Структура сервисов Morphcluster

`Service` — это основа для всех сервисов. Каждый сервис обладает:

Свойства сервиса

Свойство	Тип	Описание
`name`	`string`	Имя сервиса (по умолчанию имя класса)
`schema`	`Schema`	Публичный контракт сервиса
`started`	`boolean`	Запущен ли сервис в данный момент
`starting`	`boolean`	Находится ли в процессе запуска
`private` / `local`	`boolean`	Сервис не публикуется в глобальной системе
`remote`	`boolean`	Виртуальный сервис, не добавляется в глобальную систему
`Logger`	`Logger`	Экземпляр для журнала
`Config`	`Config`	Глобальной конфигурация приложения
`clients`	`ServiceClient[]`	Клиенты, через которые сервис обращается к другим сервисам
`timers`	`Timer[]`	Периодические задачи
`triggers`	`Trigger[]`	Обработчики событий других сервисов
`senders`	`ChannelSender[]`	Каналы для публикации данных
`aggregators`	`ChannelAggregator[]`	Агрегаторы данных из каналов

Схема сервиса

Схема задаётся через объект `ServiceSchema` и определяет публичный контракт сервиса.

Определяется через методы:

- this.addRequest(requestSchema)
- this.addEvent(requestSchema)
- this.setSvcSchema(jsonSchema)

Структуры описаны ниже

Жизненный цикл

```
[создание] → host.addService(service) → host.startService(service)
                                      → service.start(log)
                                      → service.started = true
                                      → service.stop()
                                      → service.started = false
```

При возникновении ошибки во время старта хост автоматически попытается перезапустить сервис через 5 секунд.

Запросы

Каждый сервис может предоставлять один или несколько **запросов** — методов, доступных для вызова извне (другими сервисами, через HTTP-мост, клиентами).
Запрос описывается схемой `RequestSchema` и реализуется методом сервиса с определённой сигнатурой.

Схема запроса (`RequestSchema`)

При добавлении запроса через `this.addRequest({...})` создаётся объект `RequestSchema` со следующими полями:

Поле	Тип	Обязательное	Описание
`name`	`string`	да	Уникальное имя запроса в рамках сервиса. **Должно совпадать с именем метода-обработчика**.
`description`	`string`	нет	Человекочитаемое описание.
`request`	`object`	нет	JSON-схема (стандарт JSON Schema) для валидации входящих параметров.
`response`	`object`	нет	JSON-схема, описывающая структуру ответа.
`anonymous`	`boolean`	нет	Если `true`, запрос доступен без аутентификации.
`needAdmin`	`boolean`	нет	Если `true`, требует прав администратора.
`noLogs`	`boolean`	нет	Отключает автоматическое создание лога.

Поле	Тип	Обязательное	Описание
`http`	`string`	нет	HTTP-метод (`GET`, `POST` и т.д.) при экспорте через `ServiceBridge`.

Пример добавления запроса:

```
this.addRequest({
  name: 'getReport',
  description: 'Возвращает отчёт за период',
  request: {
    type: 'object',
    properties: {
      from: { type: 'string', format: 'date' },
      to:   { type: 'string', format: 'date' }
    },
    required: ['from', 'to']
  },
  response: {
    type: 'object',
    properties: {
      rows: { type: 'array' }
    }
  }
});
```

Метод-обработчик запроса

Для каждого запроса, объявленного в схеме, в классе сервиса должен быть реализован **асинхронный метод** с точно таким же именем.

```
async methodName(params, ws, log) {
  // params – объект с параметрами запроса
  // ws – Устаревший параметр, всегда null
  // log – экземпляр Logger для данного вызова
}
```

- Параметры **не валидируются автоматически** на уровне ядра. Рекомендуется проверять входные данные внутри метода.
- Если запрос помечен как `noLogs: true`, всё равно можно писать в `log` — это безопасно, так как логгер проигнорирует запись.

- Метод может возвращать любое сериализуемое значение (объект, массив, число, строку и т.д.). Этот результат будет передан обратно вызывающей стороне.
- Все обработчики должны быть `async`, даже если внутри нет асинхронных операций.
- Не забывайте закрывать дочерние логи, созданные внутри метода, или используйте `log.wrap()` для автоматического закрытия.

Обработка ошибок

`ComplexError` – специальный класс ошибки, позволяющий передать дополнительную информацию:

- `name` – тип ошибки (строка),
- `payload` – любые данные,
- `options.showUser` – показывать ли сообщение пользователю,
- `options.httpStatus` – HTTP-статус (при использовании `ServiceBridge`),

Если метод выбрасывает `ComplexError`, он будет проброшен до вызывающего кода с сохранением всех свойств. `ServiceBridge` автоматически добавляет в ошибку `logId` текущего лога.

Любое другое исключение (`Error`) будет преобразовано в `ComplexError` с именем класса ошибки.

```
throw new ComplexError('Отчёт не найден', 'ReportNotFound', { from, to }, { showUser: true });
```

Объявление зависимостей

```
class MyService extends ServiceRequire {
   constructor(host) {
     super(host);
     /** @type {Database} */
     this.Database = null //Будет автоматически заполнен после start
     /** @type {Cache} */
     this.Cache = null //Будет автоматически заполнен после start
     this.requirements = ['Database', 'Cache'];   // жёсткие зависимости
   }

   async start(log) {
     await super.start(log)
     //Здесь доступен this.Database и this.Cache
   }
```

```
}
```

1. Запускается `start(log)` родительского класса.
2. Для каждого имени в `optionalServices` вызывается `host.waitService(optName)`, но **без ожидания**.
3. Для `requirements` формируются Promise на `waitRequire(svcName, log)`, которые резолвятся после получения сервиса.
4. Все требования загружаются параллельно (`Promise.all`).
5. Загруженные сервисы сохраняются в свойства `this[svcName]`.

Состояние загрузки можно отследить через `requirementsStatus()`.

Логирование

Каждый сервис получает экземпляр `Logger` с предустановленными `service` и `host`.

Основные методы:

Метод	Описание
`write(message, payload?, options?)`	Записать простое сообщение
`createWriter(message, payload?, options?)`	Создать дочерний логгер, привязанный к текущей записи как к родительской. Новая запись остаётся открытой (`closed: false`).
`close()`	Закрыть текущую запись (помечает `closed: true`, закрывать только те логи, которые создал сам).
`writeException(error, options?)`	Записать исключение и **закрыть запись**.
`writeExceptionOnly(error, options?)`	Записать только данные исключения без закрытия записи.
`wrap(callback)`	Обернуть асинхронную функцию: автоматически закрыть лог при успехе или записать исключение при ошибке.
`createWrapped(message, callback, payload?, options?)`	Создать логи и обернуть им асинхронную функцию.

Логирование метода сервиса (запроса)

Каждый метод сервиса, объявленный через `addRequest`, принимает аргумент `log`. Это - **дочерний логгер**, созданный вызывающей стороной. Внутри метода рекомендуется использовать именно этот `log` для всех записей, чтобы сохранить иерархию.

Пример:

```
async generateReport(params, ws, log) {
  log.write('Начало генерации отчёта', { from: params.from, to: params.to });
```

```
  //При вызове другого сервиса обязательно передаем log
  const data = await this.DataService.fetchData(params, null, log);
  log.write('Данные получены', { recordCount: data.length });
  return { report: data };
}
```

Дочерние доги

Если код вызывается асинхронно (без await), необходимо создать ему дочерний лог

```
async complexTask(params, workspace, log) {
    // log уже передан извне
    const ChildLog = log.createWriter('Шаг 1', { details: '...' });
    //`wrap` автоматически оборачивает функцию, закрывая лог при успехе и записывая ошибку при
неудаче.
    await ChildLog.wrap(async () => {
      await this.step1();
    });
}
```

События

Схема события (`EventSchema`)

Поле	Тип	Описание
`name`	`string`	Имя события
`description`	`string`	Описание
`structure`	`object`	Структура передаваемых данных

Реализация событий с поддержкой:

- подписки/отписки (`on`, `off`),
- вложенных событий (`addSubEvent`, `removeSubEvent`),
- автоматического вызова `onSubscribe`/`onUnsubscribe` при появлении первого/последнего слушателя.

Таймеры и триггеры

Таймер (`Timer`) — периодическая асинхронрная задача. Таймаут начнется только после завершения асихронного вызова.

```
this.timers.push(new Timer('cleanup', 60000, async () => {
    // очистка каждые 60 секунд
}));
```

Триггер (`Trigger`) — реакция на событие другого сервиса. Автоматически создает лог.

```
this.onDataChanged = new Event();
this.trgDataChanged = new Trigger('onDataChanged', async (log, data) => {
    // обработать событие
});
this.triggers.push(trgDataChanged);
// подключить триггер к событию
trigger.connect(this.onDataChanged);
```

Оба компонента автоматически инициализируются при старте сервиса (им присваиваются `serviceName` и `logger`).

Каналы (ChannelSender / ChannelAggregator)

- **ChannelSender** — позволяет сервису публиковать данные в именованный канал.
- **ChannelAggregator** — собирает данные от нескольких сервисов в одном канале, вызывая событие при изменении.

Пример использования:

```
this.senders.push(new ChannelSender('metrics'));
// где-то в коде
this.senders[0].data = { cpu: 50 };

this.aggregators.push(new ChannelAggregator('metrics'));
this.aggregators[0].onDataChanged.on((ws, allData) => {
    // allData — массив { serviceName, data }
});
```

Config

Глобальная конфигурация, загружаемая при старте приложения. Сервис получает её через `this.Config`.

ServiceClient — клиент к сервису

Клиент позволяет сервису обращаться к другому сервису, не заботясь о его физическом расположении (локальный или удалённый).

На данный момент `ServiceClient` - это экспериментальная функциональность

Revision #13
Created 26 June 2026 12:08:58 by Admin
Updated 26 June 2026 13:52:58 by Admin