Создание новых хостов

В MorphCluster приложение строится из **хостов** (ServiceHost) и **сервисов** (Service), которые могут объединяться в единую систему через GlobalServices. Ниже приведены пошаговые инструкции по созданию новых хостов и сервисов на основе исходного кода и документации.

---

1. Основные понятия

- **ServiceHost** – контейнер, управляющий жизненным циклом сервисов: запуск, остановка, перезапуск при ошибках, ожидание зависимостей.
- **Service** – базовый класс любого сервиса. Имеет схему (ServiceSchema) с запросами, событиями, таймерами.
- **ServiceRequire** – расширение Service с механизмом ожидания других сервисов (зависимостей) перед стартом.
- **GlobalServices** – реестр, связывающий локальные и удалённые сервисы, позволяет вызывать их по имени через схему.
- **Config** – глобальная конфигурация (загружается из `global-config.json`).
- **Logger** – иерархическое логирование с привязкой к сервису и запросу.

---

2. Создание нового хоста

Хост – это экземпляр `ServiceHost`. Обычно он создаётся в точке входа приложения.

Пример (`main.js`):

```
import { ServiceHost, Config, Logger, LoggerBackendConsole } from '@morphcluster/core';
import { Fastify, HttpProxy } from '@morphcluster/fastify';
import MyService from './my-service.js';

const config = new Config();
await config.load();

const host = new ServiceHost('main');
host.Config = config;

// Настройка логгера
const logger = new Logger({
  service: 'host',
```

```
  host: 'main',
  loggerErrorsFile: './logs/errors.log'
});
logger.backends.push(new LoggerBackendConsole());
host.Logger = logger;

// Добавление сервисов
host.addService(new Fastify(host, { port: 3000 }));
host.addService(new MyService(host));

await host.start(logger);
```

Основные шаги:

- Загрузить конфигурацию через `Config.load()`.
- Создать `ServiceHost` с уникальным именем.
- Присвоить хосту `Config` и `Logger`.
- Добавить сервисы через `host.addService(service)`.
- Запустить хост через `host.start(logger)`.

---

3. Создание простого сервиса

Наследуйте класс от `Service` или `ServiceRequire`. Определите схему и реализуйте обработчики. Зависимости объявляются в массиве `this.requirements`.

```
import { ServiceRequire } from '@morphcluster/core';

export default class MyDependentService extends ServiceRequire {
  constructor(host, config) {
    super(host); // обязательно передаём host для ServiceRequire
    /** @type {MySimpleService} */
    this.MySimpleService = null; // будет заполнено автоматически
    this.requirements = ['MySimpleService'];
    this.addRequest({
      name: 'echoTwice',
      description: 'Вызывает echo дважды',
      request: {
        type: 'object',
        properties: { text: { type: 'string' } },
        required: ['text']
```

```
    }
   });
  }

  async echoTwice(req, workspace, log) {
   const first = await this.MySimpleService.echo(req, workspace, log);
   const second = await this.MySimpleService.echo(req, workspace, log);
   return { first, second };
  }

  async start(log) {
   await super.start(log); // здесь произойдёт ожидание всех requirements
  }
}
```

Сервис с именем `MySimpleService` будет автоматически найден в том же хосте и присвоен в свойство с таким же именем.

---

4. Схема сервиса: запросы, события, таймеры

Полную схему задают через методы `addRequest`, `addEvent` или прямо через `this.setSvcSchema(json)`.

4.1. Запросы (requests)

Объект запроса:

```
{
  name: 'methodName',       // имя метода
  description: '...',
  request: { ... },         // JSON-схема входных данных (может быть null)
  response: { ... },        // JSON-схема ответа
  anonymous: false,         // доступно без токена (для HTTP)
  needAdmin: false,         // требуются права админа
  noLogs: false,            // отключить логирование
  http: 'POST',             // HTTP-метод (для автоматической генерации маршрутов)
}
```

После добавления запроса обработчик должен быть методом с таким же именем в классе.

4.2. События (events)

```
this.addEvent({
  name: 'onUserAdded',
  description: 'Вызывается при добавлении пользователя',
  structure: { ... }   // JSON-схема передаваемых данных
});

// В коде сервиса событие становится экземпляром Event
this.onUserAdded = new Event();
// Отправка события
this.onUserAdded.emit(workspace, data);
```

Клиенты подписываются через `svc.getEvent('onUserAdded').on(callback)`.

4.3. Таймеры

```
import { Timer } from '@morphcluster/core';

const timer = new Timer('cleanupTimer', 60000, async (log) => {
  // код, выполняемый каждую минуту
});
timer.maxRunningTime = 10000; // таймаут выполнения в мс
timer.serviceName = this.name; // будет установлено при старте
timer.logger = this.Logger;    // то же
this.timers.push(timer);
```

---

5. Конфигурация сервиса

Параметры сервиса обычно передаются вторым аргументом конструктора:

```
constructor(host, config) {
  super(host);
  this.someOption = config.someOption || 'default';
}
```

Конфигурация заполняется через `config.services` в `global-config.json`:

```
{
  "common": { "baseUrl": "http://localhost:3000" },
  "services": [
    { "name": "MyService", "config": { "someOption": "value" } }
```

```
  ]
}
```

Затем `ServiceFactory` (см. раздел 6) автоматически инстанцирует сервис с нужным конфигом.

---

6. HTTP-интерфейс и Fastify

Если сервису нужен свой HTTP-эндпоинт, удобно наследоваться от `Fastify` (из `@morphcluster/fastify`).

```
import { Fastify } from '@morphcluster/fastify';

export default class MyHttpService extends Fastify {
  constructor(host, config) {
    super(host, config);
    this.addRoute({
      method: 'GET',
      url: '/hello',
      handler: (req, reply) => reply.send('Hello World')
    });
  }
}
```

Сервис сам запустит Fastify-сервер и при необходимости зарегистрируется в `HttpProxy` (если он есть в системе).

Для создания API Gateway, объединяющего все сервисы, используется `FastifyGateway`. Он автоматически строит маршруты на основе схем сервисов с полем `http`.

7. Глобальные сервисы и межхостовое взаимодействие

Если приложение разбито на несколько хостов (например, отдельные процессы), для общения используется `GlobalServices`.

- На каждом хосте запускается `GlobalServices` (обычно через `ServiceFactory`).
- При добавлении локального сервиса он автоматически публикуется в реестре.
- Для вызова удалённого сервиса используется `GlobalServiceInterface` (например, через NATS или HTTP-мост `ServiceBridge`).

Подробнее в документации к пакетам `@morphcluster/nats` и `@morphcluster/fastify`.

8. Обработка ошибок

Используйте `ComplexError` для ошибок, которые могут быть показаны пользователю:

```
throw new ComplexError('Текст ошибки', 'MyErrorCode', { доп_данные }, {
  showUser: true,
  httpStatus: 422
});
```

В логах ошибки автоматически обрабатываются, в HTTP-ответе от FastifyGateway они возвращаются клиенту с указанным статусом.

---

10. Резюме

- **Хост** управляет сервисами; создайте `ServiceHost`, назначьте `Config` и `Logger`, вызовите `start`.
- **Сервис** – класс с методами, соответствующими `addRequest`, и наследующий от `Service` / `ServiceRequire`.
- Зависимости указываются в массиве `requirements` (только для `ServiceRequire`).
- HTTP-функциональность добавляется через классы из `@morphcluster/fastify`.
- Глобальная коммуникация – через `GlobalServices` и `GlobalServiceInterface`.

Эти шаблоны покрывают большинство сценариев при разработке на MorphCluster.

Revision #3
Created 21 June 2026 23:37:45 by Admin
Updated 26 June 2026 13:52:59 by Admin