**Prisma Notes**

We are using **Neon** database which is a cloud database.

It is basically for if you want to use postgres Sql.

To install prisma: **‘npm i –D prisma @prisma/client’**

To create a prisma schema file: ‘**npx prisma init’**

Make sure to add database url in **.env** file

After creating a model, you must add a script in package.json file:

"scripts": {

"postinstall": "prisma generate"

},

Then you run it locally: **‘npx prisma generate’**

This will generate prisma Client.

We must migrate our data to database:

**‘npx prisma migrate dev --name list’**

This our database will be created.

To see the database of postgres SQL : **‘npx prisma studio’**

This command will show the database tables.

Server Actions are those that are created on server, in order to use it we must **‘use server’**

To write query in prisma we use queryRaw from prisma.

const salesDataRaw = await prisma.$queryRaw<

Array<{ month: string; totalSales: Prisma.Decimal }>

>`SELECT to\_char("createdAt", 'MM/YY') as "month", sum("totalPrice") as "totalSales" FROM "Order" GROUP BY to\_char("createdAt", 'MM/YY')`;

**In prisma methods we have:**

**findFirst, findMany, count (please view order.actions.ts in lib/actions)**

const data = await prisma.user.findMany({

where: {

...queryFilter,

},

orderBy: { createdAt: "desc" },

take: limit,

skip: (page - 1) \* limit,

});

skip means?

In Prisma (and databases generally), skip means:

How many records to skip before starting to return results.

This is used for pagination.

Example:

page = 1, limit = 10 → skip = 0 → return first 10

page = 2, limit = 10 → skip = 10 → skip first 10, return next 10

page = 3, limit = 10 → skip = 20 → skip first 20, return next 10

**Zod**

* It offers typescript-first schema validation with static type inference.
* Offers runtime validation.

In order to use Zod, we first install it, then we create a schema using zod:

import { z } from "zod";

// Schema for inserting products

export const insertProductSchema = z.object({

name: z.string().min(3, "Name must be at least 3 characters"),

slug: z.string().min(3, "Slug must be at least 3 characters"),

category: z.string().min(3, "Category must be at least 3 characters"),

brand: z.string().min(3, "Brand must be at least 3 characters"),

description: z.string().min(3, "Description must be at least 3 characters"),

stock: z.coerce.number(),

images: z.array(z.string()).min(1, "Product must have at least one image"),

isFeatured: z.boolean(),

banner: z.string().nullable(),

});

**To use this schema we use it like this:**   
import { z } from "zod";

import { insertProductSchema } from "@/lib/validators";

export type Product = z.infer<typeof insertProductSchema> & {

id: string;

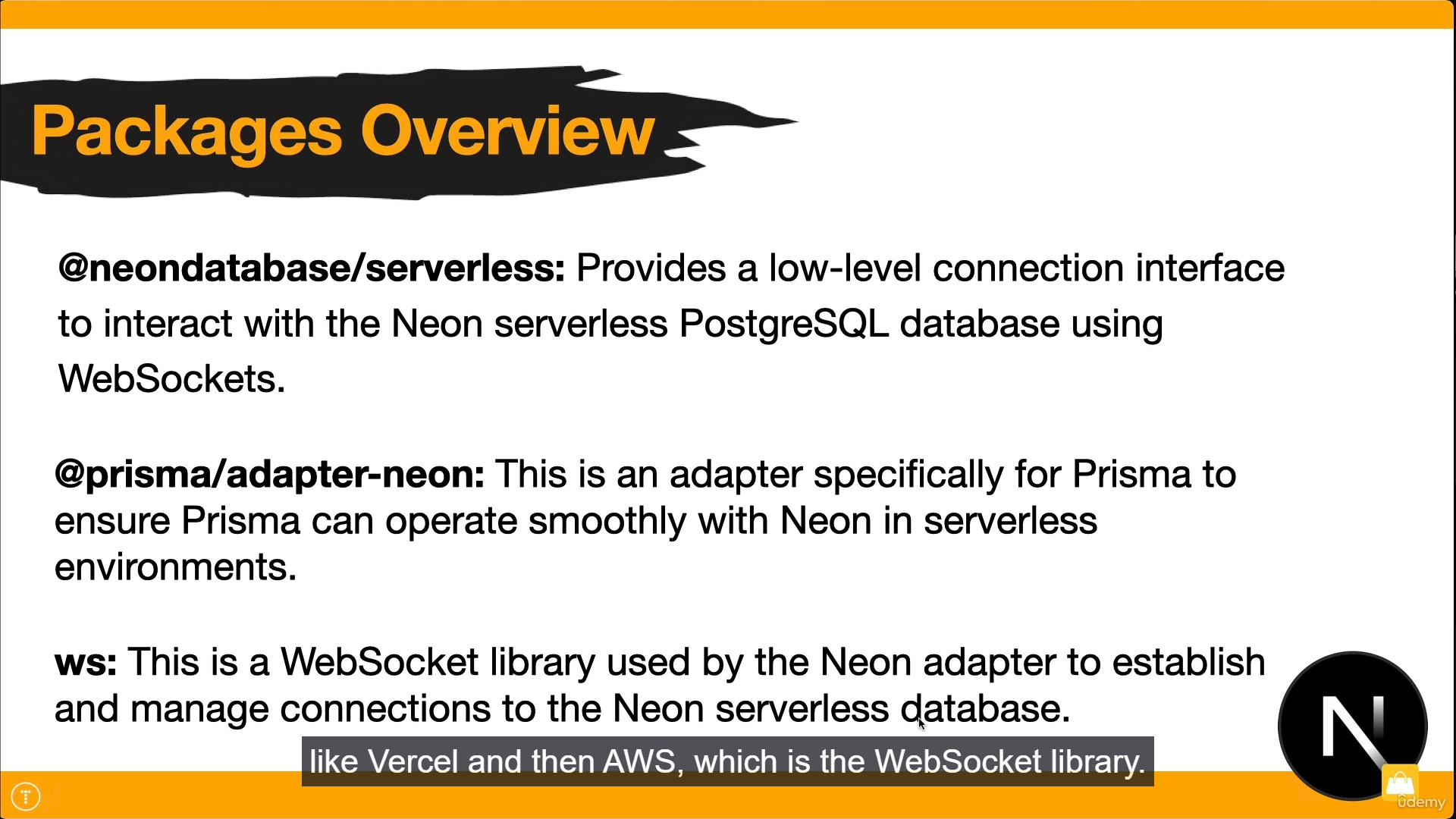
}

**Serverless Environment**

A **serverless environment** is a cloud computing model where I don’t have to manage any servers. The cloud provider automatically handles the infrastructure, scaling, and server maintenance behind the scenes.

My code is deployed as functions, and these functions run only when triggered—like when an HTTP request comes in. Once the task is complete, the function shuts down. This makes it **cost-efficient and scalable.**

For example, I used **Prisma** with a **serverless** PostgreSQL database from Neon, and deployed my API routes on Vercel. When a user makes a request, a Vercel function is triggered, connects to Neon using Prisma, runs the database query, sends the response, and then shuts down. I didn’t have to manage any server or worry about traffic spikes—everything scales automatically.



**Authentication with Next Auth**

import NextAuth from "next-auth";

export const config = {};

**/\*\***

**\* Handlers:**

**\* It is an object that contains the HTTP Handlers for different endpoints in Next Auth uses.**

**\***

**\*/**

export const { handlers, auth, signIn, signOut } = NextAuth(config);

**Use Action State:**

It is used to handle form submissions.

A new react hook called **‘use action state’**

**First argument:** Previous State

**Second argument:** Actual formData

Often used in server actions (Next.js) to process and respond to forms easily.

it’s the ideal tool for modern form handling when you're submitting data to the server and want instant feedback in the UI.

Also keeps the code clean.

**Also Use Form Status**

**Difference between Use Action State and Use Form Status?**

**useActionState**: React 19 Feature, in React 18 it was called useFormState.

**useFormStatus:** React DOM Feature, can be used to have a user experience like showing a loading or a spinner when user is performing an action for example when user signs in using sign in button , show a spinner that user’s request is sent and will soon show result of an action.

**useTransition Hook: (React 19)**

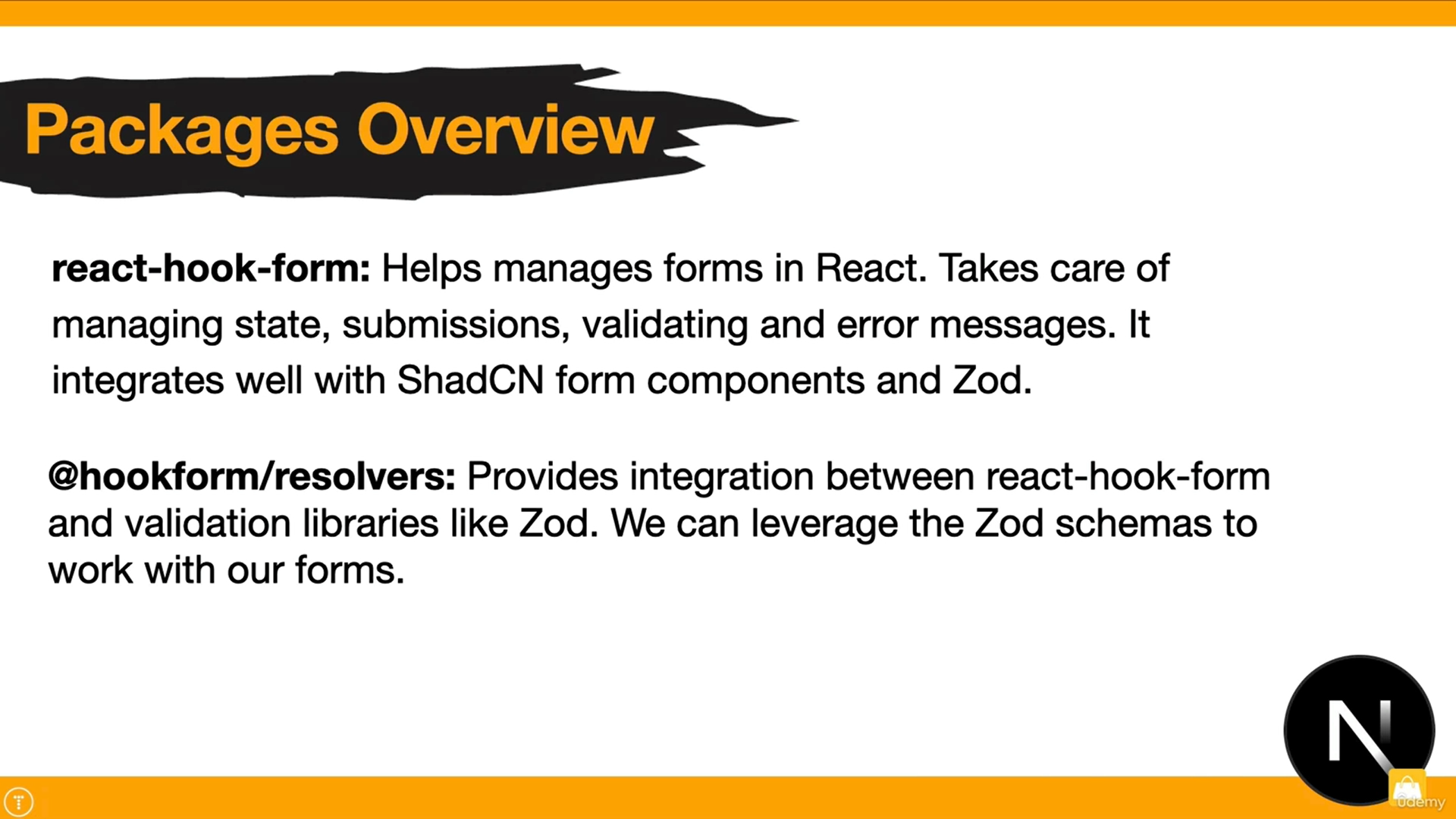
It helps manage the UI updates by transitioning.

Like showing a loading spinner while that update is happening.

It provides two values:

**isPending:** which indicates whether that transition is ongoing.

**startTransition:** a function that starts a transition.



**SubmitHandler Hook:**

* type-safe submit function that only runs with validated data.
* You avoid runtime errors from wrong field names/types.

**Transactions:**

* This will ensure that if any part fails, the entire transaction is rolled back
* If all operations succeed → MongoDB commits the transaction (saves everything).
* If any operation fails → MongoDB aborts the transaction (rolls back everything).

**PayPal Section 8 Course**

* We go in sandbox mode.
* You create sandbox account for both buyer and seller.
* **Seller Account Type:** Business
* **Buyer Account Type:** Personal

You must have both.

First go to Testing Tools and navigate to sandbox accounts.

Then go to Apps and Credentials and create your account.

Access Tokens (secure identifier) will allow our app to interact with PayPal services.

**New way to destructure process.env**

const { PAYPAL\_CLIENT\_ID, PAYPAL\_APP\_SECRET } = process.env;

First you create an order intent and then you capture payment for order.

**All in documentation is given, all steps, please refer there.**

**JEST Section 8 Course**

* It is a testing library.
* We can write unit test cases using it.

**Config file:**

jest.config.ts

For example to write test cases, create a folder:

tests/paypa.test.ts

**Test Case:**

import { generateAccessToken } from "../lib/paypal";

// Test to generate access token from paypal

test("generates token from paypal", async () => {

const tokenResponse = await generateAccessToken();

console.log(tokenResponse);

expect(typeof tokenResponse).toBe("string");

expect(tokenResponse.length).toBeGreaterThan(0);

});

**To run Test files command:**

npm test.

**Output:**

console.log

A21AAKmbh12hOSE3COFMFzm6-Dd-8-vWZjnHovU6I5pb7qHNRzQU6fgPzK0P0FfpgHCkDTUihsWH2-tkF\_kgcQXzQTSDghpZw

at Object.<anonymous> (tests/paypal.test.ts:6:11)

PASS tests/paypal.test.ts √ generates token from paypal (1415 ms)

Test Suites: 1 passed, 1 total

Tests: 1 passed, 1 total

Snapshots: 0 total

Time: 2.053 s

Ran all test suites.

**Helper function to minimize duplication logic**

async function handleResponse(response: Response) {

if (response.ok) {

return response.json();

} else {

const errorMessage = await response.text();

throw new Error(errorMessage);

}

}

**Session Provider: (next-auth/react)**

It is required when a component is client component and we cant use session like this directly

const session = await auth();

Since by default they are server components and we are using client component to fetch user’s session.

In order to use it in server component:

We must wrap it using Session Provider, for example:

const Profile = async () => {

const session = await auth();

return (

<SessionProvider session={session}>

<div className="max-w-md mx-auto space-y-4">

<h2 className="h2-bold">Profile</h2>

<ProfileForm />

</div>

</SessionProvider>

);

};

export default Profile;

The client component that is using this Session Provider for example in Profile Form component it must have

Use Session hook to consume it.

import { useSession } from "next-auth/react";

const { data: session, update } = useSession();

**Note:**

**To protect admin pages we must use this everywhere in admin routes.**

import { requireAdmin } from "@/lib/auth-guard";

  // Protect the admin route

  // This will ensure that only authenticated users with admin privileges can access this page

  // No normal user should be able to access this page

  await requireAdmin();

**What does asChild in shadcn button mean?**

<Button asChild>

<Link href="/admin/products/create">Create Product</Link>

</Button>

In shadcn, **asChild** means:

* Use my child element (like <Link>) instead of a real <button>.
* Style the <Link> to look and act like a button.
* Use a Link instead of a button
* But still get all the button styles (like padding, hover, etc.)
* It keeps the HTML valid and looks like a proper button.

**Slugify Package:**

For example Hello World

Slugify will create a slug for it: hello-world

Useful for navigating like: product/hello-world

Basically product name id, rather than going to id like 123 etc.

**For Image Upload Service:**   
**uploadthing**

It is a free tier service and allows upto 2 GB storage.

Just like cloudinary, we can make use of this as well.

We also must configure in next.config.ts:

import type { NextConfig } from "next";

const nextConfig: NextConfig = {

images: {

remotePatterns: [

{

protocol: "https",

hostname: "utfs.io",

port: "",

},

],

},

};

export default nextConfig;

To use this service, configuration is must in next.config.ts

**Stripe**

**Steps for payment Intent:**

1. Create a payment intent:
2. Confirm payment on client
3. Check Payment status:

* Payment method
* Payment confirmation
* Processing of payment
* Succeed payment
* Failed payment

1. Handle success/failure cases.

**Payment Intent:**

All the information for collecting payment like the user will be charged in US dollars etc. Track the status of payment and handle any authentication.