return (
        <main className='error'>
            <h1> error occured </h1>
        </main>
    )
}
```

we get an error object

this class can be used anywhere since its defined in globals.css (no import required)

Not found State

not-found.js file
catches all the not found errors

```
c/ export default function NotFound() {

    return (
      .
      .
      .
    )
}
```

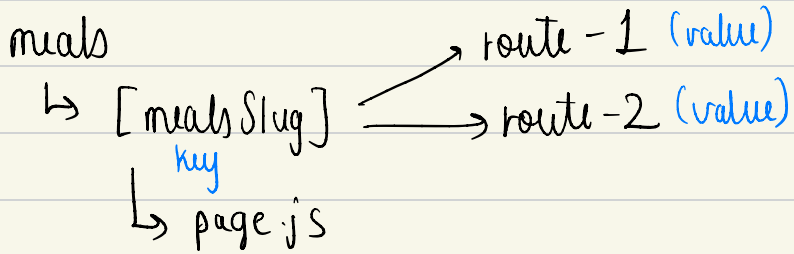
Loading and Rendering details via Dynamic Routes & Route Parameters

→ creating api function

search query

```
export function getMeal(slug) {
    return db.prepare('SELECT * FROM meals WHERE
    slug = ?').get(slug)
}
```

Folder Structure



- Calling function and injecting values

c|

```
export default function MealDetailsPage({ params })
{
    const meal = getMeal(params.mealSlug)
                                  ↑ this is the name of the folder in []

    return (

        <header> {meal.title} </header>
        <p> {meal.creator} </p>

    )
}
```

Throwing Error

c/

```
import {notFound} from 'next/navigation'

export default function MealDetailPage({params}) {

  const meal = getMeal(params.mealSlug)

  if(!meal) {
    return notFound()
  }

  ...
}
```

stops the component from executing and displays the closest notFound page when there is a error condition

Image Picker Component

c/ 'use Client'

```
import { useRef } from 'react'

export default function ImagePicker({ label, name }) {

  const imageInput = useRef()
  function handlePickClick() {
    imageInput.current.click()
  }

  return ( <label> ... </label>
    <input
      type='file'
      id={name}
      accept='image/png, image/jpg'
      name={name}
      ref={imageInput} />
    <button type='button'
      onClick={handlePickClick}></button>
  )
}
```

props we accept

connecting hidden input to our custom button

we basically hide this

and instead use this button

Adding an Image Preview after selecting an Image

↳ in image picker component

c/

```
const [ pickedImage, setPickedImage ] = useState( )

function handleImageChange (event) {

    const file = event.target.files[0]

    if (!file) {
        setPickedImage (null)
        return
    }                                   } if file is not picked

    // convert image into data url
    const fileReader = new FileReader()

    fileReader.readAsDataUrl (file)
    fileReader.onload = () => {
        setPickedImage (fileReader.result)
    }
}
```

```
return (
    {pickedImage && (
        <Image src={pickedImage} />
    )}
)
```

Server Actions for handling form submissions

c/

```
export default function ShareMeal() {

  async function shareMeal(formData) {
    'use server'

    const meal = {
      title : formData.get('title')
      email : formData.get('email')
    }

  return (
    <form action={shareMeal}>
      ...
    </form>
```

this creates a server action which is then only executed on the server

this is the name we set on input field

Storing Server Actions into seperate files

* The previous method will only work if component is not a client component
* server actions wont work if it is a client component

c/ 'use server' ⟵ now all the function difined in this file are server actions

```
export async function shareMeal(formData) {
    :
}
```

---

2 now just import this function in file you want to use

```
import { shareMeal } from '@/fetch-function/actions'
    :
<form action={shareMeal}>
```

Creating a slug and sanitizing user input for XSS Protection

```
npm install slugify xss
```

↖ to prevent cross site scripting attacks

- we need xss because we are outputting instructions as html

```
c/ ... <p dangerouslySetInnerHTML={{
        __html: meal.instructions
      }} />
   </p>
```

- so we can add following statements in our function and modify data on the fly while storing data into our database

```
c/ export function saveMeal(meal) {

    meal.slug = slugify(meal.title, {lower: true})
    meal.instructions = xss(meal.instructions)
```

Storing uploaded images & storing data in a database

```
import fs from 'node:fs'

export async function saveMeal(meal) {

  meal.slug = slugify(meal.title, { lower: true })
  meal.instructions = xss(meal.instruction);
```

→ get rid of the photo file name extension &
→ create a unique filename

```
  const extension = meal.image.name.split('.').pop()
  const fileName = `${meal.slug}.${extension}`
```

→ store image in public folder

```
  const stream = fs.createWriteStream(`public/${fileName}`)
  const bufferedImage = await meal.image.arrayBuffer()
```

→ Stream.write requires a buffered image
- thing you want to write to disk (buffered img)
- callback fn when execution is finished

```
stream.write(Buffer.from(bufferedImage), (error) =>
    {
    if (error) {
        throw new Error('Saving Image Faild')
        }
    })
```

→ now we want to store image path in database, so modify it

```
real.image = `/images/${fileName}`
```

→ using sqlite now write whole data to database

```
db.prepare (`
    INSERT INTO meals
    (title, summary, instruction, creator,
     creator_email, image, slug)
    VALUES (
        @title
        @summary
        @instruction,
        @creator,
        @creator_email,
        @image,
        @slug
    )
`).run(meal)
```

place holders should match the order difind above

}

→ so now in action import this function

```
'use server'

import { redirect } from 'next/navigation'
import { saveMeal } from './meals'

export async function shareMeal(formData) {

    const meal = {
        title : formData.get('title')
        :
        :
    }

    await saveMeal(meal)
    redirect('/meals')
}
```

redirect to other page after meals are saved to database

Managing form Submission Status with useFormStatus

if we want to provide user with some feedback as something like to let user know that the request is pending / ongoing. hence we use this hook

- creating seperate component button that displays button's text as 'loading...' when req is ongoing
- also this only works in client component, so instead of making the whole component → client component where the form is located. just create client component for button

c/

```
'use client'

import { useFormStatus } from 'react-dom'

export default function MealsFormButton() {

    const { pending } = useFormStatus()

    return (
        <button disabled={pending}>
            {pending? 'submitting...' : 'Share Meal'}
        </button>
        )
}
```

pending is true if there is an ongoing request

Adding Server Side Input Validation

↳ in actions

easy eg

```
c/ if (!mial.title || mial.title.trim()=== '') {
        throw New Error ('Invalid Input')
   }
```

but now due to this the whole data entered by user is gone as we are redirected to error page

so instead its better to show error on same page ---> (Next)

previously we used useFormStatus

Working with Server Action Responses and useFormState

→ to show error on same page
→ so as to not loose all the user input in the form

- in server component we can also return values
- in actions

* although we dont use it we need to accept this arg

```
// export async function shareMeal(prevState, formData) {
  ...
  if (!meal.title || meal.title.trim() === '') {
    return { message: 'Invalid Input' }
  }
  ...
}
```

→ in our component where we have a form

c/

```
'use client'

import { useFormState } from 'react-dom'

export default function ShareMealPage() {

const [currState, formAction] = useFormState(shareMeal, {message: null})
```

this is our server action

argument 1: initial state of component i.e. initial value that should be returned by useFormState

argument 2: actual server action that should be triggered when form is submitted

```
<form action={formAction}>...</form>

// display notification
{currState.message && <p>{currState.message}</p>}
```

Building for production and understanding NextJS caching

```
npm run build
```

prepares nextjs application for production

```
npm start
```

start the production server

↳ to prepare app for production, nextjs generates and pre renders all the non-dynamic pages
∴ it is very fast during production

↳ the downside to this is it never re-fetches the meals (dynamic data)
hence new data we add using form is not seen on the page

so how can we fix the problem of nextJS caching aggressively ?

Triggering cache revalidations

so when do we want to revalidate path?
→ when a new meal is added

hence
we add the following in the server action function

```
export async function shareMeal(prevState, formData)
{
    :
    await saveMeal(meal)
    revalidatePath('/meals')
}
```

↑ only rerenders the meals path and not its nested paths

```
revalidatePath('/meals', 'layout')
```

if you add this second argument it will also rerender its nested components

Dont Store Files on the Local System

- use AWS S3 to store files/images which are user generated

Adding static metadata

c/
```
export const metadata = {
    title: ' '
    description: ' '
}
```

- if you have added this to root layout.js
  it will automatically be added to all the pages
  that are wrapped by the layout
  unless
  a page specifies its own metadata

Adding Dynamic Metadata

this function receives same data our page component receives as props

this name can't be changed

```
export async function generateMetaData({params})
{

    const meal = getMeal(params.mealSlug)

    if(!meal) { notFound() }

    return (
        title: meal.title
    discription: meal.summary
        )
}
```

additional check because this metadata wouldnt load for incorrect dynamic path