

NEXT.JS

NEXT講座  
ブログシステム編

# この資料について



オンライン学習プラットフォーム『Udemy』  
で公開している  
『NEXTjs 基本講座』の説明用資料です

<https://www.udemy.com/course/nextjs-basic>



# 簡易 ブログシステム

# 簡易ブログシステム

The screenshot shows a blog homepage with a header containing 'Blog', a search bar, and login/authenticate buttons. Below the header are two blog posts and a user registration form.

**Post 1:** **たちおる**  
これは最初のブログ投稿です。  
Test User 3日前

**Post 2:** **初めてのブログ投稿**  
Test User 21日前

**ユーザー登録:**

- 名前: [Input Field]
- メールアドレス: [Input Field]
- パスワード: [Input Field]
- パスワード(確認): [Input Field]
- 登録: [Submit Button]

The illustration features various characters like students, a teacher, and a graduation cap, all surrounded by educational icons like books, a lightbulb, and a globe. The text 'BASICS' is prominently displayed at the top.

**投稿者:** Test User **2025年01月28日**

**たちおる**

**列の揃え方**

左寄せ	中央揃え	右寄せ
左揃えの値	中央の値	右揃えの値
テスト1	テスト2	テスト3
サンプル1	サンプル2	サンプル3

**チェックリストの例**

- 未完了のタスク1
- 完了済みのタスク2
- 未完了のタスク3

```
# Pythonの例
def greet(name):
    return f"Hello, {name}!"

print(greet("World"))
```

# 簡易ブログシステム



Next.js15 Typescript

認証: Auth.js (旧NextAuth v5)

DB: Prisma(ORM), Sqlite(DB)

UI: shadcn/ui, tailwindcss

バリデーション: zod

機能: 検索、Markdown, 画像アップロード

# 環境構築



Node.js ver20以上推奨

デスクトップに

next-udemy-blog フォルダを作成

ターミナルなどで

npx create-next-app@^15

# インストール時の選択



What is your project name? .

Ok to processd? y

src/ directory? Yes

App Router? Yes

Turbopack? No

import alias? No

# GitHub管理



GitHub側で

next-udemy-blog リポジトリ作成

ローカル側で

git init

git add .

git commit -m "first"

git branch -M main

git remote add origin GitHubのURL

git push -u origin main

# フォルダ構成 app

src

  app

**(auth)**

      login/page.tsx

      register/page.tsx

      layout.tsx

**(public)** · · ログインなしでもみれる

      posts/page.tsx トップページ・記事一覧

      posts/[id]/page.tsx 記事詳細

      posts/layout.tsx

**(private)** · · ログイン後

      manage/ · · ログイン後とわかりやすくする

      dashboard/page.tsx ダッシュボード

      posts/page.tsx 記事管理

      create/page.tsx 新規記事

      [id]/page.tsx 記事プレビュー

      [id]/edit/page.tsx 記事編集



# Prisma

# 今回作成するテーブル

User			1対多	Post		
物理	型	キー オプション		物理	型	キー オプション
id	cuid	PK		id	cuid	PK
name	string			title	string	
email	string	unique		content	string	
password	string			topImage	string	nullable
createdAt	datetime			published	boolean	
updatedAt	datetime			authorId	String	FK
				createdAt	datetime	
				updatedAt	datetime	

# Prisma インストール



パスワードも扱うため暗号化のbcryptjsもインストール

```
npm install prisma@^6 @prisma/client@^6
```

```
npm install -D ts-node@^10
```

```
npm install bcryptjs@^2
```

```
npm install @types/bcryptjs@^2
```

# Prismaの設定



npx prisma init // Prismaの初期化

prisma/schema.prismaと.envが生成される

prisma/schema.prismaの設定

provider= "sqlite" に変更 (ファイルベース 開発向けDB)

.env ファイル

postgresqlはコメントアウト

DATABASE\_URL="file:./dev.db" を追記

# Prisma クライアントインスタンス

## lib/prisma.ts

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

```
// グローバルスコープでPrismaインスタンスを保持できる場所を作る
```

```
const globalForPrisma = globalThis as unknown as {  
    prisma: PrismaClient | undefined  
}
```

```
// Prismaインスタンスがあれば使う、なければ作成
```

```
export const prisma = globalForPrisma.prisma ?? new PrismaClient()
```

```
// 開発環境でのみ使用
```

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

参考: <https://www.prisma.io/docs/orm/prisma-client/setup-and-configuration/databases-connections#prevent-hot-reloading-from-creating-new-instances-of-prismaclient>

# prisma/schema.prisma (DBの構造)

データベース内 テーブルの列や属性を定義

@id … PK @default … デフォルト値 @unique … 重複不可

? … 必須ではない Post[] … 1対nを表す @updatedAt … 更新年月

```
model User {  
    id      String  @id @default(cuid())  
    email   String  @unique  
    password String  
    name    String  
    posts   Post[]  
    createdAt DateTime @default(now())  
    updatedAt DateTime @updatedAt  
}
```

# prisma/schema.prisma

@relation · · リレーション fields: [authorId] · · このモデルのどの列を使うか references: [id] · · 参照先(User)のどの列と紐付けるか  
onDelete: Cascade ユーザーが削除された場合、ユーザー投稿も一緒に削除される

```
model Post {  
    id      String  @id @default(cuid())  
    title   String  
    content String  
    topImage String?  
    published Boolean @default(true)  
    authorId String  
    author   User    @relation(fields: [authorId], references: [id], onDelete: Cascade)  
    createdAt DateTime @default(now())  
    updatedAt DateTime @updatedAt  
}
```

# prisma/seed.ts | ダミーデータ

```
// prisma.対象テーブル名.メソッド のように記述
import { PrismaClient } from '@prisma/client'
import * as bcrypt from 'bcryptjs'
const prisma = new PrismaClient()

async function main() {
    // クリーンアップ
    await prisma.post.deleteMany()
    await prisma.user.deleteMany()
    const hashedPassword = await bcrypt.hash('password123', 12) // 暗号化
    const dummyImages = [ 'https://picsum.photos/seed/post1/600/400', // ダミー画像
        'https://picsum.photos/seed/post2/600/400' ] }

    main().catch((e) => { console.error(e) process.exit(1) })
    .finally(async () => { await prisma.$disconnect() })
```

# prisma/seed.ts 2 main関数の中

```
// ユーザー作成
const user = await prisma.user.create({
  data: {
    email: 'test@example.com',
    name: 'Test User',
    password: hashedPassword,
    posts: {
      create: [
        { title: 'はじめてのブログ投稿',
          content: 'これは最初のブログ投稿です。Next.jsとPrismaでブログを作成しています。',
          topImage: dummyImages[0],
          published: true,
        },
        { title: '2番目の投稿',
          content: 'ブログの機能を少しずつ追加していきます。認証機能やダッシュボードなども実装予定です。',
          topImage: dummyImages[1],
          published: true,
        }
      ]
    }
  }
})
console.log({ user }) }
```

# next.config.tsに追記

```
import type { NextConfig } from "next";
```

```
const nextConfig: NextConfig = {  
  images: {  
    remotePatterns: [  
      {  
        protocol: 'https',  
        hostname: 'picsum.photos',  
      },  
    ],  
  },  
}
```

# package.jsonに追記



コマンドでダミーデータも追加できるように

```
"prisma": {  
    "seed": "ts-node --compiler-options  
    {\\"module\\":\\"CommonJS\\"} prisma/seed.ts"  
},
```

# マイグレーションとシード実行



// マイグレーション(テーブル作成)

npx prisma migrate dev --name init

// シード実行(ダミーデータ)

npx prisma db seed

// DBの内容を確認

npx prisma studio

// DBリセット

npx prisma migrate reset

# エラーが出たら



// 全てのマイグレーションファイルを削除

rm -rf prisma/migrations

// dev.dbファイルも削除

rm prisma/dev.db

// 新しいマイグレーションファイルを作成

npx prisma migrate dev --name init

// シードを実行

npx prisma db seed

# Prisma Studio

The screenshot shows the Prisma Studio interface for managing a database. At the top, there's a navigation bar with back, forward, and search icons, and a URL field showing "localhost:5555". Below the navigation is a header with a "User" tab and a "+" button. Underneath are filter and field selection buttons: "Filters None", "Fields All", "Showing 1 of 1", and a prominent "Add record" button.

The main area displays a table with four columns: "id", "email", "password", and "name". A single row is shown for a user named "Test User" with the ID "cm4mgmtwt0000zp3w05ey5on4", email "test@example.com", and password "\$2a\$12\$dJc7ygDKiRnJHzA9B...".

Below this table is another table with columns "id", "title", "content", and "published". It lists two posts. Both posts have their "published" status set to "true". The first post has the ID "cm4mgmtwt0001zp3w9zdfdrib" and the title "はじめてのブログ投稿", with content "これは最初のブログ投稿で…". The second post has the ID "cm4mgmtwt0002zp3wxc2w3mdx" and the title "2番目の投稿", with content "ブログの機能を少しずつ追…".

At the bottom of the interface, there are two buttons: "Open in new tab" and "Skip to unconnected records".

	id	email	password	name
	cm4mgmtwt0000zp3w05ey5on4	test@example.com	\$2a\$12\$dJc7ygDKiRnJHzA9B...	Test User

	id	title	content	published
	cm4mgmtwt0001zp3w9zdfdrib	はじめてのブログ投稿	これは最初のブログ投稿で…	true
	cm4mgmtwt0002zp3wxc2w3mdx	2番目の投稿	ブログの機能を少しずつ追…	true



shadcn/ui

# shadcn/ui



<https://ui.shadcn.com/>

RadixuiとTailwindCSSを用いて開発された

自由度の高いUIコンポーネント集

2023 JS Rising Starsで1位

必要なコンポーネントのみを選択してインストールで

きる 軽量、柔軟に扱える

# shadcn/ui インストール・設定



npx shadcn@latest init

base color > Neutral

Use --force

# shadcn/ui



必要なコンポーネントをインストール

```
npx shadcn@latest add navigation-menu
```

```
button input label alert dropdown-menu
```

```
alert-dialog
```

--forceで強制インストール

src/components/uiに各コンポーネントがイ

ンストールされる



認証なし画面

# 認証なしヘッダー



src/components/layouts/PublicHeader.tsx

各コンポーネントをimportしてデザインする

src/app/(public)/layout.tsx内で

import PublicHeader 追加し表示確認

# 記事一覧の取得

prisma.対象テーブル名.メソッド

where 条件 include リレーション(別テーブル) select 表示列

orderBy ソート

## src/lib/post.ts

```
import { prisma } from "@/lib/prisma"
```

```
export async function getPosts() {
```

```
    return await prisma.post.findMany({
```

```
        where: { published: true },
```

```
        include: { author: { select: { name: true } } },
```

```
        orderBy: { createdAt: 'desc' } }) }
```

# ブロガー一覧のカード 日付フォーマット

```
npm install date-fns@^4 //日付フォー  
マット
```

```
npx shadcn@latest add card
```

# 記事の型



src/types/post.ts

```
export type Post = {
```

```
    id: string
```

```
    title: string
```

```
    content: string
```

```
    topImage: string | null
```

```
    createdAt: Date
```

```
    author: {
```

```
        name: string
```

```
}
```

```
}
```

```
export type PostCardProps = { post: Post }
```

# 記事一覧 カード

## src/components/post/PostCard.tsx

```
import { formatDistanceToNow } from 'date-fns'  
import { ja } from 'date-fns/locale'  
import Link from 'next/link'  
import { Card, CardHeader, CardTitle,CardContent } from "@/components/ui/card"  
import { PostCardProps } from '@/types/post.ts'  
  
export function PostCard({ post }: PostCardProps) {  
    略}
```

# 記事一覧 カード (画像最適化)

```
import Image from 'next/image'
```

```
{ post.topImage && (
  <div className="relative w-full h-48">
    <Image src={post.topImage}
      alt={post.title}
      fill
      sizes="(max-width: 768px) 100vw, (max-width: 1200px) 50vw, 33vw"
      className="rounded-t-md object-cover"
      priority />
  </div>
)}
```

参考: <https://nextjs.org/docs/pages/api-reference/components/image>

# 記事一覧ページ

## app/(public)/page.tsx

```
import { getPosts } from '@/lib/post'
import PostCard from '@/components/post/PostCard'
import { Post } from '@/types/post'

export default async function PostsPage(){
  const posts = await getPosts() as Post[]
  略
  {posts.map((post) => (
    <PostCard key={post.id} post={post} />
  )))
}
```

# 1記事の取得

## src/lib/post.ts

略

```
export async function getPost(id: string) {  
  return await prisma.post.findUnique({  
    where: { id },  
    include: { author: { select: { name: true, },  
      }, }, } ) }
```

# 記事詳細 app/(public)/posts/[id]/page.tsx

```
import {getPost} from '@/lib/post'
import {Card,CardContent,CardHeader,CardTitle} from '@/components/ui/card'
import {format} from 'date-fns'
import {ja} from 'date-fns/locale'
import {notFound} from 'next/navigation'
import Image from 'next/image'

type Params = {params: Promise<{id: string}>} // urlの情報はparamsで渡ってくる

export default async function PostPage({params}: Params){
    const {id} = await params // paramsはPromise型で定義しつつawaitで非同期処理
    const post = await getPost(id) // DBからid指定して情報取得

    if (!post) {
        notFound()
    }
}
```

# app/(public)/posts/[id]/not-found.tsx



```
export default function NotFound(){  
  return (  
    <div>  
      お探しの記事は存在しないか、削除された可能性があります。  
    </div>  
  )  
}
```

# 記事の検索クエリ

src/lib/post.ts 検索キーワード例 next.js prisma

```
export async function searchPosts(search: string){
```

```
// 全角スペースを半角スペースに変換しつつスペースで分割 (空文字などを除外)
```

```
const decodedSearch = decodeURIComponent(search)
```

```
const normalizedSearch = decodedSearch.replace(/[\s ]+/g, ' ').trim()
```

```
const searchWords = normalizedSearch.split(' ').filter(Boolean)
```

```
const filters = searchWords.map( word =>{
```

```
    OR: [ { title: { contains: word }}, { content: { contains: word }}, ]
```

```
})
```

```
return await prisma.post.findMany({
```

```
    where: { AND: filters }
```

```
    略
```

```
})}
```

# 検索ボックスとして分離 その1

src/components/post/SearchBox.tsx

```
'use client'

import { useState, useEffect } from 'react'
import { useRouter } from 'next/navigation'
import {Input} from '@/components/ui/input'
export default function SearchBox() {

    const [search, setSearch] = useState("");
    const [debouncedSearch, setDebouncedSearch] = useState("")
    const router = useRouter();
    // デバウンス (高頻度に呼び出されるのを防ぐ 500ms後に実行
    useEffect(() => {
        const timer = setTimeout(() => {
            setDebouncedSearch(search);
        }, 500);
        return () => clearTimeout(timer);
    }, [search]);
```

## 検索ボックス その2

```
// debouncedSearch が更新されたときに検索を実行
// スペースだけを入力した場合は/にリダイレクト、trimで不要なスペースを削除
useEffect(() => {
  if (debouncedSearch.trim()) {
    router.push(`/?search=${debouncedSearch.trim()}`);
  } else { router.push('/'); }
}, [debouncedSearch, router]);
return (
  <>
  <Input
    placeholder="記事を検索..."
    value={search}
    onChange={(e)=> setSearch(e.target.value)}
    className="w-[200px] lg:w-[300px]"
  /></>)
}
```

# 記事一覧ページの修正

Next.jsではsearchParamsでURLパラメータを取得できる

参考: <https://nextjs.org/docs/app/building-your-application/upgrading/version-15#params--searchparams>

**src/app/(public)/page.tsx**

```
import { searchPosts, getPosts } from '@/lib/post'
```

```
typeSearchParams = { search?: string }
```

```
export default async function PostsPage(  
  {searchParams}: {searchParams: Promise<SearchParams>}){
```

```
  const resolvedSearchParams = await searchParams
```

```
  const query = resolvedSearchParams.search || "
```

```
  const posts = query
```

```
    ? await searchPosts(query)
```

```
    : await getPosts()
```

# このセクションのまとめ



認証なしのページ

環境構築

Prisma設定・初期データ登録・スキーマ

shadcn/ui カードコンポーネント等

記事一覧ページ

記事詳細ページ

記事検索機能



# 認証

# Auth.js (旧NextAuth.js)



<https://authjs.dev/>

元々 Next.js向けのライブラリ 認証・セッション管理

Next.js以外にも対象拡大 v5時に Auth.jsに名称変更

認証方式

OAuth (Google, X, GitHub, Facebookなど50以上), JWT,

EmailCredentials

セッション永続化 対応DB MongoDB, PostgreSQL, MySQL, SQLite...

# Learn Next.js



Learn Next.jsのChapter 15 をベースに  
一部改良しています

[https://nextjs.org/learn/dashboard-  
app/adding-authentication](https://nextjs.org/learn/dashboard-app/adding-authentication)

# インストール & シークレットキー生成



インストール方法 `npm install next-auth@beta`

`npm install zod@^3`

シークレットキー生成

`npx auth secret`

.env.local ファイルに AUTH\_SECRET が発行される

.env に統合

```
import type { NextAuthConfig } from 'next-auth';
```

## src/auth.config.ts 認証設定ファイル

```
export const authConfig = {  
  pages: { signIn: '/login' },  
  callbacks: {  
    authorized({ auth, request: { nextUrl } }) { // authはユーザーセッションが含まれる  
      const isLoggedIn = !!auth?.user; // ユーザーがログインしているか  
      const isOnDashboard = nextUrl.pathname.startsWith('/dashboard') || nextUrl.pathname.startsWith('/manage')  
      if (isOnDashboard) {  
        if (isLoggedIn) return true;  
        return false; // ログインしてなければloginページにリダイレクト  
      } else if (isLoggedIn && nextUrl.pathname === '/login') {  
        return Response.redirect(new URL('/dashboard', nextUrl));  
      }  
      return true;  
    },  
  },  
  providers: [], // ログインオプション auth/index.ts側で設定  
} satisfies NextAuthConfig;
```

# src/middleware.ts

auth.configファイルを読み込んでAuth.jsを初期化

authプロパティをエクスポート

```
import NextAuth from 'next-auth';
```

```
import { authConfig } from './auth.config';
```

```
export default NextAuth(authConfig).auth;
```

```
export const config = {
```

```
    // https://nextjs.org/docs/app/building-your-application/routing/
```

```
    middleware#matcher
```

```
        matcher: [ '/((?!api|_next/static|_next/image|.*\\.png$).*)' ],
```

```
};
```

# src/auth.ts 認証ファイル

auth.configの内容を展開する

Credentials(認証の種類)を設定できるようにする

```
import NextAuth from 'next-auth';
```

```
import { authConfig } from './auth.config';
```

```
import Credentials from 'next-auth/providers/credentials';
```

```
export const { auth, signIn, signOut, handlers } = NextAuth({  
  ...authConfig,  
  providers: [Credentials({})],  
});
```

```
import { z } from 'zod';
import { prisma } from './lib/prisma'; import bcryptjs from 'bcryptjs';

async function getUser(email: string) { // ユーザー取得関数
  return await prisma.user.findUnique({
    where: { email: email }})
}

Credentials({
  async authorize(credentials) {
    // メールアドレスとパスワードをZodで検証
    const parsedCredentials = z
      .object({ email: z.string().email(), password: z.string().min(8) })
      .safeParse(credentials);

    if (parsedCredentials.success) {
      const { email, password } = parsedCredentials.data;
      const user = await getUser(email); // ユーザー取得
      if (!user) return null;

      const passwordsMatch = await bcryptjs.compare(password, user.password); // パスワード比較
      if (passwordsMatch) return user;
    }
    return null;
}
```

ユーザー情報 | null を返却  
バリデーション  
ユーザー情報取得  
パスワードを比較

# 認証ロジック src/lib/actions/authenticate.ts

```
'use server';

import { signIn } from '@/auth'; // signIn関数のインポート
import { AuthError } from 'next-auth';
export async function authenticate(
  prevState: string | undefined,
  formData: FormData,
) {
  try {
    await signIn('credentials', formData);
  } catch (error) {
    if (error instanceof AuthError) {
      switch (error.type) {
        case 'CredentialsSignin':
          return 'メールアドレスまたはパスワードが正しくありません。';
        default:
          return 'エラーが発生しました。';
      }
    } throw error; }}
```

ログインフォーム用  
ServerAction



# ログイン処理

## 認証関連のページ



app/(auth)/layout.tsx 認証レイアウト

app/(auth)/login/page.tsx ログインページ

app/(auth)/register/page.tsx ユーザー登録

components/auth/LoginForm.tsx

components/auth/RegisterForm.tsx

# レイアウト

src/app/(auth)/layout.tsx

```
export default function AuthLayout({  
    children,  
}: Readonly<{  
    children: React.ReactNode;  
}>) {  
  
    return (  
        <div className="min-h-screen flex items-center justify-center p-4">  
            {children}  
        </div>  
    );  
}
```

# ログインページ



src/app/(auth)/login/page.tsx

```
import LoginForm from "@/components/auth/LoginForm"
```

```
export default function LoginPage() {
```

```
    return (
```

```
        <LoginForm />
```

```
    )
```

```
}
```

# src/components/auth/LoginForm.tsx



```
'use client'

import { Button } from "@/components/ui/button"
import { Input } from "@/components/ui/input"
import { Label } from "@/components/ui/label"
import { Card,CardContent,CardHeader,CardTitle } from "@/components/ui/card"
import { useActionState } from 'react'
import { authenticate } from '@/lib/actions/authenticate' // ServerAction

export function LoginForm() {
  const [errorMessage, formAction, isPending] = useActionState(
    authenticate,
    undefined,
  );
  return ( <form action={formAction}></form>
}
```



# ログアウト処理

```
import Link from "next/link";
import {NavigationMenu, NavigationMenuItem, NavigationMenuLink, NavigationMenuList} from "@/components/ui/navigation-menu";
import Setting from "./Setting";
import { auth } from "@/auth"; // 認証情報
```

src/components/layouts/PrivateHeader.tsx

```
export default async function PrivateHeader() {
  const session = await auth(); // サーバーサイドでセッション情報を取得
  if (!session?.user?.email) throw new Error("不正なリクエストです");

  return (
    <Link href="/dashboard" legacyBehavior passHref>
      <NavigationMenuLink className="font-bold text-xl">
        管理ページ
      </NavigationMenuLink>
    </Link>
  )
}
```

```
import { signOut } from "@/auth" // ログアウト処理
import { Button } from "@/components/ui/button"
import { DropdownMenu, DropdownMenuContent, DropdownMenuItem, DropdownMenuTrigger } from "@/components/ui/dropdown-menu"
import { Session } from "next-auth" // Session 型をインポート

export default function Setting({session}: {session: Session}) {
  const handleLogout = async () => {
    'use server'
    await signOut() // CSRF対応済み
  }

  return (
    <DropdownMenu>
      <DropdownMenuTrigger asChild>
        <Button variant="ghost" className="font-medium">
          {session.user?.name}
        </Button>
      </DropdownMenuTrigger>
      <DropdownMenuContent align="end" className="w-48">
        <DropdownMenuItem onClick={handleLogout} className="cursor-pointer">
          ログアウト
        </DropdownMenuItem>
      </DropdownMenuContent>
    </DropdownMenu>
  )
}
```

src/components/layouts/Setting.tsx

# 不具合修正 (適切にリダイレクトをかける)

## src/auth.config.ts

```
if (isOnDashboard) {  
  if (isLoggedIn) return true;  
  return Response.redirect(new URL('/login', nextUrl)); // 修正箇所
```

## src/lib/actions/authenticate.ts

```
import { redirect } from 'next/navigation'  
try {  
  await signIn('credentials', {  
    ...Object.fromEntries(formData),  
    redirect: false, // 自動リダイレクトを無効化  
  });
```

```
// 認証成功後に手動でリダイレクト  
redirect('/dashboard');
```



# ユーザー登録

# ユーザー登録



ユーザー登録機能はAuth.jsにはない  
以前のセクションで作成した  
コンタクトフォームも参考に作っていきます

# src/validations/user.ts バリデーション

```
import { z } from "zod"

export const registerSchema = z.object({
    name: z.string().min(1, "名前は必須です"),
    email: z.string({ required_error: "メールアドレスは必須です" })
        .min(1, "メールアドレスは必須です")
        .email("不正なメールアドレスです"),
    password: z.string({ required_error: "パスワードは必須です" })
        .min(1, "パスワードは必須です")
        .min(8, "パスワードは最低8文字必要です")
        .max(32, "パスワードは最大32文字以内にしてください"),
    confirmPassword: z.string({ required_error: "確認用パスワードは必須です" })
        .min(1, "確認用パスワードは必須です")
}).refine((data) => data.password === data.confirmPassword, {
    message: "パスワードが一致しません",
    path: ["confirmPassword"], // エラーを表示するフィールドを指定
});
```

# TypeScript



Record<K, V> ユーティリティ型の1つ  
エラーの内容をまとめる時などに便利  
K・・キー, V・・バリュー(値)

Record<string, string[]>  
キーは文字列, 値は 文字列の配列

# lib/actions/createUser.ts その1

```
'use server'

import { registerSchema } from "@/validations/user"
import bcryptjs from "bcryptjs"
import { prisma } from "@/lib/prisma"
import { redirect } from "next/navigation"

// ActionStateの型定義
type ActionState = { success: boolean; errors: Record<string, string[]>;}

export async function createUser(prevState: ActionState, formData: FormData): Promise<ActionState> {
    const rawFormData = Object.fromEntries( // 4つの値を取得
        ["name", "email", "password", "confirmPassword"].map((field) => [
            field,
            formData.get(field) as string,
        ])
    ) as Record<string, string>;
```

# lib/actions/createUser.ts その2

関数

```
// バリデーションエラー処理

function handleValidationError(error: any): ActionState {
    const { fieldErrors, formErrors } = error.flatten();
    // zodの仕様でパスワード一致確認のエラーは formErrorsで渡ってくる
    // formErrorsがある場合は、 confirmPasswordフィールドにエラーを追加
    if (formErrors.length > 0) {
        return { success: false, errors: { ...fieldErrors, confirmPassword: formErrors
            }}}
    return { success: false, errors: fieldErrors };
}

// カスタムエラー処理

function handleError(customErrors: Record<string, string[]>): ActionState {
    return { success: false, errors: customErrors };
}
```

# lib/actions/createUser.ts その3

```
// バリデーション  
  
const validationResult = registerSchema.safeParse(rawFormData);  
if (!validationResult.success) { return handleValidationError(validationResult.error) }  
  
// メールアドレスが既に登録されているか確認  
  
const existingUser = await prisma.user.findUnique({where: { email:  
rawFormData.email }})  
  
if (existingUser) {  
    return handleError({ email: ['このメールアドレスはすでに登録されています'] })  
}  
  
// パスワードのハッシュ化  
  
const hashedPassword = await bcryptjs.hash(rawFormData.password, 12)
```

## lib/actions/createUser.ts その4

```
// ユーザー登録

await prisma.user.create({
  data: {
    name: rawFormData.name,
    email: rawFormData.email,
    password: hashedPassword, }})

// リダイレクトまたは成功メッセージの返却

await signIn('credentials', {
  ...Object.fromEntries(formData),
  redirect: false, // 自動リダイレクトを無効化
});

redirect('/dashboard')
```

## ユーザー登録画面

src/app/(auth)/register/page.tsx · · URL用

src/components/auth/RegisterForm.tsx · · コンポーネント

レイアウトはLoginForm.tsxとほぼ同じ

'use client'

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

```
import { createUser } from "@/lib/actions/createUser";
```

```
export function RegisterForm() {
```

```
  const [state, formAction] = useState(createUser, {
```

```
    success: false, errors: {}
```

```
});
```

```
<form action={formAction} className="space-y-4">
```

```
{state.errors.name && (
```

```
  <p className="text-red-500 text-sm mt-1">{state.errors.name.join(',')}</p>
```

```
)}
```

# npm run build

LoginForm の isPendingを削除

src/lib/action/createUser.tsで型エラー

```
import { ZodError } as 'zod'
```

```
function handleValidationError(error: ZodError): ActionState {
  const { fieldErrors, formErrors } = error.flatten();
  // fieldErrorsにundefinedが入らないようにキャスト
  const castedFieldErrors = fieldErrors as Record<string, string[]>

  if (formErrors.length > 0) {
    return { success: false, errors: {
      ...fieldErrors, confirmPassword: formErrors
    } }
  }

  return { success: false, errors: castedFieldErrors };
}
```

# このセクションのまとめ



Auth.js (旧NextAuth.js)

ログイン機能・ログアウト機能

ユーザー登録



# CRUD

# CRUDとRESTful API

データ操作の一連の方法 (Create, Read, Update, Delete)

HTTPメソッド	URI	画面・機能	CRUD
GET	/dashboard (今回) /posts	一覧画面	Read
GET	/posts/create	作成画面	-
POST	/posts	保存	Create
GET	/posts/[id]	詳細画面	Read
GET	/posts/[id]/edit	編集画面	Read
PUT/PATCH	/posts/[id]	更新	Update
DELETE	/posts/[id]	削除	Delete



# 記事一覧画面

# sessionにid情報も含める

現在のsessionはname, emailのみ token.subにidが含まれているので  
session.user.idで取得できるようにしておく

## src/auth.ts

```
providers: [略],  
callbacks: {  
  async session({ session, token }) {  
    if (session.user) {  
      session.user.id = (token.id || token.sub || '') as string;  
      session.user.name = token.name ?? '';  
      session.user.email = token.email ?? '';  
    }  
    return session;  
  }  
}
```

# ログインしているユーザーの記事のみ表示

## src/lib/ownPost.ts

```
import { prisma } from '@/lib/prisma';
export async function getOwnPosts(userId: string) {
    return await prisma.post.findMany({
        where: {
            authorId: userId
        },
        select: {
            id: true,
            title: true,
            published: true,
            updatedAt: true,
        },
        orderBy: { createdAt: "desc" },
    })
}
```

# 一覧ページ

```
import PostDropdownMenu from "@/components/post/PostDropdownMenu";
import { Button } from "@/components/ui/button";
import Link from "next/link";
import { getOwnPosts } from '@/lib/ownPost'
import { auth } from '@/auth'

export default async function DashboardPage() {
    const session = await auth()
    const userId = session?.user?.id;
    if (!session?.user?.email || !userId) {
        throw new Error("不正なリクエストです");
    }
    const posts = await getOwnPosts(userId);
    return (
        <Button>
            <Link href="/manage/posts/create">新規記事作成</Link>
        </Button>
    )
}
```

# 一覧ページ 続き

```
{posts.map((post) => (
  <tr key={post.id}> {/* 繰り返しはkeyをつけておく */}
    <td className="border p-2">{post.title}</td>
    <td className="border p-2 text-center">
      {post.published ? '表示' : '非表示'} {/* boolean 表示変換 */}
    </td>
    <td className="border p-2 text-center">
      {new Date(post.updatedAt).toLocaleString()}
    </td>
    <td className="border p-2 text-center">
      <PostDropdownMenu postId={post.id} />
    </td>
  </tr>
))}
```

# Dropdown (詳細・編集・削除)

src/components/post/PostDropdownMenu.tsx

```
<DropdownMenu>
  <DropdownMenuTrigger className="px-2 py-1 border rounded-md">…</
  DropdownMenuTrigger>
  <DropdownMenuContent>
    <DropdownMenuItem asChild>
      <Link href={`/posts/${postId}`} className="cursor-pointer">詳細</Link>
    </DropdownMenuItem>
    <DropdownMenuItem asChild>
      <Link href={`/posts/edit/${postId}`} className="cursor-pointer">編集</Link>
    </DropdownMenuItem>
    <DropdownMenuItem className="text-red-600 cursor-pointer">削除</
    DropdownMenuItem>
  </DropdownMenuContent>
</DropdownMenu>
```



# 記事作成画面

# 記事作成画面



やりたいこと

Markdownで執筆・プレビュー表示

文字数カウント

バリデーション

トップ画像のアップロード

画像保存先はローカル環境と本番環境で切り替える

(追って記事編集・更新も控えているので共有できる箇所は共有する)

# Markdown関連ライブラリ



```
npm install react-markdown@^9 remark-gfm@^4 rehype-highlight@^7 react-textarea-autosize@^8 @tailwindcss/typography@^0
```

react-markdown MarkdownをReactコンポーネントとして表示  
remark-gfm (GitHub Flavored Markdown)対応(Markdownの拡張機能)  
rehype-highlight コードハイライト表示  
react-textarea-autosize textarea自動調整  
tailwindcss/typography TailwindCSS公式プラグイン Markdownをキレイにスタイルリングするクラスを提供

# tailwind.config.tsに追加



```
// typographyを追加  
plugins: [  
    require("tailwindcss-animate"),  
    require("@tailwindcss/typography")],
```

# 記事作成画面

```
'use client'

import { useState, useActionState } from "react";
import createPost from "@/lib/actions/createPost";
import ReactMarkdown from "react-markdown";
import remarkGfm from "remark-gfm";
import rehypeHighlight from "rehype-highlight";
import TextareaAutosize from "react-textarea-autosize";
import "highlight.js/styles/github.css"; // コードハイライト用のスタイル

const [content, setContent] = useState("") // 記事の文章
const [contentLength, setContentLength] = useState(0) // 文字数
const [preview, setPreview] = useState(false) // プレビュー

const handleContentChange = (e: React.ChangeEvent<HTMLTextAreaElement>) => {
    const value = e.target.value;
    setContent(value);
    setContentLength(value.length) };
const [state, formAction ] = useActionState(createPost, { success: false, errors: {} })
```

```
const [preview, setPreview] = useState(false);
<Label htmlFor="content">内容(Markdown)</Label>
<TextareaAutosize
  id="content" value={content} name="content"
  onChange={(e) => setContent(e.target.value)}
  className="w-full border p-2"
  minRows={8}
  placeholder="Markdown形式で入力してください"
/>
```

```
{preview && (
  <div className="border p-4 bg-gray-50 prose max-w-none">
    <ReactMarkdown
      remarkPlugins={[remarkGfm]}
      rehypePlugins={[rehypeHighlight]}
      skipHtml={false} // HTMLスキップを無効化
      unwrapDisallowed={true} // Markdownの改行を解釈
    >
      {content}
    </ReactMarkdown>
  </div>
)}
```

## 記事作成画面

app/(private)/manage/posts/create/  
page.tsx

ReactMarkdownを使って  
Markdownのプレビュー  
proseクラスを使うことでキレイに描画

# Markdownサンプル

## 列の揃え方

左寄せ	中央揃え	右寄せ	
:----- :-----: -----:			
左揃えの値	中央の値	右揃えの値	
テスト1	テスト2	テスト3	
サンプル1	サンプル2	サンプル3	

## チェックリストの例

- [ ] 未完了のタスク1
- [x] 完了済みのタスク2
- [ ] 未完了のタスク3

```python

# Pythonの例

```
def greet(name):  
    return f"Hello, {name}!"
```

```
print(greet("World"))
```

```

# 画像保存関連 UI側

```
<div>
    <Label htmlFor="topImage">トップ画像</Label>
    <Input
        type="file"
        id="topImage"
        accept="image/*"
        name="topImage"
    />
</div>
```

# 画像保存 処理



1. ブラウザのFileオブジェクト(arrayBuffer形式)

2. ファイルのデータ変換 (保存するため)

file.arrayBuffer() で arrayBufferを取得

Buffer.from(arrayBuffer) で Node.jsのBufferに変換

3. Node.js側で保存

writeFileでバイナリデータ(Buffer)をファイルに書き込む

arrayBuffer ブラウザやNode.jsでも使えるデータ構造

Buffer (Node.js独自クラス) fsなどからデータを読み込むときに使う

writeFile (fsの関数 ファイル保存)

# src/utils/image.ts

```
import { writeFile } from "fs/promises";
import path from "path";

export async function saveImage(file: File): Promise<string | null> {
    const buffer = Buffer.from(await file.arrayBuffer()); // バイナリデータをBufferに変換
    const fileName = `${Date.now()}_${file.name}`; // ファイル名生成 日時_ファイル名
    const uploadDir = path.join(process.cwd(), "public/images"); // アップロードフォルダ

    try {
        const filePath = path.join(uploadDir, fileName); // 保存先の完全なファイル名
        await writeFile(filePath, buffer); // 指定パスにファイル(buffer)を書き込む
        return `/images/${fileName}`; // URLパスを返す
    } catch (error) {
        console.error("画像保存エラー:", error);
        return null;
    }
}
```

# 画像サイズ拡張



next.config.ts デフォルト1MBのため拡張

```
images: {略},
```

```
experimental: {
```

```
serverActions: {
```

```
  bodySizeLimit: '5mb', // 必要に応じて値を変更
```

```
},
```

```
},
```

# バリデーション



## src/validations/post.ts

```
import { z } from "zod"

export const postSchema = z.object({
    title: z.string()
        .min(3, { message: "タイトルは3文字以上で入力してください" })
        .max(255, { message: "タイトルは255文字以内で入力してください" }),
    content: z.string()
        .min(10, { message: "内容は10文字以上で入力してください" }),
    topImage: z.instanceof(File).nullable().optional()
});
```

# src/lib/actions/createPost.ts

```
export async function createPost(){
    // フォームの情報を取得
    // バリデーション
    // 画像保存

    const imageUrl = toplmage ? await savelmage(toplimage) : null;
    if (toplimage && !imageUrl) {
        return { success: false, errors: { image: ['画像の保存に失敗しました'] } };
    }

    // DB登録
}
```



# 記事詳細画面

# 記事取得クエリ src/lib/ownPost.ts

```
export async function getOwnPost(userId: string, postId: string) {  
    return await prisma.post.findFirst({  
        where: {  
            AND: [  
                { authorId: userId },  
                { id: postId }  
            ]  
        },  
        select: {  
            id: true,  
            title: true,  
            content: true,  
            topImage: true,  
            published: true,  
            createdAt: true, updatedAt: true,  
        }  
    })
```

# src/components/posts/PostDropdownMenu.tsx

略

```
export default function PostDropdownMenu({ postId }: { postId: string }) {
  return (
    <DropdownMenu>
      <DropdownMenuTrigger className="px-2 py-1 border rounded-md"></
    DropdownMenuTrigger>
      <DropdownMenuContent>
        <DropdownMenuItem asChild>
          <Link href={`/manage/posts/${postId}`} className="cursor-pointer">
            詳細
          </Link>
        </DropdownMenuItem>
      </DropdownMenuContent>
    </DropdownMenu>
  )
}
```

# app/(private)/manage/posts/[id]/page.tsx

```
type PageProps = { params: Promise<{ id: string }>};
```

```
export default async function PostPage({ params }: PageProps) {
  const { id } = await params;
```

```
const session = await auth()
const userId = session?.user?.id
if(!session?.user?.email || !userId){
  throw new Error('不正なリクエストです')
}
```

```
const post = await getOwnPost(userId, id)
(public)/posts/[id]/page.tsxも合わせて調整する
```



# 記事編集画面

# 記事 編集画面



記事登録画面のUI と記事詳細画面の記事取得処理  
を組み合わせる

DB取得は非同期関数コンポーネント (RSC)  
Reactフック(useActionStateなど) (RCC)  
の併用は不可 ファイルを分離して対応

src/app/(private)/manage/posts/[id]/edit/page.tsx (RSC)  
同階層に EditPostForm.tsx を作成 (RCC)

# src/app/(private)/manage/posts/[id]/edit/page.tsx

```
import EditPostForm from "./EditPostForm"
import { auth } from "@/auth";
import { getOwnPost } from "@/lib/ownPost";
import { notFound } from "next/navigation";

type PageProps = { params: Promise<{ id: string }> }

export default async function EditPage({ params }: PageProps) {
  const { id } = await params;
  const session = await auth();
  const userId = session?.user?.id;
  if (!session?.user?.email || !userId) { throw new Error("不正なリクエストです") }
  const post = await getOwnPost(userId, id);
  if (!post) { notFound() }

  return ( <div><EditPostForm post={post} /></div> );
}
```

# EditPostForm.tsx

```
// propsで受け取る 型指定
type EditPostFormProps = {
  post: {
    id: string; title: string; content: string; topImage?: string | null ; published: boolean
  }
};

// タイトル、表示非表示、画像のプレビューもuseStateで変更できるように
const [title, setTitle] = useState(post.title);
const [published, setPublished] = useState(post.published);
const [imagePreview, setImagePreview] = useState(post.topImage);

<Input value={title} onChange={(e) => setTitle(e.target.value)} />
<input type="hidden" name="postId" value={post.id} /> // 記事id
<input type="hidden" name="oldImageUrl" value={post.topImage || ''} /> //DB内画像url
```

# 画像の編集・更新について

input type="file" にはDB内の画像urlを設定できない  
->そのまま更新すると画像urlが消える可能性

対策:

DB内の画像urlをinput type="hidden"で持たせる  
画像が変更されたら新しい画像を保存する  
プレビューを表示し現在の画像を表示する  
etc...

# 画像プレビュー機能

```
const handleImageChange = async (e: React.ChangeEvent<HTMLInputElement>) => {
  const file = e.target.files?.[0];
  if (file) {
    // プレビュー用URL生成 ブラウザのメモリに保存される
    const previewUrl = URL.createObjectURL(file)
    setImagePreview(previewUrl);
  }
}

// プレビューURLはブラウザのメモリに保存される
// コンポーネントが破棄されるかimagePreview変更時に プレビューURLを解放
useEffect(() => {
  return () => {
    if (imagePreview && imagePreview !== post.topImage) {
      URL.revokeObjectURL(imagePreview); // プレビューurlはブラウザのメモリに保存される
    }
  };
}, [imagePreview, post.topImage]);
```

# 表示非表示



<https://ui.shadcn.com/docs/components/radio-group>

npx shadcn@latest add radio-group

表示非表示の切り替え用

```
<RadioGroup value={published.toString()} name="published"  
onValueChange={(value) => setPublished(value === 'true')}>
```



# 記事更新機能

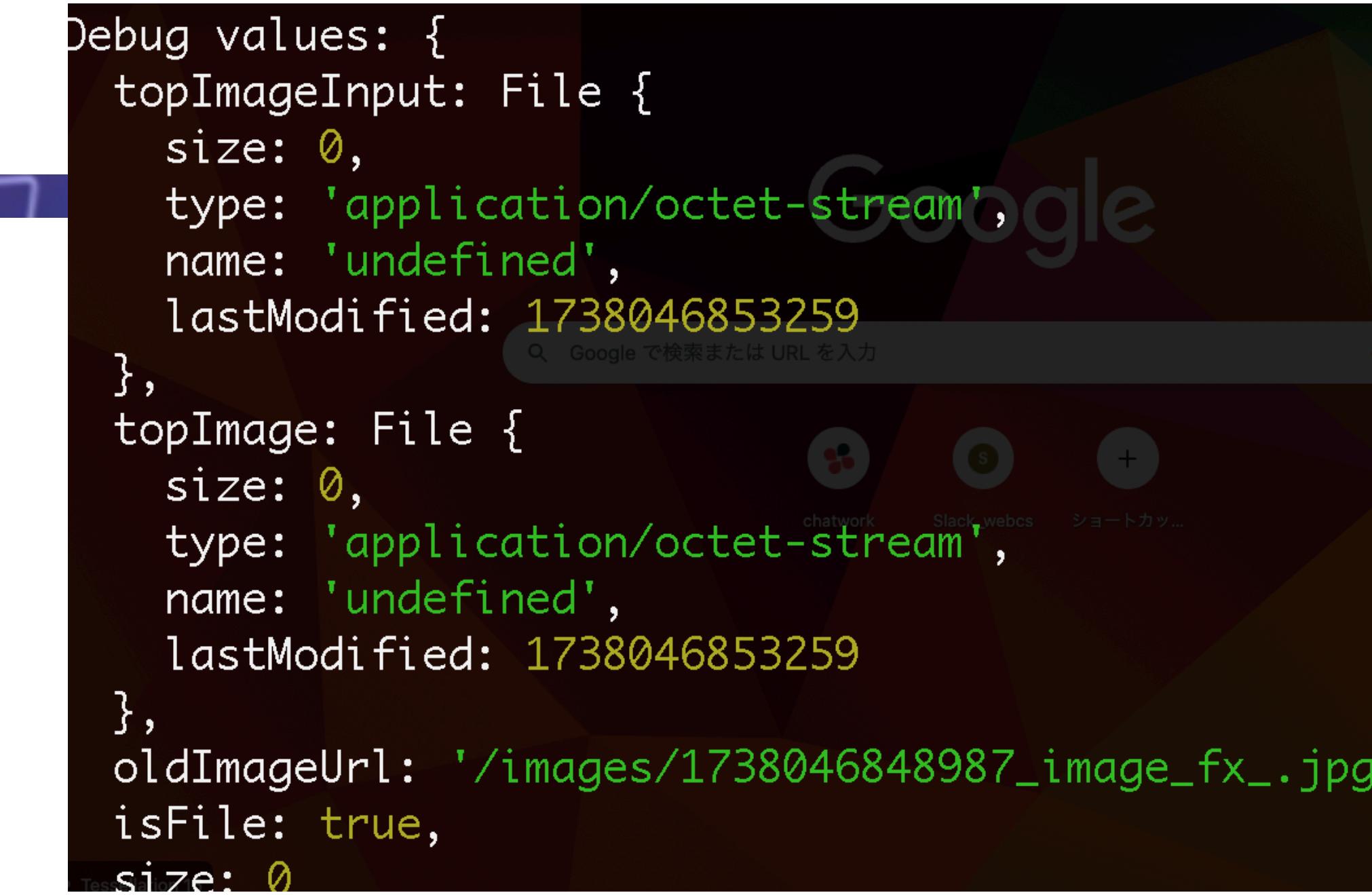
# 記事 更新機能

src/lib/actions/updatePost.ts

```
const postId = formData.get("postId") as string
const published = formData.get("published") === 'true'
const oldImageUrl = formData.get("oldImageUrl") as string
```

// 画像保存

```
let imageUrl = oldImageUrl;
if (topImage instanceof File && topImage.size > 0 && topImage.name !== 'undefined') {
    const newImageUrl = await saveImage(topImage);
    if (!newImageUrl) {
        return { success: false, errors: { image: ['画像の保存に失敗しました'] } };
    }
    imageUrl = newImageUrl;
}
```



# DB情報の更新



```
// DB更新  
const updatedPost = await prisma.post.update({  
    where: { id: postId },  
    data: {  
        title,  
        content,  
        published,  
        topImage: imageUrl,  
    },  
})
```



# 記事削除機能

# 記事削除機能 その1

削除 -> 確認用のダイアログを表示 削除を選択した場合はDBから削除

ドロップダウンとダイアログの組み合わせ useStateで状態管理 必要

**src/components/post/PostDropdownMenu.tsx**

```
import { DeletePostDialog } from '@/components/post/DeletePostDialog';
import { useState } from 'react';

const [isDropdownOpen, setIsDropdownOpen] = useState(false);
const [showDeleteDialog, setShowDeleteDialog] = useState(false);
// 削除ダイアログの状態が変わったら、ドロップダウンの状態をリセット
const handleDeleteDialogChange = (open: boolean) => {
  setShowDeleteDialog(open);
  if (!open) { setIsDropdownOpen(false) }}
```

## 記事削除機能 その2

```
<DropdownMenu open={isDropdownOpen} onOpenChange={setIsDropdownOpen}>
// 削除を選んだらドロップダウンはfalse, ダイアログはtrue
<DropdownMenuItem className="text-red-600"
onSelect={() => {
  setIsDropdownOpen(false);
  setShowDeleteDialog(true); }}>削除</DropdownMenuItem>

{showDeleteDialog && (
  <DeletePostDialog
    postId={postId}
    isOpen={showDeleteDialog}
    onOpenChange={handleDeleteDialogChange}
  />
)}
```

# src/components/post/DeletePostDialog.tsx

```
type DeletePostProps = {  
  postId: string;  
  isOpen: boolean;  
  onOpenChange: (open: boolean) => void; }  
  
export function DeletePostDialog({ postId, isOpen, onOpenChange }:  
  DeletePostProps) {  
  
  <AlertDialog open={isOpen} onOpenChange={onOpenChange}>  
    <AlertDialogAction  
      onClick={()=> deletePost(postId)}>  
      削除する</AlertDialogAction>  
  </AlertDialog>  
}
```

# src/lib/actions/deletePost.ts

```
'use server'

import { redirect } from "next/navigation"
import { prisma } from '@/lib/prisma'

type ActionState = {
    success: boolean;
    errors: Record<string, string[]>
}

export async function deletePost(postId: string): Promise<ActionState> {
    await prisma.post.delete({ where: { id: postId } })
    redirect('/dashboard')
}
```

# このセクションのまとめ



認証なし画面・・記事一覧、記事の詳細

認証関連・・ログイン・ユーザー登録・ログアウト

認証あり画面・・一覧、作成、保存、詳細、編集、更新、削除

shadcn/ui, React フック(useState, useEffect,

useStateAction), serverAction でサーバー側の処理,

prisma ORM、zod型バリデーション、Auth.js認証

画像アップロード、Markdown表示、プレビュー etc...