**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’**

**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;

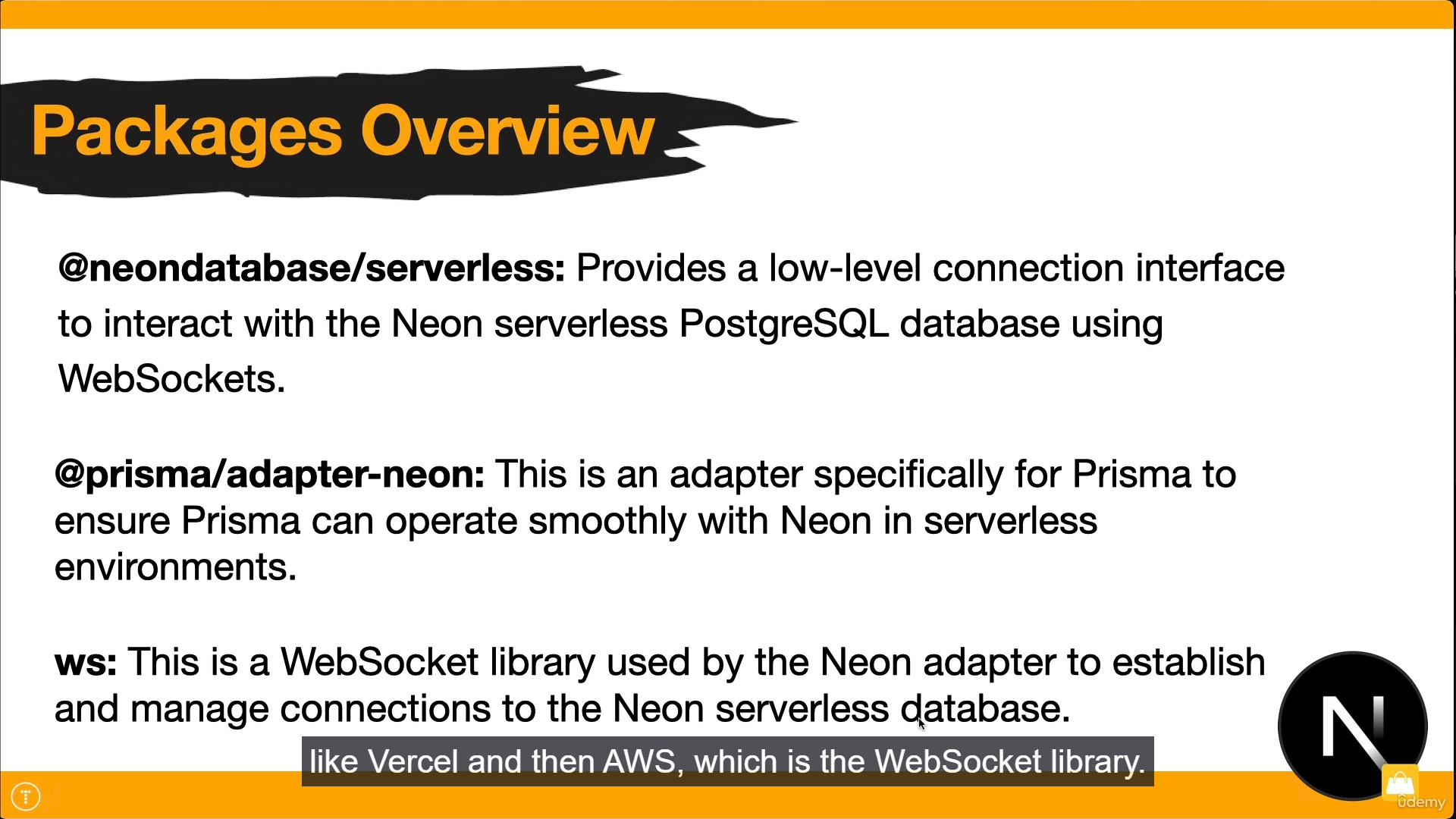
}

**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)**

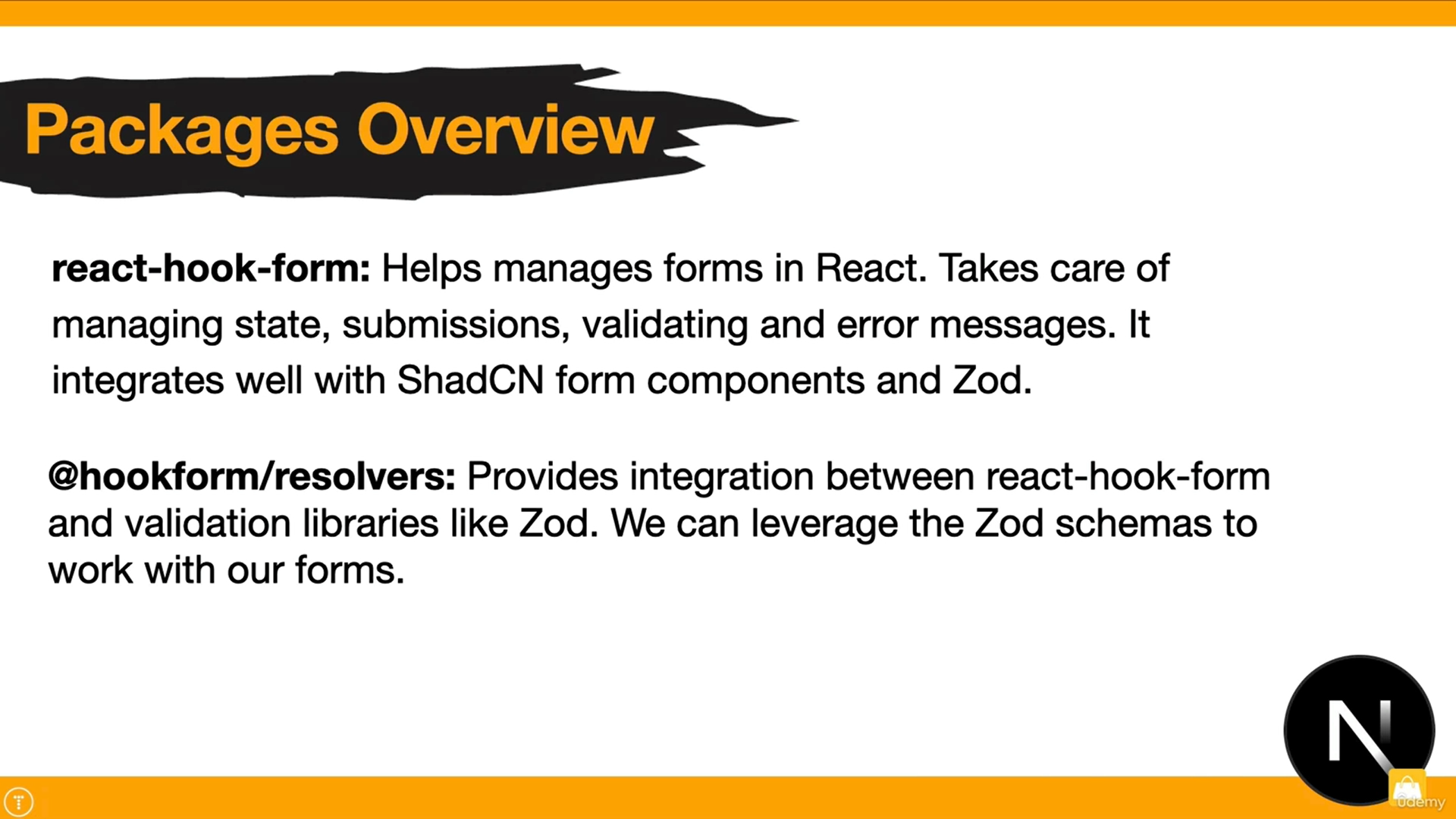
It helps manage the UI updates by transitioning.

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).