MorphCluster Logger

Подсистема логирования состоит из двух пакетов:

- **@morphcluster/logger** — серверная часть, предоставляющая сервисы для сбора, хранения и администрирования логов.
- **@morphcluster/logger-client** — клиентский бэкенд для `Logger` из ядра, обеспечивающий передачу логов на сервер логирования.

Серверная часть (@morphcluster/logger)

Пакет регистрирует в `ServiceHost` четыре сервиса: `LogStoreBackend`, `LogStoreAdmin`, `LogOraList` и вспомогательный `PostgresMigrator` (из отдельного пакета). Основное хранилище — PostgreSQL, взаимодействие с которым вынесено в сервис `LogPgStorage2`.

LogStoreBackend

Главный сервис приёма логов. Реализует два запроса: `write` и `close`. Полученные данные не пишутся в базу сразу, а накапливаются в оперативной памяти в виде дерева объектов `LogRecord` (класс `ActiveLogs`). При закрытии корневого лога вся ветка отправляется в `LogPgStorage2.insertLogTree()`.

```
// Пример использования (происходит автоматически при подключении LoggerBackendStore)
await logStoreBackend.write({ id, parentId, message, level, ... });
await logStoreBackend.close({ id, timestamp });
```

Жизненный цикл активного лога:

1. Клиент вызывает `write` при создании лога и для каждого дочернего сообщения.
2. Сообщения собираются в древовидную структуру `LogRecord` внутри `ActiveLogs`.
3. Когда клиент вызывает `close` для записи (или по тайм-ауту), `ActiveLogs` проверяет, все ли дочерние записи закрыты. Если да — вся ветка помечается на архивацию и через 5 секунд (`flushLogTime`) вызывается `onArchive`, передающий корневой `LogRecord` в `LogPgStorage2`.
4. `LogPgStorage2` вставляет всё дерево одной массовой вставкой в таблицу `logs2`.

Свойства и зависимости:

- `LogPgStorage2` — обязательный сервис для работы с БД.
- `RegistryHelper` — для получения настроек логирования из общего реестра (`minLevel`, `maxRecords`).
- `activeLogs: ActiveLogs` — хранилище незавершённых логов.

LogRecord

Модель одной записи лога. Образует древовидную структуру: родитель (`parent`) и массив дочерних записей (`children`).

```
const record = new LogRecord({
  id: 'uuid',
  message: 'Запрос выполнен',
  level: 3,
  service: 'MyService',
  payload: { ... }
});
record.addChild(childRecord);
```

Основные поля:

- `id`, `message`, `level`, `host`, `service`, `username`, `ip`
- `payload` — произвольные данные
- `order` — порядок среди дочерних одного родителя
- `duration` — длительность (заполняется при закрытии)
- `timestamp` — временная метка
- `closed` — флаг завершения записи
- `parent` / `children` — связи дерева
- `serverTimestamp` — момент попадания на сервер (для детекта зависших логов)
- `dontArchive` — если `true`, запись не будет сохранена в БД (например, автоматически созданный родитель «Unknown Parent»).

Методы:

- `addChild(record)` — добавить дочернюю запись (только если текущая ещё не закрыта).
- `levelToRoot()` — протолкнуть максимальный уровень вверх по дереву до корня.
- `findRecordById(id)` — рекурсивный поиск записи по идентификатору.
- `getPath()` — путь от корня до текущей записи.
- `getRoot()` — корневая запись.
- `sortChildren()` — сортировка детей по `order` (рекурсивно).
- `getJson()` — сериализация всего поддерева в объект (для ответа API).
- `forEach(callback)` — обход всех узлов дерева в ширину.

ActiveLogs

Хранилище всех активных (ещё не заархивированных) записей в памяти. Управляет их жизненным циклом: вставка, закрытие, автоматическое закрытие по тайм-ауту, архивация.

```
const activeLogs = new ActiveLogs();
activeLogs.onArchive = (rootLog) => { /* сохранить в БД */ };
activeLogs.insert({ id: '...', message: 'Старт' });
```

```
activeLogs.close('id', Date.now());
```

Основные методы и логика:

- `insert(request)` — создаёт или обновляет `LogRecord`.
  - Если передан `parentId`, находит или создаёт родителя. При отсутствии реального родителя создаётся запись-заглушка с `dontArchive = true` и сообщением `'Unknown Parent'`.
  - Вызывает `levelToRoot()` для подъёма уровня.
  - Обрабатывает ситуацию, когда `close` пришёл раньше `write` (через очередь `unknownCloses`).
- `close(id, timestamp)` — закрывает запись. Если запись ещё не известна, сохраняет «закрытие» в `unknownCloses` на 10 секунд, ожидая появления записи. После закрытия рекурсивно проверяет родительские ветки (`checkBranch`); если корень полностью закрыт, запускает таймер (5 секунд), после которого ветка удаляется из памяти и передаётся в `onArchive`.
- `checkTimeouts()` — периодическая проверка (каждую секунду):
  - Закрытые ветки, готовые к архивации, отправляются в `onArchive`.
  - Неиспользованные `unknownCloses` удаляются через 10 секунд.
  - Незакрытые логи, неактивные более 15 минут (`autoCloseTime`), автоматически закрываются с уровнем 50 и сообщением `'Log Timeout'`.
- `removeBranch(curLog)` — удаляет всё поддерево из `logsPlain`.
- `getRootLogs()` — возвращает только корневые логи (для `listActive`).

Свойства:

- `logsPlain` — плоский массив всех известных записей (и корневых, и дочерних).
- `unknownCloses` — очередь «закрытий», ожидающих свои записи.
- `onArchive` — колбэк, вызываемый при готовности корневого лога к сохранению.

LogPgStorage2

Сервис для взаимодействия с PostgreSQL. Выполняет миграции, создаёт пул соединений, вставляет деревья логов, предоставляет методы для чтения и очистки.

```
await logPgStorage2.insertLogTree(rootLogRecord);
const result = await logPgStorage2.list({ service: 'MyService', limit: 20 });
const log = await logPgStorage2.byId('uuid');
```

Управление:

- `minLevel: number` — логи с уровнем ниже не сохраняются (по умолчанию 0, берётся из реестра).
- `maxRecords: number` — после вставки, если счётчик корневых записей превышает лимит, вызывается `cleanup()` (полная очистка таблицы). Значение по умолчанию 500.
- `recordsCount` — текущее количество корневых записей.

Методы:

- `insertLogTree(rootLog)` — рекурсивно обходит дерево, собирает массив записей для вставки. Игнорирует записи с `level < minLevel`. Выполняет массовый `INSERT`. При дубликате `id` корневого лога переименовывает его, генерирует новый `id` и ставит уровень 50.
- `list(request)` — выборка корневых логов с фильтрацией (`service`, `username`, `minLevel`, `dateFrom/dateTo` и др.) и пагинацией.
- `listChildren(parentId)` / `listChildrenRecursive(parentId)` — получение дочерних записей.
- `byId(logId)` — одна запись по идентификатору.
- `cleanup()` — `TRUNCATE` таблицы.
- `tableSize()` — размер таблицы в человекочитаемом виде.
- `serviceOptions()` — список уникальных сервисов, встречающихся в логах.
- `start(log)` — инициализирует миграции, создаёт пул соединений, проверяет подключение (`PGTest`) и считает количество корневых записей.

Схема БД (подразумевается): таблица `logs2` с колонками `id`, `parent_id`, `message`, `level`, `duration`, `timestamp`, `order`, `host`, `service`, `username`, `ip`, `created`, `payload`, `num` (автоинкрементный номер для курсорной пагинации).

LogStoreAdmin

Сервис для административного доступа к логам. Все запросы требуют прав администратора (`needAdmin: true`).

Запросы:

- `list(request)` — делегирует в `LogPgStorage2.list`.
- `listActive(req)` — возвращает текущие незавершённые корневые логи из `activeLogs`. Поддерживает фильтрацию по `minLevel` и `service`.
- `details(request)` — сначала ищет лог в активных (`activeLogs`), если не найден — загружает из базы с рекурсивными дочерними записями.
- `stats` — возвращает `recordsCount` и размер таблицы.
- `cleanup` — принудительная очистка таблицы логов.
- `serviceOptions` — делегирует в `LogPgStorage2`.

Зависимости: `LogStoreBackend`, `LogPgStorage2`.

LogOraList

Специализированное представление для получения логов запросов к Oracle (тип сообщения `'OraQueries/exec'` или `'OraQueries/execFunc'`). Извлекает из древовидной структуры входные и выходные параметры, собирая их из фиксированной иерархии дочерних записей (4 уровня вложенности). Возвращает плоский список с полями `queryName`, `duration`, `in_payload`, `out_payload`.

Запрос:

- `list(filter)` — `filter` может содержать `username`, `payload` (поиск по тексту во входном payload), `minLevel`, `minDuration`, `dateFrom/dateTo`, плюс стандартные `limit` и `offset`.

Схемы сервисов

- **LogStoreBackend** — два внутренних запроса `write` и `close` (не требуют аутентификации, не логируются сами).
- **LogStoreAdmin** — запросы `list`, `listActive`, `details`, `serviceOptions`, `stats`, `cleanup` (требуют прав администратора).
- **LogOraList** — запрос `list` с фильтром.

Клиентская часть (@morphcluster/logger-client)

LoggerBackendStore

Реализация бэкенда `LoggerBackend` из ядра, которая отправляет все логи на серверный `LogStoreBackend`. Обеспечивает прозрачную буферизацию логов до момента появления сервиса логирования в хосте.

```
import { LoggerBackendStore } from '@morphcluster/logger-client';


const logStoreBackend = new LoggerBackendStore(host);
logger.backends.push(logStoreBackend);
```

Принцип работы:

1. При создании получает ссылку на `ServiceHost` и подписывается на событие `onServiceStarted`.
2. Как только в хосте появляется сервис с именем `LogStoreBackend` (или `LogStoreAcc`, если есть), он сохраняет ссылку на него.
3. До этого момента все вызовы `write` и `close` помещаются в очередь (`queue`).
4. При обнаружении сервиса очередь «сбрасывается» — все накопленные операции последовательно отправляются в реальный сервис. После этого `flushed` устанавливается в `true`, и дальнейшие вызовы выполняются напрямую, без очереди.

Таким образом, логирование не теряется даже на этапе запуска приложения, когда сервис логирования ещё не готов.

Свойства:

- `host: ServiceHost`
- `LogStore` — ссылка на обнаруженный сервис (например, `LogStoreBackend`).
- `queue: Array` — очередь отложенных вызовов `{ method, data, resolve, reject }`.
- `flushed: boolean` — флаг завершения «слива» очереди.

Взаимодействие компонентов

1. Сервис приложения использует обычный `Logger` из ядра, добавив в него бэкенд `LoggerBackendStore`.
2. `LoggerBackendStore` отправляет запросы `write` и `close` в локальный или удалённый `LogStoreBackend` (через `GlobalService`).
3. `LogStoreBackend` сохраняет все записи в `ActiveLogs` в оперативной памяти.
4. Когда ветка лога полностью закрыта, `ActiveLogs` вызывает `LogPgStorage2.insertLogTree()`, который одной массовой вставкой записывает всё дерево в PostgreSQL.
5. Для чтения логов используются сервисы `LogStoreAdmin` (универсальный) и `LogOraList` (специализированный для Oracle-запросов). Они читают данные напрямую из `ActiveLogs` (активные) и из `LogPgStorage2` (архивные).

Такая архитектура минимизирует количество обращений к базе данных и позволяет гибко настраивать уровни логирования через центральный реестр.

Revision #2
Created 21 June 2026 19:40:43 by Admin
Updated 26 June 2026 13:52:59 by Admin