План поэтапной миграции грузинской бухгалтерской системы

🎯 Общая стратегия миграции

Подход: Strangler Fig Pattern - постепенная замена компонентов без остановки работы системы

Продолжительность: 18-24 месяца

Бюджет: Распределен поэтапно для минимизации рисков

📋 Фаза 0: Подготовка и анализ (Месяцы 1-2)

Цели:

- Аудит текущего состояния
- Подготовка инфраструктуры
- Создание команды миграции

Задачи:

Неделя 1-2: Технический аудит

```bash
# Анализ кода
- Инвентаризация всех модулей и зависимостей
- Оценка тестового покрытия (текущий: ~10%, цель: 85%)
- Выявление критических узких мест производительности
- Анализ данных: объемы, структура, quality issues
```

Неделя 3-4: Создание MVP инфраструктуры

```yaml
```

```yaml
# docker-compose-migration.yml
version: '3.8'
services:
  # Текущая система (legacy)
  legacy-app:
    build: ./legacy
    ports:
      - "8080:8080"
    networks:
      - migration-network

  # Новая система (target)
  new-api-gateway:
    build: ./new-system/gateway
    ports:
      - "8000:8000"
    networks:
      - migration-network

  # Shared resources
  postgres-new:
    image: postgres:15
    environment:
      POSTGRES_DB: accounting_new
    networks:
      - migration-network

  redis:
    image: redis:7-alpine
    networks:
      - migration-network

networks:
  migration-network:
    driver: bridge
```

Неделя 5-6: Создание Data Pipeline

```python
```

```python
# migration/data_sync.py
from sqlalchemy import create_engine
import asyncpg
import pandas as pd

class DataSynchronizer:
    def __init__(self):
        self.legacy_engine = create_engine('postgresql://legacy_db')
        self.new_pool = None

    async def sync_accounts(self):
        """Синхронизация справочника счетов"""
        df = pd.read_sql("SELECT * FROM chart_of_accounts", self.legacy_engine)

        # Transform data
        df['id'] = df.apply(lambda x: uuid4(), axis=1)
        df['created_at'] = pd.Timestamp.now()

        # Load to new system
        async with self.new_pool.acquire() as conn:
            await conn.executemany(
                "INSERT INTO accounts (id, code, name, type) VALUES ($1, $2, $3, $4)",
                df[['id', 'code', 'name', 'account_type']].values.tolist()
            )
```

Неделя 7-8: CI/CD Pipeline

yaml

```yaml
# .github/workflows/migration.yml
name: Migration Pipeline
on:
  push:
    branches: [main, migration/*]

jobs:
  test-legacy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Test Legacy System
        run: |
          docker-compose -f legacy/docker-compose.test.yml up --abort-on-container-exit

  test-new-system:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Test New System
        run: |
          docker-compose -f new-system/docker-compose.test.yml up --abort-on-container-exit

  deploy-staging:
    needs: [test-legacy, test-new-system]
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to Staging
        run: |
          kubectl apply -f k8s/staging/
```

Результаты Фазы 0:

- ✅ Детальный план миграции с временными рамками
- ✅ Настроенная инфраструктура для параллельной работы систем
- ✅ Команда готова к миграции (3-5 разработчиков)
- ✅ Базовая система мониторинга и алертов

🔧 Фаза 1: Модернизация фундамента (Месяцы 3-6)

Цели:

- Замена устаревших компонентов
- Повышение безопасности
- Улучшение производительности

1.1 Модернизация базы данных (Месяц 3)

Неделя 1-2: Новая схема БД

sql

```sql
-- migrations/001_new_schema.sql
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pg_stat_statements";

-- Новая таблица счетов с UUID
CREATE TABLE accounts_v2 (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    code VARCHAR(20) NOT NULL,
    name VARCHAR(100) NOT NULL,
    account_type account_type_enum NOT NULL,
    parent_id UUID REFERENCES accounts_v2(id),
    is_active BOOLEAN DEFAULT TRUE,
    metadata JSONB DEFAULT '{}',
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMPTZ DEFAULT NOW(),
    version INTEGER DEFAULT 1,

    -- Индексы для производительности
    CONSTRAINT unique_code_per_company UNIQUE (code, company_id)
);

CREATE INDEX idx_accounts_v2_code ON accounts_v2(code);
CREATE INDEX idx_accounts_v2_type ON accounts_v2(account_type);
CREATE INDEX idx_accounts_v2_parent ON accounts_v2(parent_id);
```

Неделя 3-4: Партиционирование для больших таблиц

sql

```sql
-- Партиционирование журнала проводок по месяцам
CREATE TABLE journal_entries_v2 (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    transaction_id UUID NOT NULL,
    account_id UUID NOT NULL REFERENCES accounts_v2(id),
    debit DECIMAL(15,2) DEFAULT 0.00,
    credit DECIMAL(15,2) DEFAULT 0.00,
    currency_code CHAR(3) NOT NULL DEFAULT 'GEL',
    exchange_rate DECIMAL(15,5) DEFAULT 1.00000,
    description TEXT,
    transaction_date DATE NOT NULL,
    created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    CONSTRAINT check_debit_credit CHECK (
        (debit > 0 AND credit = 0) OR (credit > 0 AND debit = 0)
    )
) PARTITION BY RANGE (transaction_date);

-- Создание партиций на год вперед
CREATE TABLE journal_entries_202501 PARTITION OF journal_entries_v2
    FOR VALUES FROM ('2025-01-01') TO ('2025-02-01');
-- ... остальные месяцы
```

1.2 Новый API слой (Месяц 4)

FastAPI с современными паттернами

python

```python
# new_system/api/main.py
from fastapi import FastAPI, Depends, HTTPException, BackgroundTasks
from fastapi.security import HTTPBearer
from fastapi.middleware.cors import CORSMiddleware
from contextlib import asynccontextmanager
import structlog

logger = structlog.get_logger()

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Startup
    await init_database_pool()
    await init_kafka_producer()
    logger.info("Application started")

    yield

    # Shutdown
    await close_database_pool()
    await close_kafka_producer()
    logger.info("Application stopped")

app = FastAPI(
    title="Georgian Accounting System v2.0",
    description="Modern IFRS-compliant accounting system",
    version="2.0.0",
    lifespan=lifespan
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Configure properly in production
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# API Routes
from .routes import accounts, transactions, reports

app.include_router(accounts.router, prefix="/api/v1/accounts")
```

```
app.include_router(transactions.router, prefix="/api/v1/transactions")
app.include_router(reports.router, prefix="/api/v1/reports")
```

Современная архитектура с DI

```python
```

```python
# new_system/core/dependencies.py
from dependency_injector import containers, providers
from dependency_injector.wiring import Provide

class Container(containers.DeclarativeContainer):
    # Configuration
    config = providers.Configuration()

    # Database
    db_pool = providers.Singleton(
        create_async_pool,
        config.database.url
    )

    # Repositories
    account_repository = providers.Factory(
        AccountRepository,
        db_pool=db_pool
    )

    transaction_repository = providers.Factory(
        TransactionRepository,
        db_pool=db_pool
    )

    # Services
    accounting_service = providers.Factory(
        AccountingService,
        account_repo=account_repository,
        transaction_repo=transaction_repository
    )

# Dependency injection
async def get_accounting_service(
    service: AccountingService = Depends(Provide[Container.accounting_service])
) -> AccountingService:
    return service
```

1.3 Система безопасности (Месяц 5)

JWT с refresh tokens

```
python
```

```python
# new_system/auth/jwt_handler.py
from jose import JWTError, jwt
from datetime import datetime, timedelta
import secrets

class JWTHandler:
    def __init__(self, secret_key: str):
        self.secret_key = secret_key
        self.algorithm = "HS256"
        self.access_token_expire = timedelta(minutes=30)
        self.refresh_token_expire = timedelta(days=7)

    async def create_tokens(self, user_id: str, permissions: List[str]) -> TokenPair:
        access_payload = {
            "sub": user_id,
            "permissions": permissions,
            "type": "access",
            "exp": datetime.utcnow() + self.access_token_expire,
            "iat": datetime.utcnow(),
            "jti": secrets.token_hex(16)  # JWT ID для отзыва
        }

        refresh_payload = {
            "sub": user_id,
            "type": "refresh",
            "exp": datetime.utcnow() + self.refresh_token_expire,
            "iat": datetime.utcnow(),
            "jti": secrets.token_hex(16)
        }

        access_token = jwt.encode(access_payload, self.secret_key, self.algorithm)
        refresh_token = jwt.encode(refresh_payload, self.secret_key, self.algorithm)

        # Сохранить refresh token в Redis с TTL
        await self.redis.setex(f"refresh:{refresh_payload['jti']}",
                               int(self.refresh_token_expire.total_seconds()),
                               user_id)

        return TokenPair(
            access_token=access_token,
            refresh_token=refresh_token,
```

```
        expires_in=int(self.access_token_expire.total_seconds())
    )
```

Role-Based Access Control

python

```python
# new_system/auth/rbac.py
from enum import Enum
from dataclasses import dataclass
from typing import Set

class Permission(Enum):
    ACCOUNTS_READ = "accounts:read"
    ACCOUNTS_WRITE = "accounts:write"
    TRANSACTIONS_READ = "transactions:read"
    TRANSACTIONS_WRITE = "transactions:write"
    TRANSACTIONS_APPROVE = "transactions:approve"
    REPORTS_FINANCIAL = "reports:financial"
    REPORTS_TAX = "reports:tax"
    ADMIN_USERS = "admin:users"
    ADMIN_SYSTEM = "admin:system"

@dataclass
class Role:
    name: str
    permissions: Set[Permission]

class GeorgianAccountingRoles:
    ACCOUNTANT = Role("accountant", {
        Permission.ACCOUNTS_READ,
        Permission.TRANSACTIONS_READ,
        Permission.TRANSACTIONS_WRITE,
        Permission.REPORTS_FINANCIAL
    })

    CHIEF_ACCOUNTANT = Role("chief_accountant", {
        *ACCOUNTANT.permissions,
        Permission.TRANSACTIONS_APPROVE,
        Permission.REPORTS_TAX,
        Permission.ADMIN_USERS
    })

    TAX_SPECIALIST = Role("tax_specialist", {
        Permission.ACCOUNTS_READ,
        Permission.TRANSACTIONS_READ,
        Permission.REPORTS_TAX
    })

def require_permission(permission: Permission):
```

```
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, current_user = Depends(get_current_user), **kwargs):
            if permission not in current_user.permissions:
                raise HTTPException(403, "Insufficient permissions")
            return await func(*args, **kwargs, current_user=current_user)
        return wrapper
    return decorator
```

1.4 Мониторинг и логирование (Месяц 6)

Структурированное логирование

python

```python
# new_system/core/logging.py
import structlog
from pythonjsonlogger import jsonlogger

def setup_logging():
    structlog.configure(
        processors=[
            structlog.stdlib.filter_by_level,
            structlog.stdlib.add_logger_name,
            structlog.stdlib.add_log_level,
            structlog.stdlib.PositionalArgumentsFormatter(),
            structlog.processors.TimeStamper(fmt="iso"),
            structlog.processors.StackInfoRenderer(),
            structlog.processors.format_exc_info,
            structlog.processors.UnicodeDecoder(),
            structlog.processors.JSONRenderer()
        ],
        context_class=dict,
        logger_factory=structlog.stdlib.LoggerFactory(),
        wrapper_class=structlog.stdlib.BoundLogger,
        cache_logger_on_first_use=True,
    )

# Использование в коде
logger = structlog.get_logger()

async def create_transaction(transaction_data: TransactionCreate):
    logger.info(
        "Transaction creation started",
        transaction_id=transaction_data.id,
        user_id=current_user.id,
        amount=float(transaction_data.total_amount)
    )

    try:
        result = await service.create_transaction(transaction_data)
        logger.info(
            "Transaction created successfully",
            transaction_id=result.id,
            duration_ms=(time.time() - start_time) * 1000
        )
        return result
    except Exception as e:
```

```
        logger.error(
            "Transaction creation failed",
            error=str(e),
            transaction_data=transaction_data.dict()
        )
        raise
```

Prometheus метрики

```python
```

```python
# new_system/core/metrics.py
from prometheus_client import Counter, Histogram, Gauge
import time

# Business metrics
transaction_counter = Counter(
    'accounting_transactions_total',
    'Total number of accounting transactions',
    ['status', 'transaction_type']
)

transaction_amount_histogram = Histogram(
    'accounting_transaction_amount_gel',
    'Distribution of transaction amounts in GEL',
    buckets=[10, 50, 100, 500, 1000, 5000, 10000, 50000, float('inf')]
)

account_balance_gauge = Gauge(
    'accounting_account_balance_gel',
    'Current account balance in GEL',
    ['account_code', 'account_type']
)

# Technical metrics
request_duration = Histogram(
    'http_request_duration_seconds',
    'HTTP request duration',
    ['method', 'endpoint', 'status']
)

class MetricsMiddleware:
    async def __call__(self, request, call_next):
        start_time = time.time()
        response = await call_next(request)
        duration = time.time() - start_time

        request_duration.labels(
            method=request.method,
            endpoint=request.url.path,
            status=response.status_code
        ).observe(duration)
```

```
    return response
```

Результаты Фазы 1:

- ✅ Новая БД схема с UUID и партиционированием
- ✅ Modern FastAPI с async/await
- ✅ JWT authentication с RBAC
- ✅ Структурированное логирование и метрики
- ✅ 60% покрытие тестами новых компонентов

🧬 Фаза 2: Внедрение Event-Driven Architecture (Месяцы 7-10)

Цели:

- Реализация Event Sourcing для audit trail
- CQRS для разделения чтения/записи
- Интеграция с Kafka

2.1 Event Store и Event Sourcing (Месяц 7)

Event Store на PostgreSQL

```sql
```

```sql
-- events/001_event_store.sql
CREATE TABLE event_store (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    aggregate_id UUID NOT NULL,
    aggregate_type VARCHAR(100) NOT NULL,
    event_type VARCHAR(100) NOT NULL,
    event_data JSONB NOT NULL,
    event_metadata JSONB DEFAULT '{}',
    version INTEGER NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL DEFAULT NOW(),

    CONSTRAINT unique_version_per_aggregate UNIQUE (aggregate_id, version)
);

CREATE INDEX idx_event_store_aggregate ON event_store(aggregate_id);
CREATE INDEX idx_event_store_type ON event_store(event_type);
CREATE INDEX idx_event_store_timestamp ON event_store(timestamp);

-- Snapshots для производительности
CREATE TABLE aggregate_snapshots (
    aggregate_id UUID PRIMARY KEY,
    aggregate_type VARCHAR(100) NOT NULL,
    snapshot_data JSONB NOT NULL,
    version INTEGER NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

Базовые классы для Event Sourcing

```python
```

```python
# new_system/events/base.py
from dataclasses import dataclass
from typing import Any, List, Dict
from abc import ABC, abstractmethod
import uuid
from datetime import datetime

@dataclass(frozen=True)
class DomainEvent:
    """Базовый класс для доменных событий"""
    aggregate_id: uuid.UUID
    event_id: uuid.UUID
    event_type: str
    event_data: Dict[str, Any]
    version: int
    timestamp: datetime
    metadata: Dict[str, Any]

class Aggregate(ABC):
    """Базовый класс для агрегатов"""
    def __init__(self, aggregate_id: uuid.UUID):
        self.id = aggregate_id
        self.version = 0
        self.uncommitted_events: List[DomainEvent] = []

    def apply_event(self, event: DomainEvent):
        """Применить событие к агрегату"""
        self._apply_event(event)
        if event.version > self.version:
            self.version = event.version

    def raise_event(self, event_type: str, event_data: Dict[str, Any], metadata: Dict[str, Any] = None):
        """Поднять новое событие"""
        event = DomainEvent(
            aggregate_id=self.id,
            event_id=uuid.uuid4(),
            event_type=event_type,
            event_data=event_data,
            version=self.version + 1,
            timestamp=datetime.utcnow(),
            metadata=metadata or {}
        )
        self.uncommitted_events.append(event)
```

```
        self.apply_event(event)

    @abstractmethod
    def _apply_event(self, event: DomainEvent):
        """Применить событие к состоянию агрегата"""
        pass

    def mark_events_as_committed(self):
        """Пометить события как сохраненные"""
        self.uncommitted_events.clear()
```

Accounting Aggregate

python

```python
# new_system/domain/aggregates.py
from decimal import Decimal
from dataclasses import dataclass
from typing import List, Optional

@dataclass
class JournalEntryData:
    account_id: uuid.UUID
    debit: Decimal
    credit: Decimal
    description: str

class AccountingTransaction(Aggregate):
    """Агрегат бухгалтерской транзакции"""

    def __init__(self, aggregate_id: uuid.UUID):
        super().__init__(aggregate_id)
        self.transaction_date: Optional[datetime] = None
        self.description: str = ""
        self.entries: List[JournalEntryData] = []
        self.status: str = "draft"
        self.total_debit: Decimal = Decimal('0.00')
        self.total_credit: Decimal = Decimal('0.00')

    def create_transaction(self, transaction_date: datetime, description: str, entries: List[JournalEntryData]):
        """Создать новую транзакцию"""
        if self.status != "":
            raise ValueError("Transaction already exists")

        # Валидация двойной записи
        total_debit = sum(entry.debit for entry in entries)
        total_credit = sum(entry.credit for entry in entries)

        if total_debit != total_credit:
            raise ValueError(f"Unbalanced transaction: debit={total_debit}, credit={total_credit}")

        self.raise_event("TransactionCreated", {
            "transaction_date": transaction_date.isoformat(),
            "description": description,
            "entries": [
                {
                    "account_id": str(entry.account_id),
                    "debit": str(entry.debit),
```

```
                "credit": str(entry.credit),
                "description": entry.description
            }
            for entry in entries
        ],
        "total_amount": str(total_debit)
    })

def approve_transaction(self, approved_by: uuid.UUID):
    """Одобрить транзакцию"""
    if self.status != "draft":
        raise ValueError(f"Cannot approve transaction with status: {self.status}")

    self.raise_event("TransactionApproved", {
        "approved_by": str(approved_by),
        "approved_at": datetime.utcnow().isoformat()
    })

def post_transaction(self, posted_by: uuid.UUID):
    """Провести транзакцию"""
    if self.status != "approved":
        raise ValueError(f"Cannot post transaction with status: {self.status}")

    self.raise_event("TransactionPosted", {
        "posted_by": str(posted_by),
        "posted_at": datetime.utcnow().isoformat()
    })

def _apply_event(self, event: DomainEvent):
    """Применить событие к состоянию транзакции"""
    if event.event_type == "TransactionCreated":
        self.transaction_date = datetime.fromisoformat(event.event_data["transaction_date"])
        self.description = event.event_data["description"]
        self.entries = [
            JournalEntryData(
                account_id=uuid.UUID(entry["account_id"]),
                debit=Decimal(entry["debit"]),
                credit=Decimal(entry["credit"]),
                description=entry["description"]
            )
            for entry in event.event_data["entries"]
        ]
        self.total_debit = self.total_credit = Decimal(event.event_data["total_amount"])
        self.status = "draft"
```

```
        elif event.event_type == "TransactionApproved":
            self.status = "approved"

        elif event.event_type == "TransactionPosted":
            self.status = "posted"
```

2.2 CQRS Implementation (Месяц 8)

Command и Query разделение

python

```python
# new_system/cqrs/commands.py
from dataclasses import dataclass
from abc import ABC, abstractmethod

class Command(ABC):
    """Базовый класс для команд"""
    pass

class CommandHandler(ABC):
    @abstractmethod
    async def handle(self, command: Command) -> Any:
        pass

@dataclass
class CreateTransactionCommand(Command):
    transaction_date: datetime
    description: str
    entries: List[JournalEntryData]
    created_by: uuid.UUID

class CreateTransactionHandler(CommandHandler):
    def __init__(self, event_store: EventStore, event_bus: EventBus):
        self.event_store = event_store
        self.event_bus = event_bus

    async def handle(self, command: CreateTransactionCommand) -> uuid.UUID:
        # Создать агрегат
        transaction_id = uuid.uuid4()
        transaction = AccountingTransaction(transaction_id)

        # Выполнить бизнес-логику
        transaction.create_transaction(
            command.transaction_date,
            command.description,
            command.entries
        )

        # Сохранить события
        await self.event_store.save_events(
            transaction.id,
            transaction.uncommitted_events,
            expected_version=0
        )
```

```
    # Опубликовать события
    for event in transaction.uncommitted_events:
        await self.event_bus.publish(event)

    transaction.mark_events_as_committed()
    return transaction.id
```

Query side (Read Models)

python

```python
# new_system/cqrs/queries.py
@dataclass
class AccountBalanceQuery:
    account_id: uuid.UUID
    as_of_date: Optional[datetime] = None

class AccountBalanceQueryHandler:
    def __init__(self, read_db_pool):
        self.read_db = read_db_pool

    async def handle(self, query: AccountBalanceQuery) -> Decimal:
        async with self.read_db.acquire() as conn:
            if query.as_of_date:
                result = await conn.fetchval(
                    """
                    SELECT balance FROM account_balances_history
                    WHERE account_id = $1 AND date <= $2
                    ORDER BY date DESC LIMIT 1
                    """,
                    query.account_id,
                    query.as_of_date
                )
            else:
                result = await conn.fetchval(
                    "SELECT current_balance FROM account_balances WHERE account_id = $1",
                    query.account_id
                )

            return Decimal(str(result or '0.00'))

# Projection для поддержания read models
class AccountBalanceProjection:
    def __init__(self, read_db_pool):
        self.read_db = read_db_pool

    async def handle_transaction_posted(self, event: DomainEvent):
        """Обновить балансы счетов при проведении транзакции"""
        entries = event.event_data["entries"]

        async with self.read_db.acquire() as conn:
            async with conn.transaction():
                for entry_data in entries:
                    account_id = uuid.UUID(entry_data["account_id"])
```

```
        debit = Decimal(entry_data["debit"])
        credit = Decimal(entry_data["credit"])

        # Обновить текущий баланс
        await conn.execute(
            """
            INSERT INTO account_balances (account_id, current_balance, last_updated)
            VALUES ($1, $2, $3)
            ON CONFLICT (account_id) DO UPDATE SET
                current_balance = account_balances.current_balance + $2,
                last_updated = $3
            """,
            account_id,
            debit - credit,
            event.timestamp
        )

        # Добавить историческую запись
        await conn.execute(
            """
            INSERT INTO account_balance_history
            (account_id, date, balance_change, running_balance, transaction_id)
            VALUES ($1, $2, $3,
                (SELECT current_balance FROM account_balances WHERE account_id = $1),
                $4)
            """,
            account_id,
            event.timestamp.date(),
            debit - credit,
            event.aggregate_id
        )
```

2.3 Kafka Integration (Месяц 9)

Event Bus с Kafka

```
python
```

```python
# new_system/infrastructure/event_bus.py
from aiokafka import AIOKafkaProducer, AIOKafkaConsumer
import json
from typing import Dict, Callable

class KafkaEventBus:
    def __init__(self, bootstrap_servers: str):
        self.bootstrap_servers = bootstrap_servers
        self.producer: Optional[AIOKafkaProducer] = None
        self.consumer: Optional[AIOKafkaConsumer] = None
        self.handlers: Dict[str, List[Callable]] = {}

    async def start(self):
        self.producer = AIOKafkaProducer(
            bootstrap_servers=self.bootstrap_servers,
            value_serializer=lambda v: json.dumps(v, default=str).encode('utf-8')
        )
        await self.producer.start()

    async def publish(self, event: DomainEvent):
        """Опубликовать событие"""
        topic = f"accounting.{event.event_type.lower()}"

        event_payload = {
            "event_id": str(event.event_id),
            "aggregate_id": str(event.aggregate_id),
            "event_type": event.event_type,
            "event_data": event.event_data,
            "version": event.version,
            "timestamp": event.timestamp.isoformat(),
            "metadata": event.metadata
        }

        await self.producer.send(topic, event_payload)

        logger.info(
            "Event published",
            event_type=event.event_type,
            aggregate_id=str(event.aggregate_id),
            topic=topic
        )

    async def subscribe(self, event_type: str, handler: Callable):
```

```
    """Подписаться на тип события"""
    if event_type not in self.handlers:
        self.handlers[event_type] = []
    self.handlers[event_type].append(handler)

async def start_consuming(self):
    """Начать обработку событий"""
    topics = [f"accounting.{event_type.lower()}" for event_type in self.handlers.keys()]

    self.consumer
```