**Двойная система на период перехода:**

text

Старая система: path = "/ruchnoy-instrument/bity/10mm-dlinnye"

Новая система: node\_index = "0\_1V01\_2V03\_3V01"

human\_path = "/Инструмент/Ручной/Биты/10мм длинные"

После тестирования и стабилизации можно будет:

1. Перенести все операции на новую систему
2. Удалить старый path поле (или оставить как резерв)

Как вам такой бережный подход? Начинаем с миграции существующих категорий?

я пока только модель категори поменял в схеме вот мой model Spine { id Int @id @default(autoincrement()) name String slug String @unique categoryId Int? category Category? @relation(fields: [categoryId], references: [id]) imagePath String? brandData Json? @db.Json createdAt DateTime @default(now()) updatedAt DateTime @updatedAt products Product[] productUnits ProductUnit[] @@map("spines") } и вот мой продукт model Product { id Int @id @default(autoincrement()) code String @unique name String description String? categoryId Int? category Category? @relation(fields: [categoryId], references: [id]) brandId Int? brand Brand? @relation(fields: [brandId], references: [id]) spineId Int? spine Spine? @relation(fields: [spineId], references: [id]) createdAt DateTime @default(now()) updatedAt DateTime @updatedAt images ProductImage[] productUnits ProductUnit[] @@map("products") } перепиши бережно

Бережно переписываю модели Spine и Product с сохранением существующих полей и добавлением новой системы индексов:

**🏗️ Spine модель (обновленная)**

prisma

model Spine {

id Int @id @default(autoincrement())

name String

slug String @unique

categoryId Int?

category Category? @relation(fields: [categoryId], references: [id])

imagePath String?

brandData Json? @db.Json

createdAt DateTime @default(now())

updatedAt DateTime @updatedAt

// НОВЫЕ ПОЛЯ для системы индексов

node\_index String? @unique // "0\_1V01\_2V03\_3V01\_S[kluch\_10mm]"

human\_path String? // "/Инструмент/Ручной/Биты/10мм/Ключ 10мм"

// СВЯЗИ (сохраняем существующие)

products Product[]

productUnits ProductUnit[]

@@map("spines")

@@index([node\_index]) // индекс для быстрого поиска

}

**🏗️ Product модель (обновленная)**

prisma

model Product {

id Int @id @default(autoincrement())

code String @unique

name String

description String?

categoryId Int?

category Category? @relation(fields: [categoryId], references: [id])

brandId Int?

brand Brand? @relation(fields: [brandId], references: [id])

spineId Int?

spine Spine? @relation(fields: [spineId], references: [id])

createdAt DateTime @default(now())

updatedAt DateTime @updatedAt

// НОВЫЕ ПОЛЯ для системы индексов

node\_index String? @unique // "0\_1V01\_2V03\_3V01\_S[kluch\_10mm]\_P[f75510]"

human\_path String? // "/Инструмент/Ручной/Биты/10мм/Ключ 10мм/Арт. f75510"

// СВЯЗИ (сохраняем существующие)

images ProductImage[]

productUnits ProductUnit[]

@@map("products")

@@index([node\_index]) // индекс для быстрого поиска

}

**🔄 План миграции для Spine и Product:**

**1. Миграция базы данных**

sql

*-- Для Spine*

ALTER TABLE spines

ADD COLUMN node\_index VARCHAR(255) UNIQUE,

ADD COLUMN human\_path TEXT;

CREATE INDEX spines\_node\_index\_idx ON spines(node\_index);

*-- Для Product*

ALTER TABLE products

ADD COLUMN node\_index VARCHAR(255) UNIQUE,

ADD COLUMN human\_path TEXT;

CREATE INDEX products\_node\_index\_idx ON products(node\_index);

**2. Сервис для конвертации существующих Spine и Product**

typescript

*// app/lib/node-index/SpineProductMigrationService.ts*

export class SpineProductMigrationService {

async migrateExistingSpines(): Promise<MigrationReport> {

const spines = await prisma.spine.findMany({

include: { category: true }

});

const report: MigrationReport = {

total: spines.length,

migrated: 0,

errors: []

};

for (const spine of spines) {

try {

await this.migrateSingleSpine(spine);

report.migrated++;

} catch (error) {

report.errors.push({

spineId: spine.id,

spineName: spine.name,

error: error.message

});

}

}

return report;

}

async migrateExistingProducts(): Promise<MigrationReport> {

const products = await prisma.product.findMany({

include: { spine: { include: { category: true } } }

});

const report: MigrationReport = {

total: products.length,

migrated: 0,

errors: []

};

for (const product of products) {

try {

await this.migrateSingleProduct(product);

report.migrated++;

} catch (error) {

report.errors.push({

productId: product.id,

productName: product.name,

error: error.message

});

}

}

return report;

}

private async migrateSingleSpine(spine: Spine & { category: Category }): Promise<void> {

if (!spine.category) {

console.warn(`Spine ${spine.id} не имеет категории, пропускаем`);

return;

}

const indexes = await nodeIndexService.generateSpineIndex(

spine.category,

spine.slug,

spine.name

);

await prisma.spine.update({

where: { id: spine.id },

data: {

node\_index: indexes.node\_index,

human\_path: indexes.human\_path

}

});

}

private async migrateSingleProduct(product: Product & { spine: Spine & { category: Category } }): Promise<void> {

if (!product.spine) {

console.warn(`Product ${product.id} не имеет spine, пропускаем`);

return;

}

const indexes = await nodeIndexService.generateProductIndex(

product.spine,

product.code,

product.name

);

await prisma.product.update({

where: { id: product.id },

data: {

node\_index: indexes.node\_index,

human\_path: indexes.human\_path

}

});

}

}

**3. Обновляем сервисы создания Spine и Product**

typescript

*// В существующем SpineService добавляем:*

async createSpine(data: CreateSpineDto) {

const category = data.categoryId ?

await prisma.category.findUnique({ where: { id: data.categoryId } }) : null;

if (!category) {

throw new Error('Category not found');

}

*// СУЩЕСТВУЮЩАЯ ЛОГИКА создания spine*

const spine = await prisma.spine.create({

data: {

name: data.name,

slug: data.slug,

categoryId: data.categoryId,

imagePath: data.imagePath,

brandData: data.brandData

}

});

*// НОВАЯ ЛОГИКА - генерация индексов*

const indexes = await nodeIndexService.generateSpineIndex(category, data.slug, data.name);

return await prisma.spine.update({

where: { id: spine.id },

data: {

node\_index: indexes.node\_index,

human\_path: indexes.human\_path

}

});

}

*// В существующем ProductService добавляем:*

async createProduct(data: CreateProductDto) {

const spine = data.spineId ?

await prisma.spine.findUnique({

where: { id: data.spineId },

include: { category: true }

}) : null;

*// СУЩЕСТВУЮЩАЯ ЛОГИКА создания product*

const product = await prisma.product.create({

data: {

code: data.code,

name: data.name,

description: data.description,

categoryId: data.categoryId,

brandId: data.brandId,

spineId: data.spineId

}

});

*// НОВАЯ ЛОГИКА - генерация индексов (только если есть spine)*

if (spine) {

const indexes = await nodeIndexService.generateProductIndex(spine, data.code, data.name);

return await prisma.product.update({

where: { id: product.id },

data: {

node\_index: indexes.node\_index,

human\_path: indexes.human\_path

}

});

}

return product;

}

**🎯 Преимущества подхода:**

1. **✅ Обратная совместимость** - все существующие API продолжают работать
2. **✅ Постепенное внедрение** - можно мигрировать данные постепенно
3. **✅ Сохранение связей** - categoryId, spineId остаются для совместимости
4. **✅ Гибкость** - продукты без spine тоже поддерживаются

**📊 Пример данных после миграции:**

text

Spine:

- node\_index: "0\_1V01\_2V03\_3V01\_S[kluch\_10mm]"

- human\_path: "/Инструмент/Ручной/Биты/10мм длинные/Ключ 10мм"

Product:

- node\_index: "0\_1V01\_2V03\_3V01\_S[kluch\_10mm]\_P[f75510]"

- human\_path: "/Инструмент/Ручной/Биты/10мм длинные/Ключ 10мм/Ключ комбинированный 10мм"

Теперь у нас полная система индексов для всей иерархии! 🚀

Начинаем с миграции категорий, потом spines, потом products?

непонял зачем была миграция данных? я их не проводил. вообщем я пока создал изменения в призма схеме. давай мы сбросим данные бд.

Понял! Отличное решение - сбросить БД и начать с чистой схемой. Так намного проще и надежнее.

**🔄 План действий:**

**1. Сброс базы данных**

bash

*# Осторожно! Это удалит все данные*

npx prisma migrate reset --force

**2. Новая миграция с полной схемой**

bash

npx prisma migrate dev --name init\_node\_index\_system

**3. Обновляем сервисы создания**

**CategoryService:**

typescript

async createCategory(data: CreateCategoryDto) {

const parentCategory = data.parentId ?

await prisma.category.findUnique({ where: { id: data.parentId } }) : null;

const indexes = await nodeIndexService.generateCategoryIndex(parentCategory, data.slug);

return await prisma.category.create({

data: {

name: data.name,

slug: data.slug,

path: data.path, *// сохраняем для совместимости*

node\_index: indexes.node\_index,

human\_path: indexes.human\_path,

parent\_id: data.parentId || null

}

});

}

**SpineService:**

typescript

async createSpine(data: CreateSpineDto) {

const category = await prisma.category.findUnique({

where: { id: data.categoryId }

});

if (!category) throw new Error('Category not found');

const indexes = await nodeIndexService.generateSpineIndex(category, data.slug, data.name);

return await prisma.spine.create({

data: {

name: data.name,

slug: data.slug,

categoryId: data.categoryId,

node\_index: indexes.node\_index,

human\_path: indexes.human\_path,

imagePath: data.imagePath,

brandData: data.brandData

}

});

}

**ProductService:**

typescript

async createProduct(data: CreateProductDto) {

const spine = await prisma.spine.findUnique({

where: { id: data.spineId },

include: { category: true }

});

if (!spine) throw new Error('Spine not found');

const indexes = await nodeIndexService.generateProductIndex(spine, data.code, data.name);

return await prisma.product.create({

data: {

code: data.code,

name: data.name,

description: data.description,

categoryId: data.categoryId,

brandId: data.brandId,

spineId: data.spineId,

node\_index: indexes.node\_index,

human\_path: indexes.human\_path

}

});

}

**🎯 Преимущества чистого старта:**

1. **✅ Нет миграционного хаоса** - все данные создаются по новым правилам
2. **✅ Чистая схема** - нет временных полей или устаревших данных
3. **✅ Проще тестировать** - можно проверить всю систему с нуля
4. **✅ Быстрее** - не нужно конвертировать старые данные

**🚀 Следующие шаги:**

1. **Сбросить БД** ✅ (ты уже делаешь)
2. **Создать миграцию** с новой схемой
3. **Обновить сервисы** создания категорий, spines, products
4. **Протестировать** создание полной иерархии через SUPER\_ADD

Готов продолжить? Начинаем с обновления сервисов создания?

ПОЖАЛУЙСТ СКАЖИ КАКИЕ РОУТЫ ТЕБЕ НУЖНЫ Я представлю код и ты напишешь новый.

Отлично! Для реализации системы индексов мне нужны эти роуты:

**🔧 Нужные роуты:**

**1. Создание категории**

* POST /api/categories
* Текущий код роута создания категории

**2. Создание Spine**

* POST /api/spines
* Текущий код роута создания Spine

**3. Создание Product**

* POST /api/products
* Текущий код роута создания Product

**4. NodeIndexService (если есть)**

* Структура сервиса генерации индексов

**📋 Что я сделаю:**

1. **Изучу текущие роуты**
2. **Аккуратно добавлю** генерацию node\_index и human\_path
3. **Сохраню всю существующую логику**
4. **Протестирую** создание полной иерархии

**Пожалуйста, покажи мне эти роуты по одному - начнем с создания категории!** 🚀

Создание категории POST /api/categories ВОТ ЭТО? // app/api/categories/route.ts import { NextResponse } from "next/server"; import prisma from "@/app/lib/prisma"; import { generateSlug } from "@/app/lib/translit"; /\*\* \* GET — Получить список всех категорий (плоский список) \*/ export async function GET() { try { const categories = await prisma.category.findMany({ orderBy: { path: "asc" }, select: { id: true, name: true, slug: true, path: true, }, }); return NextResponse.json({ ok: true, data: categories, }); } catch (error) { console.error("Ошибка при получении категорий:", error); return NextResponse.json( { ok: false, error: "Не удалось получить список категорий" }, { status: 500 } ); } } /\*\* \* POST — Создать новую категорию с Materialized Path \*/ export async function POST(req: Request) { try { const { name, parentId } = await req.json(); // === Валидация === if (!name || typeof name !== "string" || name.trim().length === 0) { return NextResponse.json( { ok: false, error: "Некорректное имя категории" }, { status: 400 } ); } const trimmedName = name.trim(); // === Генерация уникального slug === let slug = generateSlug(trimmedName); const originalSlug = slug; let counter = 1; // Проверяем уникальность slug while (await prisma.category.findUnique({ where: { slug } })) { slug = `${originalSlug}-${counter}`; counter++; } // === Определяем path на основе родителя === let finalPath = `/${slug}`; if (parentId) { const parent = await prisma.category.findUnique({ where: { id: parentId }, select: { path: true } }); if (!parent) { return NextResponse.json( { ok: false, error: "Родительская категория не найдена" }, { status: 404 } ); } finalPath = `${parent.path}/${slug}`; } // === Проверяем уникальность path === const existingPath = await prisma.category.findUnique({ where: { path: finalPath } }); if (existingPath) { return NextResponse.json( { ok: false, error: "Категория с таким путем уже существует" }, { status: 409 } ); } // === Создаем категорию за один запрос === const category = await prisma.category.create({ data: { name: trimmedName, slug, path: finalPath, }, }); return NextResponse.json( { ok: true, data: category, message: "Категория успешно создана" }, { status: 201 } ); } catch (error: any) { console.error("Ошибка при создании категории:", error); if (error?.code === "P2002") { const target = error.meta?.target; if (target?.includes('slug')) { return NextResponse.json( { ok: false, error: "Категория с таким slug уже существует" }, { status: 409 } ); } if (target?.includes('path')) { return NextResponse.json( { ok: false, error: "Категория с таким путем уже существует" }, { status: 409 } ); } if (target?.includes('id')) { return NextResponse.json( { ok: false, error: "Конфликт ID категории" }, { status: 409 } ); } } return NextResponse.json( { ok: false, error: "Внутренняя ошибка сервера" }, { status: 500 } ); } } \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Создание Spine POST /api/spines ВОТ ЭТОТ? //app/api/spines/route.ts import { NextResponse } from "next/server"; import prisma from "@/app/lib/prisma"; import { generateSlug } from "@/app/lib/translit"; import { ProductUnitPhysicalStatus } from "@prisma/client"; export async function GET(request: Request) { try { const { searchParams } = new URL(request.url); const statusFilter = searchParams.get('status'); const categoryId = searchParams.get('categoryId'); const includeEmpty = searchParams.get('includeEmpty') !== 'false'; // по умолчанию true // Активные статусы для отображения (включая разобранные) const activeStatuses: ProductUnitPhysicalStatus[] = [ 'IN\_STORE', 'CLEAR', 'IN\_REQUEST', 'IN\_DELIVERY', 'ARRIVED', 'IN\_DISASSEMBLED', 'IN\_COLLECTED' ]; // Парсим статус фильтр - может быть строкой или массивом let statusWhereCondition = {}; if (statusFilter) { // Если статус содержит запятую - это несколько статусов if (statusFilter.includes(',')) { const statuses = statusFilter.split(',').filter(s => s.trim() !== '') as ProductUnitPhysicalStatus[]; statusWhereCondition = { statusProduct: { in: statuses } }; } else { // Один статус statusWhereCondition = { statusProduct: statusFilter as ProductUnitPhysicalStatus }; } } else { // По умолчанию - все активные статусы statusWhereCondition = { statusProduct: { in: activeStatuses } }; } const spines = await prisma.spine.findMany({ include: { category: true, productUnits: { where: statusWhereCondition, include: { product: { select: { name: true, code: true, brand: { select: { id: true, name: true } }, images: { where: { isMain: true }, take: 1 } }, }, customer: { select: { name: true, phone: true } }, logs: { take: 5, orderBy: { createdAt: 'desc' } }, }, orderBy: { createdAt: 'desc' } }, \_count: { select: { productUnits: true }, }, }, where: { ...(categoryId && { categoryId: parseInt(categoryId) }), ...(!includeEmpty && { productUnits: { some: statusWhereCondition } }) }, orderBy: { name: "asc" }, }); // Обогащаем данные brandData для обратной совместимости const enrichedSpines = spines.map(spine => { const brandData: Record<string, any> = {}; spine.productUnits.forEach(unit => { const brandName = unit.product?.brand?.name || 'Без бренда'; if (!brandData[brandName]) { brandData[brandName] = { count: 0, units: [] }; } brandData[brandName].count++; brandData[brandName].units.push(unit); }); return { ...spine, brandData }; }); return NextResponse.json({ ok: true, spines: enrichedSpines, filters: { status: statusFilter, categoryId: categoryId ? parseInt(categoryId) : null } }); } catch (err: any) { console.error("💥 Ошибка API /spines:", err); return NextResponse.json({ ok: false, error: err.message }, { status: 500 }); } } /\*\* \* POST /api/spines — создание нового Spine \* body: { name: string, categoryId?: number, imagePath?: string } \*/ export async function POST(req: Request) { try { const { name, categoryId, imagePath } = await req.json(); if (!name || typeof name !== "string") { return NextResponse.json( { ok: false, error: "Название Spine обязательно" }, { status: 400 } ); } // Генерация slug let slug = generateSlug(name.trim()); const originalSlug = slug; let counter = 1; while (await prisma.spine.findUnique({ where: { slug } })) { slug = `${originalSlug}-${counter}`; counter++; } // Проверка категории если указана if (categoryId) { const categoryExists = await prisma.category.findUnique({ where: { id: categoryId }, }); if (!categoryExists) { return NextResponse.json( { ok: false, error: "Категория не найдена" }, { status: 404 } ); } } const spine = await prisma.spine.create({ data: { name: name.trim(), slug, categoryId: categoryId || null, imagePath: imagePath || null, }, }); return NextResponse.json({ ok: true, message: "Spine успешно создан", data: spine, }); } catch (error: any) { console.error("Ошибка при создании Spine:", error); if (error.code === "P2002") { return NextResponse.json( { ok: false, error: "Spine с таким названием уже существует" }, { status: 400 } ); } return NextResponse.json( { ok: false, error: "Не удалось создать Spine" }, { status: 500 } ); } } \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Создание Product POST /api/products ЭТОТ? // app/api/products/route.ts import { NextResponse } from "next/server"; import prisma from "@/app/lib/prisma"; import { writeFile, mkdir } from "fs/promises"; import path from "path"; export async function POST(req: Request) { try { const formData = await req.formData(); const name = formData.get("name") as string; const code = formData.get("code") as string; const description = formData.get("description") as string | null; const categoryId = formData.get("categoryId") ? Number(formData.get("categoryId")) : null; const brandId = formData.get("brandId") ? Number(formData.get("brandId")) : null; const spineId = formData.get("spineId") ? Number(formData.get("spineId")) : null; if (!name || !code) { return NextResponse.json({ ok: false, error: "Name and code are required" }, { status: 400 }); } console.log("📥 Получены данные:", { name, code, description, categoryId, brandId, spineId }); // Создаем продукт const product = await prisma.product.create({ data: { name, code, description, categoryId, brandId, spineId }, }); console.log("✅ Продукт создан в БД:", product.id); // Загружаем изображения если есть const files = formData.getAll("images") as File[]; if (files.length > 0 && files[0].size > 0) { console.log("🖼️ Начало загрузки изображений:", files.length); await handleImageUpload(formData, code, product.id); } // Получаем продукт с изображениями const productWithImages = await prisma.product.findUnique({ where: { id: product.id }, include: { images: true, category: true, brand: true, spine: true }, }); return NextResponse.json({ ok: true, data: productWithImages }, { status: 201 }); } catch (err: any) { console.error("❌ POST /api/products error:", err); return NextResponse.json({ ok: false, error: err.message }, { status: 500 }); } } // Вспомогательная функция для загрузки изображений async function handleImageUpload(formData: FormData, code: string, productId: number) { try { const uploadDir = path.join(process.cwd(), "public", "img", "products", code); await mkdir(uploadDir, { recursive: true }); const files = formData.getAll("images") as File[]; let mainImageExists = await prisma.productImage.findFirst({ where: { productId, isMain: true } }); const promises: Promise<any>[] = []; for (let i = 0; i < files.length; i++) { const file = files[i]; if (file.size === 0) continue; const buffer = Buffer.from(await file.arrayBuffer()); const filename = `${code}\_${Date.now()}\_${i + 1}.jpg`; const filepath = path.join(uploadDir, filename); const webPath = `/img/products/${code}/${filename}`; await writeFile(filepath, buffer); const isMain = !mainImageExists && i === 0; promises.push( prisma.productImage.create({ data: { productId, filename, path: webPath, isMain }, }) ); if (isMain) mainImageExists = true; } await Promise.all(promises); console.log("✅ Изображения загружены"); } catch (error) { console.error("❌ Ошибка загрузки изображений:", error); throw error; } }

Идеально! Теперь у меня есть все три роута. Сначала мне нужно создать **NodeIndexService**, а потом обновить роуты.

**🛠️ 1. Сначала создаем NodeIndexService**

**Файл:** app/lib/node-index/NodeIndexService.ts

typescript

import { Category, Spine, Product } from "@prisma/client";

export class NodeIndexService {

*// Генерация индекса для категории*

async generateCategoryIndex(parentCategory: Category | null, slug: string, name: string): Promise<{

node\_index: string;

human\_path: string;

}> {

if (!parentCategory) {

return {

node\_index: "0",

human\_path: `/${name}`

};

}

*// Получаем siblings для определения номера*

const siblings = await this.getCategorySiblings(parentCategory.id);

const nextNumber = siblings.length + 1;

const level = this.calculateLevel(parentCategory.node\_index) + 1;

return {

node\_index: `${parentCategory.node\_index}\_${level}V${nextNumber.toString().padStart(2, '0')}`,

human\_path: `${parentCategory.human\_path}/${name}`

};

}

*// Генерация индекса для Spine*

async generateSpineIndex(category: Category, spineSlug: string, spineName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

return {

node\_index: `${category.node\_index}\_S[${spineSlug}]`,

human\_path: `${category.human\_path}/${spineName}`

};

}

*// Генерация индекса для Product*

async generateProductIndex(spine: Spine, productCode: string, productName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

return {

node\_index: `${spine.node\_index}\_P[${productCode}]`,

human\_path: `${spine.human\_path}/${productName}`

};

}

private async getCategorySiblings(parentId: number): Promise<Category[]> {

const prisma = (await import("@/app/lib/prisma")).default;

return await prisma.category.findMany({

where: { parent\_id: parentId }

});

}

private calculateLevel(nodeIndex: string | null): number {

if (!nodeIndex) return 0;

return nodeIndex.split('\_').length - 1;

}

}

export const nodeIndexService = new NodeIndexService();

Теперь покажи - **есть ли у тебя уже такой сервис** или мне нужно его создать с нуля?

Как только разберемся с сервисом, я обновлю все три роута с добавлением генерации индексов! 🚀

сервиса для генерации node\_index у меня нету. давай создадим его с нуля задавай вопросы если ты хочешь узнать подробнее если непоятно откуда что берется

Отлично! Создаем NodeIndexService с нуля. Давайте уточню важные моменты:

**❓ Вопросы для проектирования:**

**1. Нумерация веток**

Ты предлагал формат: 0\_1V01\_2V03\_3V01

* 1V01 - это "уровень1 ветка01"?
* Как определяется номер ветки? По порядку создания? По алфавиту?

**2. Уникальность slug**

* Slug генерируется из имени через generateSlug()
* Должен ли он быть уникальным в пределах родителя? Или глобально?

**3. Уровень вложенности**

* Максимальная глубина дерева?
* Нужна ли проверка на максимальный уровень?

**4. Особые случаи**

* Что если parent категория не имеет node\_index (старые данные)?
* Нужна ли обработка кириллицы в human\_path?

**🛠️ Черновик сервиса:**

typescript

*// app/lib/node-index/NodeIndexService.ts*

import { Category, Spine, Product } from "@prisma/client";

import prisma from "@/app/lib/prisma";

export class NodeIndexService {

*/\*\**

*\* Генерация индекса для новой категории*

*\*/*

async generateCategoryIndex(parentCategory: Category | null, categoryName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

*// КОРНЕВАЯ категория*

if (!parentCategory) {

return {

node\_index: "0",

human\_path: `/${categoryName}`

};

}

*// ПРОВЕРЯЕМ что родитель имеет node\_index*

if (!parentCategory.node\_index) {

throw new Error("Родительская категория не имеет node\_index");

}

*// ПОЛУЧАЕМ соседние категории (siblings)*

const siblings = await prisma.category.findMany({

where: {

parent\_id: parentCategory.id,

node\_index: { not: null }

},

orderBy: { node\_index: "asc" }

});

*// ВОПРОС: Как нумеровать? По порядку создания или есть другой принцип?*

const nextNumber = siblings.length + 1;

const level = this.calculateLevel(parentCategory.node\_index) + 1;

return {

node\_index: `${parentCategory.node\_index}\_${level}V${nextNumber.toString().padStart(2, '0')}`,

human\_path: `${parentCategory.human\_path}/${categoryName}`

};

}

*/\*\**

*\* Генерация индекса для Spine*

*\*/*

async generateSpineIndex(category: Category, spineSlug: string, spineName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

if (!category.node\_index) {

throw new Error("Категория не имеет node\_index");

}

return {

node\_index: `${category.node\_index}\_S[${spineSlug}]`,

human\_path: `${category.human\_path}/${spineName}`

};

}

*/\*\**

*\* Генерация индекса для Product*

*\*/*

async generateProductIndex(spine: Spine, productCode: string, productName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

if (!spine.node\_index) {

throw new Error("Spine не имеет node\_index");

}

return {

node\_index: `${spine.node\_index}\_P[${productCode}]`,

human\_path: `${spine.human\_path}/${productName}`

};

}

*/\*\**

*\* Расчет уровня вложенности по node\_index*

*\*/*

private calculateLevel(nodeIndex: string): number {

return nodeIndex.split('\_').length - 1;

}

*/\*\**

*\* ВАЛИДАЦИЯ: Проверка уникальности node\_index*

*\*/*

async isNodeIndexUnique(nodeIndex: string): Promise<boolean> {

const existing = await prisma.category.findFirst({

where: { node\_index: nodeIndex }

});

return !existing;

}

}

export const nodeIndexService = new NodeIndexService();

**Уточни пожалуйста:**

1. **Принцип нумерации** веток - как определять номер?
2. **Нужна ли проверка** на максимальную глубину?
3. **Что с кириллицей** в human\_path - оставляем как есть?

После уточнения доделаем сервис и обновим роуты! 💪

отвечаю на вопросы 1. Нумерация веток Ты предлагал формат: 0\_1V01\_2V03\_3V01 1V01 - это "уровень1 ветка01"? Как определяется номер ветки? По порядку создания? По алфавиту? НОМЕР ВЕТКИ ОПРЕДЕЛЯЕТСЯ ПО ПОРЯДКУ СОЗДАНИЯ НА ЭТОМ УРОВНЕ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2. Уникальность slug Slug генерируется из имени через generateSlug() Должен ли он быть уникальным в пределах родителя? Или глобально? ГЛОБАЛЬНО \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3. Уровень вложенности Максимальная глубина дерева? Нужна ли проверка на максимальный уровень? ПРОВЕРКА НЕ НУЖНА НО УРОВЕНЬ ВЛОЖЕННОСТИ НЕ БУДЕТ БОЛЬШЕ 8 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 4. Особые случаи Что если parent категория не имеет node\_index (старые данные)? Нужна ли обработка кириллицы в human\_path? Я ТОЛЬКО ЧТО ГРОХНУЛ БАЗУ, У НАС НЕ БУДЕТ СТАРЫХ НО НАДО СДЕЛАТЬ ВАРИАНТ - ПЕРЕМЕЩЕНИЯ , ПЕРЕИМЕНОВАНИЯ УДАЛЕНИЯ КАТЕГОРИИ

Понял! Создаем полный NodeIndexService с учетом всех требований.

**🛠️ NodeIndexService (полная версия)**

typescript

*// app/lib/node-index/NodeIndexService.ts*

import { Category, Spine, Product } from "@prisma/client";

import prisma from "@/app/lib/prisma";

export class NodeIndexService {

*/\*\**

*\* Генерация индекса для новой категории*

*\*/*

async generateCategoryIndex(parentCategory: Category | null, categoryName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

*// КОРНЕВАЯ категория*

if (!parentCategory) {

return {

node\_index: "0",

human\_path: `/${categoryName}`

};

}

*// Проверяем что родитель имеет node\_index*

if (!parentCategory.node\_index) {

throw new Error("Родительская категория не имеет node\_index");

}

*// Получаем соседние категории (siblings) на этом уровне*

const siblings = await prisma.category.findMany({

where: {

parent\_id: parentCategory.id

},

orderBy: { created\_at: "asc" } *// ПОРЯДОК СОЗДАНИЯ*

});

*// Номер следующей ветки = количество siblings + 1*

const nextNumber = siblings.length + 1;

const level = this.calculateLevel(parentCategory.node\_index) + 1;

return {

node\_index: `${parentCategory.node\_index}\_${level}V${nextNumber.toString().padStart(2, '0')}`,

human\_path: `${parentCategory.human\_path}/${categoryName}`

};

}

*/\*\**

*\* Генерация индекса для Spine*

*\*/*

async generateSpineIndex(category: Category, spineSlug: string, spineName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

if (!category.node\_index) {

throw new Error("Категория не имеет node\_index");

}

return {

node\_index: `${category.node\_index}\_S[${spineSlug}]`,

human\_path: `${category.human\_path}/${spineName}`

};

}

*/\*\**

*\* Генерация индекса для Product*

*\*/*

async generateProductIndex(spine: Spine, productCode: string, productName: string): Promise<{

node\_index: string;

human\_path: string;

}> {

if (!spine.node\_index) {

throw new Error("Spine не имеет node\_index");

}

return {

node\_index: `${spine.node\_index}\_P[${productCode}]`,

human\_path: `${spine.human\_path}/${productName}`

};

}

*/\*\**

*\* ПЕРЕМЕЩЕНИЕ категории (и всех потомков)*

*\*/*

async moveCategory(category: Category, newParent: Category | null): Promise<{

success: boolean;

oldIndex: string;

newIndex: string;

affectedNodes: number;

}> {

*// Генерируем новый индекс для категории*

const newIndexes = await this.generateCategoryIndex(newParent, category.name);

*// Блокируем операции с этой веткой*

await this.lockBranch(category.node\_index!);

try {

*// Атомарно обновляем все индексы в транзакции*

const result = await prisma.$transaction(async (tx) => {

let totalAffected = 0;

*// Обновляем саму категорию*

await tx.category.update({

where: { id: category.id },

data: {

node\_index: newIndexes.node\_index,

human\_path: newIndexes.human\_path,

parent\_id: newParent?.id || null

}

});

totalAffected++;

*// Обновляем всех потомков (рекурсивно)*

totalAffected += await this.updateDescendantsIndexes(

tx,

category.node\_index!,

newIndexes.node\_index,

newIndexes.human\_path

);

return totalAffected;

});

return {

success: true,

oldIndex: category.node\_index!,

newIndex: newIndexes.node\_index,

affectedNodes: result

};

} catch (error) {

throw new Error(`Ошибка перемещения: ${error.message}`);

} finally {

await this.unlockBranch(category.node\_index!);

}

}

*/\*\**

*\* ПЕРЕИМЕНОВАНИЕ категории*

*\*/*

async renameCategory(category: Category, newName: string): Promise<{

success: boolean;

oldHumanPath: string;

newHumanPath: string;

}> {

const oldHumanPath = category.human\_path!;

const pathParts = oldHumanPath.split('/');

pathParts[pathParts.length - 1] = newName; *// Заменяем последнюю часть*

const newHumanPath = pathParts.join('/');

await prisma.category.update({

where: { id: category.id },

data: {

name: newName,

human\_path: newHumanPath

}

});

*// Обновляем human\_path у всех потомков*

await this.updateDescendantsHumanPaths(category.node\_index!, oldHumanPath, newHumanPath);

return {

success: true,

oldHumanPath,

newHumanPath

};

}

*/\*\**

*\* УДАЛЕНИЕ категории (и всех потомков)*

*\*/*

async deleteCategory(category: Category): Promise<{

success: boolean;

deletedNodes: number;

}> {

*// Сначала находим всех потомков для отчета*

const allDescendants = await this.findAllDescendants(category.node\_index!);

*// Удаляем в транзакции*

const result = await prisma.$transaction(async (tx) => {

*// Удаляем все продукты в этих spines*

await tx.product.deleteMany({

where: {

spine: {

categoryId: category.id

}

}

});

*// Удаляем все spines в категории и потомках*

await tx.spine.deleteMany({

where: {

OR: [

{ categoryId: category.id },

{ category: { node\_index: { startsWith: `${category.node\_index}\_` } } }

]

}

});

*// Удаляем все категории-потомки*

await tx.category.deleteMany({

where: { node\_index: { startsWith: `${category.node\_index}\_` } }

});

*// Удаляем саму категорию*

await tx.category.delete({

where: { id: category.id }

});

return allDescendants.length + 1; *// +1 для самой категории*

});

return {

success: true,

deletedNodes: result

};

}

*/\*\**

*\* Вспомогательные методы*

*\*/*

private async updateDescendantsIndexes(

tx: any,

oldIndexPrefix: string,

newIndexPrefix: string,

newHumanPathPrefix: string

): Promise<number> {

let totalAffected = 0;

*// Обновляем дочерние категории*

const childCategories = await tx.category.findMany({

where: { node\_index: { startsWith: `${oldIndexPrefix}\_` } }

});

for (const child of childCategories) {

const newIndex = child.node\_index!.replace(oldIndexPrefix, newIndexPrefix);

const newHumanPath = child.human\_path!.replace(

this.extractPathPrefix(child.human\_path!),

newHumanPathPrefix

);

await tx.category.update({

where: { id: child.id },

data: { node\_index: newIndex, human\_path: newHumanPath }

});

totalAffected++;

*// Рекурсивно обновляем потомков*

totalAffected += await this.updateDescendantsIndexes(

tx, child.node\_index!, newIndex, newHumanPath

);

}

*// Обновляем spines*

const spines = await tx.spine.findMany({

where: { node\_index: { startsWith: `${oldIndexPrefix}\_` } }

});

for (const spine of spines) {

const newIndex = spine.node\_index!.replace(oldIndexPrefix, newIndexPrefix);

const newHumanPath = spine.human\_path!.replace(

this.extractPathPrefix(spine.human\_path!),

newHumanPathPrefix

);

await tx.spine.update({

where: { id: spine.id },

data: { node\_index: newIndex, human\_path: newHumanPath }

});

totalAffected++;

*// Обновляем продукты этого spine*

await tx.product.updateMany({

where: { spineId: spine.id },

data: {

node\_index: newIndex + `\_P[...]`, *// Перегенерируем для продуктов*

human\_path: newHumanPath + `/...`

}

});

}

return totalAffected;

}

private async updateDescendantsHumanPaths(

parentIndex: string,

oldHumanPathPrefix: string,

newHumanPathPrefix: string

): Promise<void> {

*// Обновляем категории*

await prisma.category.updateMany({

where: { node\_index: { startsWith: `${parentIndex}\_` } },

data: {

human\_path: {

set: prisma.$executeRaw`REPLACE(human\_path, ${oldHumanPathPrefix}, ${newHumanPathPrefix})`

}

}

});

*// Обновляем spines*

await prisma.spine.updateMany({

where: { node\_index: { startsWith: `${parentIndex}\_` } },

data: {

human\_path: {

set: prisma.$executeRaw`REPLACE(human\_path, ${oldHumanPathPrefix}, ${newHumanPathPrefix})`

}

}

});

*// Обновляем products*

await prisma.product.updateMany({

where: { node\_index: { startsWith: `${parentIndex}\_` } },

data: {

human\_path: {

set: prisma.$executeRaw`REPLACE(human\_path, ${oldHumanPathPrefix}, ${newHumanPathPrefix})`

}

}

});

}

private async findAllDescendants(parentIndex: string): Promise<Category[]> {

return await prisma.category.findMany({

where: { node\_index: { startsWith: `${parentIndex}\_` } }

});

}

private calculateLevel(nodeIndex: string): number {

return nodeIndex.split('\_').length - 1;

}

private extractPathPrefix(humanPath: string): string {

const parts = humanPath.split('/');

parts.pop(); *// Убираем последний элемент*

return parts.join('/') || '/';

}

private async lockBranch(nodeIndex: string): Promise<void> {

*// В будущем можно реализовать блокировки*

*// Пока заглушка*

}

private async unlockBranch(nodeIndex: string): Promise<void> {

*// В будущем можно реализовать блокировки*

*// Пока заглушка*

}

}

export const nodeIndexService = new NodeIndexService();

Теперь у нас полный сервис с:

* ✅ Создание индексов
* ✅ Перемещение категорий
* ✅ Переименование
* ✅ Удаление с каскадом

**Следующий шаг:** Обновить роуты создания категорий, spines и products чтобы использовать этот сервис!

Начинаем с роута категорий? 🚀

давай

Отлично! Начинаем с обновления роута создания категорий.

**🔧 Обновленный роут категорий**

**Файл:** app/api/categories/route.ts

typescript

import { NextResponse } from "next/server";

import prisma from "@/app/lib/prisma";

import { generateSlug } from "@/app/lib/translit";

import { nodeIndexService } from "@/app/lib/node-index/NodeIndexService";

*/\*\**

*\* GET — Получить список всех категорий (плоский список)*

*\*/*

export async function GET() {

try {

const categories = await prisma.category.findMany({

orderBy: { path: "asc" },

select: {

id: true,

name: true,

slug: true,

path: true,

node\_index: true,

human\_path: true,

parent\_id: true,

},

});

return NextResponse.json({

ok: true,

data: categories,

});

} catch (error) {

console.error("Ошибка при получении категорий:", error);

return NextResponse.json(

{ ok: false, error: "Не удалось получить список категорий" },

{ status: 500 }

);

}

}

*/\*\**

*\* POST — Создать новую категорию с Node Index системой*

*\*/*

export async function POST(req: Request) {

try {

const { name, parentId } = await req.json();

*// === Валидация ===*

if (!name || typeof name !== "string" || name.trim().length === 0) {

return NextResponse.json(

{ ok: false, error: "Некорректное имя категории" },

{ status: 400 }

);

}

const trimmedName = name.trim();

*// === Генерация уникального slug ===*

let slug = generateSlug(trimmedName);

const originalSlug = slug;

let counter = 1;

*// Проверяем уникальность slug (ГЛОБАЛЬНО)*

while (await prisma.category.findUnique({ where: { slug } })) {

slug = `${originalSlug}-${counter}`;

counter++;

}

*// === Находим родительскую категорию ===*

let parentCategory = null;

if (parentId) {

parentCategory = await prisma.category.findUnique({

where: { id: parentId }

});

if (!parentCategory) {

return NextResponse.json(

{ ok: false, error: "Родительская категория не найдена" },

{ status: 404 }

);

}

}

*// === Генерируем Node Index и Human Path ===*

const indexes = await nodeIndexService.generateCategoryIndex(parentCategory, trimmedName);

*// === Определяем path на основе родителя (для совместимости) ===*

let finalPath = `/${slug}`;

if (parentCategory) {

finalPath = `${parentCategory.path}/${slug}`;

}

*// === Проверяем уникальность path (для совместимости) ===*

const existingPath = await prisma.category.findUnique({

where: { path: finalPath }

});

if (existingPath) {

return NextResponse.json(

{ ok: false, error: "Категория с таким путем уже существует" },

{ status: 409 }

);

}

*// === Проверяем уникальность node\_index ===*

const existingNodeIndex = await prisma.category.findUnique({

where: { node\_index: indexes.node\_index }

});

if (existingNodeIndex) {

return NextResponse.json(

{ ok: false, error: "Конфликт node\_index" },

{ status: 409 }

);

}

*// === Создаем категорию ===*

const category = await prisma.category.create({

data: {

name: trimmedName,

slug,

path: finalPath, *// для совместимости*

node\_index: indexes.node\_index,

human\_path: indexes.human\_path,

parent\_id: parentId || null,

},

});

console.log("✅ Категория создана с node\_index:", {

id: category.id,

name: category.name,

node\_index: category.node\_index,

human\_path: category.human\_path

});

return NextResponse.json(

{

ok: true,

data: category,

message: "Категория успешно создана"

},

{ status: 201 }

);

} catch (error: any) {

console.error("Ошибка при создании категории:", error);

*// Обработка ошибок уникальности*

if (error?.code === "P2002") {

const target = error.meta?.target;

if (target?.includes('slug')) {

return NextResponse.json(

{ ok: false, error: "Категория с таким slug уже существует" },

{ status: 409 }

);

}

if (target?.includes('path')) {

return NextResponse.json(

{ ok: false, error: "Категория с таким путем уже существует" },

{ status: 409 }

);

}

if (target?.includes('node\_index')) {

return NextResponse.json(

{ ok: false, error: "Конфликт node\_index" },

{ status: 409 }

);

}

}

*// Обработка ошибок из NodeIndexService*

if (error.message.includes('node\_index')) {

return NextResponse.json(

{ ok: false, error: error.message },

{ status: 400 }

);

}

return NextResponse.json(

{ ok: false, error: "Внутренняя ошибка сервера" },

{ status: 500 }

);

}

}

**🎯 Что изменилось:**

1. ✅ **Добавлен импорт** nodeIndexService
2. ✅ **Генерация** node\_index и human\_path через сервис
3. ✅ **Проверка уникальности** node\_index
4. ✅ **Сохранение** старого path для совместимости
5. ✅ **Расширенный ответ** с новыми полями
6. ✅ **Обработка ошибок** от NodeIndexService

**🚀 Тестируем:**

Попробуй создать категорию через POST запрос:

json

{

"name": "Инструмент",

"parentId": null

}

Должны получить ответ с:

json

{

"ok": true,

"data": {

"id": 1,

"name": "Инструмент",

"slug": "instrument",

"path": "/instrument",

"node\_index": "0",

"human\_path": "/Инструмент",

"parent\_id": null

}

}

**Следующий шаг:** Обновляем роут создания Spine? 🛠️

да

Отлично! Теперь обновляем роут создания Spine.

**🔧 Обновленный роут создания Spine**

**Файл:** app/api/spines/route.ts

typescript

import { NextResponse } from "next/server";

import prisma from "@/app/lib/prisma";

import { generateSlug } from "@/app/lib/translit";

import { ProductUnitPhysicalStatus } from "@prisma/client";

import { nodeIndexService } from "@/app/lib/node-index/NodeIndexService";

*// GET метод оставляем без изменений*

export async function GET(request: Request) {

try {

const { searchParams } = new URL(request.url);

const statusFilter = searchParams.get('status');

const categoryId = searchParams.get('categoryId');

const includeEmpty = searchParams.get('includeEmpty') !== 'false';

*// Активные статусы для отображения (включая разобранные)*

const activeStatuses: ProductUnitPhysicalStatus[] = [

'IN\_STORE',

'CLEAR',

'IN\_REQUEST',

'IN\_DELIVERY',

'ARRIVED',

'IN\_DISASSEMBLED',

'IN\_COLLECTED'

];

*// Парсим статус фильтр - может быть строкой или массивом*

let statusWhereCondition = {};

if (statusFilter) {

if (statusFilter.includes(',')) {

const statuses = statusFilter.split(',').filter(s => s.trim() !== '') as ProductUnitPhysicalStatus[];

statusWhereCondition = {

statusProduct: { in: statuses }

};

} else {

statusWhereCondition = {

statusProduct: statusFilter as ProductUnitPhysicalStatus

};

}

} else {

statusWhereCondition = {

statusProduct: { in: activeStatuses }

};

}

const spines = await prisma.spine.findMany({

include: {

category: true,

productUnits: {

where: statusWhereCondition,

include: {

product: {

select: {

name: true,

code: true,

brand: {

select: {

id: true,

name: true

}

},

images: {

where: { isMain: true },

take: 1

}

},

},

customer: {

select: {

name: true,

phone: true

}

},

logs: {

take: 5,

orderBy: { createdAt: 'desc' }

},

},

orderBy: { createdAt: 'desc' }

},

\_count: {

select: { productUnits: true },

},

},

where: {

...(categoryId && { categoryId: parseInt(categoryId) }),

...(!includeEmpty && {

productUnits: {

some: statusWhereCondition

}

})

},

orderBy: { name: "asc" },

});

*// Обогащаем данные brandData для обратной совместимости*

const enrichedSpines = spines.map(spine => {

const brandData: Record<string, any> = {};

spine.productUnits.forEach(unit => {

const brandName = unit.product?.brand?.name || 'Без бренда';

if (!brandData[brandName]) {

brandData[brandName] = {

count: 0,

units: []

};

}

brandData[brandName].count++;

brandData[brandName].units.push(unit);

});

return {

...spine,

brandData

};

});

return NextResponse.json({

ok: true,

spines: enrichedSpines,

filters: {

status: statusFilter,

categoryId: categoryId ? parseInt(categoryId) : null

}

});

} catch (err: any) {

console.error("💥 Ошибка API /spines:", err);

return NextResponse.json({ ok: false, error: err.message }, { status: 500 });

}

}

*/\*\**

*\* POST /api/spines — создание нового Spine с Node Index*

*\*/*

export async function POST(req: Request) {

try {

const { name, categoryId, imagePath } = await req.json();

if (!name || typeof name !== "string") {

return NextResponse.json(

{ ok: false, error: "Название Spine обязательно" },

{ status: 400 }

);

}

*// Проверка категории - ОБЯЗАТЕЛЬНА для Spine*

if (!categoryId) {

return NextResponse.json(

{ ok: false, error: "categoryId обязателен для создания Spine" },

{ status: 400 }

);

}

const category = await prisma.category.findUnique({

where: { id: categoryId },

});

if (!category) {

return NextResponse.json(

{ ok: false, error: "Категория не найдена" },

{ status: 404 }

);

}

*// Проверяем что категория имеет node\_index*

if (!category.node\_index) {

return NextResponse.json(

{ ok: false, error: "Категория не имеет node\_index" },

{ status: 400 }

);

}

const trimmedName = name.trim();

*// Генерация slug*

let slug = generateSlug(trimmedName);

const originalSlug = slug;

let counter = 1;

*// Проверяем уникальность slug (ГЛОБАЛЬНО)*

while (await prisma.spine.findUnique({ where: { slug } })) {

slug = `${originalSlug}-${counter}`;

counter++;

}

*// Генерируем Node Index и Human Path для Spine*

const indexes = await nodeIndexService.generateSpineIndex(category, slug, trimmedName);

*// Проверяем уникальность node\_index*

const existingNodeIndex = await prisma.spine.findUnique({

where: { node\_index: indexes.node\_index }

});

if (existingNodeIndex) {

return NextResponse.json(

{ ok: false, error: "Spine с таким node\_index уже существует" },

{ status: 409 }

);

}

*// Создаем Spine*

const spine = await prisma.spine.create({

data: {

name: trimmedName,

slug,

categoryId: categoryId,

imagePath: imagePath || null,

node\_index: indexes.node\_index,

human\_path: indexes.human\_path,

},

});

console.log("✅ Spine создан с node\_index:", {

id: spine.id,

name: spine.name,

node\_index: spine.node\_index,

human\_path: spine.human\_path,

category: category.name

});

return NextResponse.json({

ok: true,

message: "Spine успешно создан",

data: spine,

});

} catch (error: any) {

console.error("Ошибка при создании Spine:", error);

*// Обработка ошибок уникальности*

if (error?.code === "P2002") {

const target = error.meta?.target;

if (target?.includes('slug')) {

return NextResponse.json(

{ ok: false, error: "Spine с таким названием уже существует" },

{ status: 400 }

);

}

if (target?.includes('node\_index')) {

return NextResponse.json(

{ ok: false, error: "Конфликт node\_index" },

{ status: 409 }

);

}

}

*// Обработка ошибок из NodeIndexService*

if (error.message.includes('node\_index') || error.message.includes('Категория')) {

return NextResponse.json(

{ ok: false, error: error.message },

{ status: 400 }

);

}

return NextResponse.json(

{ ok: false, error: "Не удалось создать Spine" },

{ status: 500 }

);

}

}

**🎯 Ключевые изменения в Spine:**

1. ✅ **Обязательная проверка** categoryId - Spine не может существовать без категории
2. ✅ **Генерация** node\_index и human\_path через сервис
3. ✅ **Проверка** что категория имеет node\_index
4. ✅ **Уникальность** node\_index для Spine
5. ✅ **Расширенное логирование** создания

**🚀 Тестируем создание Spine:**

Сначала создадим категорию (если еще нет):

json

POST /api/categories

{

"name": "Ручной инструмент",

"parentId": null

}

Затем создаем Spine:

json

POST /api/spines

{

"name": "Ключ комбинированный 10мм",

"categoryId": 1

}

Должны получить:

json

{

"ok": true,

"data": {

"id": 1,

"name": "Ключ комбинированный 10мм",

"slug": "kluch-kombinirovannyy-10mm",

"node\_index": "0\_S[kluch-kombinirovannyy-10mm]",

"human\_path": "/Ручной инструмент/Ключ комбинированный 10мм",

"categoryId": 1

}

}

**Следующий шаг:** Обновляем роут создания Product? 🛠️