PROMPT Jan 19, 2026, 09:11 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 Claude Sonnet 4

Connecting Nuxt to Neon Database: A Complete Tutorial for Beginners

Prerequisites

Before starting this tutorial, make sure you have:

- Node.js (version 16 or higher) installed on your machine
- npm or yarn package manager
- Basic knowledge of JavaScript and Vue.js
- A Neon account (free tier available at neon.tech)
- A code editor like VS Code
- Basic understanding of databases and SQL

Learning Objectives

By the end of this tutorial, you will be able to:

- Set up a new Nuxt 3 application
- Create and configure a Neon PostgreSQL database
- Connect your Nuxt frontend to Neon using server API routes
- Perform basic CRUD operations (Create, Read, Update, Delete)
- Handle database connections securely using environment variables
- Implement error handling for database operations

Step 1: Setting Up Your Nuxt 3 Project

Create a New Nuxt Application

```bash
npx nuxi@latest init nuxt-neon-tutorial
cd nuxt-neon-tutorial
npm install
```

Install Required Dependencies

```bash
npm install @neondatabase/serverless dotenv
npm install -D @types/node
```

Important: The @neondatabase/serverless package is optimized for serverless environments and works perfectly with Nuxt's server API routes.

Start the Development Server

```bash
npm run dev
```

You should see:

```
Nuxt 3.8.0 with Nitro 2.7.2
> Local:    http://localhost:3000/
> Network:  use --host to expose
```

Step 2: Setting Up Your Neon Database

Create a Neon Project

1. Go to neon.tech and sign up for a free account
2. Click "Create Project"
3. Choose your region (select the one closest to your users)
4. Enter a project name: nuxt-tutorial-db
5. Click "Create Project"

Get Your Connection String

After creating your project, you'll see a connection string like:

```
postgresql://username:password@ep-example.us-east-2.aws.neon.tech/dbname?sslmode=require
```

Important: Copy this connection string - you'll need it in the next step.

Create Your First Table

1. Go to the SQL Editor in your Neon dashboard
2. Run this SQL command to create a users table:

```sql
CREATE TABLE users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. Insert some sample data:

```sql
INSERT INTO users (name, email) VALUES
('John Doe', 'john@example.com'),
('Jane Smith', 'jane@example.com');
```

Step 3: Configure Environment Variables

Create Environment File

Create a .env file in your project root:

```bash
touch .env
```

Add Your Database URL

```env
DATABASE_URL=postgresql://username:password@ep-example.us-east-2.aws.neon.tech/dbname?sslmode=require
```

Important: Replace the URL with your actual Neon connection string.

Update .gitignore

Make sure your .gitignore includes:

```gitignore
# Environment variables
.env
.env.*
```

Step 4: Create Database Connection Utility

Create the Database Utility File

Create utils/database.ts:

typescript

```
import { neon } from '@neondatabase/serverless'

// Initialize the Neon client
export const sql = neon(process.env.DATABASE_URL!)

// Test connection function
export async function testConnection() {
 try {
 const result = await sql`SELECT version()`
 console.log('Database connected successfully:', result[0].version)
 return { success: true, version: result[0].version }
} catch (error) {
 console.error('Database connection failed:', error)
 return { success: false, error: error.message }
}
}
```

Step 5: Create Server API Routes

Create the API Directory Structure

```bash
mkdir -p server/api/users
```

Create User API Endpoints

Create server/api/users/index.get.ts (Get all users):

```typescript
import { sql } from '~/utils/database'

export default defineEventHandler(async (event) => {
 try {
 const users = await sql`
 SELECT id, name, email, created_at
 FROM users
 ORDER BY created_at DESC
`

 return {
 success: true,
 data: users
}
} catch (error) {
 throw createError({
 statusCode: 500,
 statusMessage: 'Failed to fetch users',
 data: error.message
})
}
})
```

Create server/api/users/index.post.ts (Create new user):

```typescript
```

```
import { sql } from '~/utils/database'

export default defineEventHandler(async (event) => {
 try {
 const body = await readBody(event)
 const { name, email } = body

// Validate input
 if (!name || !email) {
 throw createError({
 statusCode: 400,
 statusMessage: 'Name and email are required'
})
}

// Insert new user
 const result = await sql`
 INSERT INTO users (name, email)
 VALUES (${name}, ${email})
 RETURNING id, name, email, created_at
`

 return {
 success: true,
 data: result[0],
 message: 'User created successfully'
}
} catch (error) {
// Handle unique constraint violation
 if (error.code === '23505') {
 throw createError({
 statusCode: 409,
 statusMessage: 'Email already exists'
})
}

 throw createError({
 statusCode: 500,
 statusMessage: 'Failed to create user',
 data: error.message
})
}
})
```

Create server/api/users/[id].delete.ts (Delete user):

typescript

```
import { sql } from '~/utils/database'

export default defineEventHandler(async (event) => {
 try {
 const id = getRouterParam(event, 'id')

 if (!id || isNaN(Number(id))) {
 throw createError({
 statusCode: 400,
 statusMessage: 'Valid user ID is required'
})
}

 const result = await sql`
 DELETE FROM users
 WHERE id = ${id}
 RETURNING id, name, email
`

 if (result.length === 0) {
 throw createError({
 statusCode: 404,
 statusMessage: 'User not found'
})
}

 return {
 success: true,
 data: result[0],
 message: 'User deleted successfully'
}
} catch (error) {
 if (error.statusCode) {
 throw error
}

 throw createError({
 statusCode: 500,
 statusMessage: 'Failed to delete user',
 data: error.message
})
}
})
```

Step 6: Create the Frontend Interface

Update the Main Page

Replace the content of app.vue:

```vue
<template>
<div class="container">
<h1>Nuxt + Neon Database Tutorial</h1>

<!-- Add User Form -->
<div class="form-section">
<h2>Add New User</h2>
<form @submit.prevent="addUser" class="user-form">
<div class="form-group">
<label for="name">Name:</label>
<input
 id="name"
 v-model="newUser.name"
 type="text"
 required
 placeholder="Enter user name"
/>
</div>
<div class="form-group">
<label for="email">Email:</label>
<input
 id="email"
 v-model="newUser.email"
 type="email"
 required
 placeholder="Enter user email"
/>
</div>
<button type="submit" :disabled="loading">
{{ loading ? 'Adding...' : 'Add User' }}
</button>
</form>
</div>

<!-- Error Message -->
<div v-if="error" class="error">
{{ error }}
</div>

<!-- Success Message -->
<div v-if="success" class="success">
{{ success }}
</div>

<!-- Users List -->
<div class="users-section">
<h2>Users ({{ users.length }})</h2>
<div v-if="users.length === 0" class="no-users">
 No users found. Add some users above!
</div>
<div v-else class="users-grid">
<div v-for="user in users" :key="user.id" class="user-card">
<h3>{{ user.name }}</h3>
<p class="email">{{ user.email }}</p>
<p class="date">
 Created: {{ formatDate(user.created_at) }}
</p>
<button
@click="deleteUser(user.id)"
 class="delete-btn"
:disabled="loading"
>
 Delete
</button>
</div>
</div>
</div>
</div>
</template>

<script setup>
// Page meta
useHead({
 title: 'Nuxt + Neon Tutorial'
})

// Reactive data
const users = ref([])
const loading = ref(false)
const error = ref('')
const success = ref('')
const newUser = ref({
 name: '',
 email: ''
})

// Fetch users on page load
await fetchUsers()
```

```
// Functions
async function fetchUsers() {
 try {
 loading.value = true
 error.value = ''

 const response = await $fetch('/api/users')
 users.value = response.data
} catch (err) {
 error.value = 'Failed to fetch users: ' + err.data
 console.error('Fetch error:', err)
} finally {
 loading.value = false
}
}

async function addUser() {
 try {
 loading.value = true
 error.value = ''
 success.value = ''

 const response = await $fetch('/api/users', {
 method: 'POST',
 body: {
 name: newUser.value.name.trim(),
 email: newUser.value.email.trim()
}
})

// Add new user to the list
 users.value.unshift(response.data)

// Reset form
 newUser.value = { name: '', email: '' }
 success.value = response.message

// Clear success message after 3 seconds
 setTimeout(() => {
 success.value = ''
}, 3000)
} catch (err) {
 error.value = err.data || 'Failed to add user'
 console.error('Add user error:', err)
} finally {
 loading.value = false
}
}

async function deleteUser(userId) {
 if (!confirm('Are you sure you want to delete this user?')) {
 return
}

 try {
 loading.value = true
 error.value = ''

 const response = await $fetch(`/api/users/${userId}`, {
 method: 'DELETE'
})

// Remove user from the list
 users.value = users.value.filter(user => user.id !== userId)
 success.value = response.message

// Clear success message after 3 seconds
 setTimeout(() => {
 success.value = ''
}, 3000)
} catch (err) {
 error.value = err.data || 'Failed to delete user'
 console.error('Delete error:', err)
} finally {
 loading.value = false
}
}

function formatDate(dateString) {
 return new Date(dateString).toLocaleDateString('en-US', {
 year: 'numeric',
 month: 'short',
 day: 'numeric',
 hour: '2-digit',
 minute: '2-digit'
})
}
</script>

<style scoped>
.container {
 max-width: 800px;
 margin: 0 auto;
 padding: 20px;
 font-family: Arial, sans-serif;
}

h1 {
 color: #2c3e50;
 text-align: center;
 margin-bottom: 30px;
}

.form-section {
 background: #f8f9fa;
 padding: 20px;
 border-radius: 8px;
 margin-bottom: 20px;
}

.user-form {
 display: flex;
 flex-direction: column;
 gap: 15px;
}

.form-group {
 display: flex;
 flex-direction: column;
 gap: 5px;
}

label {
 font-weight: bold;
 color: #34495e;
}

input {
 padding: 10px;
 border: 1px solid #ddd;
 border-radius: 4px;
 font-size: 16px;
}

input:focus {
 outline: none;
 border-color: #3498db;
}

button {
 background: #3498db;
 color: white;
 border: none;
 padding: 12px 20px;
 border-radius: 4px;
 cursor: pointer;
 font-size: 16px;
 font-weight: bold;
}

button:hover:not(:disabled) {
 background: #2980b9;
}

button:disabled {
 background: #bdc3c7;
 cursor: not-allowed;
}

.error {
 background: #e74c3c;
 color: white;
 padding: 10px;
 border-radius: 4px;
 margin: 10px 0;
}

.success {
 background: #27ae60;
 color: white;
 padding: 10px;
 border-radius: 4px;
 margin: 10px 0;
}

.users-section h2 {
 color: #2c3e50;
 margin-bottom: 15px;
}

.no-users {
 text-align: center;
 color: #7f8c8d;
 font-style: italic;
 padding: 20px;
}

.users-grid {
 display: grid;
 grid-template-columns: repeat(auto-fill, minmax(300px, 1fr));
 gap: 20px;
}

.user-card {
 background: white;
 border: 1px solid #ddd;
 border-radius: 8px;
 padding: 20px;
 box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

.user-card h3 {
 margin: 0 0 10px 0;
 color: #2c3e50;
}

.email {
 color: #3498db;
 margin: 5px 0;
 font-weight: 500;
}

.date {
 color: #7f8c8d;
 font-size: 0.9em;
 margin: 10px 0;
}

.delete-btn {
 background: #e74c3c;
 font-size: 14px;
 padding: 8px 16px;
}

.delete-btn:hover:not(:disabled) {
 background: #c0392b;
}
</style>
```

Step 7: Testing Your Application

Test the Connection

1. Start your development server:

```bash
npm run dev
```

2. Visit http://localhost:3000 in your browser
3. You should see a form to add users and a list of existing users

Expected Output

When the page loads successfully, you should see:

```
Users (2)
[User cards displaying John Doe and Jane Smith]
```

Hands-On Exercises

Exercise 1: Add User Update Functionality

Goal: Implement the ability to edit existing users.

Steps:

1. Create server/api/users/[id].put.ts
2. Add an edit mode to the user cards
3. Handle form submission for updates

Solution for the API route:

```typescript
// server/api/users/[id].put.ts
import { sql } from '~/utils/database'

export default defineEventHandler(async (event) => {
 try {
 const id = getRouterParam(event, 'id')
 const body = await readBody(event)
 const { name, email } = body

 if (!name || !email) {
 throw createError({
 statusCode: 400,
 statusMessage: 'Name and email are required'
})
}

 const result = await sql`
 UPDATE users
 SET name = ${name}, email = ${email}
 WHERE id = ${id}
 RETURNING id, name, email, created_at
`

 if (result.length === 0) {
 throw createError({
 statusCode: 404,
 statusMessage: 'User not found'
})
}

 return {
 success: true,
 data: result[0],
 message: 'User updated successfully'
}
} catch (error) {
 if (error.statusCode) {
 throw error
}

 throw createError({
 statusCode: 500,
 statusMessage: 'Failed to update user',
 data: error.message
})
}
})
```

Exercise 2: Add Pagination

Goal: Implement pagination for the users list.

Hint: Modify the GET API to accept page and limit query parameters and use SQL LIMIT and OFFSET.

Exercise 3: Add Search Functionality

Goal: Allow users to search by name or email.

Hint: Use SQL ILIKE operator for case-insensitive search.

Troubleshooting Tips

Common Issues and Solutions

1. "connect ENOTFOUND" Error

Problem: Cannot connect to Neon database.

Solutions:

- Check your DATABASE_URL in .env
- Ensure your Neon project is not suspended
- Verify your internet connection
- Check if your region is accessible

```bash
# Test your connection string manually
echo $DATABASE_URL
```

2. "relation 'users' does not exist" Error

Problem: The users table hasn't been created.

Solution:

```sql
-- Run this in Neon's SQL Editor
CREATE TABLE IF NOT EXISTS users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100) NOT NULL,
 email VARCHAR(100) UNIQUE NOT NULL,
 created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. Environment Variables Not Loading

Problem: process.env.DATABASE_URL is undefined.

Solutions:

- Restart your development server after adding .env
- Check .env file is in the project root
-

Ensure no spaces around the = in .env

```env
# Wrong L
DATABASE_URL = postgresql://...

# Correct
DATABASE_URL=postgresql://...
```

4. CORS Issues in Production

Problem: API calls failing in production.

Solution: Ensure your API routes are properly configured:

```typescript
// nuxt.config.ts
export default defineNuxtConfig({
 nitro: {
 experimental: {
 wasm: true
}
}
})
```

5. SSL Connection Issues

Problem: SSL-related connection errors.

Solution: Ensure your connection string includes ?sslmode=require:

```env
DATABASE_URL=postgresql://user:pass@host/db?sslmode=require
```

Debugging Commands

Check if your server routes are working:

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

Expected response:

```json
{
"success": true,
"data": [
{
"id": 1,
"name": "John Doe",
"email": "john@example.com",
"created_at": "2024-01-01T12:00:00.000Z"
}
]
}
```

Development vs Production

Important differences:

Environment	Database Connection	Error Handling
Development	Direct connection pool	Detailed error messages
Production	Serverless/Edge compatible	Generic error messages

For production deployment, ensure:

- Environment variables are set in your hosting platform
- Connection pooling is properly configured
- Error messages don't expose sensitive information

Summary

Congratulations! <‰You've successfully:

- Set up a Nuxt 3 application
- Created a Neon PostgreSQL database
- Connected your frontend to the database using server API routes
- Implemented CRUD operations (Create, Read, Delete)
- Added proper error handling and user feedback
- Created a responsive user interface

Next Steps

To further enhance your application, consider:

1. Authentication: Add user authentication with libraries like @sidebase/nuxt-auth
2. Validation: Implement schema validation with zod or joi
3. Caching: Add caching layers for better performance
4. Testing: Write unit and integration tests
5. Deployment: Deploy to Vercel, Netlify, or other platforms

Additional Resources

- Nuxt 3 Documentation
- Neon Documentation
- Neon Serverless Driver
- Vue 3 Composition API

Happy coding! =€