RUDN university

НАДЕЖНОСТЬ **ПО**

ЛОГИРОВАНИЕ И МОНИТОРИНГ

ПОДГОТОВКА ОКРУЖЕНИЯ

```
4   LOG_DIR=logs # Каталог где будут лежать логи
5   LOG_FILE=bot.log # Имя основного файла лога
6   LOG_ENCODING=utf-8 # Кодировка файла логов
7   LOG_MAX_BYTES=5000000 # Максимальный размер одного файла лога в байтах (≈ 5 МБ) до ротации
8   LOG_BACKUP_COUNT=5 # Сколько старых файлов логов хранить (bot.log.1 ... bot.log.5)
9   LOG_LEVEL=INFO # Минимальный уровень логирования
```

В файл **.env** добавляем
параметры для логирования

СОЗДАНИЕ ТАБЛИЦЫ ДЛЯ ХРАНЕНИЯ ВЫЗОВОВ

```
-- Журнал вызовов внешних сервисов (OpenRouter и т.п.)
CREATE TABLE IF NOT EXISTS service_call_log (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  created_at TEXT NOT NULL,
  service TEXT NOT NULL,
  request TEXT NOT NULL,
  response TEXT,
  status_code INTEGER,
  duration_ms INTEGER,
  error TEXT
);
```

В файл **db.py** добавляем запрос создания Журнала вызовов внешних сервисов

created_at – время создания записи

service – имя сервиса, например «Openrouter»

request – строка запроса

response – строка ответа

status_code – HTTP статус (если применимо)

duration_ms – длительность вызова в миллисекундах

error – текст ошибки

СОЗДАНИЕ ТАБЛИЦЫ ДЛЯ ХРАНЕНИЯ ОШИБОК

```sql
-- Журнал ошибок
CREATE TABLE IF NOT EXISTS error_log (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        created_at TEXT NOT NULL,
        level TEXT NOT NULL,
        logger TEXT NOT NULL,
        message TEXT NOT NULL,
        user_id INTEGER,
        command TEXT,
        details TEXT
    );
```

В файл **db.py** добавляем запрос создания Журнала ошибок

created_at – время создания записи

level – уровень ошибки (ERROR, WARNING и т.п.)

logger – имя модуля, где произошла ошибка

message – текстовое описание ошибки

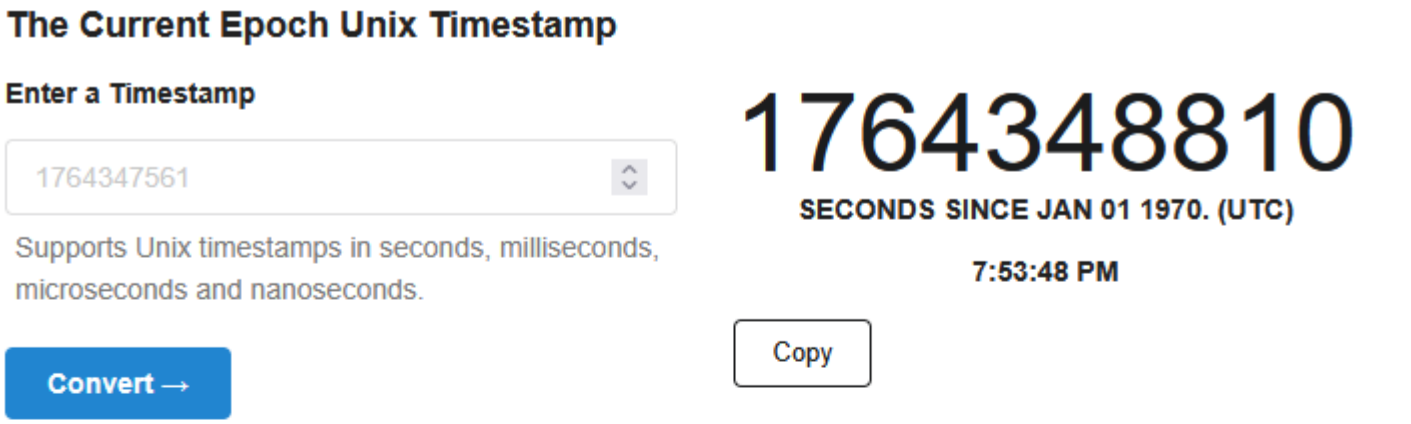
user_id – ID пользователя

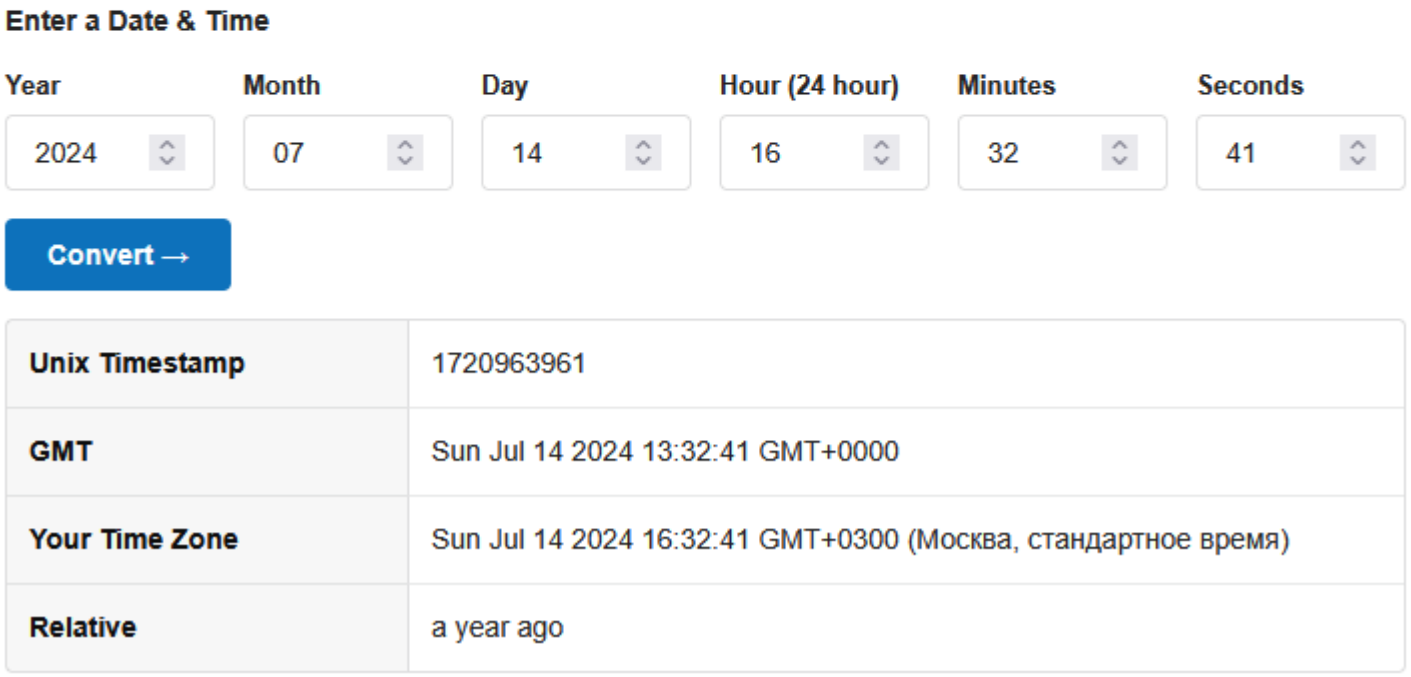
command – команда, во время которой произошла ошибка

details – дополнительные детали: стек-трейс, контекст, параметры и т.п.

TIMESTAMP

https://www.unixtimestamp.com





Unix Timestamp	1720963961
GMT	Sun Jul 14 2024 13:32:41 GMT+0000
Your Time Zone	Sun Jul 14 2024 16:32:41 GMT+0300 (Москва, стандартное время)
Relative	a year ago

Timestamp – это метка времени, количество секунд, прошедшее с 00:00:00 UTC 1 января, 1970 года

МОДУЛЬ **DATETIME**

Метод **strptime()**

Принимает в качестве одного из аргументов строку и создает на основе её объект типа datetime

```
from datetime import datetime as dt

first_strdate='10.05.2025'
second_strdate='26-June-2005'
third_strdate='5 Jan, 11'

first_date=dt.strptime(first_strdate, '%d.%m.%Y')
second_date=dt.strptime(second_strdate, '%d-%B-%Y')
third_date=dt.strptime(third_strdate, '%d %b, %y')

print(first_strdate, '->', first_date)
print(second_strdate, '->', second_date)
print(third_strdate, '->', third_date)
```

Вывод:

```
10.05.2025 -> 2025-05-10 00:00:00
26-June-2005 -> 2005-06-26 00:00:00
5 Jan, 11 -> 2011-01-05 00:00:00
```

Метод **strftime()**

Преобразует объект типа datetime в строку. Метод имеет следующий синтаксис

```
from datetime import datetime as dt

time = dt.now()

day_of_the_month = time.strftime("%d")
day_of_the_week = time.strftime(("%A"))
month = time.strftime("%B")
year = time.strftime("%Y")

format_of_time = time.strftime("%H:%M")
print('Today is', day_of_the_week+'.', 'It is',
day_of_the_month, 'day of the', month, year)
print('Current time', format_of_time)
```

Вывод:

```
Today is Thursday. It is 10 day of the November
2022
Current time 15:40
```

Одна из проблем, связанных с любыми датами и временем — это его формат. В Америке распространен формат MM-DD-YYYY, в России более привычным является формат DD-MM-YYYY.

Помимо них существуют иные записи даты и времени. В некоторых случаях возникает необходимость считывать и конвертировать дату. Чтобы сделать этот процесс более комфортным,в Python существует два метода для преобразования строк в формат datetime и обратно — **strptime()** и **strftime()**.

СОЗДАНИЕ НАСТРОЕК ЛОГИРОВАНИЯ

```python
"""
Настройка логирования.

Логи пишем:
- в консоль (StreamHandler)
- в файл в каталоге LOG_DIR (RotatingFileHandler с ротацией)

Пример формата строки лога:
2025-11-23 18:19:31.834 [MainThread] INFO main - Сообщение
"""

import logging
import os
import time
from logging.handlers import RotatingFileHandler

from dotenv import load_dotenv

load_dotenv()


class DotTimeFormatter(logging.Formatter):  2 usages
    """
    Кастомный форматтер времени.

    Формат: YYYY-MM-DD HH:MM:SS.mmm
    """

    def formatTime(self, record, datefmt=None):
        t = time.localtime(record.created)
        datetime_str = time.strftime( format: "%Y-%m-%d %H:%M:%S", t)
        return f"{datetime_str}.{int(record.msecs):03d}"
```

1. Создаем файл **logging_config.py**.
2. Добавляем класс DotTimeFormatter.

СОЗДАНИЕ НАСТРОЕК ЛОГИРОВАНИЯ

```python
def setup_logging() -> None:  2 usages
    """
    Инициализируем корневой логгер

    - создаём каталог LOG_DIR (если нет)
    - настраиваем консольный и файловый хендлеры
    - вешаем их на root-логгер
    """

    log_dir = os.getenv("LOG_DIR", "logs")
    log_file_name = os.getenv("LOG_FILE", "bot.log")
    log_encoding = os.getenv("LOG_ENCODING", "utf-8")
    log_max_bytes = int(os.getenv("LOG_MAX_BYTES", "5000000"))
    log_backup_count = int(os.getenv("LOG_BACKUP_COUNT", "5"))
    log_level_name = os.getenv("LOG_LEVEL", "INFO").upper()

    # Уровень логирования (INFO / DEBUG / WARNING ...)
    log_level = getattr(logging, log_level_name, logging.INFO)

    # Убедимся, что каталог для логов существует
    os.makedirs(log_dir, exist_ok=True)

    # Полный путь к файлу логов
    log_path = os.path.join(log_dir, log_file_name)

    fmt = "%(asctime)s [%(threadName)s] %(levelname)s %(name)s - %(message)s"

    console = logging.StreamHandler()
    console.setLevel(log_level)
    console.setFormatter(DotTimeFormatter(fmt))

    file_handler = RotatingFileHandler(
        log_path,
        maxBytes=log_max_bytes,
        backupCount=log_backup_count,
        encoding=log_encoding,
    )
    file_handler.setLevel(log_level)
    file_handler.setFormatter(DotTimeFormatter(fmt))

    # Чистим старые хендлеры root-логгера, если они были
    logging.root.handlers.clear()

    logging.basicConfig(
        level=log_level,
        handlers=[console, file_handler],
    )
```

3. Добавляем метод setup_logging()

ДОБАВЛЕНИЕ ЛОГИРОВАНИЯ В МЕТОДЫ

```python
from logging_config import setup_logging

# Настраиваем логирование до того, как делаем что-либо еще
setup_logging()
log = logging.getLogger(__name__)

log.info("Старт приложения (инициализация бота)")
```

```python
@bot.message_handler(commands=["start", "help"])
def cmd_start(message: types.Message) -> None:
    """
    Поприветствовать пользователя и кратко описать команды.
    """
    log.debug("Запущена команда /start")
    text = (
        "Привет! Это заметочник на SQLite.\n\n"
        "Команды:\n"
        "  /note_add <текст>\n"
        "  /note_list [N]\n"
        "  /note_find <подстрока>\n"
        "  /note_edit <id> <текст>\n"
        "  /note_del <id>\n"
        "  /note_count\n"
        "  /note_export\n"
        "  /note_stats [days]\n"
        "  /models \n"
        "  /model <id>\n"
        "  /ask <вопрос>\n"
        "  /ask_random <вопрос>\n"
        "  /characters \n"
        "  /character <id>\n"
        "  /whoami \n"
    )
    log.debug(f"Команда start вернула текст:\n{text}")
    bot.reply_to(message, text)
```

Добавляем логи в файл main.py для хендлера /start и попробуем выполнить

ДОБАВЛЕНИЕ ЛОГИРОВАНИЯ В МЕТОДЫ

```python
from logging_config import setup_logging

# Настраиваем логирование до того, как делаем что-либо еще
setup_logging()
log = logging.getLogger(__name__)

log.info("Старт приложения (инициализация бота)")
```

```python
@bot.message_handler(commands=["start", "help"])
def cmd_start(message: types.Message) -> None:
    """
    Поприветствовать пользователя и кратко описать команды.
    """
    log.debug("Запущена команда /start")
    text = (
        "Привет! Это заметочник на SQLite.\n\n"
        "Команды:\n"
        "  /note_add <текст>\n"
        "  /note_list [N]\n"
        "  /note_find <подстрока>\n"
        "  /note_edit <id> <текст>\n"
        "  /note_del <id>\n"
        "  /note_count\n"
        "  /note_export\n"
        "  /note_stats [days]\n"
        "  /models \n"
        "  /model <id>\n"
        "  /ask <вопрос>\n"
        "  /ask_random <вопрос>\n"
        "  /characters \n"
        "  /character <id>\n"
        "  /whoami \n"
    )
    log.debug(f"Команда start вернула текст:\n{text}")
    bot.reply_to(message, text)
```

Добавляем логи в файл main.py для хендлера /start и попробуем выполнить

> Посмотрим на содержимое файла /logs/bot.log или в консоль.
>
> Почему логи не отображаются?

СОЗДАНИЕ НАСТРОЕК МОНИТОРИНГА

```python
"""
Реестр метрик для бота.

Метрики хранятся в памяти процесса, без Prometheus и внешних систем.
Используются для команды /stats и простой диагностики.
"""

import threading
import time
import functools
import logging
from dataclasses import dataclass, asdict
from typing import Dict, Any, Callable, TypeVar

T = TypeVar("T")

class Counter:  3 usages
    """
    Простой счетчик.

    Пример использования:

        total = metric.counter("commands_total")
        total.inc()
        total.inc(5)

    Значения не сбрасываются автоматически при рестарте процесса.
    """

    def __init__(self, name: str) -> None:
        self.name = name
        self.value = 0

    def inc(self, amount: int = 1) -> None:  19 usages (19 dynamic)
        # Увеличить счетчик на amount (по умолчанию на 1)
        if amount < 0:
            raise ValueError("Ошибка при увеличении счетчика метрик: количество должно быть >= 0")
        self.value += amount

    def get(self) -> int:  1 usage
        return self.value
```

1. Создаем файл **metrics.py**.
2. Добавляем класс Counter.

СОЗДАНИЕ НАСТРОЕК МОНИТОРИНГА

```python
@dataclass  1 usage
class LatencyStats:
    """
    Статистика по задержке в миллисекундах.

    count    – сколько раз мы замерили время;
    total_ms – сумма всех задержек;
    min_ms   – минимальное значение;
    max_ms   – максимальное значение.
    """
    count: int = 0
    total_ms: int = 0
    min_ms: int = 0
    max_ms: int = 0

    def observe(self, ms: int) -> None:  2 usages (1 dynamic)
        # Учесть одно измерение задержки
        if ms < 0:
            return
        if self.count == 0:
            self.min_ms = ms
            self.max_ms = ms
        else:
            self.min_ms = min(self.min_ms, ms)
            self.max_ms = max(self.max_ms, ms)
        self.count += 1
        self.total_ms += ms

    @property  1 usage
    def avg_ms(self) -> float:
        # Средняя задержка в мс
        if self.count == 0:
            return 0.0
        return self.total_ms / self.count
```

3. Добавляем класс LatencyStats.

СОЗДАНИЕ НАСТРОЕК МОНИТОРИНГА

```python
class LatencyMetric:  3 usages
    """
    Обертка для LatencyStats, чтобы иметь единообразный API

    Пример использования:

        metric.latency("openrouter_latency_ms").observe(500)
    """

    def __init__(self, name: str) -> None:
        self.name = name
        self.stats = LatencyStats()

    def observe(self, ms: int) -> None:  2 usages (1 dynamic)
        self.stats.observe(ms)

    def snapshot(self) -> Dict[str, Any]:  2 usages (1 dynamic)
        """
        Срез статистики для этой метрики

        Возвращает dict с полями:
        count, total_ms, min_ms, max_ms, avg_ms
        """
        data = asdict(self.stats)
        data["avg_ms"] = self.stats.avg_ms
        return data
```

4. Добавляем класс LatencyMetric.

СОЗДАНИЕ НАСТРОЕК МОНИТОРИНГА

```python
class MetricsRegistry:  1 usage
    """
    Реестр метрик

    Основное API:

        from metrics import metric

        metric.counter("commands_total").inc()
        metric.latency("openrouter_latency_ms").observe(812)

    и метод snapshot() для команды /stats
    """

    def __init__(self) -> None:
        self._lock = threading.Lock()
        self._counters: Dict[str, Counter] = {}
        self._latencies: Dict[str, LatencyMetric] = {}

    # --- Счетчики ---

    def counter(self, name: str) -> Counter:  19 usages (19 dynamic)
        """
        Получить (или создать) счетчик с именем name.
        Если счётчик еще не зарегистрирован, создается новый.
        """
        with self._lock:
            c = self._counters.get(name)
            if c is None:
                c = Counter(name)
                self._counters[name] = c
            return c

    # --- Метрики задержки ---

    def latency(self, name: str) -> LatencyMetric:  2 usages (1 dynamic)
        """
        Получить (или создать) метрику задержки с именем name.

        Если не было — создается новая.
        """
        with self._lock:
            m = self._latencies.get(name)
            if m is None:
                m = LatencyMetric(name)
                self._latencies[name] = m
            return m

    # --- Срез всех метрик ---

    def snapshot(self) -> Dict[str, Any]:  1 usage (1 dynamic)
        """
        Вернуть все метрики в виде словаря:

        {
          "counters": {"commands_total": 10, ...},
          "latencies": {
             "openrouter_latency_ms": {
                 "count": ...,
                 "total_ms": ...,
                 "min_ms": ...,
                 "max_ms": ...,
                 "avg_ms": ...
             },
             "build_messages_ms": { ... },
             ...
          }
        }

        Используется для команды /stats.
        """
        with self._lock:
            counters = {name: c.get() for name, c in self._counters.items()}
            latencies = {
                name: metric.snapshot()
                for name, metric in self._latencies.items()
            }
        return {"counters": counters, "latencies": latencies}


# Глобальный реестр метрик
metric = MetricsRegistry()
```

5. Добавляем класс MetricsRegistry.

СОЗДАНИЕ НАСТРОЕК МОНИТОРИНГА

```python
def timed(metric_name: str, logger: logging.Logger | None = None) -> Callable[[Callable[..., T]], Callable[..., T]]:
    """
    Декоратор для замера времени выполнения функции.

    Пример:

        from metrics import metric, timed
        import logging

        log = logging.getLogger(__name__)

        @timed("build_messages_ms", logger=log)
        def build_messages(...):
            ...

    При каждом вызове:
    - замеряется время выполнения функции в миллисекундах
    - значение записывается в metric.latency(metric_name)
    - если передан logger, пишется DEBUG-запись со временем
    """

    def decorator(func: Callable[..., T]) -> Callable[..., T]:
        @functools.wraps(func)
        def wrapper(*args, **kwargs) -> T:
            t0 = time.perf_counter()
            try:
                return func(*args, **kwargs)
            finally:
                dt_ms = int((time.perf_counter() - t0) * 1000)
                metric.latency(metric_name).observe(dt_ms)
                if logger is not None:
                    logger.debug( msg: "timed %s: %s ms",  *args: func.__qualname__, dt_ms)

        return wrapper

    return decorator
```

6. Добавляем метод timed().

СОЗДАНИЕ МЕТОДА ЗАПИСИ В ЖУРНАЛ ВЫЗОВОВ

```python
def write_service_call(  3 usages  new *
    service: str,
    request: str,
    response: Optional[str],
    status_code: Optional[int],
    duration_ms: Optional[int],
    error: Optional[str] = None,
) -> None:
    """
    Записать информацию о вызове внешнего сервиса в service_call_log.

    service      — имя сервиса, например 'openrouter'
    request      — строковое представление запроса
    response     — строка ответа (может быть None)
    status_code  — HTTP статус, если есть
    duration_ms  — длительность вызова в мс
    error        — текст ошибки, если вызов завершился неуспешно
    """
    try:
        conn = _connect()
        cur = conn.cursor()
        cur.execute(
            _sql: """
            INSERT INTO service_call_log
                (created_at, service, request, response, status_code, duration_ms, error)
            VALUES (?, ?, ?, ?, ?, ?, ?)
            """,
            _parameters: (
                datetime.utcnow().isoformat(timespec="seconds"),
                service,
                request,
                response,
                status_code,
                duration_ms,
                error,
            ),
        )
        conn.commit()
        conn.close()
    except Exception as e:
        log.error( msg: "Не удалось записать запись в service_call_log: %s",  *args: e, exc_info=True)
```

Добавляем метод **write_service_call**() в файл db.py.

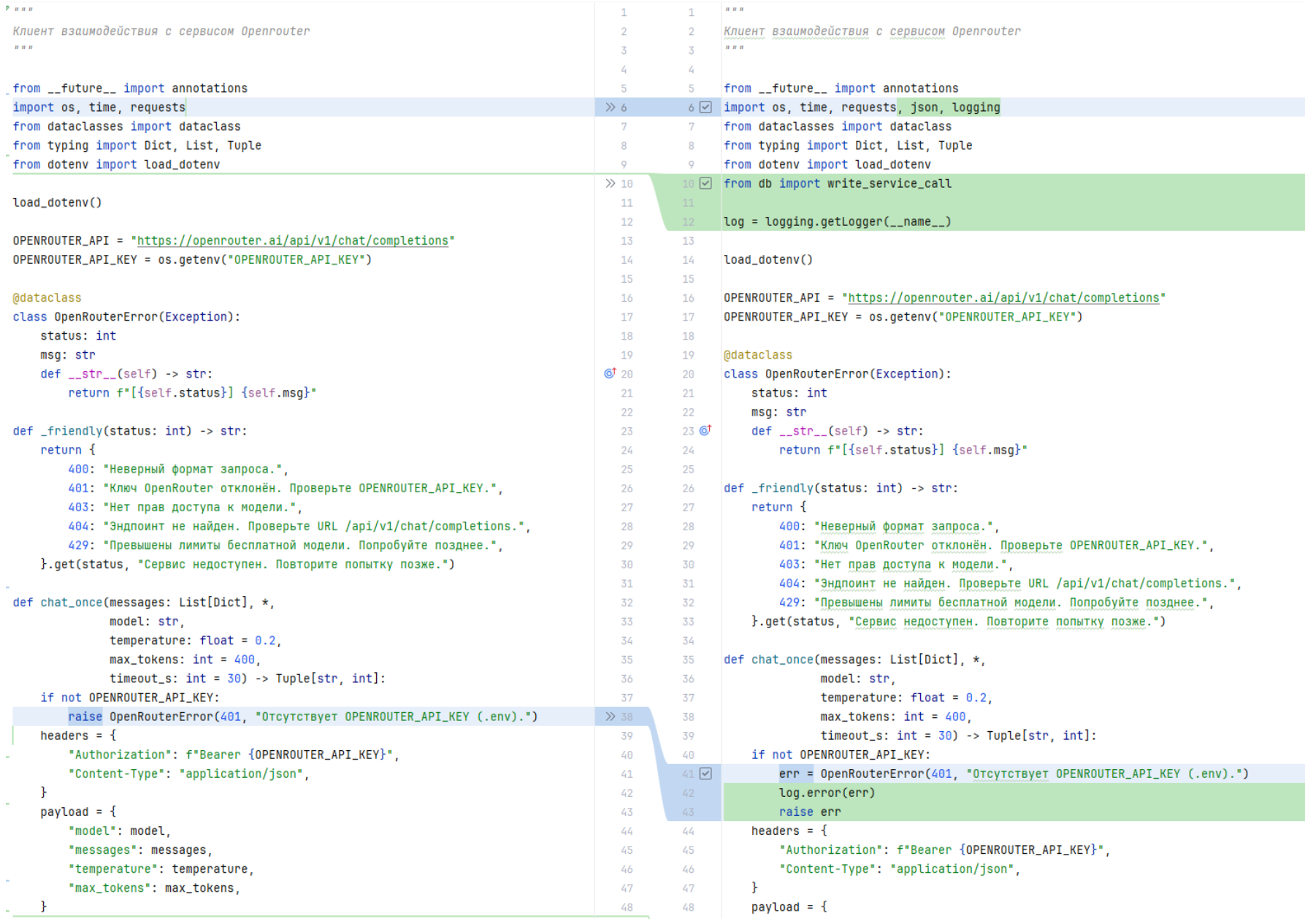
СОЗДАНИЕ МЕТОДА ЗАПИСИ В ЖУРНАЛ ОШИБОК

```python
def write_error_log(  3 usages  new *
    level: str,
    logger_name: str,
    message: str,
    user_id: Optional[int] = None,
    command: Optional[str] = None,
    details: Optional[str] = None,
) -> None:
    """
    Записать одну строку в таблицу error_log.

    Используем для важных ошибок:
    - OpenRouterError (401/429/5xx),
    - серьезные ошибки БД,
    - падения хендлеров
    """
    try:
        conn = _connect()
        cur = conn.cursor()
        cur.execute(
            _sql: """
            INSERT INTO error_log (created_at, level, logger, message, user_id, command, details)
            VALUES (?, ?, ?, ?, ?, ?, ?)
            """,
            _parameters: (
                datetime.utcnow().isoformat(timespec="seconds"),
                level,
                logger_name,
                message,
                user_id,
                command,
                details,
            ),
        )
        conn.commit()
        conn.close()
    except Exception as e:
        log.error( msg: "Не удалось записать ошибку в error_log: %s",  *args: e, exc_info=True)
```

Добавляем метод **write_error_log**() в файл db.py.

РЕФАКТОРИНГ **OPENROUTER_CLIENT**



```
"""
Клиент взаимодействия с сервисом Openrouter
"""

from __future__ import annotations
import os, time, requests
from dataclasses import dataclass
from typing import Dict, List, Tuple
from dotenv import load_dotenv

load_dotenv()

OPENROUTER_API = "https://openrouter.ai/api/v1/chat/completions"
OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")

@dataclass
class OpenRouterError(Exception):
    status: int
    msg: str
    def __str__(self) -> str:
        return f"[{self.status}] {self.msg}"

def _friendly(status: int) -> str:
    return {
        400: "Неверный формат запроса.",
        401: "Ключ OpenRouter отклонён. Проверьте OPENROUTER_API_KEY.",
        403: "Нет прав доступа к модели.",
        404: "Эндпоинт не найден. Проверьте URL /api/v1/chat/completions.",
        429: "Превышены лимиты бесплатной модели. Попробуйте позднее.",
    }.get(status, "Сервис недоступен. Повторите попытку позже.")

def chat_once(messages: List[Dict], *,
              model: str,
              temperature: float = 0.2,
              max_tokens: int = 400,
              timeout_s: int = 30) -> Tuple[str, int]:
    if not OPENROUTER_API_KEY:
        raise OpenRouterError(401, "Отсутствует OPENROUTER_API_KEY (.env).")
    headers = {
        "Authorization": f"Bearer {OPENROUTER_API_KEY}",
        "Content-Type": "application/json",
    }
    payload = {
        "model": model,
        "messages": messages,
        "temperature": temperature,
        "max_tokens": max_tokens,
    }
```

```
"""
Клиент взаимодействия с сервисом Openrouter
"""

from __future__ import annotations
import os, time, requests, json, logging
from dataclasses import dataclass
from typing import Dict, List, Tuple
from dotenv import load_dotenv
from db import write_service_call

log = logging.getLogger(__name__)

load_dotenv()

OPENROUTER_API = "https://openrouter.ai/api/v1/chat/completions"
OPENROUTER_API_KEY = os.getenv("OPENROUTER_API_KEY")

@dataclass
class OpenRouterError(Exception):
    status: int
    msg: str
    def __str__(self) -> str:
        return f"[{self.status}] {self.msg}"

def _friendly(status: int) -> str:
    return {
        400: "Неверный формат запроса.",
        401: "Ключ OpenRouter отклонён. Проверьте OPENROUTER_API_KEY.",
        403: "Нет прав доступа к модели.",
        404: "Эндпоинт не найден. Проверьте URL /api/v1/chat/completions.",
        429: "Превышены лимиты бесплатной модели. Попробуйте позднее.",
    }.get(status, "Сервис недоступен. Повторите попытку позже.")

def chat_once(messages: List[Dict], *,
              model: str,
              temperature: float = 0.2,
              max_tokens: int = 400,
              timeout_s: int = 30) -> Tuple[str, int]:
    if not OPENROUTER_API_KEY:
        err = OpenRouterError(401, "Отсутствует OPENROUTER_API_KEY (.env).")
        log.error(err)
        raise err
    headers = {
        "Authorization": f"Bearer {OPENROUTER_API_KEY}",
        "Content-Type": "application/json",
    }
    payload = {
```

РЕФАКТОРИНГ **OPENROUTER CLIENT**

```python
def _friendly(status: int) -> str:
    return {
        400: "Неверный формат запроса.",
        401: "Ключ OpenRouter отклонён. Проверьте OPENROUTER_API_KEY.",
        403: "Нет прав доступа к модели.",
        404: "Эндпоинт не найден. Проверьте URL /api/v1/chat/completions.",
        429: "Превышены лимиты бесплатной модели. Попробуйте позднее.",
    }.get(status, "Сервис недоступен. Повторите попытку позже.")

def chat_once(messages: List[Dict], *,
              model: str,
              temperature: float = 0.2,
              max_tokens: int = 400,
              timeout_s: int = 30) -> Tuple[str, int]:
    if not OPENROUTER_API_KEY:
        raise OpenRouterError(401, "Отсутствует OPENROUTER_API_KEY (.env).")
    headers = {
        "Authorization": f"Bearer {OPENROUTER_API_KEY}",
        "Content-Type": "application/json",
    }
    payload = {
        "model": model,
        "messages": messages,
        "temperature": temperature,
        "max_tokens": max_tokens,
    }
    t0 = time.perf_counter()
    r = requests.post(OPENROUTER_API, json=payload, headers=headers, timeout=timeout
    dt_ms = int((time.perf_counter() - t0) * 1000)
    if r.status_code // 100 != 2:
        raise OpenRouterError(r.status_code, _friendly(r.status_code))
    try:
        data = r.json()
        text = data["choices"][0]["message"]["content"]
    except Exception:
        raise OpenRouterError(500, "Неожиданная структура ответа OpenRouter.")
    return text, dt_ms
```

```python
        "temperature": temperature,
        "max_tokens": max_tokens,
    }

    request_str = json.dumps(payload, ensure_ascii=False)
    log.debug(
        "Запрос к OpenRouter: model=%s, temperature=%s, max_tokens=%s",
        model,
        temperature,
        max_tokens
    )

    t0 = time.perf_counter()
    r = requests.post(OPENROUTER_API, json=payload, headers=headers, timeout=timeout
    dt_ms = int((time.perf_counter() - t0) * 1000)

    if r.status_code // 100 != 2:
        raise OpenRouterError(r.status_code, _friendly(r.status_code))
    try:
        data = r.json()
        text = data["choices"][0]["message"]["content"]

        write_service_call(
            service="openrouter",
            request=request_str,
            response=r.text,
            status_code=r.status_code,
            duration_ms=dt_ms,
            error=None if r.status_code // 100 == 2 else _friendly(r.status_code),
        )
    except Exception as e:
        log.error(e)
        write_service_call(
            service="openrouter",
            request=request_str,
            response=None,
            status_code=None,
            duration_ms=None,
            error=repr(e),
        )
        raise OpenRouterError(500, "Неожиданная структура ответа OpenRouter.")

    return text, dt_ms
```

СОЗДАНИЕ МЕТОДА ПОЛУЧЕНИЯ МЕТРИК

```python
@bot.message_handler(commands=["stats"])  new *
def handle_stats(message: types.Message) -> None:
    """
    Показать все накопленные метрики:
    - все счетчики из metric.counter(...);
    - все latency-метрики
    """
    user_id = message.from_user.id
    metric.counter("commands_total").inc()
    log.info( msg: "Команда /stats от user_id=%s",  *args: user_id)

    stats = metric.snapshot()
    counters = stats["counters"]
    latencies = stats["latencies"]

    lines: list[str] = []
    lines.append("Статистика бота\n")

    # Счетчики
    lines.append("Счетчики:")
    if counters:
        for name, value in sorted(counters.items()):
            lines.append(f"- {name}: {value}")
    else:
        lines.append("- нет данных")

    # Замеры времени
    lines.append("\nЗамеры времени (мс):")
    if latencies:
        for name, data in sorted(latencies.items()):
            lines.append(
                f"- {name}: count={data['count']}, "
                f"avg={data['avg_ms']:.0f}, "
                f"min={data['min_ms']}, max={data['max_ms']}"
            )
    else:
        lines.append("- нет данных")

    bot.reply_to(message, "\n".join(lines))
```

Добавляем хендлер **/stats** в файл main.py для вывода значений метрик

СОЗДАНИЕ МЕТОДА ПОЛУЧЕНИЯ МЕТРИК

```python
def _setup_bot_commands() -> None:
    """
    Регистрирует команды в меню клиента Telegram (удобно для новичков).
    """
    cmds = [
        types.BotCommand("start", "Приветствие и помощь"),
        types.BotCommand("note_add", "Добавить заметку"),
        types.BotCommand("note_list", "Список заметок"),
        types.BotCommand("note_find", "Поиск заметок"),
        types.BotCommand("note_edit", "Изменить заметку"),
        types.BotCommand("note_del", "Удалить заметку"),
        types.BotCommand("note_count", "Сколько заметок"),
        types.BotCommand("note_export", "Экспорт заметок в .txt"),
        types.BotCommand("note_stats", "Статистика по датам"),
        types.BotCommand("model", "Установить активную модель"),
        types.BotCommand("models", "Получить список моделей"),
        types.BotCommand("ask", "Задать вопрос модели"),
        types.BotCommand("ask_random", "Задать вопрос случайной модели"),
        types.BotCommand("character", "Установить активного персонажа"),
        types.BotCommand("characters", "Получить список персонажей"),
        types.BotCommand("whoami", "Получить активную модель и активного персонажа"),
        types.BotCommand("stats", "Мониторинг бота"),
    ]
    bot.set_my_commands(cmds)
```

```
Команды:
  /start                  – приветствие + список команд
  /note_add <текст>       – добавить заметку
  /note_list [N]          – показать последние N заметок (по умолчанию 10, максимум 50)
  /note_find <подстрока>  – поиск заметок по тексту (без учёта регистра)
  /note_edit <id> <текст> – изменить текст заметки
  /note_del <id>          – удалить заметку
  /note_count             – количество заметок
  /note_export            – выгрузка заметок в .txt
  /note_stats [days]      – статистика по датам (ASCII-гистограмма, по умолчанию 7 дней)
  /models                 – список LLM моделей
  /model <id>             – установить активную LLM модель
  /ask <вопрос>           – задать вопрос LLM модели
  /ask_random <вопрос>    – задать вопрос случайной LLM модели
  /characters             – список персонажей
  /character <id>         – установить активного персонажа
  /whoami                 – активная модель и активный персонаж
  /stats                  – мониторинг бота
```

Добавляем команду **/stats** в список команд

ДОБАВЛЕНИЕ МЕТРИК В МЕТОДЫ

```
48  from metrics import metric, timed
```

```
483 @bot.message_handler(commands=["character"])
484 def cmd_character(message: types.Message) -> None:
485     """
486     Установить активным персонажа
487     """
488     metric.counter("commands_total").inc()
489     user_id = message.from_user.id
490     arg = message.text.replace("/character", "", 1).strip
```

Добавляем общий счетчик всех команд бота в каждый хэндлер (в начале каждого метода)

ДОБАВЛЕНИЕ МЕТРИК В МЕТОДЫ

```python
@bot.message_handler(commands=["whoami"])
def cmd_whoami(message: types.Message) -> None:
    """
    Показать активную модель и активного персонажа
    """
    metric.counter("commands_total").inc()
    metric.counter("whoami_requests_total").inc()
```

Добавляем индивидуальный счетчик каждой команды бота в каждый хэндлер по схеме:

```python
metric.counter("команда_requests_total").inc()
```

кроме **/stats**, т.к эта команда предназначена для вывода метрик и ее отслеживание не несет практической пользы

ДОБАВЛЕНИЕ МЕТРИК В МЕТОДЫ

```python
@timed("build_messages_ms", logger=log)
def _build_messages(user_id: int, user_text: str) -> list[dict]:
    p = get_user_character(user_id)
    system = (
```

К необходимым методам добавляем декоратор @timed для снятия метрик выполнения методов