

## PROMPT

Jan 19, 2026, 10:52 AM

Write a step-by-step tutorial for Nuxt aimed at beginner developers for connecting their frontend to Neon. Include prerequisites, clear learning objectives, hands-on exercises, and troubleshooting tips.

## Response from GPT-5

### Nuxt + Neon: Beginner-Friendly, End-to-End Tutorial

Build a simple full-stack Nuxt app that securely reads and writes to a Neon (serverless Postgres) database using Nuxt server routes. You'll learn best practices, see real results, and get exercises to practice.

#### What You'll Build

- A Nuxt 3 app with:
  - æ Secure server-side API routes for CRUD operations
  - æ A simple Todo UI in the frontend
  - æ A Neon-backed Postgres database

#### Prerequisites

- Node.js 18+ (Nuxt and Neon's serverless driver rely on fetch)
- npm or pnpm
- A Neon account and project (free tier is fine)
- Basic familiarity with JavaScript/TypeScript and Nuxt

#### Learning Objectives

- Understand how to securely connect a Nuxt app to Neon
- Create server-side API routes in Nuxt that talk to Postgres
- Use the Neon serverless driver efficiently
- Build a minimal CRUD UI and test results
- Troubleshoot common connection and runtime issues

#### Architecture Overview

- Client (Vue/Nuxt pages): Calls your own API routes (never talks to Neon directly).
- Server (Nuxt Nitro server routes): Runs queries against Neon with env-protected credentials.
- Database (Neon): Serverless Postgres with TLS.

Important:

- Never expose your DATABASE\_URL to the browser.
- All queries must run in Nuxt server routes (in server/).

## Step 1: Create a Nuxt 3 project

```
# Create a new Nuxt app
npx nuxi init nuxt-neon
cd nuxt-neon

# Install dependencies
npm install
# or: pnpm install
```

bash

Run the dev server to confirm it boots:

```
npm run dev
```

bash

Expected outcome:

```
Nuxt 3 ready on http://localhost:3000
```

txt

## Step 2: Create a Neon project and get your connection string

1. Sign in at <https://neon.tech>
2. Create a project and a database
3. Copy your Postgres connection string, which looks like:

```
postgresql://USER:PASSWORD@HOST/dbname?sslmode=require
```

txt

Notes:

- Ensure `sslmode=require` is present.
- Keep this string private; we'll store it in an env variable.

## Step 3: Configure environment variables in Nuxt

Create a `.env` file at the project root:

bash

```
touch .env
```

Add your Neon connection string:

```
# .env
DATABASE_URL="postgresql://USER:PASSWORD@HOST/dbname?sslmode=require"
```

bash

Add it to runtimeConfig (private by default) in nuxt.config.ts:

```
// nuxt.config.ts
export default defineNuxtConfig({
  devtools: { enabled: true },
  runtimeConfig: {
    databaseUrl: process.env.DATABASE_URL, // private (server-only)
  },
})
```

ts

## Step 4: Install the Neon serverless driver

```
npm install @neondatabase/serverless
```

bash

Why this driver?

- It talks to Neon over HTTP/fetch, great for serverless/edge runtimes.
- Simple API with automatic parameterization.

## Step 5: Create a reusable database helper

Create a small helper to initialize and cache the Neon client.

```
// server/utils/db.ts
import { neon, neonConfig } from '@neondatabase/serverless'
import { useRuntimeConfig } from '#imports'

// Cache connections across hot reloads / serverless invocations
neonConfig.fetchConnectionCache = true

let _sql: ReturnType<typeof neon> | null = null

export function sql() {
  if (!_sql) {
    const { databaseUrl } = useRuntimeConfig()
    if (!databaseUrl) {
```

ts

```
throw new Error('DATABASE_URL is not set')
}
_sql = neon(databaseUrl)
}
return _sql
}
```

## Step 6: One-time setup route (create table + seed)

This makes it easy to create your schema while learning. In production, use migrations.

```
ts
// server/api/setup.post.ts
import { defineEventHandler } from 'h3'
import { sql } from '~/server/utils/db'

export default defineEventHandler(async () => {
  const db = sql()

  await db/* sql */`
CREATE TABLE IF NOT EXISTS todos (
  id serial PRIMARY KEY,
  title text NOT NULL,
  done boolean NOT NULL DEFAULT false,
  created_at timestamptz NOT NULL DEFAULT now()
)
`

  const countRow = await db/* sql */`
SELECT COUNT(*)::int AS count FROM todos
`

  const count = countRow[0].count as number

  if (count === 0) {
    await db/* sql */`
INSERT INTO todos (title, done)
VALUES ('Try Neon + Nuxt', false), ('Star the repo', true)
`
  }

  return { ok: true, created: count === 0 }
})
```

Run it:

```
bash
curl -X POST http://localhost:3000/api/setup
```

Expected result:

```
json
{ "ok": true, "created": true }
```

Run again and it should not re-seed:

json

```
{ "ok": true, "created": false }
```

## Step 7: CRUD API routes (server-side)

GET all todos:

ts

```
// server/api/todos.get.ts
import { defineEventHandler } from 'h3'
import { sql } from '~/server/utils/db'

export default defineEventHandler(async () => {
  const db = sql()
  const rows = await db/* sql */`
  SELECT id, title, done, created_at
  FROM todos
  ORDER BY id DESC
  `
  return rows
})
```

POST a new todo:

ts

```
// server/api/todos.post.ts
import { defineEventHandler, readBody, createError } from 'h3'
import { sql } from '~/server/utils/db'

export default defineEventHandler(async (event) => {
  const body = await readBody<{ title?: string }>(event)
  const title = (body?.title || '').trim()

  if (!title || title.length > 200) {
    throw createError({ statusCode: 400, statusMessage: 'Invalid title' })
  }

  const db = sql()
  const rows = await db/* sql */`
  INSERT INTO todos (title)
  VALUES (${title})
  RETURNING id, title, done, created_at
  `
  return rows[0]
})
```

PATCH toggle done:

ts

```
// server/api/todos/[id].patch.ts
import { defineEventHandler, readBody, createError } from 'h3'
import { sql } from '~/server/utils/db'

export default defineEventHandler(async (event) => {
  const id = Number(event.context.params?.id)
  if (!Number.isInteger(id)) {
    throw createError({ statusCode: 400, statusMessage: 'Invalid id' })
  }

  const body = await readBody<{ done?: boolean }>(event)
  const done = Boolean(body?.done)

  const db = sql()
  const rows = await db/* sql */`
  UPDATE todos
  SET done = ${done}
  WHERE id = ${id}
  RETURNING id, title, done, created_at
  `

  if (rows.length === 0) {
    throw createError({ statusCode: 404, statusMessage: 'Not found' })
  }
  return rows[0]
})
```

DELETE a todo:

```
// server/api/todos/[id].delete.ts
import { defineEventHandler, createError } from 'h3'
import { sql } from '~/server/utils/db'

export default defineEventHandler(async (event) => {
  const id = Number(event.context.params?.id)
  if (!Number.isInteger(id)) {
    throw createError({ statusCode: 400, statusMessage: 'Invalid id' })
  }

  const db = sql()
  const rows = await db/* sql */`
  DELETE FROM todos
  WHERE id = ${id}
  RETURNING id
  `

  if (rows.length === 0) {
    throw createError({ statusCode: 404, statusMessage: 'Not found' })
  }
  return { ok: true, id }
})
```

Test the GET route:

```
curl http://localhost:3000/api/todos
```

Example result:

json

```
[
  { "id": 2, "title": "Star the repo", "done": true, "created_at": "2024-01-01T10:00:00.000Z" },
  { "id": 1, "title": "Try Neon + Nuxt", "done": false, "created_at": "2024-01-01T10:00:00.000Z" }
]
```

## Step 8: Build a simple frontend

Create a minimal page to list/add/toggle/delete todos.

vue

```
<!-- pages/index.vue -->
<template>
<main class="container">
<h1>Nuxt + Neon Todos</h1>

<form @submit.prevent="addTodo" class="row">
<input v-model="newTitle" placeholder="New todo..." />
<button :disabled="adding || !newTitle.trim()">Add</button>
</form>

<div v-if="pending">Loading...</div>
<div v-else-if="error">Error: {{ error.message }}</div>

<ul>
<li v-for="t in todos" :key="t.id" class="todo">
<label>
<input type="checkbox" :checked="t.done" @change="toggle(t)" />
<span :style="{ textDecoration: t.done ? 'line-through' : 'none' }">{{ t.title }}</span>
</label>
<button @click="remove(t)">Delete</button>
</li>
</ul>
</main>
</template>

<script setup lang="ts">
const { data, pending, error, refresh } = await useFetch('/api/todos')
const todos = computed(() => data.value ?? [])

const newTitle = ref('')
const adding = ref(false)

async function addTodo() {
  if (!newTitle.value.trim()) return
  adding.value = true
  try {
    await $fetch('/api/todos', {
      method: 'POST',
      body: { title: newTitle.value },
    })
    newTitle.value = ''
    await refresh()
  } finally {
    adding.value = false
  }
}
```

```
async function toggle(t: any) {
  await $fetch(`/api/todos/${t.id}`, {
    method: 'PATCH',
    body: { done: !t.done },
  })
  await refresh()
}

async function remove(t: any) {
  await $fetch(`/api/todos/${t.id}`, { method: 'DELETE' })
  await refresh()
}
</script>

<style scoped>
.container { max-width: 640px; margin: 2rem auto; font: 16px/1.4 system-ui, sans-serif; }
.row { display: flex; gap: 0.5rem; margin-bottom: 1rem; }
.todo { display: flex; align-items: center; justify-content: space-between; }
</style>
```

Open <http://localhost:3000> and interact with the app.

Example outcomes:

- Adding a todo creates a new row in Neon and appears in the list
- Checking a todo toggles its done status
- Deleting a todo removes it from Neon and the list

## Step 9: Verify database changes directly (optional)

From the Neon SQL editor or psql:

```
SELECT id, title, done, created_at FROM todos ORDER BY id DESC;
```

Example output:

| id | title              | done | created_at             |
|----|--------------------|------|------------------------|
| 3  | Learn prepared SQL | f    | 2024-01-01 10:05:00+00 |
| 2  | Star the repo      | t    | 2024-01-01 10:00:00+00 |
| 1  | Try Neon + Nuxt    | f    | 2024-01-01 10:00:00+00 |

## Hands-on Exercises

1. Add edit title support

- æ Create PUT /api/todos/[id] to update the title
  - æ Add an inline edit UI in the list
  - 2. Add filtering
    - æ Add a query param done=true|false to GET /api/todos
    - æ Add UI buttons to filter client-side or server-side
  - 3. Add pagination
    - æ Accept limit/offset query params in GET
  - 4. Improve validation
    - æ Disallow empty or overly long titles (already partially done)
    - æ Return 422 with detailed message
  - 5. Handle errors in UI
    - æ Show toast or inline messages when API calls fail
  - 6. Security hardening
    - æ Add a simple rate limiter per IP on POST/DELETE
    - æ Confirm deletion with a modal
  - 7. Bonus: use an ORM
    - æ Try Drizzle ORM or Prisma with Neon for typed queries
- 

## Troubleshooting Tips

Connection issues:

- Error: DATABASE\_URL is not set

```
```txt
```

Fix: Ensure .env exists and Nuxt sees it. Restart dev server after changes.

```
```
```

- TLS/SSL errors

```
```txt
```

Fix: Ensure your Neon URL has ?sslmode=require

```
```
```

- Password authentication failed

```
```txt
```

Fix: Verify user/password/DB match your Neon project.

```
```
```

- Timeout on first query

```
```txt
```

Neon free tier may cold start. Retry once; subsequent queries will be fast.

...

Driver/runtime issues:

- TypeError: fetch is not defined

```
```txt
```

Fix: Use Node 18+ (or run inside Nuxt/Nitro which polyfills fetch).

...

- CORS error in browser when calling Neon directly

```
```txt
```

Fix: Do NOT call Neon from the browser. Use Nuxt server routes as shown.

...

Nuxt specifics:

- Accessing runtime config

```
```ts
```

```
// server-only (works in API routes and server utils)
```

```
const { databaseUrl } = useRuntimeConfig()
```

...

- Exposing secrets

```
```txt
```

Do not put DATABASE\_URL into public runtime config. Keep it private (default).

...

- Performance

```
```txt
```

We enabled `neonConfig.fetchConnectionCache = true` to reuse connections.

...

Deployment notes:

- Set DATABASE\_URL in your hosting provider's environment variables.
- Most serverless platforms (Vercel/Netlify/Fly/Render) work out-of-the-box with the Neon serverless driver.
- After deploy, hit `/api/setup` once (or run a migration) to create tables.

---

## What You Learned

- How to securely connect a Nuxt app to Neon using server routes
- How to structure CRUD endpoints and a simple UI
- How to configure environment variables and avoid leaking secrets
- How to test and troubleshoot end-to-end

You're ready to extend this app, add migrations, auth, and deploy!