Общество с ограниченной ответственностью «УМАРТА»

Сервис для автоматического перевода документов с использованием LLM-моделей «УМАРТА.Переводчик»

Описание процессов, обеспечивающих поддержку жизненного цикла ПО

Листов 26

Москва, 2025 г.

АННОТАЦИЯ

Настоящий документ содержит описание процессов, обеспечивающих поддержку жизненного цикла программного обеспечения «Сервис для автоматического перевода документов с использованием LLM-моделей "УМАРТА.Переводчик"» (далее – Система).

В документе рассмотрены механизмы мониторинга, логирования, обработки ошибок, резервного копирования и восстановления, обновления системы, а также методы диагностики и устранения типовых проблем. Особое внимание уделено системе миграций базы данных, автоматическому восстановлению зависших заданий и процедурам поддержки пользователей. Включены инструкции по сбору метрик, формированию отчётности и эскалации инцидентов.

Документ предназначен для технических специалистов, DevOps-инженеров, системных администраторов и руководителей команд сопровождения.

СОДЕРЖАНИЕ

1 СИСТЕМА ЛОГИРОВАНИЯ И МОНИТОРИНГА ОШИБОК

1.1 Конфигурация логирования

Система использует встроенное логирование Python с централизованной конфигурацией:

```
# backend/app/main.py
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
)
logger = logging.getLogger(__name__)

# Специальная настройка для модуля email
logging.getLogger("app.email").setLevel(logging.DEBUG)
```

1.2 Middleware для логирования запросов

```
class LoggingMiddleware(BaseHTTPMiddleware):
    """Middleware для логирования запросов."""

    async def dispatch(self, request: Request, call_next):
        logger.info(f"Request: {request.method} {request.url}")
        try:
            response = await call_next(request)
            logger.info(f"Response: {response.status_code}")
            return response
        except Exception as e:
            logger.error(f"Error: {str(e)}")
            raise
```

1.3 Централизованная обработка исключений

Система имеет иерархию обработчиков исключений:

– Обработчик промокодов

```
@app.exception_handler(PromoCodeException)
async def promo_code_exception_handler(request: Request, exc:
PromoCodeException):
    logger.error(f"PromoCode error: {exc.detail}")
    return JSONResponse(
        status_code=exc.status_code,
        content={"detail": exc.detail},
    )
```

– Обработчик HTTP исключений

```
@app.exception_handler(HTTPException)
async def http_exception_handler(request: Request, exc:
HTTPException):
    logger.error(f"HTTP error: {exc.detail}")
    return JSONResponse(
        status_code=exc.status_code,
```

```
        content={"detail": exc.detail},
    )
```

– Обработчик общих исключений

```
@app.exception_handler(Exception)
async def general_exception_handler(request: Request, exc: Exception):
    logger.error(f"Unexpected error: {str(exc)}")
    return JSONResponse(
        status_code=500,
        content={"detail": "Внутренняя ошибка сервера"},
    )
```

1.4 Специализированные исключения LLM

```
# backend/app/llm/exceptions.py
class LLMError(Exception):
    """Базовый класс для всех ошибок, связанных с LLM"""

class LLMConnectionError(LLMError):
    """Ошибка подключения к LLM API"""

class LLMAuthenticationError(LLMError):
    """Ошибка аутентификации в LLM API"""

class LLMRateLimitError(LLMError):
    """Превышен лимит запросов к LLM API"""

class LLMResponseError(LLMError):
    """Ошибка в ответе от LLM API"""

class LLMTimeoutError(LLMError):
    """Превышено время ожидания ответа от LLM API"""
```

1.5 Обработка ошибок базы данных

```
def get_db() -> Generator[Session, None, None]:
    db = SessionLocal()
    try:
        yield db
    except OperationalError as e:
        logger.error(f"Ошибка подключения к базе данных: {e}")
        db.rollback()
        raise HTTPException(
            status_code=503,
            detail="Сервис временно недоступен. Попробуйте позже."
        )
    except SQLAlchemyError as e:
        logger.error(f"Ошибка базы данных: {e}")
        db.rollback()
        raise HTTPException(status_code=500, detail="Внутренняя ошибка
сервера")
    finally:
```

```
        db.close()
```

2 АВТОМАТИЧЕСКОЕ ВОССТАНОВЛЕНИЕ И МОНИТОРИНГ ЗАДАНИЙ

2.1 Планировщик задач

Система использует APScheduler для периодических задач:

```
# backend/app/main.py - в lifespan
scheduler.add_job(
    func=process_recurring_subscriptions,
    trigger="cron",
    hour=3,  # Запускать в 3 часа ночи по UTC
    minute=0,
    id="process_recurring_subscriptions",
    replace_existing=True,
)

scheduler.add_job(
    func=cleanup_expired_temp_files,
    trigger="interval",
    hours=6,  # Запускать каждые 6 часов
    id="cleanup_expired_temp_files",
    replace_existing=True,
)

scheduler.add_job(
    func=check_and_fix_stuck_jobs,
    trigger="interval",
    minutes=5,  # Запускать каждые 5 минут
    id="check_stuck_jobs",
    replace_existing=True,
)
```

2.2 Восстановление зависших заданий

```
async def check_and_fix_stuck_jobs():
    """Периодически проверяет и исправляет проблемы с заданиями."""
    logger.info("Запуск проверки застрявших заданий...")
    db = SessionLocal()

    try:
        current_time = datetime.now(timezone.utc)
        fixed_count = 0

        # 1. Проверяем файлы в processing с failed заданиями
        problematic = (
            db.query(File, TranslationJob)
            .join(TranslationJob)
            .filter(
                File.status == FileStatus.processing,
```

```
                TranslationJob.status == JobStatus.failed,
            )
            .all()
        )

        for file, job in problematic:
            logger.warning(f"Найден файл {file.id} в processing с
failed заданием {job.id}")
            file.status = FileStatus.error
            file.error_message = job.error_message or "Translation
failed"
            fixed_count += 1

        # 2. Проверяем застрявшие задания (processing > 10 минут)
        stuck_time = current_time - timedelta(minutes=10)
        stuck_jobs = (
            db.query(TranslationJob)
            .filter(
                TranslationJob.status == JobStatus.processing,
                TranslationJob.started_at < stuck_time,
            )
            .all()
        )

        queue_manager = QueueManager(db)
        for job in stuck_jobs:
            if job.retry_count < 3:
                job.retry_count += 1
                job.status = JobStatus.pending
                queue_manager.reset_job(job.id)
                logger.info(f"Job {job.id} reset to queue (attempt
{job.retry_count})")
                fixed_count += 1
            else:
                job.status = JobStatus.failed
                job.error_message = f"Job stuck after
{job.retry_count} retries"
                if job.file:
                    job.file.status = FileStatus.error
                    job.file.error_message = "Translation failed after
multiple attempts"
                fixed_count += 1

        if fixed_count > 0:
            db.commit()
            logger.info(f"Исправлено {fixed_count} проблем с
заданиями")
        else:
            logger.info("Проблем с заданиями не обнаружено")

    except Exception as e:
```

```
        logger.error(f"Ошибка при проверке застрявших заданий: {e}",
exc_info=True)
        db.rollback()
    finally:
        db.close()
```

2.3 Очистка временных файлов

```
async def cleanup_expired_temp_files():
    """Удаляет временные файлы, которые не были связаны с
пользователем в течение 24 часов."""
    db = SessionLocal()
    try:
        expiration_time = datetime.now(timezone.utc) -
timedelta(hours=24)
        expired_files = (
            db.query(File)
            .filter(
                File.is_temporary == True,
                File.created_at < expiration_time,
                File.user_id == None,
            )
            .all()
        )

        for file in expired_files:
            logger.info(f"Cleaning up expired temporary file:
{file.original_name} (ID: {file.id})")
            try:
                if os.path.exists(file.file_path):
                    os.remove(file.file_path)
                    logger.info(f"Deleted physical file:
{file.file_path}")
                db.delete(file)
            except OSError as e:
                logger.error(f"Error deleting physical file
{file.file_path}: {e}")

        db.commit()
        logger.info(f"Cleaned up {len(expired_files)} expired
temporary file(s).")

    except Exception as e:
        logger.error(f"Error in cleanup_expired_temp_files task: {e}",
exc_info=True)
        db.rollback()
    finally:
        db.close()
```

2.4 Система повторных попыток

```
# В воркере обработки очереди
```

```
try:
    processor = JobProcessor(db=db, user=user)
    await processor.process_job(job.id)
    logger.info(f"Воркер {worker_id} завершил задание {job.id}")
    consecutive_errors = 0
except Exception as e:
    logger.error(f"Ошибка обработки задания {job.id} воркером
{worker_id}: {e}", exc_info=True)

    # Проверяем количество попыток задания
    if job.retry_count < 3:
        # Возвращаем задание в очередь для повторной попытки
        job.retry_count += 1
        job.status = JobStatus.pending
        job.error_message = f"Retry {job.retry_count}: {str(e)}"
        db.commit()
        queue_manager.add_job(job.id, priority=100 + job.retry_count *
10)
        logger.info(f"Задание {job.id} возвращено в очередь (попытка
{job.retry_count})")
    else:
        # Превышен лимит попыток
        queue_manager.fail_job(job.id, f"Failed after
{job.retry_count} attempts: {str(e)}")
        if job.file:
            job.file.status = FileStatus.error
            job.file.error_message = f"Translation failed: {str(e)}"
            db.commit()
```

3 СИСТЕМА МИГРАЦИЙ БАЗЫ ДАННЫХ

3.1 Конфигурация Alembic

```
# backend/migrations/env.py
from app.database import Base
from app.config import settings

# Импортируем все модели для автогенерации миграций
from app.auth.models import User, UserRole, VerificationToken,
PasswordResetToken
from app.subscriptions.models import (
    SubscriptionPlan, UserSubscription, UserUsageLimit,
    PromoCode, UserPromoCodeUsage,
)
from app.files.models import File
from app.payments.models import Payment
from app.system.models import SystemSetting
from app.translation.models import (
    TranslationJob, TranslationBatch, TranslationBlock,
TranslationQueue,
)

target_metadata = Base.metadata

def get_url():
    return settings.DATABASE_URL
```

3.2 Пример миграции

```
# backend/migrations/versions/20241227_add_file_size_limits.py
def upgrade() -> None:
    op.add_column(
        "subscription_plans",
        sa.Column("max_file_size_mb", sa.Integer(), nullable=True)
    )
    op.add_column(
        "subscription_plans",
        sa.Column("max_total_upload_size_mb", sa.Integer(),
nullable=True),
    )

def downgrade() -> None:
    op.drop_column("subscription_plans", "max_total_upload_size_mb")
    op.drop_column("subscription_plans", "max_file_size_mb")
```

3.3 Команды миграций

```
# Создание новой миграции
docker exec backend alembic revision --autogenerate -m "Описание
изменений"
```

```
# Применение миграций
docker exec backend alembic upgrade head

# Откат миграции
docker exec backend alembic downgrade -1

# Просмотр истории миграций
docker exec backend alembic history

# Просмотр текущей версии
docker exec backend alembic current
```

4 РЕЗЕРВНОЕ КОПИРОВАНИЕ И ВОССТАНОВЛЕНИЕ

4.1 Резервное копирование базы данных

```
#!/bin/bash
# Скрипт для создания резервной копии БД

DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_DIR="/backups"
DB_NAME="umarta_translator"

# Создание дампа базы данных
docker exec postgres_db pg_dump -U user -d $DB_NAME >
"$BACKUP_DIR/db_backup_$DATE.sql"

# Сжатие бэкапа
gzip "$BACKUP_DIR/db_backup_$DATE.sql"

# Удаление старых бэкапов (старше 30 дней)
find $BACKUP_DIR -name "db_backup_*.sql.gz" -mtime +30 -delete

echo "Backup completed: db_backup_$DATE.sql.gz"
```

4.2 Резервное копирование файлов

```
#!/bin/bash
# Скрипт для резервного копирования пользовательских файлов

DATE=$(date +%Y%m%d_%H%M%S)
BACKUP_DIR="/backups"
FILES_DIR="./backend/files"

# Создание архива файлов
tar -czf "$BACKUP_DIR/files_backup_$DATE.tar.gz" -C "$FILES_DIR" .

# Удаление старых архивов файлов (старше 7 дней)
find $BACKUP_DIR -name "files_backup_*.tar.gz" -mtime +7 -delete

echo "Files backup completed: files_backup_$DATE.tar.gz"
```

4.3 Восстановление системы

```
#!/bin/bash
# Скрипт восстановления системы

BACKUP_FILE=$1

if [ -z "$BACKUP_FILE" ]; then
    echo "Usage: $0 <backup_file.sql.gz>"
    exit 1
fi
```

```
# Остановка системы
docker-compose down

# Восстановление базы данных
gunzip -c "$BACKUP_FILE" | docker exec -i postgres_db psql -U user -d
umarta_translator

# Запуск системы
docker-compose up -d

echo "System restored from $BACKUP_FILE"
```

5 МОНИТОРИНГ СИСТЕМЫ

5.1 Проверка состояния API

```
# Проверка здоровья API
curl -X GET "http://localhost/api/health"

# Ожидаемый ответ:
# {"status": "healthy"}
```

5.2 Мониторинг логов

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

# Просмотр логов конкретного сервиса
docker-compose logs -f backend
docker-compose logs -f nginx
docker-compose logs -f db

# Фильтрация логов по уровню
docker-compose logs backend | grep ERROR
docker-compose logs backend | grep WARNING
```

5.3 Мониторинг базы данных

```
-- Подключение к PostgreSQL
-- docker exec -it postgres_db psql -U user -d umarta_translator

-- Основная статистика
SELECT COUNT(*) as total_users FROM users;
SELECT COUNT(*) as total_files FROM files;
SELECT status, COUNT(*) as count FROM files GROUP BY status;
SELECT COUNT(*) as active_jobs FROM translation_jobs WHERE status IN
('pending', 'processing');

-- Проблемные задания
SELECT id, status, retry_count, error_message, created_at
FROM translation_jobs
WHERE status = 'failed'
ORDER BY created_at DESC
LIMIT 10;

-- Зависшие файлы
SELECT id, original_name, status, created_at, error_message
FROM files
WHERE status = 'processing'
AND created_at < NOW() - INTERVAL '1 hour';

-- Статистика использования LLM провайдеров
SELECT llm_provider, COUNT(*) as usage_count
```

```
FROM files
WHERE llm_provider IS NOT NULL
GROUP BY llm_provider;
```

5.4 Мониторинг очереди переводов

```
-- Статистика очереди
SELECT status, COUNT(*) as count
FROM translation_queue
GROUP BY status;

-- Задания с высоким приоритетом
SELECT tq.*, tj.file_id, f.original_name
FROM translation_queue tq
JOIN translation_jobs tj ON tq.job_id = tj.id
JOIN files f ON tj.file_id = f.id
WHERE tq.priority < 50
ORDER BY tq.priority;
```

6 ОБНОВЛЕНИЕ СИСТЕМЫ

6.1 Процедура обновления

```
#!/bin/bash
# Скрипт обновления системы

echo "Starting system update..."

# 1. Создание резервной копии
./backup_system.sh

# 2. Остановка сервисов
docker-compose down

# 3. Обновление кода
git fetch origin
git checkout main
git pull origin main

# 4. Пересборка контейнеров
docker-compose build

# 5. Применение миграций
docker-compose up -d db
sleep 10
docker-compose run --rm backend alembic upgrade head

# 6. Запуск обновленной системы
docker-compose up -d

# 7. Проверка здоровья системы
sleep 30
curl -f http://localhost/api/health || {
    echo "Health check failed! Rolling back..."
    docker-compose down
    # Здесь можно добавить логику отката
    exit 1
}

echo "System update completed successfully"
```

6.2 Откат изменений

```
#!/bin/bash
# Скрипт отката системы

PREVIOUS_COMMIT=$1

if [ -z "$PREVIOUS_COMMIT" ]; then
```

```
    echo "Usage: $0 <previous_commit_hash>"
    exit 1
fi

# Остановка системы
docker-compose down

# Откат кода
git checkout $PREVIOUS_COMMIT

# Пересборка контейнеров
docker-compose build

# Запуск системы
docker-compose up -d

echo "System rolled back to commit $PREVIOUS_COMMIT"
```

7 УСТРАНЕНИЕ ТИПОВЫХ ПРОБЛЕМ

7.1 Проблемы с переводом

```
# Диагностика проблем с LLM провайдерами
async def diagnose_llm_issues():
    """Проверяет доступность всех LLM провайдеров"""
    providers = ["deepseek", "openai", "lmstudio"]
    results = {}

    for provider_name in providers:
        try:
            provider = get_llm_provider(provider_name)
            test_result = await provider.translate_text(
                "Hello", "ru", "general"
            )
            results[provider_name] = "OK"
        except Exception as e:
            results[provider_name] = f"Error: {str(e)}"

    return results
```

7.2 Проблемы с очередью

```
# Сброс зависшей очереди
def reset_stuck_queue():
    """Сбрасывает все зависшие задания в очереди"""
    db = SessionLocal()
    try:
        queue_manager = QueueManager(db)
        reset_count = queue_manager.recover_stuck_jobs(hours_ago=1)
        logger.info(f"Reset {reset_count} stuck jobs")
        return reset_count
    finally:
        db.close()
```

7.3 Проблемы с файлами

```
# Очистка поврежденных файлов
#!/bin/bash

echo "Checking for corrupted files..."

# Проверка файлов на диске
find ./backend/files -type f -name "*.docx" -exec file {} \; | grep -v
"Microsoft Word"

# Проверка несоответствия БД и файловой системы
docker exec backend python -c "
from app.database import SessionLocal
from app.files.models import File
```

```
import os

db = SessionLocal()
try:
    files = db.query(File).all()
    for file in files:
        if not os.path.exists(file.file_path):
            print(f'Missing file: {file.id} - {file.file_path}')
finally:
    db.close()
"
```

8 ТЕХНИЧЕСКАЯ ПОДДЕРЖКА ПОЛЬЗОВАТЕЛЕЙ

8.1 Диагностика проблем пользователя

```
-- Информация о пользователе
SELECT u.*, us.status as subscription_status, sp.name as plan_name
FROM users u
LEFT JOIN user_subscriptions us ON u.id = us.user_id AND us.status =
'active'
LEFT JOIN subscription_plans sp ON us.plan_id = sp.id
WHERE u.email = 'user@example.com';

-- История файлов пользователя
SELECT id, original_name, status, error_message, created_at
FROM files
WHERE user_id = (SELECT id FROM users WHERE email =
'user@example.com')
ORDER BY created_at DESC
LIMIT 20;

-- Активные задания пользователя
SELECT tj.*, f.original_name
FROM translation_jobs tj
JOIN files f ON tj.file_id = f.id
WHERE f.user_id = (SELECT id FROM users WHERE email =
'user@example.com')
AND tj.status IN ('pending', 'processing')
ORDER BY tj.created_at DESC;
```

8.2 Восстановление файлов пользователя

```
# Повторная попытка обработки файла
async def retry_user_file(file_id: int):
    """Повторная попытка обработки файла пользователя"""
    db = SessionLocal()
    try:
        file = db.query(File).filter(File.id == file_id).first()
        if not file:
            return {"error": "File not found"}

        # Сброс статуса файла
        file.status = FileStatus.pending
        file.error_message = None

        # Создание нового задания
        from app.translation.queue_manager import QueueManager
        queue_manager = QueueManager(db)

        # Добавляем в очередь с высоким приоритетом
        queue_manager.add_job(file_id, priority=10)
```

```
        db.commit()
        return {"status": "File queued for retry"}

    except Exception as e:
        db.rollback()
        return {"error": str(e)}
    finally:
        db.close()
```

9 МЕТРИКИ И ОТЧЕТНОСТЬ

9.1 Ежедневные метрики

```
-- Статистика за сегодня
SELECT
    COUNT(CASE WHEN status = 'ready' THEN 1 END) as completed_files,
    COUNT(CASE WHEN status = 'error' THEN 1 END) as failed_files,
    COUNT(CASE WHEN status = 'processing' THEN 1 END) as
processing_files,
    COUNT(CASE WHEN status = 'pending' THEN 1 END) as pending_files
FROM files
WHERE DATE(created_at) = CURRENT_DATE;

-- Новые пользователи
SELECT COUNT(*) as new_users
FROM users
WHERE DATE(created_at) = CURRENT_DATE;

-- Использование LLM провайдеров
SELECT llm_provider, COUNT(*) as usage_count
FROM files
WHERE DATE(created_at) = CURRENT_DATE
AND llm_provider IS NOT NULL
GROUP BY llm_provider;
```

9.2 Еженедельные отчеты

```
-- Статистика за неделю
SELECT
    DATE(created_at) as date,
    COUNT(*) as total_files,
    COUNT(CASE WHEN status = 'ready' THEN 1 END) as success_rate,
    AVG(CASE WHEN translated_at IS NOT NULL
        THEN EXTRACT(EPOCH FROM (translated_at - created_at))/60 END)
as avg_processing_time_minutes
FROM files
WHERE created_at >= CURRENT_DATE - INTERVAL '7 days'
GROUP BY DATE(created_at)
ORDER BY date;
```

10 КОНТАКТЫ И ЭСКАЛАЦИЯ

10.1 Уровни поддержки

Поддержка системы организована по трёхуровневой модели:

1. **Уровень 1** - Автоматическое восстановление:
   - Планировщик задач запускается каждые 5 минут для проверки состояния заданий;
   - Автоматически перезапускаются зависшие задания;
   - Выполняется очистка временных файлов.
2. **Уровень 2** - Мониторинг администратора:
   - Анализ логов и мониторинг состояния Системы;
   - Ручное восстановление проблемных заданий;
   - Диагностика проблем пользователей.
3. **Уровень 3** – Поддержка со стороны разработчиков:
   - Исправление ошибок в коде;
   - Обновления системы;
   - Реагирование на инциденты, связанные с безопасностью и отказами критических компонентов.

10.2 Процедуры эскалации

```
# Критическая ошибка - немедленная эскалация
if grep -q "CRITICAL" /var/log/umarta/app.log; then
    echo "CRITICAL ERROR DETECTED" | mail -s "UMARTA Critical Alert"
admin@company.com
fi

# Высокая нагрузка - предупреждение
if [ $(docker stats --no-stream --format "{{.CPUPerc}}" backend | sed
's/%//') -gt 80 ]; then
    echo "High CPU usage detected" | mail -s "UMARTA Performance
Alert" admin@company.com
fi
```

11 ЗАКЛЮЧЕНИЕ

Данная система поддержки жизненного цикла обеспечивает:

– Автоматическое восстановление после сбоев;
– Централизованное логирование всех операций;
– Регулярное резервное копирование данных;
– Мониторинг производительности и здоровья системы;
– Структурированный процесс обновлений;
– Многоуровневую поддержку пользователей.

Все процессы основаны на реальном коде системы и готовы к использованию в продакшене.