PROMPT Jan 19, 2026, 09:12 PM

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

Beginner's Guide: Connect a Nuxt App to Neon (Serverless Postgres)

This step-by-step tutorial shows you how to connect a Nuxt 3 frontend to a Neon Postgres database the right way: through secure Nuxt server routes (so your database credentials are never exposed in the browser). You'll build a tiny Todo app that reads/writes data in Neon.

Who this is for: Beginner developers with basic JavaScript knowledge and little/no experience with Nuxt or Postgres.

Prerequisites

- Node.js 18+ and npm (or pnpm/yarn) installed
- A Neon account (free): https://neon.tech
- Basic familiarity with JavaScript/TypeScript and async/await
- Git installed (optional but recommended)

Learning Objectives

By the end, you'll be able to:

- Create a Neon Postgres database and run SQL to create a table
- Store and use secrets safely in Nuxt via runtime config
- Build Nuxt server routes that talk to Neon using the @neondatabase/serverless driver
- Fetch and display records on a Nuxt page, and insert new records
- Diagnose common connection and SQL issues

What You'll Build

A minimal Todo list:

- GET /api/todos to list todos
- POST /api/todos to create a todo
- A Nuxt page to show todos and a form to add new ones

Expected behaviors:

```
- Visiting the home page lists todos from Neon
- Submitting the form adds a new todo to the database
```

---

1) Create a Neon project and database

1) Sign up or log in at Neon

2) Create a new project and database (accept defaults)

3) In the project Dashboard, copy the Postgres connection string. It looks like:

```
postgresql://<user>:<password>@<host>.neon.tech/<database>?sslmode=require
```

4) Open Neon’s SQL Editor and run the table setup below (you can also use any SQL client).

SQL to create a simple todos table:

```sql
CREATE TABLE IF NOT EXISTS todos (
 id SERIAL PRIMARY KEY,
 title TEXT NOT NULL,
 completed BOOLEAN NOT NULL DEFAULT false,
 created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Expected result in Neon’s SQL editor:

```
CREATE TABLE
```

---

2) Create a new Nuxt 3 project

In your terminal:

```bash
# Create a Nuxt app
npx nuxi@latest init nuxt-neon-demo
cd nuxt-neon-demo

# Install dependencies
npm install
```

Expected result:

```
 Generated nuxt-neon-demo
 Dependencies installed
```

3) Install the Neon serverless driver

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

Expected result:

```
+ @neondatabase/serverless@<version>
added <n> packages
```

4) Configure your database secret

Create a .env file at the root of your project:

```bash
touch .env
```

Add your Neon connection string:

```ini
# .env
DATABASE_URL="postgresql://<user>:<password>@<host>.neon.tech/<database>?sslmode=require"
```

Update nuxt.config.ts to expose this to the server runtime only:

```ts
// nuxt.config.ts
export default defineNuxtConfig({
 devtools: { enabled: true },
 runtimeConfig: {
 databaseUrl: process.env.DATABASE_URL, // Private: server only
 public: {} // Keep DB secrets out of here
}
});
```

Expected result:

```
- nuxt.config.ts updated
- .env created with DATABASE_URL
```

---

5) Create a small DB utility for server routes (optional but clean)

Create server/utils/db.ts:

```ts
// server/utils/db.ts
import { neon } from '@neondatabase/serverless';

export function useDb() {
 const { databaseUrl } = useRuntimeConfig();
 if (!databaseUrl) {
 throw new Error('DATABASE_URL is missing. Set it in your .env file.');
}
 return neon(databaseUrl);
}
```

Expected result:

```
- server/utils/db.ts created
```

---

6) Build server API routes

Create a route to list todos.

File: server/api/todos.get.ts

```ts
// server/api/todos.get.ts
import { useDb } from '../utils/db';

export default defineEventHandler(async () => {
 const sql = useDb();
// Query all todos ordered by newest first
 const rows = await sql<
{ id: number; title: string; completed: boolean; created_at: string }[]
>`SELECT id, title, completed, created_at FROM todos ORDER BY created_at DESC`;
 return rows;
});
```

Create a route to add a todo.

File: server/api/todos.post.ts

```ts
// server/api/todos.post.ts
import { useDb } from '../utils/db';

type Body = { title?: string };
```

```
export default defineEventHandler(async (event) => {
 const sql = useDb();
 const body = (await readBody(event)) as Body;

 if (!body?.title || !body.title.trim()) {
 throw createError({ statusCode: 400, statusMessage: 'Title is required' });
}

 const [row] = await sql<
{ id: number; title: string; completed: boolean; created_at: string }[]
>`INSERT INTO todos (title) VALUES (${body.title.trim()}) RETURNING id, title, completed, created_at`;

 return row;
});
```

Expected behavior:

```
- GET /api/todos returns an array of todos
- POST /api/todos with { "title": "Learn Nuxt + Neon" } returns the inserted row
```

---

7) Build the Nuxt page

Create pages/index.vue:

```vue
<!-- pages/index.vue -->
<script setup lang="ts">
const newTitle = ref('');
const { data: todos, pending, error, refresh } = await useFetch('/api/todos');

const addTodo = async () => {
 const title = newTitle.value.trim();
 if (!title) return;

 try {
 await $fetch('/api/todos', {
 method: 'POST',
 body: { title }
});
 newTitle.value = '';
 await refresh(); // refresh GET /api/todos
} catch (err) {
 console.error(err);
 alert('Failed to add todo');
}
};
</script>

<template>
<main style="max-width: 640px; margin: 2rem auto; font-family: system-ui;">
<h1>Nuxt + Neon: Todos</h1>

<form @submit.prevent="addTodo" style="margin-bottom: 1rem;">
<input
 v-model="newTitle"
 type="text"
 placeholder="What needs to be done?"
 style="padding: 0.5rem; width: 70%;"
/>
<button type="submit" style="padding: 0.5rem 1rem; margin-left: 0.5rem;">
 Add
</button>
</form>

<section>
<p v-if="pending">Loading...</p>
<p v-else-if="error">Failed to load todos.</p>
<ul v-else>
<li v-for="t in todos" :key="t.id" style="margin: 0.25rem 0;">
<span :style="{ textDecoration: t.completed ? 'line-through' : 'none' }">
{{ t.title }}
</span>
<small style="color: #666;"> — {{ new Date(t.created_at).toLocaleString() }}</small>
</li>
</ul>
</section>
</main>
</template>
```

Expected result in the browser (after you add an item):

```
- A list of todos with newest items first
- Added items appear immediately after submission
```

---

8) Run and test locally

Start your Nuxt dev server:

```bash
npm run dev
```

Expected terminal output:

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

Test your endpoints:

- GET todos

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

Expected response:

```json
[]
```

- POST a todo

```bash
curl -X POST http://localhost:3000/api/todos \
-H "Content-Type: application/json" \
-d '{"title":"Learn Nuxt + Neon"}'
```

Expected response:

```json
{
"id": 1,
"title": "Learn Nuxt + Neon",
"completed": false,
"created_at": "2024-01-01T12:34:56.789Z"
}
```

- Refresh the page at http://localhost:3000/

Expected browser results:

```
- The new todo appears in the list
```

Hands-on Exercises

1) Toggle completed

- Add a PATCH endpoint: server/api/todos/[id].patch.ts

```ts
// server/api/todos/[id].patch.ts
import { useDb } from '../../utils/db';

type Body = { completed?: boolean };

export default defineEventHandler(async (event) => {
 const sql = useDb();
 const id = Number(getRouterParam(event, 'id'));
 const body = (await readBody(event)) as Body;
 if (!Number.isFinite(id)) {
 throw createError({ statusCode: 400, statusMessage: 'Invalid id' });
}
 if (typeof body.completed !== 'boolean') {
 throw createError({ statusCode: 400, statusMessage: 'completed must be boolean' });
}

 const [row] = await sql`
 UPDATE todos SET completed = ${body.completed}
 WHERE id = ${id}
 RETURNING id, title, completed, created_at
`;
```

```
 if (!row) {
 throw createError({ statusCode: 404, statusMessage: 'Todo not found' });
}
 return row;
});
```

- Update the UI to toggle a todo’s completion and refresh the list.

2) Add pagination

- Accept query params ?limit=10&offset=0 in GET /api/todos.get.ts and update SQL with LIMIT/OFFSET.
- Add “Load more” to the UI.

3) Improve UX

- Show loading states on submit
- Disable the submit button when the title is empty
- Display error messages inline

Expected outcomes:

```
- You can mark todos complete/incomplete
- Large lists can be paginated
- The UI handles loading and error states gracefully
```

Troubleshooting Tips

- Never expose your database URL to the browser

æ Use only server routes (/server/api/*) to talk to Neon

æ Do not access process.env.DATABASE_URL in client-side code

æ In Nuxt, only variables in runtimeConfig.public are sent to the client. Keep DATABASE_URL out of public.

- Missing or wrong DATABASE_URL

æ Symptom:

```
Error: DATABASE_URL is missing. Set it in your .env file.
```

æ Fix:

æ Ensure .env contains DATABASE_URL

æ Restart dev server after creating .env

æ Log useRuntimeConfig() inside a server route if needed

- Connection or SSL issues

æ Symptom:

```
connect ECONNREFUSED / TLS / certificate errors
```

```

æ Fix:

æ Ensure your connection string ends with ?sslmode=require (Neon requires SSL)

æ Ensure your network/firewall allows outbound HTTPS

æ Copy the connection string exactly from Neon

- Table doesn’t exist

æ Symptom:

```

relation "todos" does not exist

```

æ Fix:

æ Run the CREATE TABLE SQL in Neon’s SQL Editor for your active database

æ Double-check which database name you’re connected to in the connection string

- JSON/validation errors on POST

æ Symptom:

```

400 Title is required

```

æ Fix:

æ Ensure request body contains a non-empty "title"

æ Ensure you set Content-Type: application/json

- Accidentally bundling DB code into the client

æ Symptom:

```

ReferenceError: process is not defined

```

æ Fix:

æ Only import @neondatabase/serverless from server files (server/*)

æ Avoid importing server-side utilities into components/composables that run on client

- Using the wrong driver

æ If you switch to the "pg" package in serverless/edge contexts, you may hit connection/pooling issues

æ @neondatabase/serverless is recommended for serverless and edge runtimes

- CORS issues

æ If your frontend and API are on different origins during development:

æ Prefer same-origin calls (Nuxt server routes) while developing

æ If needed, configure CORS in Nitro with Nitro presets or proxy

Expected result of applying fixes:
```

```
- API routes return data successfully
- The page lists and inserts todos reliably
```

Deploying (Overview)

When deploying to providers like Vercel, Netlify, or Cloudflare:

- Set DATABASE_URL as a server environment variable in your hosting provider’s dashboard
- No code changes required if you used runtimeConfig as shown
- Verify production logs if anything fails on first deploy

Expected production behavior:

```
- App works the same as locally
- /api/todos reads/writes to Neon in production
```

Next Steps

- Add authentication and Row Level Security (RLS) for multi-user data
- Use an ORM like Prisma or Drizzle for migrations and type-safety
- Move to edge runtime if desired; @neondatabase/serverless supports it
- Add schema validation with zod for API inputs

Recap

You:

- Created a Neon database and table
- Stored secrets safely in Nuxt runtime config
- Built secure API routes with @neondatabase/serverless
- Rendered and inserted data from a Nuxt page

Final expected outcome:

```
- A working Nuxt app that connects to Neon securely via server routes
- Ability to list and add todos
```