Telegram Shop MVP - Complete Documentation

DeepAgent by Abacus.AI

2025-12-04

🖨️ Telegram Shop MVP - Complete Documentation

Version: 1.0.0
Last Updated: 2025-12-04
Stack: NestJS + Grammy + Prisma + PostgreSQL + Docker

📑 Table of Contents

4. API DOCUMENTATION - Endpoints
5. BOT LOGIC - Telegram Bot
6. DATABASE - Schema
7. ENVIRONMENT - Variables
8. DOCKER - Containerization
9. DEVELOPMENT - Dev Guide
10. TROUBLESHOOTING - Solutions

README.md

🛍 Telegram Shop MVP

> Полнофункциональный интернет-магазин с Telegram-ботом на современном технологическом стеке

📋 Оглавление

📖 Описание проекта

Telegram Shop MVP — это полноценное решение для интернет-магазина, состоящее из:

- 🤖 **Telegram-бота** для клиентов (просмотр каталога, оформление заказов)
- 🔧 **REST API** для управления магазином (админ-панель)
- 🗄️ **PostgreSQL** базы данных для хранения данных
- 📦 **Docker** контейнеризации для легкого развертывания

Проект разработан специально для рынка России/СНГ с поддержкой до 1000 заказов в день.

🛠 Технологический стек

Backend API

- **NestJS 11.0** — прогрессивный Node.js фреймворк
- **TypeScript 5.9** — типизированный JavaScript
- **Prisma 6.0** — современная ORM для работы с БД
- **PostgreSQL 15** — надежная реляционная СУБД
- **JWT** — аутентификация для админ-панели
- **Swagger/OpenAPI** — автодокументация API
- **class-validator** — валидация входных данных

Telegram Bot

- **Grammy 1.38** — современная библиотека для Telegram Bot API
- **TypeScript 5.9** — типобезопасность
- **@grammyjs/conversations** — управление диалогами
- **Axios** — HTTP клиент для взаимодействия с API
- **Winston** — структурированное логирование

Инфраструктура

- **Docker & Docker Compose** — контейнеризация
- **Yarn 4** — менеджер пакетов
- **ESLint** — линтинг кода
- **Jest** — тестирование

✨ Ключевые возможности

Для клиентов (через Telegram-бота)

- ✅ Просмотр каталога товаров с фото
- ✅ Фильтрация по категориям
- ✅ Выбор вариантов товара (цвет, размер)
- ✅ Оформление заказа через диалог
- ✅ Отслеживание истории заказов
- ✅ Управление профилем
- ✅ Автозаполнение данных из Telegram

Для администраторов (через REST API)

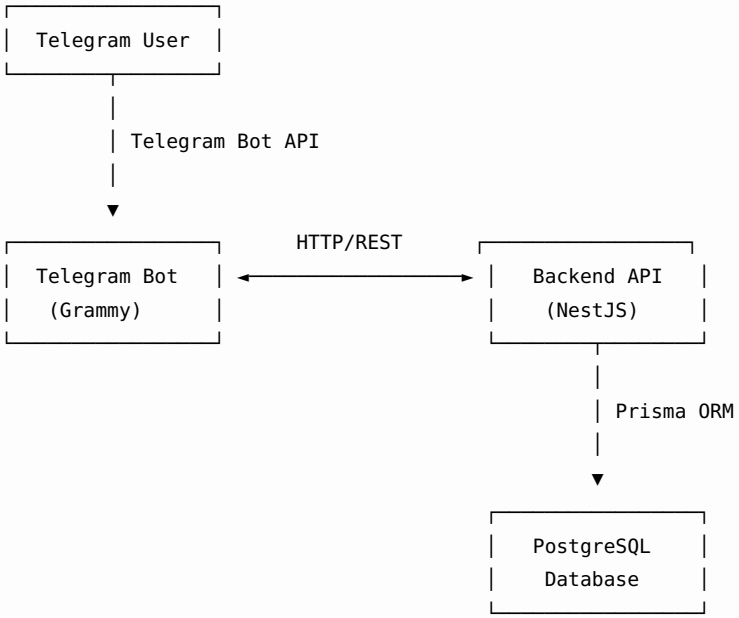
- ✅ Управление товарами и категориями
- ✅ Обработка заказов
- ✅ Управление пользователями
- ✅ JWT-аутентификация
- ✅ Ролевая система (admin, manager)
- ✅ Импорт/экспорт данных (CSV, XLSX, Google Sheets)
- ✅ Аудит действий администраторов
- ✅ Интеграция с аналитикой (GA4, Yandex Metrika)

Технические возможности

- ✅ REST API с полной OpenAPI документацией
- ✅ Webhook поддержка для Telegram

- ✅ Graceful shutdown
- ✅ Health checks
- ✅ Rate limiting
- ✅ CORS настройка
- ✅ Структурированные логи
- ✅ Обработка ошибок

🏗 Архитектура



Компоненты:

1. **Telegram Bot** (`bot/`) — обрабатывает сообщения пользователей
2. **Backend API** (`nodejs_space/`) — бизнес-логика и данные
3. **PostgreSQL** — хранение данных
4. **Docker Compose** — оркестрация контейнеров

Подробнее: ARCHITECTURE.md

🚀 Быстрый старт

Предварительные требования

- **Docker** 20.10+
- **Docker Compose** 2.0+
- **Telegram Bot Token** (получить у @BotFather)

Установка за 3 шага

1️⃣ Клонировать и настроить

```bash
# Распаковать проект
tar -xzf telegram_shop_backend.tar.gz
cd telegram_shop_backend

# Создать .env файл
cp .env.example .env
nano .env  # Указать BOT_TOKEN и JWT_SECRET
```

2️⃣ Запустить через Docker

```bash
# Запустить все сервисы
docker-compose up -d

# Проверить статус
docker-compose ps

# Просмотреть логи
docker-compose logs -f
```

3️⃣ Готово! 🎉

- **API:** http://localhost:3000/api
- **Swagger Docs:** http://localhost:3000/api-docs
- **Telegram Bot:** отправьте `/start` вашему боту

Демо админ: `username: admin` / `password: admin123`

Локальная разработка (без Docker)

Подробная инструкция: SETUP.md

📁 Структура проекта

```
telegram_shop_backend/
├── bot/                          # Telegram Bot
│   ├── src/
│   │   ├── handlers/            # Обработчики команд (/start, /catalog)
│   │   ├── conversations/        # Диалоги (оформление заказа)
│   │   ├── middleware/          # Middleware (логирование, авторизация)
│   │   ├── services/            # API клиент, уведомления
│   │   ├── utils/               # Вспомогательные функции
│   │   └── index.ts            # Точка входа бота
│   ├── package.json
│   └── Dockerfile
│
├── nodejs_space/                 # Backend API
│   ├── src/
│   │   ├── auth/                # JWT аутентификация
│   │   ├── products/            # Управление товарами
│   │   ├── orders/              # Управление заказами
```

```
│   │   ├── users/              # Управление пользователями
│   │   ├── import-export/      # Импорт/экспорт данных
│   │   ├── integrations/       # GA4, Yandex Metrika
│   │   ├── webhook/            # Telegram webhook
│   │   ├── common/             # Общие компоненты
│   │   ├── prisma/             # Prisma сервис
│   │   └── main.ts             # Точка входа API
│   ├── prisma/
│   │   ├── schema.prisma       # Схема БД
│   │   └── seed.ts             # Начальные данные
│   ├── package.json
│   └── Dockerfile
│
├── docker-compose.yml           # Docker оркестрация
├── .env.example                 # Пример переменных окружения
└── README.md                    # Этот файл
```

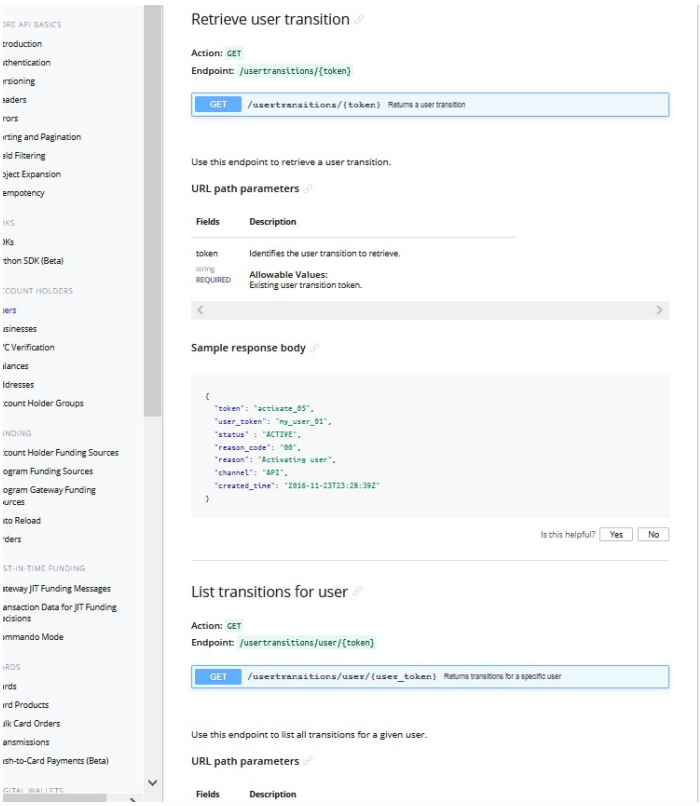
📚 Дополнительная документация

Документ	Описание
ARCHITECTURE.md	Архитектура и взаимодействие компонентов
SETUP.md	Установка локально и на сервере
API_DOCUMENTATION.md	REST API endpoints с примерами
BOT_LOGIC.md	Команды бота и сценарии диалогов
DATABASE.md	Схема БД, таблицы, миграции
ENVIRONMENT.md	Переменные окружения
DOCKER.md	Работа с Docker
DEVELOPMENT.md	Руководство для разработчиков
TROUBLESHOOTING.md	Решение проблем

🎥 Демо

Backend API (Swagger)

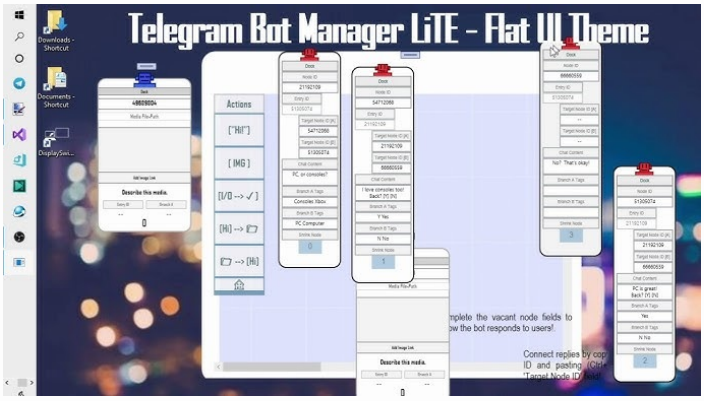
Production: https://smokyyard.abacusai.app/api-docs



Swagger UI

Telegram Bot

Demo Bot: [@SmokyYardBot](https://t.me/SmokyYardBot)



Bot Screenshot

🔐 Безопасность

- ✅ JWT токены для аутентификации админов
- ✅ Bcrypt хеширование паролей
- ✅ CORS защита
- ✅ Rate limiting
- ✅ Input validation через class-validator
- ✅ SQL injection защита через Prisma

- ✅ Environment variables для секретов

⚠ **ВАЖНО:** В production обязательно измените: - `JWT_SECRET` на случайную строку (минимум 32 символа) - Пароль админа по умолчанию - Database credentials

🔋 Тестирование

```
# Unit тесты
cd nodejs_space
yarn test

# E2E тесты
yarn test:e2e

# Coverage
yarn test:cov
```

📊 Мониторинг

Логи

```
# Backend API логи
docker-compose logs -f api

# Bot логи
docker-compose logs -f bot

# БД логи
docker-compose logs -f postgres
```

Health Check

```
# API health
curl http://localhost:3000/api/health

# Database health
docker-compose exec postgres pg_isready -U postgres
```

🤝 Вклад в проект

Проект находится в активной разработке. Будем рады вашему вкладу!

1. Fork проекта
2. Создайте feature branch (`git checkout -b feature/AmazingFeature`)
3. Commit изменения (`git commit -m 'Add AmazingFeature'`)
4. Push в branch (`git push origin feature/AmazingFeature`)
5. Откройте Pull Request

📄 Лицензия

Проект разработан для коммерческого использования. Все права защищены.

👥 Авторы

- Разработка: DeepAgent by Abacus.AI
- Telegram Bot: Grammy Framework
- Backend: NestJS Framework

📞 Поддержка

Если у вас возникли вопросы:

1. Прочитайте TROUBLESHOOTING.md
2. Проверьте Issues
3. Откройте новый Issue с описанием проблемы

🗺 Roadmap

Phase 1 (MVP) ✓ Завершено

- Backend API
- Telegram Bot
- Базовые функции магазина

Phase 2 (В разработке)

- [ ] Интеграция платежных систем (Tinkoff, YooMoney)
- [ ] Интеграция служб доставки (СДЭК, Яндекс.Доставка)
- [ ] Админ веб-панель (React)
- [ ] Многоязычность (6 языков)

Phase 3 (Планируется)

- [ ] 1С УНФ 3.0 интеграция
- [ ] Онлайн-касса (54-ФЗ)
- [ ] LLM/AI ассистент
- [ ] Мобильное приложение

☆ Если проект полезен, поставьте звезду! ☆

ARCHITECTURE.md

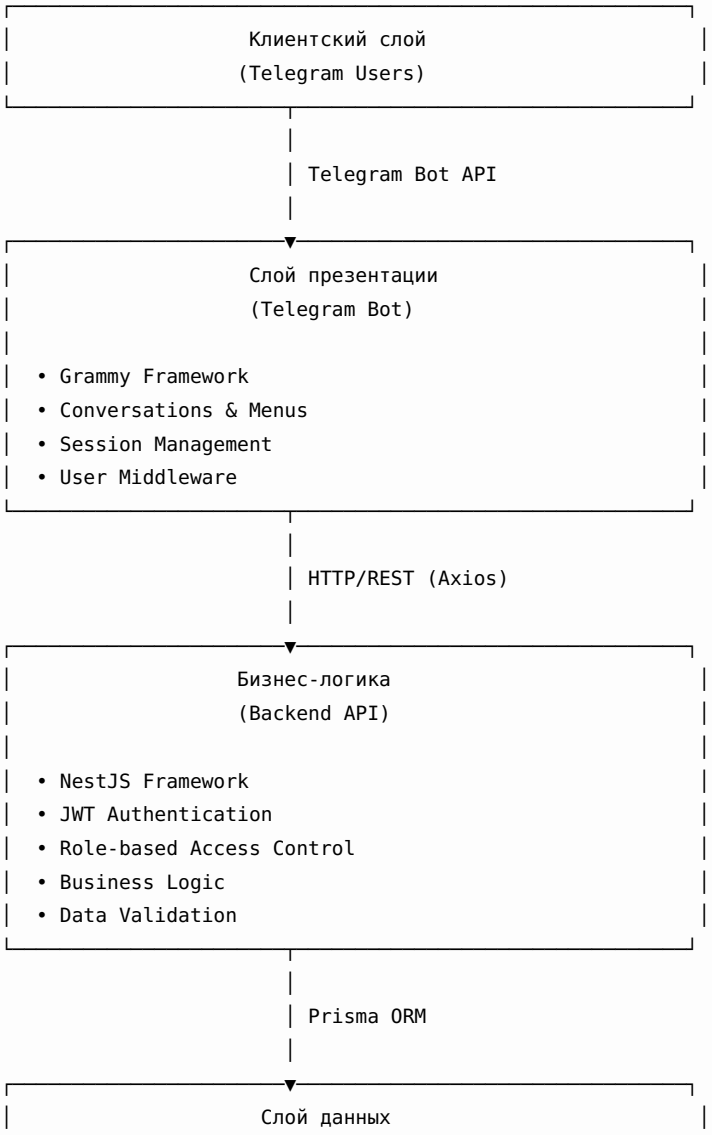
🏗 Архитектура Telegram Shop

MVP

📑 Оглавление

🎯 Общий обзор

Telegram Shop MVP построен на микросервисной архитектуре с четким разделением ответственности:



```
│                   (PostgreSQL)                          │
│                                                         │
│  • Relational Database                                  │
│  • Data Integrity                                       │
│  • Transactions                                         │
│  • Indexes & Constraints                                │
└─────────────────────────────────────────────────────────┘
```

📡 Компоненты системы

1. Telegram Bot (Grammy)

Местоположение: `bot/`

Ответственность: - Обработка сообщений пользователей - Управление состояниями диалогов - Отображение каталога товаров - Оформление заказов через conversations - Уведомления пользователям

Ключевые модули:

```
bot/
├── handlers/                 # Обработчики команд
│   ├── start.ts             # /start, главное меню
│   ├── catalog.ts           # Каталог, пагинация
│   ├── order.ts             # История заказов
│   └── profile.ts           # Профиль пользователя
│
├── conversations/           # Диалоговые сценарии
│   └── order.ts            # Оформление заказа (8 шагов)
│
├── middleware/              # Промежуточные обработчики
│   ├── user.ts             # Авто-создание пользователей
│   ├── logging.ts          # Логирование запросов
│   └── error-handler.ts    # Обработка ошибок
│
├── services/                # Сервисы
│   ├── api-client.ts       # HTTP клиент для Backend API
│   └── notification.ts     # Отправка уведомлений
│
└── utils/                   # Утилиты
    ├── logger.ts           # Winston logger
    └── formatters.ts       # Форматирование данных
```

Технологии: - **Grammy 1.38.4** - современная библиотека для Telegram Bot API - **@grammyjs/conversations** - управление диалогами - **Axios** - HTTP клиент - **Winston** - структурированное логирование

---

2. Backend API (NestJS)

Местоположение: `nodejs_space/`

Ответственность: - Бизнес-логика приложения - Управление данными - Аутентификация и авторизация - REST API для бота и админ-панели - Интеграции с внешними сервисами

Архитектура модулей:

```
nodejs_space/src/
├── auth/                      # JWT аутентификация
│   ├── auth.service.ts      # Логика аутентификации
│   ├── guards/              # JWT Guard, Roles Guard
│   └── strategies/          # JWT Strategy
│
├── products/                # Управление товарами
│   ├── products.service.ts  # Бизнес-логика товаров
│   ├── products.controller.ts  # Public endpoints
│   └── dto/                 # Data Transfer Objects
│
├── orders/                  # Управление заказами
│   ├── orders.service.ts    # Бизнес-логика заказов
│   ├── orders.controller.ts # Endpoints для заказов
│   └── dto/
│
├── users/                   # Управление пользователями
│   ├── users.service.ts     # Логика пользователей
│   ├── users.controller.ts  # Admin endpoints
│   ├── users-public.controller.ts  # Bot endpoints
│   └── dto/
│
├── import-export/           # Импорт/экспорт данных
│   ├── import-export.service.ts
│   └── dto/
│
├── integrations/            # Внешние интеграции
│   └── integrations.service.ts  # GA4, Yandex Metrika
│
├── webhook/                 # Telegram webhook
│   ├── webhook.controller.ts
│   └── webhook.service.ts
│
├── prisma/                  # Prisma сервис
│   ├── prisma.service.ts    # Database connection
│   └── prisma.module.ts
│
└── common/                  # Общие компоненты
    ├── decorators/          # @CurrentUser, @Roles
    ├── filters/             # Exception filters
    └── interceptors/        # Audit log interceptor
```

Слои архитектуры:

```
Controller Layer (HTTP)
    ↓
Service Layer (Business Logic)
```

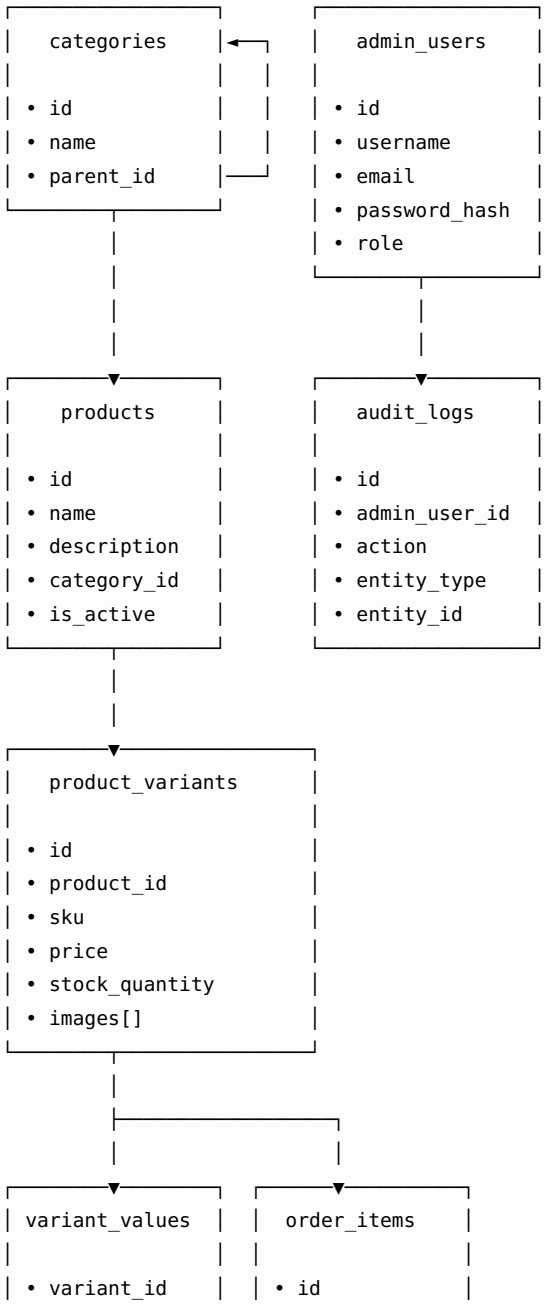
```
    ↓
Repository Layer (Prisma ORM)
    ↓
Database (PostgreSQL)
```

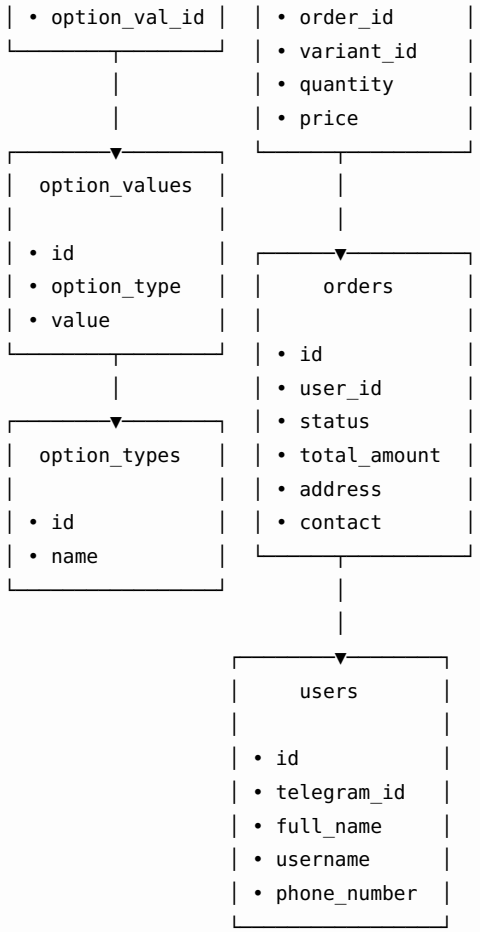
Технологии: - **NestJS 11.0** - Enterprise-ready framework - **Prisma 6.0** - Type-safe ORM - **Passport.js + JWT** - Authentication - **class-validator** - DTO validation - **Swagger/OpenAPI** - API documentation

---

3. База данных (PostgreSQL + Prisma)

Схема базы данных:



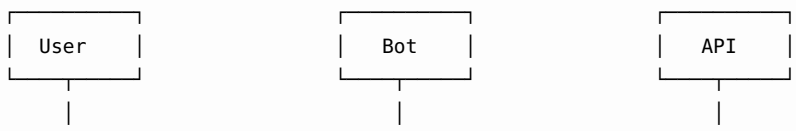


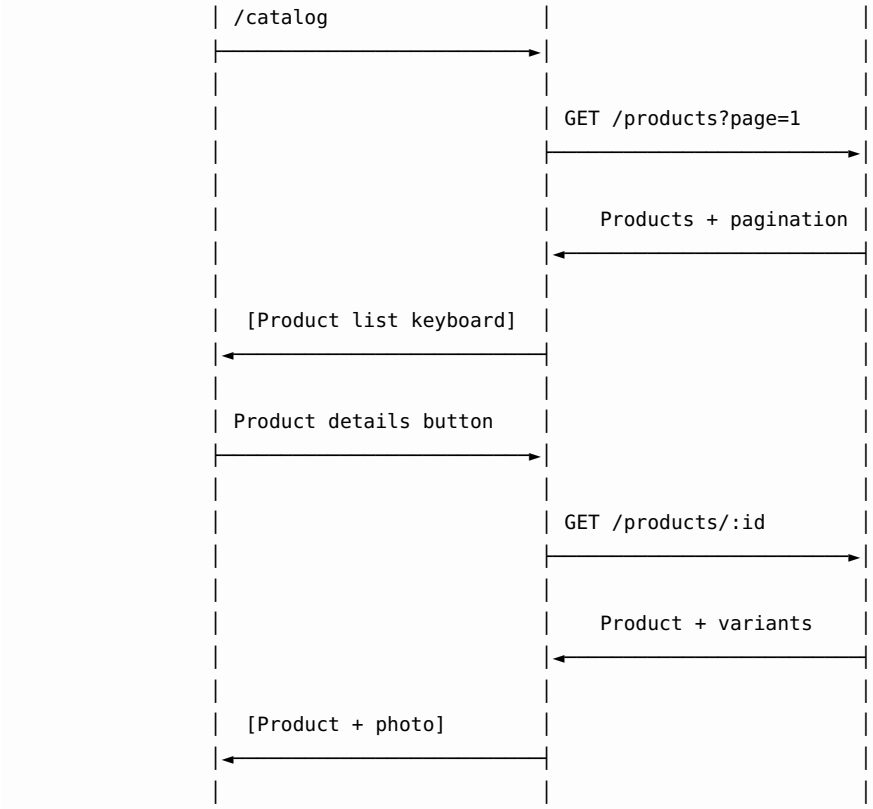
Ключевые таблицы:

1. **users** - клиенты из Telegram
2. **admin_users** - администраторы системы
3. **categories** - иерархия категорий (самосвязь)
4. **products** - товары
5. **product_variants** - варианты товаров (SKU)
6. **option_types** - типы опций (Color, Size)
7. **option_values** - значения опций (Red, Blue, M, L)
8. **variant_values** - связь вариантов и опций
9. **orders** - заказы
10. **order_items** - позиции заказа
11. **audit_logs** - аудит действий админов

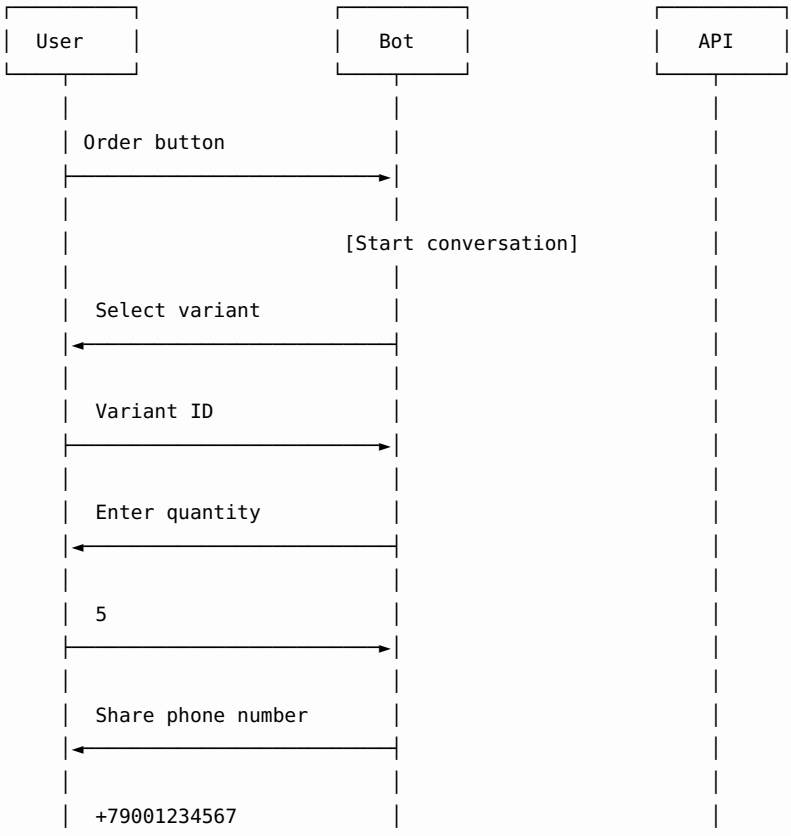
🔄 Схема взаимодействия

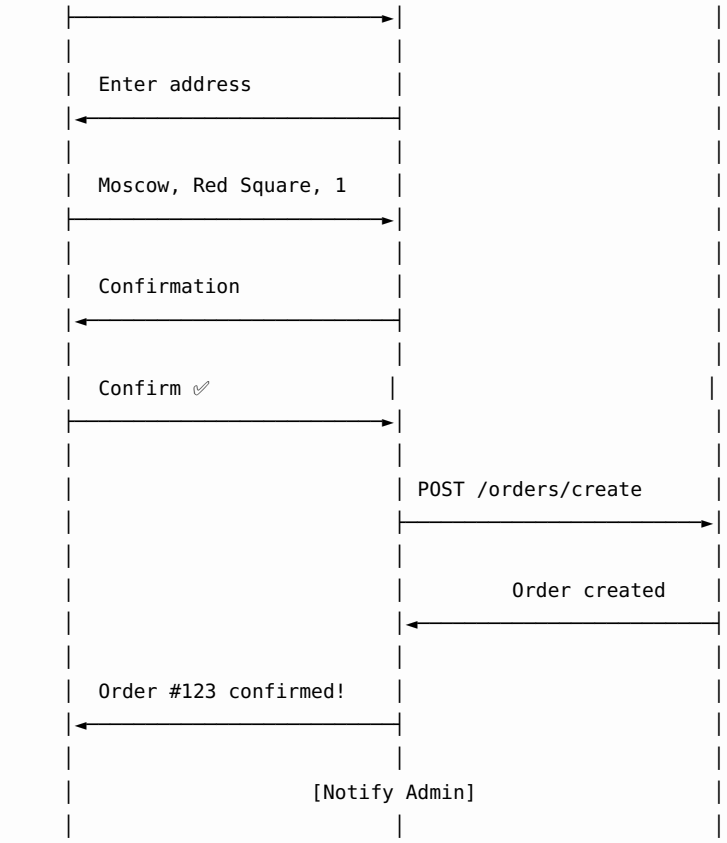
Сценарий 1: Просмотр каталога



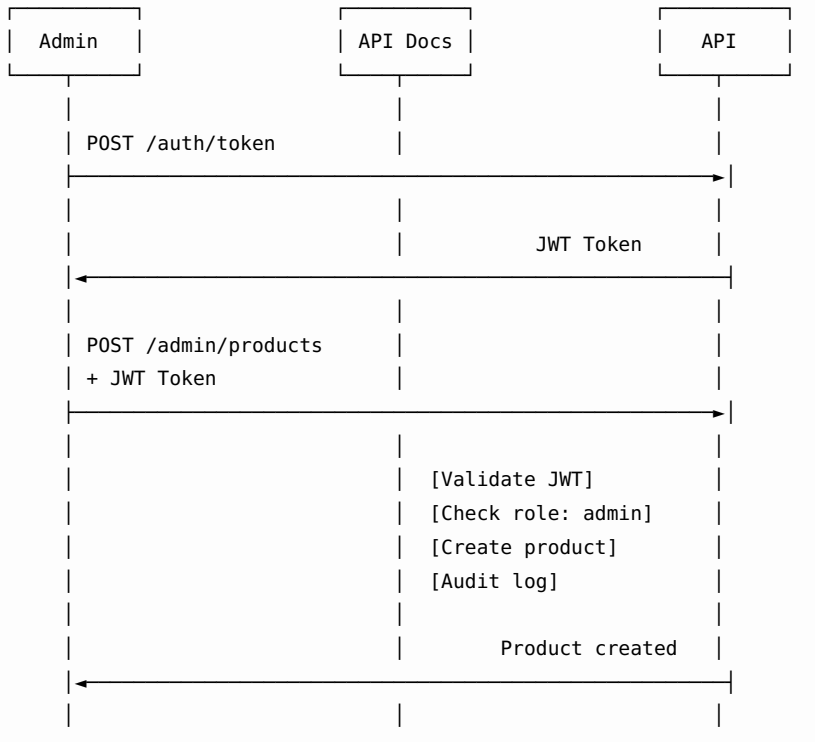


Сценарий 2: Оформление заказа



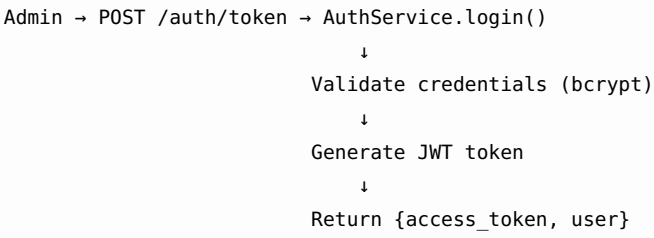


Сценарий 3: Админ управление товарами

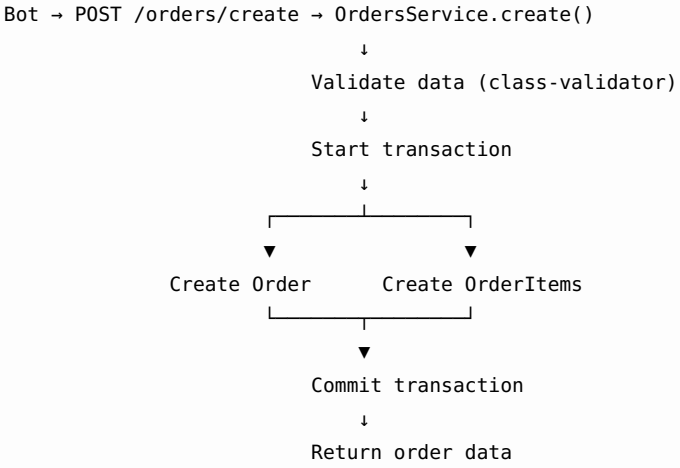


📊 Потоки данных

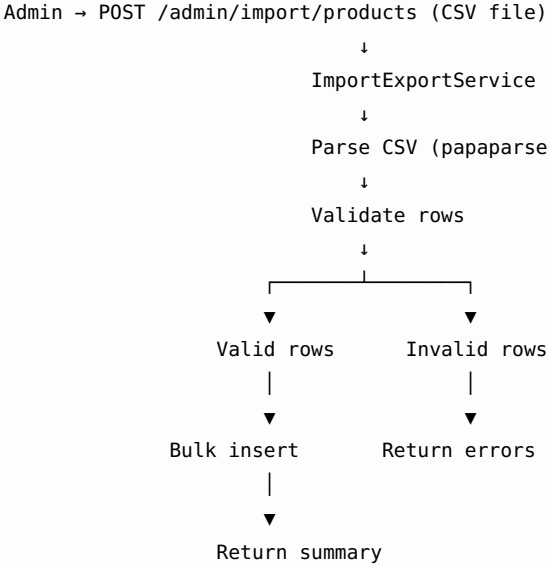
1. Поток аутентификации



2. Поток создания заказа



3. Поток импорта данных



🛡 Технологические решения

1. Аутентификация и авторизация

JWT Token Flow:

```
1. Admin → POST /auth/token {username, password}
2. API validates credentials (bcrypt)
3. API generates JWT (HS256, 30m expiration)
4. Admin stores token
5. Admin → Protected endpoint + Authorization: Bearer <token>
6. API validates JWT signature & expiration
7. API checks user role (admin/manager)
8. API executes request
```

Guards: - `JwtAuthGuard` - проверка наличия и валидности JWT - `RolesGuard` - проверка роли пользователя

2. Валидация данных

Входные данные:

```typescript
// DTO с валидацией
export class CreateProductDto {
  @IsString()
  @MinLength(3)
  @MaxLength(255)
  name: string;

  @IsString()
  @MinLength(10)
  description: string;

  @IsInt()
  @Min(1)
  category_id: number;

  @IsBoolean()
  @IsOptional()
  is_active?: boolean;
}
```

Автоматическая валидация через `ValidationPipe` на уровне контроллеров.

3. Обработка ошибок

Структура ошибки:

```
{
  "statusCode": 400,
  "message": "Validation failed",
  "errors": [
    {
      "field": "name",
      "constraints": {
```

```
        "minLength": "name must be longer than or equal to 3
        characters"
      }
    }
  ],
  "timestamp": "2025-12-04T10:30:00.000Z",
  "path": "/api/products"
}
```


Exception Filter обрабатывает все исключения и возвращает унифицированный формат.

4. Логирование

Backend API (NestJS Logger):

```
[ProductsService] Fetching products list - {"page":1,"limit":10}
[OrdersService] Creating order - {"user_id":"12345","total":1500.00}
[AuthService] Login attempt - {"username":"admin"}
```

Bot (Winston):


```
{
  "level": "info",
  "message": "Start command received",
  "userId": 325922641,
  "timestamp": "2025-12-04T10:30:00.000Z"
}
```

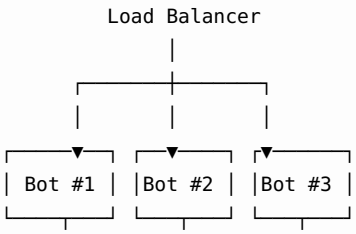

5. Аудит действий

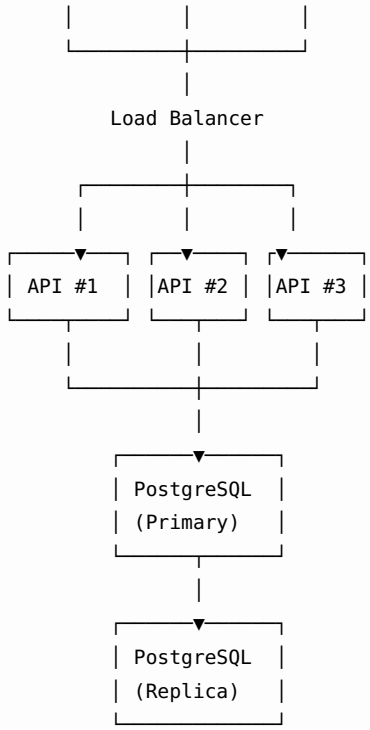
Автоматический аудит:

```
// Interceptor записывает все изменения админов
@UseInterceptors(AuditLogInterceptor)
@Put('admin/products/:id')
async updateProduct(@Param('id') id: number, @Body() data:
        UpdateProductDto) {
  // После успешного обновления создается запись в audit_logs
  return this.productsService.update(id, data);
}
```

📈 Масштабирование

Горизонтальное масштабирование





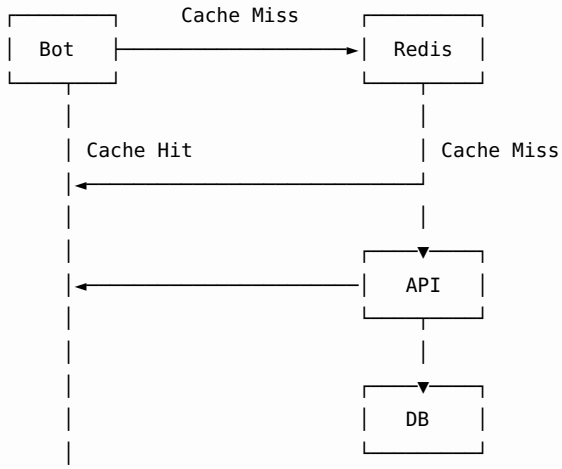
Вертикальное масштабирование

Текущие ресурсы (MVP): - Bot: 512MB RAM, 0.5 CPU - API: 1GB RAM, 1 CPU - PostgreSQL: 2GB RAM, 1 CPU

Для 1000+ заказов/день: - Bot: 1GB RAM, 1 CPU - API: 2GB RAM, 2 CPU - PostgreSQL: 4GB RAM, 2 CPU

Кэширование

Потенциальные точки кэширования: 1. Каталог товаров (Redis) 2. Категории (In-memory) 3. Настройки системы (In-memory)



🔐 Безопасность

1. Защита от атак

- **SQL Injection** → Prisma ORM (parameterized queries)
- **XSS** → Sanitization через class-transformer
- **CSRF** → JWT tokens (stateless)
- **Rate Limiting** → @nestjs/throttler (100 req/min)
- **CORS** → Настраиваемые origins

2. Секреты

Все секреты хранятся в environment variables: - `JWT_SECRET` - секрет для подписи JWT - `BOT_TOKEN` - Telegram Bot token - `DATABASE_URL` - строка подключения к БД

Никогда не коммитить секреты в Git!

3. HTTPS

В production использовать: - SSL/TLS сертификаты (Let’s Encrypt) - HTTPS для API - WSS для Telegram webhook (если используется)

🖨 Дополнительные ресурсы

- SETUP.md - установка и настройка
- API_DOCUMENTATION.md - API endpoints
- BOT_LOGIC.md - логика бота
- DATABASE.md - схема БД
- DEVELOPMENT.md - руководство разработчика

Обновлено: 2025-12-04
Версия: 1.0.0

SETUP.md

🚀 Установка и настройка Telegram Shop MVP

🫙 Оглавление

💻 Системные требования

Минимальные требования

- **CPU:** 2 ядра
- **RAM:** 2GB
- **Диск:** 10GB свободного места
- **ОС:** Ubuntu 20.04+ / Debian 11+ / macOS / Windows 10+

Программное обеспечение

Для Docker метода:

- Docker 20.10+
- Docker Compose 2.0+

Для локальной разработки:

- Node.js 18+ (LTS)
- Yarn 4+
- PostgreSQL 15+
- Git

Получение Telegram Bot Token

1. Откройте Telegram и найдите @BotFather
2. Отправьте `/newbot`
3. Следуйте инструкциям (имя бота, username)
4. Получите токен вида: `1234567890:ABCdefGhIjKlmNoPqRsTuVwXyZ`
5. Сохраните токен для дальнейшей настройки

🐳 Метод 1: Docker Compose (рекомендуется)

Самый простой и быстрый способ запуска.

Шаг 1: Распаковка проекта

```
# Распаковать архив
tar -xzf telegram_shop_backend.tar.gz

# Перейти в директорию
cd telegram_shop_backend
```

Шаг 2: Настройка environment variables

```bash
# Создать .env из примера
cp .env.example .env

# Редактировать .env
nano .env
```

Содержимое .env:

```
# Backend API
DATABASE_URL=postgresql://postgres:postgres@postgres:5432/telegram_sl
schema=public
PORT=3000
JWT_SECRET=your-super-secret-jwt-key-change-in-production-minimum-
32-characters
JWT_EXPIRES_IN=30m
NODE_ENV=production

# Telegram Bot
BOT_TOKEN=ваш_telegram_bot_token_здесь
ADMIN_TELEGRAM_IDS=123456789,987654321

# Optional: CORS origins (comma-separated)
CORS_ORIGIN=*

# Optional: Analytics
# GA4_MEASUREMENT_ID=G-XXXXXXXXXX
# YANDEX_METRIKA_ID=12345678
```

Важно: - Замените `BOT_TOKEN` на реальный токен от @BotFather - Укажите ваш Telegram ID в `ADMIN_TELEGRAM_IDS` (узнать: @userinfobot) - Обязательно смените `JWT_SECRET` на случайную строку минимум 32 символа

Шаг 3: Запуск

```bash
# Запустить все сервисы
docker-compose up -d

# Проверить статус
docker-compose ps

# Просмотр логов
docker-compose logs -f
```

Что запустится: - PostgreSQL база данных (порт 5432) - Backend API (порт 3000) - Telegram Bot

Шаг 4: Инициализация базы данных

База данных автоматически создастся и заполнится тестовыми данными при первом запуске API.

Тестовые данные включают: - Админ: `username: admin`, `password: admin123` - 10 товаров с вариантами - 3 категории

🛠 Метод 2: Локальная разработка

Для разработчиков, которым нужен прямой доступ к коду.

Шаг 1: Установка зависимостей

macOS (с Homebrew):

```
# Node.js
brew install node@18

# Yarn
npm install -g yarn

# PostgreSQL
brew install postgresql@15
brew services start postgresql@15
```

Ubuntu/Debian:

```
# Node.js 18 LTS
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt-get install -y nodejs

# Yarn
npm install -g yarn

# PostgreSQL 15
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt
        $(lsb_release -cs)-pgdg main" >
        /etc/apt/sources.list.d/pgdg.list'
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc
        | sudo apt-key add -
sudo apt-get update
sudo apt-get install -y postgresql-15
```

Windows:

1. Установите Node.js 18 LTS
2. Установите PostgreSQL 15
3. Откройте PowerShell и выполните: `npm install -g yarn`

Шаг 2: Настройка PostgreSQL

```
# Подключиться к PostgreSQL
sudo -u postgres psql

# Создать базу данных
CREATE DATABASE telegram_shop;
```

```
CREATE USER shopuser WITH PASSWORD 'shoppassword';
GRANT ALL PRIVILEGES ON DATABASE telegram_shop TO shopuser;

# Выход
\q
```

Шаг 3: Настройка Backend API

```
cd telegram_shop_backend/nodejs_space

# Установить зависимости
yarn install

# Создать .env
cat > .env << 'EOF'
DATABASE_URL=postgresql://shopuser:shoppassword@localhost:5432/telegram_shop?
        schema=public

PORT=3000
JWT_SECRET=your-super-secret-jwt-key-change-in-production
JWT_EXPIRES_IN=30m
NODE_ENV=development
CORS_ORIGIN=*
EOF

# Применить схему БД
yarn prisma db push

# Заполнить тестовыми данными
yarn prisma db seed

# Скомпилировать
yarn build

# Запустить в dev режиме
yarn start:dev

# Или в production режиме
yarn start:prod
```

API будет доступен на http://localhost:3000

Шаг 4: Настройка Telegram Bot

Откройте новый терминал:

```
cd telegram_shop_backend/bot

# Установить зависимости
yarn install

# Создать .env
cat > .env << 'EOF'
```

```
BOT_TOKEN=ваш_telegram_bot_token_здесь
API_BASE_URL=http://localhost:3000/api
API_TIMEOUT=10000
ADMIN_TELEGRAM_IDS=123456789
NODE_ENV=development
LOG_LEVEL=debug
EOF

# Скомпилировать
yarn build

# Запустить в dev режиме
yarn dev

# Или в production режиме
yarn start
```

Шаг 5: Управление процессами с PM2 (опционально)

PM2 позволяет запускать процессы в фоне с автозапуском.

```
# Установить PM2
npm install -g pm2

# Backend API
cd telegram_shop_backend/nodejs_space
pm2 start dist/main.js --name telegram-shop-api

# Telegram Bot
cd ../bot
pm2 start dist/index.js --name telegram-shop-bot

# Просмотр логов
pm2 logs

# Автозапуск при перезагрузке
pm2 startup
pm2 save

# Управление
pm2 restart telegram-shop-api
pm2 stop telegram-shop-bot
pm2 delete telegram-shop-api
```

🌐 Метод 3: Развертывание на Ubuntu сервере

Полноценное production развертывание на VPS/dedicated сервере.

Предварительные требования

- Ubuntu 20.04+ сервер
- Доступ по SSH
- Доменное имя (опционально, для HTTPS)
- Минимум 2GB RAM

Шаг 1: Подготовка сервера

```
# Обновление системы
sudo apt update && sudo apt upgrade -y

# Установка Docker
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

# Добавление пользователя в группу docker
sudo usermod -aG docker $USER

# Перелогиниться для применения изменений
exit
# SSH снова

# Установка Docker Compose
sudo curl -L
        "https://github.com/docker/compose/releases/latest/download/docker-
        compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

# Проверка установки
docker --version
docker-compose --version
```

Шаг 2: Загрузка проекта

```
# Создать директорию
mkdir -p ~/apps
cd ~/apps

# Загрузить проект (через SCP или Git)
# Вариант 1: SCP с локального компьютера
# scp telegram_shop_backend.tar.gz user@your-server-ip:~/apps/

# Вариант 2: Wget если файл на сервере
# wget https://your-domain.com/telegram_shop_backend.tar.gz

# Распаковать
tar -xzf telegram_shop_backend.tar.gz
cd telegram_shop_backend
```

Шаг 3: Настройка .env

```
nano .env
```

Production .env:

```
DATABASE_URL=postgresql://postgres:YourStrongPassword123@postgres:543
schema=public
PORT=3000
JWT_SECRET=your-production-secret-min-32-chars-random-string-here
JWT_EXPIRES_IN=30m
NODE_ENV=production
CORS_ORIGIN=https://yourdomain.com

BOT_TOKEN=ваш_реальный_telegram_bot_token
ADMIN_TELEGRAM_IDS=ваш_telegram_id

# Analytics (опционально)
GA4_MEASUREMENT_ID=G-XXXXXXXXXX
YANDEX_METRIKA_ID=12345678
```

Шаг 4: Запуск с Docker Compose

```
# Запуск
docker-compose up -d

# Проверка
docker-compose ps
docker-compose logs -f api
docker-compose logs -f bot

# Проверка API
curl http://localhost:3000/api/products
```

Шаг 5: Настройка Nginx (опционально)

Для публичного доступа по доменному имени.

```
# Установка Nginx
sudo apt install nginx -y

# Создать конфиг
sudo nano /etc/nginx/sites-available/telegram-shop
```

Содержимое конфига:

```
server {
    listen 80;
    server_name your-domain.com www.your-domain.com;

    # Redirect to HTTPS (после установки SSL)
    # return 301 https://$server_name$request_uri;

    location /api {
        proxy_pass http://localhost:3000;
```

```
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_cache_bypass $http_upgrade;
    }

    location /api-docs {
        proxy_pass http://localhost:3000/api-docs;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
    }
}

# Активировать конфиг
sudo ln -s /etc/nginx/sites-available/telegram-shop
        /etc/nginx/sites-enabled/

# Проверить конфигурацию
sudo nginx -t

# Перезапустить Nginx
sudo systemctl restart nginx

# Включить автозапуск
sudo systemctl enable nginx
```

Шаг 6: Установка SSL с Let’s Encrypt

```
# Установить Certbot
sudo apt install certbot python3-certbot-nginx -y

# Получить сертификат
sudo certbot --nginx -d your-domain.com -d www.your-domain.com

# Автоматическое обновление сертификата
sudo systemctl status certbot.timer
```

После установки SSL раскомментируйте строку redirect в Nginx конфиге.

Шаг 7: Мониторинг и логи

```
# Логи Docker
docker-compose logs -f

# Логи Nginx
sudo tail -f /var/log/nginx/access.log
```

```
sudo tail -f /var/log/nginx/error.log

# Статус сервисов
docker-compose ps
sudo systemctl status nginx
```

🎬 Первый запуск

После успешной установки любым методом:

1. Проверка API

```
# Health check
curl http://localhost:3000/api

# Список товаров
curl http://localhost:3000/api/products

# Категории
curl http://localhost:3000/api/categories

# Swagger документация
# Откройте в браузере: http://localhost:3000/api-docs
```

2. Получение JWT токена для админки

```
curl -X POST http://localhost:3000/api/auth/token \
  -H "Content-Type: application/json" \
  -d '{
    "username": "admin",
    "password": "admin123"
  }'
```

Ответ:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": 1,
    "username": "admin",
    "email": "admin@example.com",
    "role": "admin"
  }
}
```

3. Проверка Telegram бота

1. Откройте Telegram
2. Найдите вашего бота (по username)
3. Отправьте `/start`
4. Должно появиться приветственное сообщение с меню

Если бот не отвечает:

```
# Проверьте логи бота
docker-compose logs bot
# или
pm2 logs telegram-shop-bot
```

✅ Проверка работы

Checklist

- ☐ API доступен на порту 3000
- ☐ Swagger документация открывается
- ☐ PostgreSQL запущен и доступен
- ☐ Telegram бот отвечает на `/start`
- ☐ Можно получить JWT токен для админа
- ☐ Endpoint `/api/products` возвращает товары
- ☐ Можно просмотреть каталог в боте
- ☐ Можно оформить заказ через бота

Тестовые запросы

```
# 1. Получить токен
TOKEN=$(curl -s -X POST http://localhost:3000/api/auth/token \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"admin123"}' \
  | jq -r '.access_token')

# 2. Получить все товары (admin)
curl -H "Authorization: Bearer $TOKEN" \
  http://localhost:3000/api/admin/products

# 3. Создать новый товар
curl -X POST http://localhost:3000/api/admin/products \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "name": "Test Product",
    "description": "This is a test product",
    "category_id": 1,
    "is_active": true
  }'

# 4. Получить заказы
curl -H "Authorization: Bearer $TOKEN" \
  http://localhost:3000/api/admin/orders
```

🔴 Остановка и управление

Docker Compose

```
# Остановить все сервисы
docker-compose stop

# Остановить и удалить контейнеры
docker-compose down

# Остановить и удалить контейнеры + volumes (БД будет удалена!)
docker-compose down -v

# Перезапустить сервисы
docker-compose restart

# Перезапустить конкретный сервис
docker-compose restart api
docker-compose restart bot

# Просмотр логов
docker-compose logs -f
docker-compose logs -f api
docker-compose logs -f bot

# Статус сервисов
docker-compose ps

# Обновление после изменений
docker-compose up -d --build
```

PM2

```
# Остановить
pm2 stop telegram-shop-api
pm2 stop telegram-shop-bot

# Перезапустить
pm2 restart telegram-shop-api
pm2 restart telegram-shop-bot

# Удалить
pm2 delete telegram-shop-api
pm2 delete telegram-shop-bot

# Просмотр всех процессов
pm2 list

# Мониторинг в реальном времени
pm2 monit

# Логи
pm2 logs telegram-shop-api
pm2 logs telegram-shop-bot
```

🔧 Обновление проекта

```
# Остановить сервисы
docker-compose down

# Удалить старые образы
docker-compose rm -f

# Пересобрать и запустить
docker-compose up -d --build

# Проверить логи
docker-compose logs -f
```

📚 Дополнительные ресурсы

- ARCHITECTURE.md - архитектура проекта
- API_DOCUMENTATION.md - описание API endpoints
- BOT_LOGIC.md - логика работы бота
- TROUBLESHOOTING.md - решение проблем
- DEVELOPMENT.md - руководство разработчика

Обновлено: 2025-12-04
Версия: 1.0.0

API_DOCUMENTATION.md

🔌 API Documentation

📑 Table of Contents

🌐 Base URL

Development: `http://localhost:3000/api`
Production: `https://smokyyard.abacusai.app/api`

Swagger Documentation: `http://localhost:3000/api-docs`

🔐 Authentication

JWT Token

Protected endpoints require a JWT token in the Authorization header:

```
Authorization: Bearer <access_token>
```

Getting a Token

```bash
curl -X POST http://localhost:3000/api/auth/token \
  -H "Content-Type: application/json" \
  -d '{"username": "admin", "password": "admin123"}'
```

Response:

```json
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": 1,
    "username": "admin",
    "email": "admin@example.com",
    "role": "admin"
  }
}
```

📊 Common Response Codes

Code	Status	Description
200	OK	Request successful
201	Created	Resource created
204	No Content	Delete successful
400	Bad Request	Invalid input
401	Unauthorized	Authentication required
403	Forbidden	Insufficient permissions
404	Not Found	Resource not found
422	Unprocessable Entity	Validation error
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Server error

🌍 Public Endpoints

Products

GET /products

Get list of active products with pagination.

Query Parameters: - `page` (number, optional): Page number (default: 1) - `limit` (number, optional): Items per page (default: 10, max: 100) - `category` (number, optional): Filter by category ID - `search` (string, optional): Search in name/description

Example:

```
curl "http://localhost:3000/api/products?page=1&limit=10"
```

Response:

```
{
  "data": [
    {
      "id": 1,
      "name": "T-Shirt Blue",
      "description": "Comfortable cotton t-shirt",
      "category": {
        "id": 1,
        "name": "Clothing"
      },
      "variants": [
        {
          "id": 1,
          "sku": "TSHIRT-BLUE-M",
          "price": "1500.00",
          "stock_quantity": 50,
          "images": ["https://example.com/image1.jpg"]
        }
      ],
      "is_active": true
    }
  ],
  "meta": {
    "total": 100,
    "page": 1,
    "limit": 10,
    "totalPages": 10
  }
}
```

GET /products/:id

Get product details by ID.

Parameters: - `id` (number): Product ID

Example:

```
curl "http://localhost:3000/api/products/1"
```

Response:


```
{
  "id": 1,
  "name": "T-Shirt Blue",
  "description": "Comfortable cotton t-shirt",
  "category": {
    "id": 1,
    "name": "Clothing",
    "parent_id": null
  },
  "variants": [
    {
      "id": 1,
      "sku": "TSHIRT-BLUE-M",
      "price": "1500.00",
      "stock_quantity": 50,
      "weight": "0.3",
      "dimensions": "40x30x2 cm",
      "images": ["https://example.com/image1.jpg"],
      "options": [
        {"type": "Color", "value": "Blue"},
        {"type": "Size", "value": "M"}
      ]
    }
  ],
  "is_active": true,
  "created_at": "2025-12-01T10:00:00.000Z"
}
```

Categories

GET /categories

Get all categories with hierarchy.

Example:

```
curl "http://localhost:3000/api/categories"
```

Response:


```
[
  {
    "id": 1,
    "name": "Clothing",
    "parent_id": null,
    "children": [
      {
        "id": 2,
        "name": "T-Shirts",
        "parent_id": 1
      }
```

```
    ]
  }
]
```

Orders

POST /orders

Create a new order (for bot integration).

Request Body:

```
{
  "user_id": 123456789,
  "items": [
    {
      "variant_id": 1,
      "quantity": 2
    }
  ],
  "shipping_address": "Moscow, Red Square, 1",
  "customer_contact": "Ivan Petrov, +79001234567",
  "notes": "Please deliver after 6 PM"
}
```

Example:

```
curl -X POST http://localhost:3000/api/orders \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": 123456789,
    "items": [{"variant_id": 1, "quantity": 2}],
    "shipping_address": "Moscow, Red Square, 1",
    "customer_contact": "Ivan Petrov, +79001234567"
  }'
```

Response:

```
{
  "id": 42,
  "user_id": 123456789,
  "status": "pending",
  "total_amount": "3000.00",
  "shipping_address": "Moscow, Red Square, 1",
  "customer_contact": "Ivan Petrov, +79001234567",
  "notes": "Please deliver after 6 PM",
  "created_at": "2025-12-04T15:30:00.000Z",
  "items": [
    {
      "id": 1,
      "variant": {
        "id": 1,
        "sku": "TSHIRT-BLUE-M",
```

```
        "product_name": "T-Shirt Blue"
      },
      "quantity": 2,
      "price_at_purchase": "1500.00"
    }
  ]
}
```

Users (Public - Bot Integration)

GET /users/telegram/:telegramId

Get user by Telegram ID.

Parameters: - `telegramId` (number): Telegram User ID

Example:

```
curl "http://localhost:3000/api/users/telegram/123456789"
```

Response:

```
{
  "id": 1,
  "telegram_id": 123456789,
  "full_name": "Ivan Petrov",
  "username": "ivanpetrov",
  "phone_number": "+79001234567",
  "created_at": "2025-12-01T10:00:00.000Z"
}
```

POST /users/upsert

Create or update user from Telegram data.

Request Body:

```
{
  "telegram_id": 123456789,
  "username": "ivanpetrov",
  "full_name": "Ivan Petrov",
  "phone_number": "+79001234567"
}
```

Example:

```
curl -X POST http://localhost:3000/api/users/upsert \
  -H "Content-Type: application/json" \
  -d '{
    "telegram_id": 123456789,
    "username": "ivanpetrov",
    "full_name": "Ivan Petrov",
    "phone_number": "+79001234567"
  }'
```

🔑 Authentication Endpoints

POST /auth/token

Admin login and token generation.

Request Body:

```
{
  "username": "admin",
  "password": "admin123"
}
```

Response:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "user": {
    "id": 1,
    "username": "admin",
    "email": "admin@example.com",
    "role": "admin",
    "is_active": true
  }
}
```

👑 Admin Endpoints

All admin endpoints require JWT authentication and appropriate role.

Products Management

GET /admin/products

Get all products (including inactive).

Headers: - `Authorization: Bearer <token>`

Query Parameters: - `page`, `limit`, `category`, `search` (same as public endpoint) - `is_active` (boolean, optional): Filter by active status

Example:

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
  "http://localhost:3000/api/admin/products?is_active=false"
```

POST /admin/products

Create a new product.

Headers: - `Authorization: Bearer <token>`

Request Body:

```
{
  "name": "New T-Shirt",
  "description": "Amazing new t-shirt",
  "category_id": 1,
  "is_active": true
}
```

Response: Product detail object

PUT /admin/products/:id

Update existing product.

Headers: - `Authorization: Bearer <token>`

Request Body:

```
{
  "name": "Updated T-Shirt",
  "description": "Updated description",
  "is_active": false
}
```

DELETE /admin/products/:id

Delete product (cascade deletes variants).

Headers: - `Authorization: Bearer <token>`

Response: `204 No Content`

Orders Management

GET /admin/orders

Get all orders with filters.

Headers: - `Authorization: Bearer <token>`

Query Parameters: - `page`, `limit` - `status` (string): pending, processing, shipped, delivered, cancelled - `date_from` (ISO date): Start date filter - `date_to` (ISO date): End date filter - `user_id` (number): Filter by user

Example:

```
curl -H "Authorization: Bearer YOUR_TOKEN" \
  "http://localhost:3000/api/admin/orders?status=pending&page=1"
```

Response:

```
{
```

```
  "data": [
    {
      "id": 42,
      "user": {
        "id": 1,
        "telegram_id": 123456789,
        "full_name": "Ivan Petrov"
      },
      "status": "pending",
      "total_amount": "3000.00",
      "created_at": "2025-12-04T15:30:00.000Z",
      "items_count": 2
    }
  ],
  "meta": {
    "total": 50,
    "page": 1,
    "limit": 10
  }
}
```

GET /admin/orders/:id

Get detailed order information.

Headers: - `Authorization: Bearer <token>`

Response:


```
{
  "id": 42,
  "user": {
    "id": 1,
    "telegram_id": 123456789,
    "full_name": "Ivan Petrov",
    "username": "ivanpetrov",
    "phone_number": "+79001234567"
  },
  "status": "pending",
  "total_amount": "3000.00",
  "shipping_address": "Moscow, Red Square, 1",
  "customer_contact": "Ivan Petrov, +79001234567",
  "notes": "Please deliver after 6 PM",
  "created_at": "2025-12-04T15:30:00.000Z",
  "updated_at": "2025-12-04T15:30:00.000Z",
  "items": [
    {
      "id": 1,
      "variant": {
        "id": 1,
        "sku": "TSHIRT-BLUE-M",
        "product": {
          "id": 1,
```

```
          "name": "T-Shirt Blue"
        }
      },
      "quantity": 2,
      "price_at_purchase": "1500.00"
    }
  ]
}
```

PATCH /admin/orders/:id

Update order status.

Headers: - `Authorization: Bearer <token>`

Request Body:

```
{
  "status": "processing"
}
```

Valid statuses: `pending`, `processing`, `shipped`, `delivered`, `cancelled`

Users Management

GET /admin/users

Get all users.

Headers: - `Authorization: Bearer <token>`

Query Parameters: - `page`, `limit` - `search` (string): Search in name/username

---

Import/Export

POST /admin/import/csv

Import products from CSV file.

Headers: - `Authorization: Bearer <token>` - `Content-Type: multipart/form-data`

Form Data: - `file`: CSV file

CSV Format:

```
name,description,category_id,sku,price,stock_quantity
"T-Shirt Red","Comfortable red t-shirt",1,"TSHIRT-RED-M",1500,100
```

Example:

```
curl -X POST http://localhost:3000/api/admin/import/csv \
  -H "Authorization: Bearer YOUR_TOKEN" \
```

```
  -F "file=@products.csv"
```

Response:

```
{
  "success": 10,
  "failed": 2,
  "errors": [
    {
      "row": 3,
      "error": "Invalid category_id"
    }
  ]
}
```

POST /admin/import/xlsx

Import products from XLSX file (same format as CSV).

POST /admin/import/google-sheets

Import products from Google Sheets.

Request Body:

```
{
  "spreadsheet_id": "1BxiMVs0XRA5nFMdKvBdBZjgmUUqptlbs74OgvE2upms",
  "range": "Sheet1!A1:F100"
}
```

GET /admin/export/csv

Export orders to CSV.

Headers: - `Authorization: Bearer <token>`

Query Parameters: - `date_from`, `date_to` - `status`

Returns: CSV file download

GET /admin/export/xlsx

Export orders to XLSX (same as CSV).

POST /admin/export/google-sheets

Export orders to Google Sheets.

Request Body:

```
{
  "spreadsheet_id": "...",
```

```
  "sheet_name": "Orders",
  "date_from": "2025-12-01",
  "date_to": "2025-12-31"
}
```

⚠ Error Handling

All errors follow a consistent format:

```
{
  "statusCode": 400,
  "message": "Validation failed",
  "errors": [
    {
      "field": "name",
      "constraints": {
        "minLength": "name must be longer than or equal to 3
        characters"
      }
    }
  ],
  "timestamp": "2025-12-04T15:30:00.000Z",
  "path": "/api/products"
}
```

🚦 Rate Limiting

- **Default:** 100 requests per minute per IP
- **Admin endpoints:** 500 requests per minute

Rate limit headers:

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 95
X-RateLimit-Reset: 1733329800
```

🖨 Additional Resources

- **Swagger UI:** http://localhost:3000/api-docs
- **Production API:** https://smokyyard.abacusai.app/api-docs
- ARCHITECTURE.md
- SETUP.md

Updated: 2025-12-04
Version: 1.0.0

BOT_LOGIC.md

🤖 Telegram Bot Logic

📑 Table of Contents

🎯 Overview

The Telegram bot is built with **Grammy framework** and uses **conversations** plugin for dialog management.

Bot Username: @SmokyYardBot (example)

Key Features

- ✅ Command handling
- ✅ Inline keyboards
- ✅ Conversation flows
- ✅ Session state management
- ✅ Auto-user creation
- ✅ Error handling
- ✅ Logging

🎛 Bot Commands

/start

Description: Main entry point, shows welcome message and main menu

Response:

```
👋 Привет, Ivan!

Добро пожаловать в наш магазин! 🛍

Здесь вы можете:
• Просмотреть каталог товаров
• Оформить заказ
• Просмотреть историю заказов

Выберите действие:
```

```
[🛍 Каталог] [📋 Мои заказы]
[👤 Профиль] [ℹ Помощь]
```

Implementation: `bot/src/handlers/start.ts`

/catalog

Description: Browse product catalog

Flow: 1. Shows list of products (5 per page) 2. Pagination with ⬅ Prev / Next ➡ buttons 3. Click product → product details 4. [Заказать] button → start order conversation

Implementation: `bot/src/handlers/catalog.ts`

/orders

Description: View user's order history

Response:

```
🛍 Ваши заказы:

#42 - 3000.00 ₽ - Pending
Создан: 04.12.2025 15:30

#41 - 1500.00 ₽ - Delivered
Создан: 03.12.2025 10:15

[◄ Previous] [Next ►]
```

/profile

Description: Show user profile information

Response:

```
👤 Ваш профиль:

ФИО: Ivan Petrov
Username: @ivanpetrov
Телефон: +79001234567
ID: 123456789

Заказов выполнено: 15
```

🔄 Conversation Flows

Order Conversation

Complete order flow with 8 steps.

Implementation: `bot/src/conversations/order.ts`

Step 1: Product Selection

```
User clicks [Заказать] button on product
→ Bot starts conversation
→ Shows product with variants
```

Step 2: Variant Selection

```
Bot: "Выберите вариант:"
[Blue - M] [Blue - L]
[Red - M] [Red - L]

User clicks variant
→ Stored in session.orderData.variant_id
```

Step 3: Quantity Input

```
Bot: "Введите количество (доступно: 50 шт):"

User: "5"
→ Validates: number, > 0, <= stock
→ Stored in session.orderData.quantity
```

Step 4: Phone Number

```
Bot: "Введите номер телефона или нажмите кнопку:"
[📱 Поделиться контактом]

User shares contact OR types number
→ Validates format
→ Stored in session.orderData.phone
```

Step 5: Shipping Address

```
Bot: "Введите адрес доставки:"

User: "Moscow, Red Square, 1"
→ Stored in session.orderData.address
```

Step 6: Order Notes (Optional)

```
Bot: "Добавить комментарий? (или 'пропустить')"

User: "Deliver after 6 PM" OR "пропустить"
→ Stored in session.orderData.notes
```

Step 7: Confirmation

```
Bot: Shows order summary:
```

```
🛍 Ваш заказ:

Товар: T-Shirt Blue (M)
Цена: 1500.00 ₽
Количество: 5
Итого: 7500.00 ₽

Контакт: Ivan Petrov, +79001234567
Адрес: Moscow, Red Square, 1
Примечание: Deliver after 6 PM

[✅ Подтвердить] [❌ Отменить]
```

Step 8: Order Creation

```
User clicks [✅ Подтвердить]
→ Bot sends POST /orders to API
→ API creates order in database
→ API returns order ID
→ Bot shows success message
→ Bot notifies admins
```

Success message:

```
✅ Заказ #42 успешно создан!

Наш менеджер свяжется с вами в ближайшее время.

Спасибо за заказ! ❤
```

💾 State Management

Session Structure

```typescript
interface SessionData {
  conversationData: ConversationData; // Grammy conversations
  userData: UserData | null;          // Cached user info
  currentProduct: Product | null;     // Currently viewing product
  orderData: OrderData | null;        // Order being created
}

interface OrderData {
  product_id: number;
  variant_id: number;
  quantity: number;
  phone: string;
  address: string;
  notes?: string;
}
```

Session Initialization

```
bot.use(session({
  initial(): SessionData {
    return {
      conversationData: {},
      userData: null,
      currentProduct: null,
      orderData: null,
    };
  },
}));
```

🎛️ Handlers

Catalog Handler

File: `bot/src/handlers/catalog.ts`

Responsibilities: - Fetch products from API - Pagination (5 per page) - Product details display - Variant options

Key Functions:

```
// Show catalog page
async function showCatalog(ctx, page = 1)

// Show product details
async function showProductDetails(ctx, productId)

// Handle variant selection
async function handleVariantSelection(ctx, variantId)
```

Order Handler

File: `bot/src/handlers/order.ts`

Responsibilities: - Start order conversation - Show order history - Order details view

Key Functions:

```
// Show order history
async function showOrders(ctx, page = 1)

// Show specific order
async function showOrderDetails(ctx, orderId)
```

Profile Handler

File: `bot/src/handlers/profile.ts`

Responsibilities: - Display user info - Order statistics

Start Handler

File: `bot/src/handlers/start.ts`

Responsibilities: - Welcome message - Main menu display - Help information

---

🔗 Middleware

User Middleware

File: `bot/src/middleware/user.ts`

Purpose: Automatically create/update user in database from Telegram profile

Flow:

```
1. Every incoming message → middleware
2. Extract Telegram user data
3. Call POST /users/upsert
4. Store user in session
5. Continue to handler
```

Implementation:

```
export async function userMiddleware(ctx: BotContext, next:
        NextFunction) {
  const telegramUser = ctx.from;
  if (!telegramUser) {
    return next();
  }

  try {
    const user = await apiClient.upsertUser({
      telegram_id: telegramUser.id,
      username: telegramUser.username,
      full_name: `${telegramUser.first_name}
        ${telegramUser.last_name || ''}`.trim(),
    });

    ctx.session.userData = user;
  } catch (error) {
    logger.error('Failed to upsert user:', error);
  }

  return next();
}
```

Logging Middleware

File: `bot/src/middleware/logging.ts`

Purpose: Log all incoming messages/callbacks

Logs:

```
{
  "level": "info",
  "message": "Incoming message",
  "userId": 123456789,
  "username": "ivanpetrov",
  "messageType": "text",
  "text": "/start",
  "timestamp": "2025-12-04T15:30:00.000Z"
}
```

Error Handler Middleware

File: `bot/src/middleware/error-handler.ts`

Purpose: Catch and handle all errors

Behavior: - Logs error details - Shows user-friendly message - Doesn't crash bot

Error message:

```
❌ Произошла ошибка. Попробуйте позже или обратитесь в поддержку.
```

🔔 Notifications

Customer Notifications

Order Created

```
✅ Заказ #42 успешно создан!

Наш менеджер свяжется с вами в ближайшее время.
```

Order Status Changed

```
📦 Статус заказа #42 изменен:
Pending → Processing

Ваш заказ обрабатывается.
```

Admin Notifications

Sent to all admin Telegram IDs from `ADMIN_TELEGRAM_IDS` env variable.

New Order

```
🔔 Новый заказ #42

Клиент: Ivan Petrov (@ivanpetrov)
Телефон: +79001234567
Адрес: Moscow, Red Square, 1

Товары:
• T-Shirt Blue (M) x5 - 7500.00 ₽

Итого: 7500.00 ₽

[Обработать заказ →]
```

🔧 Utility Functions

API Client

File: `bot/src/services/api-client.ts`

Wrapper around Axios for API calls.

Methods:

```
class ApiClient {
  async getProducts(query): Promise<ProductListResponse>
  async getProductById(id): Promise<Product>
  async createOrder(data): Promise<Order>
  async getUserByTelegramId(id): Promise<User>
  async upsertUser(data): Promise<User>
}
```

Formatters

File: `bot/src/utils/formatters.ts`

Functions:

```
// Format price
formatPrice(1500) → "1500.00 ₽"

// Format date
formatDate(date) → "04.12.2025 15:30"

// Format product for display
formatProduct(product) → "T-Shirt Blue
1500.00 ₽
В наличии: 50 шт"

// Format order summary
formatOrderSummary(order) → "#42 - 3000.00 ₽ - Pending
Создан: 04.12.2025"
```

🐛 Error Handling

Common Errors

API Unreachable

```
try {
  const products = await apiClient.getProducts({page: 1});
} catch (error) {
  await ctx.reply('❌ Сервис временно недоступен. Попробуйте
        позже.');
  logger.error('API error:', error);
}
```

Out of Stock

```
if (variant.stock_quantity < requestedQuantity) {
  await ctx.reply(`❌ Недостаточно товара. Доступно:
        ${variant.stock_quantity} шт`);
  return;
}
```

Invalid Input

```
const quantity = parseInt(ctx.message.text);
if (isNaN(quantity) || quantity <= 0) {
  await ctx.reply('❌ Пожалуйста, введите корректное количество.');
  return;
}
```

🖨 Additional Resources

- Grammy Documentation
- Grammy Conversations
- API_DOCUMENTATION.md
- DEVELOPMENT.md

Updated: 2025-12-04
Version: 1.0.0

DATABASE.md

🗄 Database Documentation

📋 Table of Contents

🎯 Overview

Database: PostgreSQL 15
ORM: Prisma 6.0
Schema File: `nodejs_space/prisma/schema.prisma`

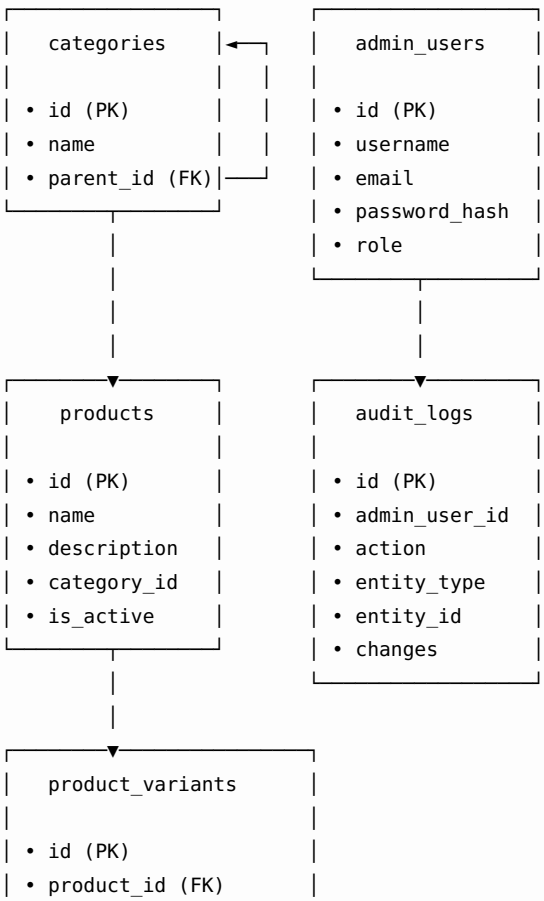
Connection String Format

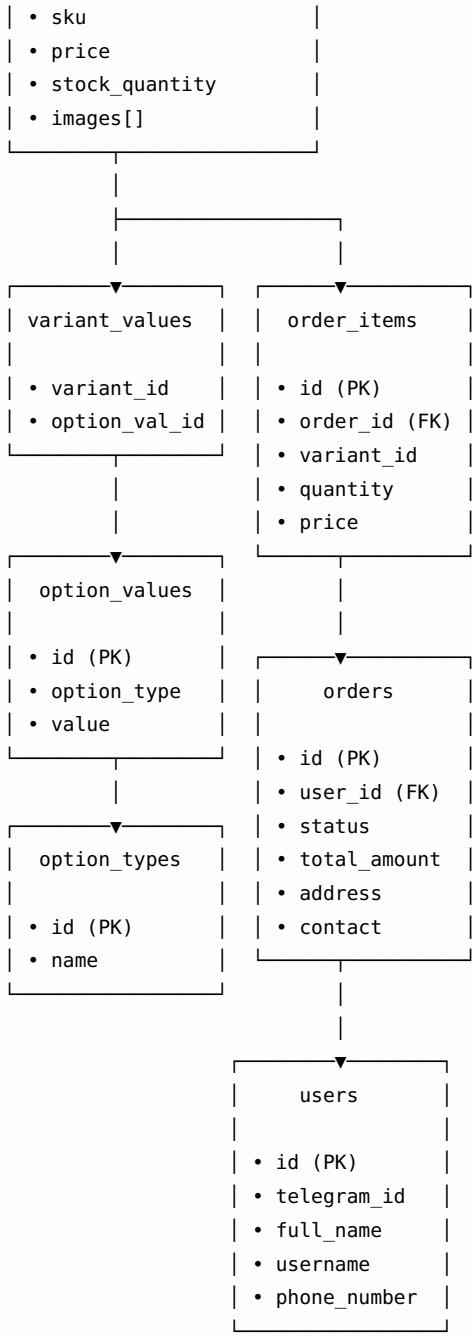
```
postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=public
```

Development:

```
postgresql://postgres:postgres@localhost:5432/telegram_shop?
schema=public
```

📊 Schema Diagram





🗃 Tables

users

Telegram customers who use the bot.

Column	Type	Nullable	Default	Descripti
id	BigInt	No	autoincrement()	Primary k
				Telegram

telegram_id	BigInt	Yes	NULL	User ID (unique)
full_name	String(255)	No	-	User’s full name
username	String(255)	Yes	NULL	Telegram username
phone_number	String(50)	Yes	NULL	Phone number
created_at	Timestamp	No	now()	Registrati date

Indexes: - PRIMARY KEY (id) - UNIQUE (telegram_id) - INDEX (username)

Example:

```
SELECT * FROM users WHERE telegram_id = 123456789;
```

---

admin_users

System administrators and managers.

Column	Type	Nullable	Default	Descripti
id	Int	No	autoincrement()	Primary k
username	String(100)	No	-	Login username
email	String(255)	No	-	Email address
password_hash	String(255)	No	-	Bcrypt password hash
role	String(50)	No	‘manager’	admin or manager
is_active	Boolean	No	true	Account status
created_at	Timestamp	No	now()	Creation date

Indexes: - PRIMARY KEY (id) - UNIQUE (username) - UNIQUE (email) - INDEX (username, email)

Example:

```
SELECT * FROM admin_users WHERE username = 'admin' AND is_active =
        true;
```

---

categories

Product categories with hierarchical structure.

Column	Type	Nullable	Default	Description
id	Int	No	autoincrement()	Primary key
name	String(255)	No	-	Category name
parent_id	Int	Yes	NULL	Parent category ID

Indexes: - PRIMARY KEY (id) - INDEX (parent_id) - FOREIGN KEY (parent_id) REFERENCES categories(id) ON DELETE CASCADE

Example:

```
-- Get root categories
SELECT * FROM categories WHERE parent_id IS NULL;

-- Get subcategories
SELECT * FROM categories WHERE parent_id = 1;
```

---

products

Products in the catalog.

Column	Type	Nullable	Default	Description
id	Int	No	autoincrement()	Primary key
name	String(255)	No	-	Product name
description	Text	No	-	Product description
category_id	Int	No	-	Category reference
is_active	Boolean	No	true	Visibility status
created_at	Timestamp	No	now()	Creation date

Indexes: - PRIMARY KEY (id) - INDEX (category_id) - INDEX (name) - INDEX (is_active) - FOREIGN KEY (category_id) REFERENCES categories(id) ON DELETE CASCADE

Example:

```
SELECT * FROM products
WHERE is_active = true AND category_id = 1
ORDER BY created_at DESC;
```

---

product_variants

SKU-level product variants (size, color combinations).

Column	Type	Nullable	Default	Descrip
id	Int	No	autoincrement()	Primary
product_id	Int	No	-	Product referenc
sku	String(100)	No	-	Stock Keeping Unit (unique)
price	Decimal(10,2)	No	-	Variant price
stock_quantity	Int	No	0	Availabl quantity
weight	Decimal(10,2)	Yes	NULL	Weight i
dimensions	String(255)	Yes	NULL	LxWxH cm
images	String[]	No	[]	Array of image U
created_at	Timestamp	No	now()	Creatior date

Indexes: - PRIMARY KEY (id) - UNIQUE (sku) - INDEX (product_id) - FOREIGN KEY (product_id) REFERENCES products(id) ON DELETE CASCADE

Example:

```
SELECT * FROM product_variants
WHERE product_id = 1 AND stock_quantity > 0;
```

---

option_types

Types of product options (Color, Size, Material, etc.).

Column	Type	Nullable	Default	Description
id	Int	No	autoincrement()	Primary key
name	String(100)	No	-	Option type name (unique)

Example:

```
INSERT INTO option_types (name) VALUES ('Color'), ('Size');
```

---

option_values

Possible values for option types (Red, Blue, M, L, etc.).

Column	Type	Nullable	Default	Descripti
id	Int	No	autoincrement()	Primary ke

option_type_id	Int	No	-	Option typ reference
value	String(100)	No	-	Option val

Indexes: - PRIMARY KEY (id) - INDEX (option_type_id) - FOREIGN KEY (option_type_id) REFERENCES option_types(id) ON DELETE CASCADE

Example:

```
-- Get all colors
SELECT ov.* FROM option_values ov
JOIN option_types ot ON ov.option_type_id = ot.id
WHERE ot.name = 'Color';
```

---

variant_values

Junction table linking variants to their option values.

Column	Type	Nullable	Default	Description
variant_id	Int	No	-	Variant reference
option_value_id	Int	No	-	Option value reference

Indexes: - PRIMARY KEY (variant_id, option_value_id) - FOREIGN KEY (variant_id) REFERENCES product_variants(id) ON DELETE CASCADE - FOREIGN KEY (option_value_id) REFERENCES option_values(id) ON DELETE CASCADE

Example:

```
-- Get all options for a variant
SELECT ot.name AS type, ov.value
FROM variant_values vv
JOIN option_values ov ON vv.option_value_id = ov.id
JOIN option_types ot ON ov.option_type_id = ot.id
WHERE vv.variant_id = 1;
```

---

orders

Customer orders.

Column	Type	Nullable	Default	Desc
id	Int	No	autoincrement()	Prim
user_id	BigInt	No	-	User refer
status	String(50)	No	‘pending’	Orde
total_amount	Decimal(10,2)	No	-	Total amou
				Deliv

shipping_address	Text	No	-	addr
customer_contact	Text	No	-	Nam phon
notes	Text	Yes	NULL	Cust notes
created_at	Timestamp	No	now()	Orde
updated_at	Timestamp	No	now()	Last

Valid statuses: `pending`, `processing`, `shipped`, `delivered`, `cancelled`

Indexes: - PRIMARY KEY (id) - INDEX (user_id) - INDEX (status) - INDEX (created_at) - FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE

Example:

```
SELECT * FROM orders
WHERE status = 'pending'
ORDER BY created_at DESC;
```

---

order_items

Items within an order.

Column	Type	Nullable	Default	Des
id	Int	No	autoincrement()	Prim
order_id	Int	No	-	Orde refe
variant_id	Int	No	-	Vari refe
quantity	Int	No	-	Qua orde
price_at_purchase	Decimal(10,2)	No	-	Pric of or

Indexes: - PRIMARY KEY (id) - INDEX (order_id) - INDEX (variant_id) - FOREIGN KEY (order_id) REFERENCES orders(id) ON DELETE CASCADE - FOREIGN KEY (variant_id) REFERENCES product_variants(id) ON DELETE RESTRICT

Example:

```
SELECT oi.*, pv.sku, p.name
FROM order_items oi
JOIN product_variants pv ON oi.variant_id = pv.id
JOIN products p ON pv.product_id = p.id
WHERE oi.order_id = 42;
```

---

audit_logs

Audit trail of admin actions.

Column	Type	Nullable	Default	Descriptio
id	Int	No	autoincrement()	Primary ke
admin_user_id	Int	No	-	Admin reference
action	String(100)	No	-	CREATE, UPDATE, DELETE
entity_type	String(100)	No	-	products, orders, users
entity_id	Int	Yes	NULL	ID of affected entity
changes	Text	Yes	NULL	JSON of changes
created_at	Timestamp	No	now()	Action timestamp

Indexes: - PRIMARY KEY (id) - INDEX (admin_user_id) - INDEX (entity_type, entity_id) - INDEX (created_at) - FOREIGN KEY (admin_user_id) REFERENCES admin_users(id) ON DELETE CASCADE

Example:

```
SELECT * FROM audit_logs
WHERE entity_type = 'products' AND entity_id = 1
ORDER BY created_at DESC;
```

🔗 Relationships

One-to-Many

```
categories (1) ←→ (N) categories (self-referencing)
categories (1) ←→ (N) products
products (1) ←→ (N) product_variants
option_types (1) ←→ (N) option_values
users (1) ←→ (N) orders
orders (1) ←→ (N) order_items
admin_users (1) ←→ (N) audit_logs
```

Many-to-Many

```
product_variants (N) ←→ (N) option_values
    (через variant_values)
```

🔍 Indexes

Purpose

- **Primary Keys:** Fast row lookup
- **Foreign Keys:** Efficient joins
- **Search Fields:** Quick filtering

Index Usage Examples

```sql
-- Uses INDEX on telegram_id
SELECT * FROM users WHERE telegram_id = 123456789;

-- Uses INDEX on category_id and is_active
SELECT * FROM products
WHERE category_id = 1 AND is_active = true;

-- Uses INDEX on status and created_at
SELECT * FROM orders
WHERE status = 'pending'
ORDER BY created_at DESC;
```

🔄 Migrations

Applying Schema

```bash
cd nodejs_space

# Push schema to database (development)
yarn prisma db push

# Generate Prisma Client
yarn prisma generate

# View current schema
yarn prisma db pull
```

Schema Location

```
nodejs_space/prisma/schema.prisma
```

Making Changes

1. Edit `schema.prisma`
2. Run `yarn prisma db push`
3. Generate client: `yarn prisma generate`
4. Restart application

Example:

```
// Add new field
model products {
  // ... existing fields
  discount_percent Int? @default(0)
}
```

```
yarn prisma db push
yarn prisma generate
```

🌱 Seed Data

Location

`nodejs_space/prisma/seed.ts`

Running Seed

```
cd nodejs_space
yarn prisma db seed
```

What Gets Seeded

1. **Admin User:**
   - Username: `admin`
   - Password: `admin123`
   - Email: `admin@example.com`
   - Role: `admin`
2. **Categories:**
   - Clothing
     - T-Shirts
     - Jeans
   - Electronics
   - Home & Garden
3. **Products:**
   - 10 sample products with variants
   - Different categories
   - Multiple images
   - Stock quantities
4. **Option Types:**
   - Color
   - Size
5. **Option Values:**
   - Colors: Red, Blue, Green, Black, White
   - Sizes: XS, S, M, L, XL, XXL

Custom Seed

Edit `prisma/seed.ts`:

```
async function main() {
  // Create admin
```

```
  await prisma.admin_users.create({
    data: {
      username: 'myAdmin',
      email: 'my@email.com',
      password_hash: await bcrypt.hash('myPassword', 10),
      role: 'admin',
    },
  });

  // Add more seed data...
}

main();
```

🖨️ Additional Resources

- Prisma Documentation
- PostgreSQL Documentation
- ARCHITECTURE.md
- DEVELOPMENT.md

Updated: 2025-12-04
Version: 1.0.0

ENVIRONMENT.md

🌍 Environment Variables

📋 Table of Contents

🎯 Overview

Environment variables are used to configure the application without changing code.

Files

- `.env` - Main environment file (root directory)
- `nodejs_space/.env` - Backend API specific

- `bot/.env` - Bot specific
- `.env.example` - Example template

⚠ **IMPORTANT:** Never commit `.env` files to Git!

🔧 Backend API Variables

Location: `nodejs_space/.env`

DATABASE_URL

Required: Yes
Type: Connection string
Description: PostgreSQL database connection string

Format:

```
postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=public
```

Examples:

```
# Development (local PostgreSQL)
DATABASE_URL=postgresql://shopuser:shoppassword@localhost:5432/telegı
schema=public

# Docker Compose
DATABASE_URL=postgresql://postgres:postgres@postgres:5432/telegram_sl
schema=public

# Production (with SSL)
DATABASE_URL=postgresql://produser:strongpass@db.example.com:5432/tel
schema=public&sslmode=require
```

PORT

Required: No
Type: Number
Default: 3000
Description: Port on which API server listens

```
PORT=3000
```

Note: In Docker, internal port is always 3000, external mapping is in `docker-compose.yml`

JWT_SECRET

Required: Yes
Type: String
Description: Secret key for signing JWT tokens

⚠ **CRITICAL:** Must be changed in production!

Requirements: - Minimum 32 characters - Random string - Include letters, numbers, symbols

Generate secure secret:

```
# Using OpenSSL
openssl rand -base64 32

# Using Node.js
node -e
        "console.log(require('crypto').randomBytes(32).toString('base64'))"
```

Example:

```
JWT_SECRET=a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0u1v2w3x4y5z6
```

JWT_EXPIRES_IN

Required: No
Type: String (time format)
Default: 30m
Description: JWT token expiration time

Formats: - `30m` - 30 minutes - `1h` - 1 hour - `24h` - 24 hours - `7d` - 7 days

```
JWT_EXPIRES_IN=30m
```

Recommendations: - Development: `24h` - Production: `30m` to `1h`

NODE_ENV

Required: No
Type: String
Default: development
Description: Application environment

Valid values: - `development` - Dev mode, verbose logs - `production` - Production mode, optimized - `test` - Testing mode

```
NODE_ENV=production
```

Impact: - Affects logging verbosity - Enables/disables debug features - Changes error messages (detailed vs generic)

CORS_ORIGIN

Required: No
Type: String (comma-separated)
Default: *
Description: Allowed CORS origins

```
# Allow all (development only!)
```

```
CORS_ORIGIN=*

# Single domain
CORS_ORIGIN=https://yourdomain.com

# Multiple domains
CORS_ORIGIN=https://yourdomain.com,https://admin.yourdomain.com,http
```

GA4_MEASUREMENT_ID

Required: No
Type: String
Description: Google Analytics 4 Measurement ID

```
GA4_MEASUREMENT_ID=G-XXXXXXXXXX
```

To get: 1. Go to Google Analytics 2. Admin → Data Streams 3. Copy Measurement ID

YANDEX_METRIKA_ID

Required: No
Type: Number
Description: Yandex.Metrika counter ID

```
YANDEX_METRIKA_ID=12345678
```

🤖 Telegram Bot Variables

Location: `bot/.env`

BOT_TOKEN

Required: Yes
Type: String
Description: Telegram Bot API token from @BotFather

Format: `<bot_id>:<hash>`

```
BOT_TOKEN=1234567890:ABCdefGhIjKlmNoPqRsTuVwXyZ
```

To get: 1. Open Telegram 2. Find @BotFather 3. Send `/newbot` 4. Follow instructions 5. Copy token

---

API_BASE_URL

Required: Yes
Type: URL
Description: Base URL of Backend API

```
# Development (local)
API_BASE_URL=http://localhost:3000/api

# Docker Compose (service name)
API_BASE_URL=http://api:3000/api

# Production
API_BASE_URL=https://api.yourdomain.com/api
```

API_TIMEOUT

Required: No
Type: Number (milliseconds)
Default: 10000
Description: HTTP request timeout

```
API_TIMEOUT=10000
```

ADMIN_TELEGRAM_IDS

Required: Yes
Type: String (comma-separated numbers)
Description: Telegram IDs of administrators who receive notifications

```
# Single admin
ADMIN_TELEGRAM_IDS=123456789

# Multiple admins
ADMIN_TELEGRAM_IDS=123456789,987654321,555444333
```

To get your Telegram ID: 1. Open Telegram 2. Find @userinfobot 3. Send `/start` 4. Copy your ID

LOG_LEVEL

Required: No
Type: String
Default: info
Description: Logging verbosity level

Valid values (from least to most verbose): - `error` - Only errors - `warn` - Warnings and errors - `info` - General information - `debug` - Debug information - `verbose` - Everything

```
# Development
LOG_LEVEL=debug

# Production
LOG_LEVEL=info
```

🐳 Docker Compose Variables

Location: `.env` (root directory)

Used by `docker-compose.yml`.

Complete Example

```
# Database
DATABASE_URL=postgresql://postgres:postgres@postgres:5432/telegram_sh
schema=public

# Backend API
PORT=3000
JWT_SECRET=your-super-secret-jwt-key-change-in-production-minimum-
32-characters
JWT_EXPIRES_IN=30m
NODE_ENV=production
CORS_ORIGIN=*

# Telegram Bot
BOT_TOKEN=1234567890:ABCdefGhIjKlmNoPqRsTuVwXyZ
ADMIN_TELEGRAM_IDS=123456789,987654321
API_BASE_URL=http://api:3000/api
API_TIMEOUT=10000
LOG_LEVEL=info

# Optional: Analytics
GA4_MEASUREMENT_ID=G-XXXXXXXXXX
YANDEX_METRIKA_ID=12345678
```

🔐 Security Best Practices

1. Never Commit Secrets

Add to `.gitignore`:

```
.env
.env.*
!.env.example
```

2. Use Strong Secrets

Bad:

```
JWT_SECRET=secret123
```

Good:

```
JWT_SECRET=8f7e6d5c4b3a2918273645aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpP
```

3. Different Secrets Per Environment

```
# Development
JWT_SECRET=dev-secret-not-for-production

# Production
JWT_SECRET=prod-8f7e6d5c4b3a2918273645aAbBcCdD
```

4. Rotate Secrets Regularly

Change sensitive secrets periodically (every 3-6 months): - JWT_SECRET - Database passwords - API keys

5. Limit Access

Only authorized personnel should have access to production `.env` files.

6. Use Environment-Specific Files

```
.env              # Default/development
.env.production   # Production
.env.staging      # Staging
.env.test         # Testing
```

7. Validate on Startup

Application should validate required environment variables on startup:

```
if (!process.env.JWT_SECRET || process.env.JWT_SECRET.length < 32) {
  throw new Error('JWT_SECRET must be at least 32 characters');
}
```

8. Document All Variables

Keep `.env.example` updated with all required variables (without real values):

```
# .env.example
DATABASE_URL=postgresql://user:password@localhost:5432/dbname
JWT_SECRET=your-secret-here-min-32-chars
BOT_TOKEN=your-telegram-bot-token
```

🔧 Loading Environment Variables

Backend API (NestJS)

```
import { ConfigService } from '@nestjs/config';

constructor(private configService: ConfigService) {}
```

```
const port = this.configService.get<number>('PORT', 3000);
const jwtSecret = this.configService.get<string>('JWT_SECRET');
```

Telegram Bot

```
import dotenv from 'dotenv';
dotenv.config();

const BOT_TOKEN = process.env.BOT_TOKEN!;
const API_BASE_URL = process.env.API_BASE_URL ||
        'http://localhost:3000/api';
```

🖨 Additional Resources

- SETUP.md - Installation guide
- DOCKER.md - Docker configuration
- TROUBLESHOOTING.md - Common issues

Updated: 2025-12-04
Version: 1.0.0

DOCKER.md

🐋 Docker Documentation

📑 Table of Contents

🎯 Overview

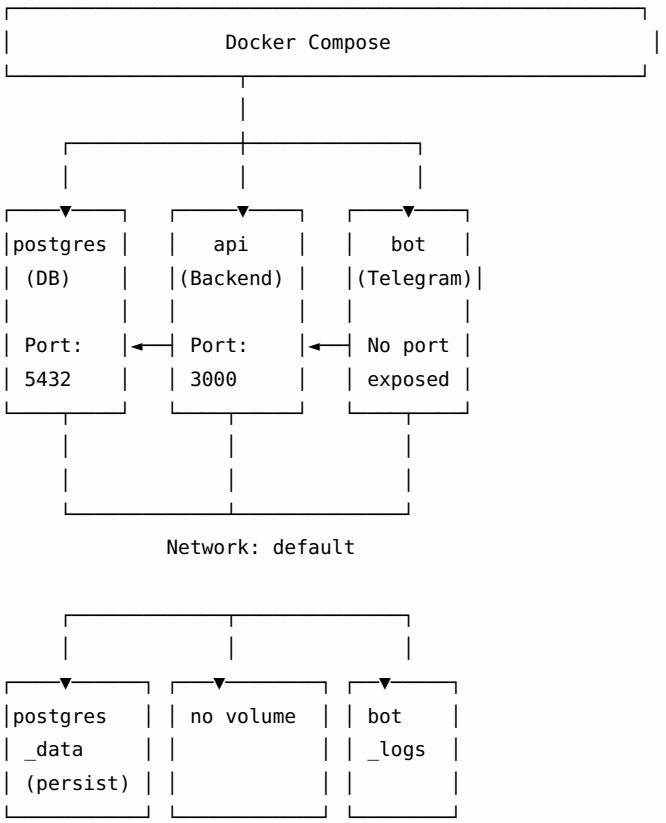
The project uses **Docker Compose** for containerization and orchestration.

Benefits

- ✅ Consistent environment across all systems
- ✅ Easy deployment
- ✅ Isolated services

- ✅ Automatic dependency management
- ✅ Health checks
- ✅ Easy scaling

🏗 Docker Architecture



🔧 Services

1. postgres

PostgreSQL database service.

Image: `postgres:15-alpine`
Container Name: `telegram_shop_db`
Restart Policy: `unless-stopped`

Environment: - `POSTGRES_USER=postgres` - `POSTGRES_PASSWORD=postgres` - `POSTGRES_DB=telegram_shop`

Ports: - `5432:5432` (host:container)

Volume: - `postgres_data:/var/lib/postgresql/data`

Health Check:

```
test: ['CMD-SHELL', 'pg_isready -U postgres']
interval: 10s
timeout: 5s
retries: 5
```

---

2. api

Backend API service (NestJS).

Build Context: .
Dockerfile: `./Dockerfile`
Container Name: `telegram_shop_api`
Restart Policy: `unless-stopped`

Environment: - `NODE_ENV=production` - `DATABASE_URL` (from .env) - `PORT=3000` - `JWT_SECRET` (from .env) - `JWT_EXPIRES_IN=30m`

Ports: - `3000:3000`

Depends On: - `postgres` (waits for healthy status)

Command:

```
sh -c "yarn prisma db push && yarn prisma:seed && node dist/main.js"
```

What happens on start: 1. Waits for PostgreSQL to be healthy 2. Applies database schema (`prisma db push`) 3. Seeds initial data (`prisma:seed`) 4. Starts NestJS application

---

3. bot

Telegram bot service (Grammy).

Build Context: `./bot`
Dockerfile: `./bot/Dockerfile`
Container Name: `telegram_shop_bot`
Restart Policy: `unless-stopped`

Environment: - `NODE_ENV=production` - `BOT_TOKEN` (from .env) - `API_BASE_URL=http://api:3000/api` - `API_TIMEOUT=10000` - `ADMIN_TELEGRAM_IDS` (from .env) - `LOG_LEVEL=info`

Depends On: - `api`

Volume: - `bot_logs:/app/logs`

🗎 docker-compose.yml

Location: `./docker-compose.yml`

```
version: '3.8'

services:
```

```yaml
postgres:
  image: postgres:15-alpine
  container_name: telegram_shop_db
  restart: unless-stopped
  environment:
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_DB: telegram_shop
  ports:
    - '5432:5432'
  volumes:
    - postgres_data:/var/lib/postgresql/data
  healthcheck:
    test: ['CMD-SHELL', 'pg_isready -U postgres']
    interval: 10s
    timeout: 5s
    retries: 5

api:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: telegram_shop_api
  restart: unless-stopped
  ports:
    - '3000:3000'
  environment:
    NODE_ENV: production
    DATABASE_URL:
      postgresql://postgres:postgres@postgres:5432/telegram_shop?
      schema=public
    PORT: 3000
    JWT_SECRET: ${JWT_SECRET:-your-secret-key-change-in-
      production}
    JWT_EXPIRES_IN: 30m
  depends_on:
    postgres:
      condition: service_healthy
  command: sh -c "yarn prisma db push && yarn prisma:seed && node
      dist/main.js"

bot:
  build:
    context: ./bot
    dockerfile: Dockerfile
  container_name: telegram_shop_bot
  restart: unless-stopped
  environment:
    NODE_ENV: production
    BOT_TOKEN: ${BOT_TOKEN}
    API_BASE_URL: http://api:3000/api
    API_TIMEOUT: 10000
    ADMIN_TELEGRAM_IDS: ${ADMIN_TELEGRAM_IDS}
    LOG_LEVEL: info
```

```
    depends_on:
      - api
    volumes:
      - bot_logs:/app/logs

volumes:
  postgres_data:
  bot_logs:
```

💾 Volumes

postgres_data

Type: Named volume
Purpose: Persist PostgreSQL data
Location: `/var/lib/postgresql/data` (inside container)

Why persistent: - Database survives container restarts - Data not lost on `docker-compose down`

To delete:

```
docker-compose down -v
```

---

bot_logs

Type: Named volume
Purpose: Persist bot logs
Location: `/app/logs` (inside container)

Access logs:

```
# From host
docker-compose exec bot ls -lh /app/logs

# Copy to host
docker cp telegram_shop_bot:/app/logs ./bot_logs_backup
```

🌐 Networks

default

Docker Compose automatically creates a default network.

Services can communicate using service names:

```
// Bot can reach API using service name
const API_BASE_URL = 'http://api:3000/api';

// API can reach PostgreSQL using service name
```

```
DATABASE_URL =
        'postgresql://postgres:postgres@postgres:5432/telegram_shop';
```

DNS Resolution: - `postgres` → `172.18.0.2` (example) - `api` → `172.18.0.3` - `bot` → `172.18.0.4`

🕸 Common Commands

Start Services

```
# Start all services
docker-compose up -d

# Start specific service
docker-compose up -d api

# Start with build
docker-compose up -d --build

# View startup logs
docker-compose up
```

Stop Services

```
# Stop all services
docker-compose stop

# Stop specific service
docker-compose stop bot

# Stop and remove containers
docker-compose down

# Stop and remove containers + volumes (⚠ data loss!)
docker-compose down -v
```

View Logs

```
# All services
docker-compose logs -f

# Specific service
docker-compose logs -f api
docker-compose logs -f bot
docker-compose logs -f postgres

# Last N lines
docker-compose logs --tail=100 api

# Since timestamp
```

```
docker-compose logs --since="2025-12-04T10:00:00"
```

Service Status

```
# List running services
docker-compose ps

# Detailed info
docker-compose ps -a

# Resource usage
docker stats

# Service health
docker-compose ps | grep healthy
```

Restart Services

```
# Restart all
docker-compose restart

# Restart specific
docker-compose restart api
docker-compose restart bot
```

Execute Commands in Containers

```
# Open shell in API container
docker-compose exec api sh

# Run Prisma migrations
docker-compose exec api npx prisma db push

# Check database
docker-compose exec postgres psql -U postgres -d telegram_shop

# View logs inside bot container
docker-compose exec bot ls -lh /app/logs
```

Rebuild Services

```
# Rebuild all images
docker-compose build

# Rebuild specific service
docker-compose build api

# Rebuild without cache
docker-compose build --no-cache
```

```
# Rebuild and start
docker-compose up -d --build
```

Clean Up

```
# Remove stopped containers
docker-compose rm

# Remove all project containers, networks, images
docker-compose down --rmi all

# Remove everything including volumes (⚠ DATA LOSS!)
docker-compose down -v --rmi all

# Clean up dangling images
docker image prune

# Full system cleanup
docker system prune -a --volumes
```

🔍 Inspecting Services

View Container Details

```
# Container info
docker inspect telegram_shop_api

# Network info
docker network inspect telegram_shop_backend_default

# Volume info
docker volume inspect telegram_shop_backend_postgres_data
```

Resource Monitoring

```
# Real-time stats
docker stats

# Specific containers
docker stats telegram_shop_api telegram_shop_bot

# Disk usage
docker system df
```

🐛 Troubleshooting

Service Won't Start

```
# Check logs for errors
docker-compose logs api

# Check last 50 lines
docker-compose logs --tail=50 api

# Remove and rebuild
docker-compose down
docker-compose up -d --build
```

Database Connection Issues

```
# Check PostgreSQL is running
docker-compose ps postgres

# Check PostgreSQL health
docker-compose exec postgres pg_isready -U postgres

# Check connection from API container
docker-compose exec api ping postgres

# Manual database connection
docker-compose exec postgres psql -U postgres -d telegram_shop -c
        "SELECT 1;"
```

Port Conflicts

```
Error: port is already allocated
```

Solution:

```
# Find process using port 3000
lsof -i :3000
# or
netstat -tulpn | grep 3000

# Kill process
kill -9 <PID>

# Or change port in docker-compose.yml
ports:
  - '3001:3000'  # Map to different host port
```

Permission Issues

```
# Fix volume permissions
docker-compose down
sudo chown -R $USER:$USER postgres_data/
docker-compose up -d
```

Out of Disk Space

```
# Check disk usage
docker system df

# Clean up
docker system prune -a --volumes

# Remove specific volume
docker volume rm telegram_shop_backend_postgres_data
```

Bot Not Receiving Messages

```
# Check bot logs
docker-compose logs -f bot

# Check API is accessible from bot
docker-compose exec bot wget -O- http://api:3000/api/products

# Check BOT_TOKEN is correct
docker-compose exec bot env | grep BOT_TOKEN

# Restart bot
docker-compose restart bot
```

🖨 Additional Resources

- Docker Documentation
- Docker Compose Documentation
- SETUP.md
- TROUBLESHOOTING.md

Updated: 2025-12-04
Version: 1.0.0

DEVELOPMENT.md

🛠 Development Guide

📋 Table of Contents

🚀 Getting Started

Prerequisites

- Node.js 18+ LTS
- Yarn 4+
- PostgreSQL 15+
- Git
- Code editor (VSCode recommended)

Setup Development Environment

```
# Clone/extract project
cd telegram_shop_backend

# Install backend dependencies
cd nodejs_space
yarn install

# Install bot dependencies
cd ../bot
yarn install

# Setup database
cd ../nodejs_space
yarn prisma db push
yarn prisma db seed
```

📁 Project Structure

Backend API (`nodejs_space/`)

```
nodejs_space/
├── prisma/
│   ├── schema.prisma          # Database schema
│   └── seed.ts                # Seed data
├── src/
│   ├── main.ts                # Entry point
│   ├── app.module.ts          # Root module
│   │
│   ├── auth/                  # Authentication
│   │   ├── auth.module.ts
│   │   ├── auth.service.ts
│   │   ├── auth.controller.ts
```

```
│   │   ├── guards/            # JWT, Roles guards
│   │   ├── strategies/        # Passport strategies
│   │   └── dto/               # Data Transfer Objects
│   │
│   ├── products/              # Products module
│   │   ├── products.module.ts
│   │   ├── products.service.ts
│   │   ├── products.controller.ts
│   │   └── dto/
│   │
│   ├── orders/                # Orders module
│   ├── users/                 # Users module
│   ├── import-export/         # Import/Export module
│   ├── integrations/          # External integrations
│   ├── webhook/               # Telegram webhook
│   │
│   ├── prisma/                # Prisma service
│   │   ├── prisma.module.ts
│   │   └── prisma.service.ts
│   │
│   └── common/                # Shared components
│       ├── decorators/        # Custom decorators
│       ├── filters/           # Exception filters
│       └── interceptors/      # Interceptors
│
├── test/                      # E2E tests
├── package.json
└── tsconfig.json
```

Telegram Bot (`bot/`)

```
bot/
├── src/
│   ├── index.ts               # Entry point
│   ├── config/                # Configuration
│   │   └── index.ts
│   │
│   ├── handlers/              # Command handlers
│   │   ├── start.ts           # /start command
│   │   ├── catalog.ts         # Catalog browsing
│   │   ├── order.ts           # Order history
│   │   └── profile.ts         # User profile
│   │
│   ├── conversations/         # Dialog flows
│   │   └── order.ts           # Order conversation
│   │
│   ├── middleware/            # Bot middleware
│   │   ├── user.ts            # User management
│   │   ├── logging.ts         # Logging
│   │   └── error-handler.ts  # Error handling
│   │
│   ├── services/              # Services
```

```
│   │   ├── api-client.ts      # API wrapper
│   │   └── notification.ts    # Notifications
│   │
│   ├── utils/                 # Utilities
│   │   ├── logger.ts          # Winston logger
│   │   └── formatters.ts      # Formatters
│   │
│   └── types/                 # TypeScript types
│       └── index.ts
│
├── package.json
└── tsconfig.json
```

💻 Development Workflow

Starting Development Servers

Terminal 1: Database

```
# Start PostgreSQL (if not using Docker)
pg_ctl start

# Or use Docker
docker run --name dev-postgres -e POSTGRES_PASSWORD=postgres -p
        5432:5432 -d postgres:15-alpine
```

Terminal 2: Backend API

```
cd nodejs_space

# Development mode with watch
yarn start:dev

# Or production mode
yarn build
yarn start:prod
```

API will be available at: http://localhost:3000

Terminal 3: Telegram Bot

```
cd bot

# Development mode with watch
yarn dev

# Or production mode
yarn build
yarn start
```

Making Changes

1. **Create feature branch**

```
git checkout -b feature/my-feature
```

2. **Make changes**
3. **Test locally**
4. **Commit**

```
git add .
git commit -m "Add: My feature description"
```

5. **Push**

```
git push origin feature/my-feature
```

✚ Adding Features

Adding a New API Endpoint

1. Create DTO

```
// src/products/dto/my-dto.ts
import { IsString, IsNumber } from 'class-validator';

export class MyDto {
  @IsString()
  name: string;

  @IsNumber()
  price: number;
}
```

2. Add Service Method

```
// src/products/products.service.ts
async myNewMethod(data: MyDto): Promise<Product> {
  return this.prisma.products.create({
    data: {
      name: data.name,
      price: data.price,
      // ...
    },
  });
}
```

3. Add Controller Endpoint

```
// src/products/products.controller.ts
@Post('my-endpoint')
@ApiOperation({ summary: 'My new endpoint' })
async myEndpoint(@Body() data: MyDto): Promise<Product> {
  return this.productsService.myNewMethod(data);
```

```
}
```

4. Test

```
curl -X POST http://localhost:3000/api/products/my-endpoint   -H
        "Content-Type: application/json"   -d
        '{"name":"Test","price":100}'
```

Adding a New Bot Command

1. Create Handler

```
// bot/src/handlers/my-handler.ts
import { Bot } from 'grammy';
import type { BotContext } from '../types';

export function registerMyHandler(bot: Bot<BotContext>) {
  bot.command('mycommand', async (ctx) => {
    await ctx.reply('Hello from my command!');
  });

  bot.callbackQuery(/^my:/, async (ctx) => {
    await ctx.answerCallbackQuery();
    // Handle callback
  });
}
```

2. Register Handler

```
// bot/src/index.ts
import { registerMyHandler } from './handlers/my-handler';

// ... in main()
registerMyHandler(bot);
```

Adding a Database Table

1. Edit Prisma Schema

```
// nodejs_space/prisma/schema.prisma
model my_table {
  id         Int      @id @default(autoincrement())
  name       String   @db.VarChar(255)
  created_at DateTime @default(now()) @db.Timestamptz()

  @@map("my_table")
}
```

2. Apply Migration

```
cd nodejs_space
yarn prisma db push
```

```
yarn prisma generate
```

3. Use in Service

```
async getMyData(): Promise<MyTable[]> {
  return this.prisma.my_table.findMany();
}
```

🧪 Testing

Unit Tests

```
cd nodejs_space

# Run all tests
yarn test

# Watch mode
yarn test:watch

# Coverage
yarn test:cov
```

E2E Tests

```
cd nodejs_space

# Run E2E tests
yarn test:e2e
```

Manual API Testing

Using cURL:

```
# Get products
curl http://localhost:3000/api/products

# Create product (with auth)
TOKEN=$(curl -s -X POST http://localhost:3000/api/auth/token   -H
        "Content-Type: application/json"   -d
        '{"username":"admin","password":"admin123"}'   | jq -r
        '.access_token')

curl -X POST http://localhost:3000/api/admin/products   -H
        "Authorization: Bearer $TOKEN"   -H "Content-Type:
        application/json"   -d '{"name":"Test
        Product","description":"Test","category_id":1}'
```

Using Swagger UI: 1. Open http://localhost:3000/api-docs 2. Click “Authorize” and enter JWT token 3. Test endpoints interactively

---

📝 Code Style

TypeScript

Follow NestJS conventions and TypeScript best practices.

Example:

```typescript
// Good
export class ProductsService {
  private readonly logger = new Logger(ProductsService.name);

  constructor(private readonly prisma: PrismaService) {}

  async getProducts(query: ProductListQueryDto):
        Promise<ProductListResponseDto> {
    this.logger.log('Fetching products');

    const products = await this.prisma.products.findMany({
      where: { is_active: true },
      take: query.limit,
      skip: (query.page - 1) * query.limit,
    });

    return {
      data: products,
      meta: { page: query.page, limit: query.limit },
    };
  }
}
```

Naming Conventions

Files: - `kebab-case.ts` for files - `my-feature.service.ts` - `my-feature.controller.ts` - `my-feature.module.ts`

Classes: - `PascalCase` for classes - `MyFeatureService` - `MyFeatureController`

Functions: - `camelCase` for functions - `getProducts()` - `createOrder()`

Constants: - `UPPER_SNAKE_CASE` for constants - `API_BASE_URL` - `MAX_ITEMS_PER_PAGE`

Linting

```bash
# Backend
cd nodejs_space
yarn lint

# Bot
cd bot
yarn lint
```

🐛 Debugging

VSCode Debug Configuration

.vscode/launch.json:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Debug API",
      "runtimeExecutable": "yarn",
      "runtimeArgs": ["start:debug"],
      "cwd": "${workspaceFolder}/nodejs_space",
      "restart": true,
      "console": "integratedTerminal",
      "internalConsoleOptions": "neverOpen"
    },
    {
      "type": "node",
      "request": "launch",
      "name": "Debug Bot",
      "runtimeExecutable": "yarn",
      "runtimeArgs": ["dev"],
      "cwd": "${workspaceFolder}/bot",
      "restart": true,
      "console": "integratedTerminal"
    }
  ]
}
```

Logging

Backend:

```
import { Logger } from '@nestjs/common';

const logger = new Logger('MyService');
logger.log('Info message');
logger.error('Error message', error.stack);
logger.warn('Warning message');
logger.debug('Debug message');
```

Bot:

```
import { logger } from './utils/logger';

logger.info('Info message', { userId: 123 });
logger.error('Error message', error);
```

Database Debugging

```
# Open Prisma Studio (GUI for database)
cd nodejs_space
yarn prisma studio

# Manual queries
yarn prisma db pull  # Update schema from DB
yarn prisma format   # Format schema file
```

🤝 Contributing

Git Workflow

1. Fork/clone repository
2. Create feature branch: `git checkout -b feature/my-feature`
3. Make changes
4. Run tests: `yarn test`
5. Run linter: `yarn lint`
6. Commit: `git commit -m "Add: My feature"`
7. Push: `git push origin feature/my-feature`
8. Create Pull Request

Commit Message Format

```
Type: Brief description

Detailed explanation if needed.

Closes #123
```

Types: - `Add:` - New feature - `Fix:` - Bug fix - `Update:` - Update existing feature - `Refactor:` - Code refactoring - `Docs:` - Documentation - `Test:` - Tests - `Chore:` - Maintenance

Examples:

```
Add: Product filtering by price range

Implement price range filter in products endpoint.
Supports min_price and max_price query parameters.

Closes #45
```

```
Fix: Order total calculation error

Fixed decimal precision issue in order total calculation
that caused rounding errors.

Closes #67
```

🔧 Useful Commands

Backend API

```
cd nodejs_space

# Development
yarn start:dev          # Start with watch mode
yarn start:debug        # Start with debug mode

# Production
yarn build              # Compile TypeScript
yarn start:prod         # Start compiled app

# Database
yarn prisma studio      # Open Prisma Studio
yarn prisma db push     # Apply schema changes
yarn prisma db seed     # Run seed script
yarn prisma generate    # Generate Prisma Client

# Testing
yarn test               # Run unit tests
yarn test:e2e           # Run E2E tests
yarn test:cov           # Run with coverage

# Quality
yarn lint               # Run ESLint
yarn format             # Run Prettier
```

Telegram Bot

```
cd bot

# Development
yarn dev                # Start with watch mode

# Production
yarn build              # Compile TypeScript
yarn start              # Start compiled app

# Quality
yarn lint               # Run ESLint
```

🖨 Additional Resources

- NestJS Documentation
- Prisma Documentation
- Grammy Documentation
- TypeScript Handbook
- API_DOCUMENTATION.md
- BOT_LOGIC.md
- DATABASE.md

Updated: 2025-12-04
Version: 1.0.0

TROUBLESHOOTING.md

🔧 Troubleshooting Guide

📑 Table of Contents

🐛 Common Issues

Issue: Port Already in Use

Error:

```
Error: listen EADDRINUSE: address already in use :::3000
```

Solution:

```bash
# Find process using port 3000
lsof -i :3000
# or
netstat -tulpn | grep 3000

# Kill the process
kill -9 <PID>

# Or use different port in .env
PORT=3001
```

Issue: Module Not Found

Error:

```
Error: Cannot find module '@nestjs/common'
```

Solution:

```bash
# Reinstall dependencies
cd nodejs_space
```

```
rm -rf node_modules
rm yarn.lock
yarn install
```

Issue: Prisma Client Out of Sync

Error:

```
The `prisma generate` command was called to generate Prisma Client.
However, your Prisma Client is already up to date!
```

Solution:

```
cd nodejs_space
yarn prisma generate --force
```

🗄 Database Problems

Issue: Cannot Connect to Database

Error:

```
Error: P1001: Can't reach database server at `localhost:5432`
```

Solution:

```
# Check PostgreSQL is running
pg_isready -h localhost -p 5432

# Or with Docker
docker ps | grep postgres

# Start PostgreSQL
# On macOS (Homebrew):
brew services start postgresql@15

# On Linux:
sudo systemctl start postgresql

# On Docker:
docker-compose up -d postgres
```

Issue: Authentication Failed

Error:

```
Error: password authentication failed for user "postgres"
```

Solution:

Check DATABASE_URL in `.env`:

```
DATABASE_URL=postgresql://correct_user:correct_password@localhost:543
```

Reset PostgreSQL password if needed:

```
sudo -u postgres psql
ALTER USER postgres PASSWORD 'new_password';
\q
```

Issue: Database Does Not Exist

Error:

```
Error: P1003: Database `telegram_shop` does not exist
```

Solution:

```
# Create database
sudo -u postgres psql
CREATE DATABASE telegram_shop;
GRANT ALL PRIVILEGES ON DATABASE telegram_shop TO postgres;
\q

# Apply schema
cd nodejs_space
yarn prisma db push
```

Issue: Migration Fails

Error:

```
Error: P3009: Failed to apply migration
```

Solution:

```
# Reset database (⚠ DATA LOSS!)
cd nodejs_space
yarn prisma migrate reset

# Or manually fix schema
yarn prisma db pull
# Edit prisma/schema.prisma
yarn prisma db push
```

🔌 API Issues

Issue: API Not Starting

Error:

```
Application failed to start
```

Solution:

```
# Check logs
cd nodejs_space
yarn start:dev

# Check for syntax errors
yarn build

# Verify environment variables
cat .env
```

Issue: 401 Unauthorized

Error:

```
{
  "statusCode": 401,
  "message": "Unauthorized"
}
```

Solution:

```
# Get fresh JWT token
curl -X POST http://localhost:3000/api/auth/token   -H "Content-
        Type: application/json"   -d
        '{"username":"admin","password":"admin123"}'

# Use token in requests
curl -H "Authorization: Bearer YOUR_TOKEN"
        http://localhost:3000/api/admin/products
```

Issue: 422 Validation Error

Error:

```
{
  "statusCode": 422,
  "message": "Validation failed",
  "errors": [
    {
      "field": "name",
      "constraints": {
        "minLength": "name must be longer than or equal to 3
        characters"
      }
    }
  ]
}
```

Solution:

Check request body matches DTO requirements:

```
// ProductDto requires:
{
  "name": "Min 3 chars",
```

```
  "description": "Min 10 chars",
  "category_id": 1,  // Must exist
  "is_active": true
}
```

Issue: CORS Error

Error:

```
Access to XMLHttpRequest has been blocked by CORS policy
```

Solution:

Add allowed origin to `.env`:

```
CORS_ORIGIN=http://localhost:3001,https://yourdomain.com
```

Or allow all (development only):

```
CORS_ORIGIN=*
```

🤖 Bot Problems

Issue: Bot Not Responding

Symptoms: Bot doesn't reply to messages

Solutions:

1. **Check bot is running:**

```
# Check Docker
docker-compose logs bot

# Check PM2
pm2 logs telegram-shop-bot

# Check process
ps aux | grep "node.*bot"
```

2. **Verify BOT_TOKEN:**

```
# Check token format
echo $BOT_TOKEN
# Should be: 1234567890:ABCdefGhIjKlmNoPqRsTuVwXyZ

# Test token with Telegram API
curl https://api.telegram.org/bot$BOT_TOKEN/getMe
```

3. **Check API connection:**

```
# From bot container
docker-compose exec bot ping api
```

```
# Test API endpoint
curl http://api:3000/api/products
```

4. **Restart bot:**

```
docker-compose restart bot
# or
pm2 restart telegram-shop-bot
```

Issue: Bot Crashes on Start

Error:

```
TypeError: Cannot read property 'id' of undefined
```

Solution:

```
# Check bot logs
docker-compose logs bot

# Common causes:
# 1. Invalid BOT_TOKEN
# 2. API not accessible
# 3. Missing environment variables

# Verify .env
cd bot
cat .env

# Should have:
# BOT_TOKEN=...
# API_BASE_URL=http://api:3000/api
# ADMIN_TELEGRAM_IDS=...
```

Issue: Conversation State Lost

Symptoms: Bot forgets conversation mid-flow

Solution:

Bot uses in-memory sessions by default. They reset on restart.

For persistent sessions, implement session storage:

```
// bot/src/index.ts
import { session } from 'grammy';

bot.use(session({
  storage: /* use file/redis storage */,
  initial(): SessionData {
    return { /* initial state */ };
  },
}));
```

Issue: Admin Not Receiving Notifications

Symptoms: Orders created but admin not notified

Solution:

```
# Check ADMIN_TELEGRAM_IDS
docker-compose exec bot env | grep ADMIN_TELEGRAM_IDS

# Should contain your Telegram ID
# Get your ID: https://t.me/userinfobot

# Update .env
ADMIN_TELEGRAM_IDS=123456789,987654321

# Restart bot
docker-compose restart bot
```

🐳 Docker Issues

Issue: Container Exits Immediately

Solution:

```
# Check logs for exit reason
docker-compose logs api

# Common causes:
# 1. Syntax error in code
# 2. Missing environment variable
# 3. Database not ready

# Try starting with verbose logs
docker-compose up api
```

Issue: Database Container Unhealthy

Error:

```
postgres unhealthy
```

Solution:

```
# Check health
docker-compose exec postgres pg_isready -U postgres

# Check logs
docker-compose logs postgres

# Remove and recreate
docker-compose down postgres
docker-compose up -d postgres
```

```
# Wait for healthy status
watch docker-compose ps
```

Issue: Volume Permission Denied

Error:

```
Error: EACCES: permission denied
```

Solution:

```
# Fix permissions
docker-compose down
sudo chown -R $USER:$USER .
docker-compose up -d
```

Issue: Network Error Between Containers

Error:

```
getaddrinfo ENOTFOUND api
```

Solution:

```
# Check network
docker network ls
docker network inspect telegram_shop_backend_default

# Recreate network
docker-compose down
docker-compose up -d
```

⚡ Performance Problems

Issue: Slow API Responses

Solution:

1. **Add database indexes:**

```
model products {
  // ...
  @@index([name])
  @@index([category_id, is_active])
}
```

2. **Enable query logging:**

```
// Log slow queries
const prisma = new PrismaClient({
  log: [
```

```
    { level: 'query', emit: 'event' },
  ],
});

prisma.$on('query', (e) => {
  if (e.duration > 1000) {
    console.log('Slow query:', e.query, `${e.duration}ms`);
  }
});
```

3. **Check database stats:**

```
docker-compose exec postgres psql -U postgres -d telegram_shop -c "
  SELECT query, calls, total_time, mean_time
  FROM pg_stat_statements
  ORDER BY mean_time DESC
  LIMIT 10;
"
```

Issue: High Memory Usage

Solution:

```
# Check usage
docker stats

# Limit memory in docker-compose.yml
services:
  api:
    deploy:
      resources:
        limits:
          memory: 512M

# Optimize Node.js
NODE_OPTIONS="--max-old-space-size=512" node dist/main.js
```

🔒 Security Issues

Issue: JWT Token Expired

Error:

```
{
  "statusCode": 401,
  "message": "Token expired"
}
```

Solution:

Get new token:

```
curl -X POST http://localhost:3000/api/auth/token   -H "Content-
        Type: application/json"   -d
        '{"username":"admin","password":"admin123"}'
```

Increase expiration in `.env`:

```
JWT_EXPIRES_IN=1h  # Instead of 30m
```

Issue: SQL Injection Attempt

Prisma automatically protects against SQL injection via parameterized queries.

Safe (using Prisma):

```
await prisma.products.findMany({
  where: { name: { contains: userInput } }
});
```

Unsafe (raw SQL):

```
// ⚠ DON'T DO THIS
await prisma.$queryRaw`SELECT * FROM products WHERE name LIKE
        '%${userInput}%'`;
```

📞 Getting Help

Check Logs

```
# Backend API
docker-compose logs -f api
# or
pm2 logs telegram-shop-api

# Telegram Bot
docker-compose logs -f bot
# or
pm2 logs telegram-shop-bot

# Database
docker-compose logs postgres
```

Enable Debug Mode

```
# API
NODE_ENV=development
LOG_LEVEL=debug

# Bot
LOG_LEVEL=debug
```

Health Checks

```
# API health
curl http://localhost:3000/api

# Database health
docker-compose exec postgres pg_isready

# Bot (check logs)
docker-compose logs --tail=10 bot
```

🖨 Additional Resources

- SETUP.md - Installation guide
- API_DOCUMENTATION.md - API reference
- BOT_LOGIC.md - Bot behavior
- DOCKER.md - Docker guide
- DATABASE.md - Database schema

Updated: 2025-12-04
Version: 1.0.0