

Начало работы

Добро пожаловать в документацию Sisk.

Наконец, что такое Sisk Framework? Это открытая библиотека с открытым исходным кодом, построенная на основе .NET, предназначенная для того, чтобы быть минимальной, гибкой и абстрактной. Она позволяет разработчикам создавать интернет-сервисы быстро, с минимальной или без необходимой конфигурации. Sisk позволяет вашему существующему приложению иметь управляемый модуль HTTP, полный и утилизируемый или полный.

Ценности Sisk включают прозрачность кода, модульность, производительность и масштабируемость, и могут обрабатывать различные типы приложений, такие как Restful, JSON-RPC, Web-sockets и многое другое.

Его основные функции включают:

| Ресурс | Описание |
|--------------------|--|
| Routing | Маршрутизатор пути, поддерживающий префиксы, пользовательские методы, переменные пути, конвертеры значений и многое другое. |
| Request Handlers | Также известные как <i>посредники</i> , предоставляют интерфейс для создания собственных обработчиков запросов, которые работают с запросом до или после действия. |
| Compression | Сжимайте содержимое ответа легко с помощью Sisk. |
| Web sockets | Предоставляет маршруты, которые принимают полные веб-сокеты, для чтения и записи на клиент. |
| Server-sent events | Предоставляет отправку событий сервера клиентам, поддерживающим протокол SSE. |
| Logging | Упрощенное ведение журнала. Ведите журнал ошибок, доступа, определяйте журналы вращения по размеру, несколько потоков вывода для одного и того же журнала и многое другое. |
| Multi-host | Имеете HTTP-сервер для нескольких портов, и каждый порт со своим маршрутизатором, и каждый маршрутизатор со своим приложением. |
| Server handlers | Расширяйте свою собственную реализацию HTTP-сервера. Настройте с помощью расширений, улучшений и новых функций. |

Первые шаги

Sisk может работать в любой среде .NET. В этом руководстве мы научим вас, как создать приложение Sisk с помощью .NET. Если вы еще не установили его, пожалуйста, скачайте SDK с [сюда](#).

В этом учебнике мы расскажем, как создать структуру проекта, получить запрос, получить параметр URL и отправить ответ. Это руководство будет сосредоточено на построении простого сервера с помощью C#. Вы также можете использовать свой любимый язык программирования.

NOTE

Вам может быть интересно начать с проекта. Проверьте [этот репозиторий](#) для получения дополнительной информации.

Создание проекта

Давайте назовем наш проект "Мое приложение Sisk". Как только вы настроите .NET, вы можете создать свой проект с помощью следующей команды:

```
dotnet new console -n my-sisk-application
```

Далее перейдите в каталог вашего проекта и установите Sisk с помощью утилиты .NET:

```
1 cd my-sisk-application  
2 dotnet add package Sisk.HttpServer
```

Вы можете найти дополнительные способы установки Sisk в вашем проекте [здесь](#).

Теперь давайте создадим экземпляр нашего HTTP-сервера. Для этого примера мы настроим его на прослушивание порта 5000.

Построение HTTP-сервера

Sisk позволяет вам строить свое приложение шаг за шагом вручную, поскольку оно маршрутизируется к объекту `HttpServer`. Однако это может быть не очень удобно для большинства проектов. Поэтому мы можем использовать метод построения, который делает его проще получить наше приложение в работу.

Program.cs

C#

```
1  class Program
2  {
3      static async Task Main(string[] args)
4      {
5          using var app = HttpServer.CreateBuilder()
6              .UseListeningPort("http://localhost:5000/")
7              .Build();
8
9          app.Router.MapGet("/", request =>
10         {
11             return new HttpResponseMessage()
12             {
13                 Status = 200,
14                 Content = new StringContent("Hello, world!")
15             };
16         });
17
18         await app.StartAsync();
19     }
20 }
```

Важно понять каждый важный компонент Sisk. Позже в этой документации вы узнаете больше о том, как работает Sisk.

Ручная (расширенная) настройка

Вы можете узнать, как работает каждый механизм Sisk в [этом разделе](#) документации, который объясняет поведение и отношения между `HttpServer`, `Router`, `ListeningPort` и другими компонентами.

Установка

Вы можете установить Sisk через Nuget, dotnet cli или [другие варианты](#). Вы можете легко настроить среду Sisk, выполнив эту команду в вашей консоли разработчика:

```
dotnet add package Sisk.HttpServer
```

Эта команда установит последнюю версию Sisk в вашем проекте.

Native AOT Поддержка

.NET Native AOT² позволяет публиковать родные приложения .NET, которые являются самодостаточными и не требуют установки среды выполнения .NET на целевом хосте. Кроме того, Native AOT предоставляет такие преимущества, как:

- Значительно меньшие приложения
- Значительно более быстрая инициализация
- Низкое потребление памяти

Sisk Framework, благодаря своей явной природе, позволяет использовать Native AOT для几乎 всех своих функций без необходимости переработки исходного кода для адаптации его к Native AOT.

Не поддерживаемые функции

Однако Sisk использует рефлексию, хотя и минимальную, для некоторых функций. Функции, упомянутые ниже, могут быть частично доступны или полностью недоступны во время выполнения родного кода:

- [Автоматическое сканирование модулей маршрутизатора](#): этот ресурс сканирует типы, встроенные в выполняемую сборку, и регистрирует типы, которые являются [модулями маршрутизатора](#). Этот ресурс требует типов, которые могут быть исключены во время обрезки сборки.

Все остальные функции совместимы с AOT в Sisk. Обычно можно найти один или другой метод, который выдает предупреждение AOT, но тот же метод, если он не упоминается здесь, имеет перегруженную версию, которая указывает на передачу типа, параметра или информации о типе, что помогает компилятору AOT компилировать объект.

Развертывание вашего приложения Sisk

Процесс развертывания приложения Sisk состоит в том, чтобы опубликовать ваш проект в производстве. Хотя процесс относительно прост, стоит отметить детали, которые могут быть опасны для безопасности и стабильности инфраструктуры развертывания.

Идеально, вы должны быть готовы развернуть ваше приложение в облаке после проведения всех возможных тестов, чтобы ваше приложение было готово.

Публикация вашего приложения

Публикация вашего приложения Sisk или сервиса заключается в генерации бинарных файлов, готовых и оптимизированных для производства. В этом примере мы скомпилируем бинарные файлы для производства, чтобы они могли работать на машине, на которой установлен .NET Runtime.

Вам понадобится .NET SDK, установленный на вашей машине, чтобы построить ваше приложение, и .NET Runtime, установленный на целевом сервере, чтобы запустить ваше приложение. Вы можете узнать, как установить .NET Runtime на вашем Linux-сервере [здесь](#), [Windows](#) и [Mac OS](#).

В папке, где находится ваш проект, откройте терминал и используйте команду .NET publish:

```
$ dotnet publish -r linux-x64 -c Release
```

Это сгенерирует ваши бинарные файлы внутри `bin/Release/publish/linux-x64`.

NOTE

Если ваше приложение запускается с помощью пакета `Sisk.ServiceProvider`, вы должны скопировать ваш `service-config.json` на ваш хост-сервер вместе со всеми бинарными файлами, сгенерированными командой `dotnet publish`. Вы можете оставить файл предварительно настроенным, с переменными окружения, портами и хостами, а также дополнительными настройками сервера.

Следующий шаг – перенести эти файлы на сервер, где будет размещено ваше приложение.

После этого, дайте права на выполнение вашему бинарному файлу. В этом случае давайте рассмотрим, что наш проект называется "my-app":

```
1 $ cd /home/htdocs  
2 $ chmod +x my-app  
3 $ ./my-app
```

После запуска вашего приложения, проверьте, не выдает ли оно какие-либо сообщения об ошибках. Если оно не выдало, это означает, что ваше приложение работает.

На этом этапе, скорее всего, не будет возможно получить доступ к вашему приложению из внешней сети, поскольку правила доступа, такие как брандмауэр, не настроены. Мы рассмотрим это в следующих шагах.

У вас должна быть адрес виртуального хоста, на котором слушает ваше приложение. Это устанавливается вручную в приложении и зависит от того, как вы создаете экземпляр вашего сервиса Sisk.

Если вы **не** используете пакет Sisk.ServiceProvider, вы должны найти его там, где вы определили экземпляр вашего HttpServer:

```
1 HttpServer server = HttpServer.Emit(5000, out HttpServerConfiguration config, out var host,  
2 out var router);  
// sisk должен слушать на http://localhost:5000/
```

Присвоение ListeningHost вручную:

```
config.ListeningHosts.Add(new ListeningHost("https://localhost:5000/", router));
```

Или если вы используете пакет Sisk.ServiceProvider, в вашем service-config.json :

```
1 {  
2   "Server": { },  
3   "ListeningHost": {  
4     "Ports": [  
5       "http://localhost:5000/"  
6     ]  
7   }  
8 }
```

Из этого мы можем создать обратный прокси, чтобы слушать ваш сервис и сделать трафик доступным по открытой сети.

Проксирование вашего приложения

Проксирование вашего сервиса означает, что вы не 直接 подвергаете ваш сервис Sisk внешней сети. Эта практика очень распространена для серверных развертываний, потому что:

- Позволяет присвоить сертификат SSL в вашем приложении;
- Создать правила доступа перед доступом к сервису и избежать перегрузок;
- Контролировать пропускную способность и ограничения запросов;
- Отделить балансировщики нагрузки для вашего приложения;
- Предотвратить повреждение безопасности из-за неисправной инфраструктуры.

Вы можете обслуживать ваше приложение через обратный прокси, такой как [Nginx](#) или [Apache](#), или вы можете использовать туннель http-over-dns, такой как [Cloudflare](#).

Также помните, что необходимо правильно разрешить заголовки прокси для получения информации о клиенте, такой как IP-адрес и хост, через [forwarding resolvers](#).

Следующий шаг после создания туннеля, настройки брандмауэра и запуска вашего приложения – создать сервис для вашего приложения.

NOTE

Использование сертификатов SSL напрямую в сервисе Sisk на не-Windows системах невозможно. Это связано с реализацией HttpListener, который является центральным модулем для управления очередью HTTP в Sisk, и эта реализация варьируется от операционной системы к операционной системе. Вы можете использовать SSL в вашем сервисе Sisk, если [присвойте сертификат виртуальному хосту с помощью IIS](#). Для других систем использование обратного прокси рекомендуется.

Создание сервиса

Создание сервиса сделает ваше приложение всегда доступным, даже после перезапуска вашего сервера или неисправной ошибки.

В этом простом учебнике мы будем использовать содержимое из предыдущего учебника в качестве примера, чтобы ваш сервис всегда был активен.

1. Получите доступ к папке, где находятся файлы конфигурации сервиса:

```
cd /etc/systemd/system
```

2. Создайте файл `my-app.service` и включите содержимое:

my-app.service

INI

```
1 [Unit]
2 Description=<описание вашего приложения>
3
4 [Service]
5 # задайте пользователя, который будет запускать сервис
6 User=<пользователь, который будет запускать сервис>
7
8 # путь к ExecStart не относителен к WorkingDirectory.
9 # задайте его как полный путь к исполняемому файлу
10 WorkingDirectory=/home/htdocs
11 ExecStart=/home/htdocs/my-app
12
13 # задайте сервис для перезапуска после краха
14 Restart=always
15 RestartSec=3
16
17 [Install]
18 WantedBy=multi-user.target
```

3. Перезапустите модуль сервиса:

```
$ sudo systemctl daemon-reload
```

4. Запустите созданный сервис с именем файла, который вы задали, и проверьте, запущен ли он:

```
1 $ sudo systemctl start my-app
2 $ sudo systemctl status my-app
```

5. Теперь, если ваше приложение запущено ("Active: active"), включите сервис, чтобы он запускался после перезапуска системы:

```
$ sudo systemctl enable my-app
```

Теперь вы готовы представить ваше приложение Sisk всем.

Работа с SSL

Работа с SSL для разработки может быть необходима в контекстах, требующих безопасности, таких как большинство сценариев веб-разработки. Sisk работает поверх HttpListener, который не поддерживает родной HTTPS, только HTTP. Однако существуют обходные пути, которые позволяют работать с SSL в Sisk. См. их ниже:

Через IIS на Windows

- Доступно на: Windows
- Уровень сложности: средний

Если вы находитесь на Windows, вы можете использовать IIS для включения SSL на вашем HTTP-сервере. Для этого рекомендуется следовать [этому учебнику](#) заранее, если вы хотите, чтобы ваше приложение слушало хост, отличный от "localhost".

Для этого необходимо установить IIS через функции Windows. IIS доступен бесплатно для пользователей Windows и Windows Server. Чтобы настроить SSL в вашем приложении, необходимо иметь сертификат SSL, даже если он самоподписанный. Далее вы можете увидеть [как настроить SSL на IIS 7 или выше](#).

Через mitmproxy

- Доступно на: Linux, macOS, Windows
- Уровень сложности: легкий

mitmproxy – это инструмент перехвата прокси, который позволяет разработчикам и тестировщикам безопасности осматривать, изменять и записывать HTTP- и HTTPS-трафик между клиентом (таким как веб-браузер) и сервером. Вы можете использовать утилиту **mitmdump**, чтобы начать обратный SSL-прокси между вашим клиентом и приложением Sisk.

1. Сначала установите [mitmproxy](#) на вашем компьютере.
2. Запустите ваше приложение Sisk. В этом примере мы будем использовать порт 8000 в качестве не安全ного HTTP-порта.
3. Запустите сервер mitmproxy, чтобы прослушивать безопасный порт на 8001:

```
mitmdump --mode reverse:http://localhost:8000/ -p 8001
```

И вы готовы! Вы уже можете использовать ваше приложение через `https://localhost:8001/`. Вашему приложению не нужно запускаться, чтобы начать `mitmdump`.

Альтернативно, вы можете добавить ссылку на [помощник mitmproxy](#) в ваш проект. Это все равно требует, чтобы mitmproxy был установлен на вашем компьютере.

Через пакет Sisk.SslProxy

- Доступно на: Linux, macOS, Windows
- Уровень сложности: легкий

Пакет Sisk.SslProxy – это простой способ включить SSL на вашем приложении Sisk. Однако это **крайне экспериментальный** пакет. Он может быть нестабильным для работы с этим пакетом, но вы можете быть частью небольшого процента людей, которые будут вносить вклад в то, чтобы этот пакет был жизнеспособным и стабильным. Чтобы начать, вы можете установить пакет Sisk.SslProxy с:

```
dotnet add package Sisk.SslProxy
```

NOTE

Вам необходимо включить "Enable pre-release packages" в менеджере пакетов Visual Studio, чтобы установить Sisk.SslProxy.

Опять же, это экспериментальный проект, поэтому даже не думайте о том, чтобы использовать его в производстве.

На данный момент Sisk.SslProxy может обрабатывать большинство функций HTTP/1.1, включая HTTP Continue, Chunked-Encoding, WebSockets и SSE. Читайте больше о SslProxy [здесь](#).

Настройка резервирования пространств имен в Windows

Sisk работает с сетевым интерфейсом HttpListener, который связывает виртуальный хост с системой для прослушивания запросов.

В Windows это связывание немного ограничено, разрешая связывать только localhost в качестве допустимого хоста. При попытке прослушивать другой хост на сервере возникает ошибка доступа. Это руководство объясняет, как предоставить авторизацию для прослушивания на любом хосте, который вы хотите на системе.

Namespace Setup.bat

BATCH

```
1 @echo off
2
3 :: вставьте префикс здесь, без пробелов или кавычек
4 SET PREFIX=
5
6 SET DOMAIN=%ComputerName%\%USERNAME%
7 netsh http add urlacl url=%PREFIX% user=%DOMAIN%
8
9 pause
```

Где в PREFIX находится префикс ("Хост→Порт прослушивания"), который будет прослушивать ваш сервер. Он должен быть отформатирован по схеме URL, хосту, порту и слэшу в конце, например:

Namespace Setup.bat

BATCH

```
SET PREFIX=http://my-application.example.test/
```

Таким образом, вы можете прослушивать в вашем приложении через:

Program.cs

C#

```
1 class Program
2 {
3     static async Task Main(string[] args)
4     {
```

```
5     using var app = HttpServer.CreateBuilder()
6         .UseListeningPort("http://my-application.example.test/")
7         .Build();
8
9     app.Router.MapGet("/", request =>
10    {
11        return new HttpResponseMessage()
12        {
13            Status = 200,
14            Content = new StringContent("Hello, world!")
15        };
16    });
17
18    await app.StartAsync();
19 }
20 }
```

Журнал изменений

Каждое изменение, внесенное в Sisk, записывается в журнал изменений. Вы можете просмотреть журналы изменений для всех версий Sisk [здесь](#).

Часто задаваемые вопросы

Часто задаваемые вопросы о Sisk.

Является ли Sisk открытым исходным кодом?

Полностью. Всё исходный код, используемый Sisk, публикуется и регулярно обновляется на [GitHub](#).

Принимаются ли вклады?

Пока они совместимы с [философией Sisk](#), все вклады очень приветствуются! Вклады не обязательно должны быть только кодом! Вы можете внести вклад в документацию, тесты, переводы, пожертвования и публикации, например.

Является ли Sisk финансируемым?

Нет. Никакая организация или проект в настоящее время не спонсирует Sisk.

Можно ли использовать Sisk в производстве?

Определенно. Проект разрабатывается более трёх лет и прошёл интенсивное тестирование в коммерческих приложениях, которые находятся в производстве с тех пор. Sisk используется в важных

коммерческих проектах в качестве основной инфраструктуры.

Руководство по [развертыванию](#) в различных системах и средах было написано и доступно.

Имеет ли Sisk аутентификацию, мониторинг и базу данных?

Нет. Sisk не имеет ни одного из этих. Это фреймворк для разработки веб-приложений HTTP, но это всё же минимальный фреймворк, который доставляет всё необходимое для работы вашего приложения.

Вы можете реализовать все сервисы, которые вам нужны, используя любую библиотеку третьего лица, которую вы предпочитаете. Sisk был создан, чтобы быть агностическим, гибким и работать с чем угодно.

Почему я должен использовать Sisk вместо <фреймворка>?

Я не знаю. Вы мне скажите.

Sisk был создан, чтобы заполнить общий сценарий для веб-приложений HTTP в .NET. Установленные проекты, такие как ASP.NET, решают различные проблемы, но с разными предубеждениями. В отличие от более крупных фреймворков, Sisk требует от пользователя знать, что он делает и строит. Базовые понятия веб-разработки и протокола HTTP необходимы для работы с Sisk.

Sisk ближе к Express в Node.js, чем к ASP.NET Core. Это высокоуровневая абстракция, которая позволяет создавать приложения с логикой HTTP, которую вы хотите.

Что мне нужно, чтобы выучить Sisk?

Вам нужно базовое понимание:

- Веб-разработки (HTTP, Restful и т. д.)
- .NET

Всего лишь это. Имея представление о этих двух темах, вы можете посвятить несколько часов разработке продвинутого приложения с Sisk.

Можно ли разрабатывать коммерческие приложения с Sisk?

Определенно.

Sisk был создан под лицензией MIT, что означает, что вы можете использовать Sisk в любом коммерческом проекте, коммерчески или некоммерчески, без необходимости в проприетарной лицензии.

Мы просим только, чтобы где-то в вашем приложении был уведомление об использованных открытых исходных проектах, и чтобы Sisk был там.

Маршрутизация

[Router](#) является первым шагом в построении сервера. Он отвечает за хранение объектов [Route](#), которые являются конечными точками, сопоставляющими URL и их методы с действиями, выполняемыми сервером. Каждое действие отвечает за получение запроса и доставку ответа клиенту.

Маршруты представляют собой пары выражений пути ("шаблон пути") и HTTP-метода, на который они могут реагировать. Когда сервер получает запрос, он попытается найти маршрут, соответствующий полученному запросу, а затем вызовет действие этого маршрута и доставит результирующий ответ клиенту.

Существует несколько способов определить маршруты в Sisk: они могут быть статическими, динамическими или автоматически сканированными, определены атрибутами или [直接](#) в объекте Router.

```
1 Router mainRouter = new Router();
2
3 // сопоставляет GET / с следующим действием
4 mainRouter.MapGet("/", request => {
5     return new HttpResponseMessage("Hello, world!");
6 });
```

Чтобы понять, что может делать маршрут, нам нужно понять, что может делать запрос. [HttpRequest](#) будет содержать все необходимое. Sisk также включает некоторые дополнительные функции, которые ускоряют общее развитие.

Для каждого действия, полученного сервером, будет вызван делегат типа [RouteAction](#). Этот делегат содержит параметр, который держит [HttpRequest](#) со всей необходимой информацией о полученном запросе. Результирующий объект от этого делегата должен быть [HttpResponse](#) или объектом, который сопоставляется с ним через [неявные типы ответов](#).

Сопоставление маршрутов

Когда сервер получает запрос, Sisk ищет маршрут, удовлетворяющий выражению полученного пути. Это выражение всегда тестируется между маршрутом и путем запроса, без учета строки запроса.

Этот тест не имеет приоритета и исключителен для одного маршрута. Когда нет маршрута, соответствующего этому запросу, возвращается ответ [Router.NotFoundErrorHandler](#) клиенту. Когда шаблон пути совпадает, но HTTP-метод не совпадает, возвращается ответ [Router.MethodNotAllowedErrorHandler](#) клиенту.

Sisk проверяет возможность коллизий маршрутов, чтобы избежать этих проблем. Когда определяются маршруты, Sisk будет искать возможные маршруты, которые могут столкнуться с определяемым маршрутом. Этот тест включает проверку пути и метода, на который настроен маршрут.

Создание маршрутов с помощью шаблонов пути

Вы можете определить маршруты, используя различные методы `SetRoute`.

```
1 // способ SetRoute
2 mainRouter.SetRoute(RouteMethod.Get, "/hey/<name>", (request) =>
3 {
4     string name = request.RouteParameters["name"].GetString();
5     return new HttpResponseMessage($"Hello, {name}");
6 });
7
8 // способ Map*
9 mainRouter.MapGet("/form", (request) =>
10 {
11     var formData = request.GetFormData();
12     return new HttpResponseMessage(); // пустой 200 ok
13 });
14
15 // помощник методов Route.-
16 mainRouter += Route.Get("/image.png", (request) =>
17 {
18     var imageStream = File.OpenRead("image.png");
19
20     return new HttpResponseMessage()
21     {
22         // StreamContent inner
23         // stream будет disposed после отправки
24         // ответа.
25         Content = new StreamContent(imageStream)
26     };
27 });
28
29 // несколько параметров
30 mainRouter.MapGet("/hey/<name>/surname/<surname>", (request) =>
31 {
```

```
32     string name = request.RouteParameters["name"].GetString();
33     string surname = request.RouteParameters["surname"].GetString();
34
35     return new HttpResponse($"Hello, {name} {surname}!");
36 });

```

Свойство `RouteParameters` объекта `HttpResponse` содержит всю информацию о переменных пути полученного запроса.

Каждый полученный сервером путь нормализуется перед выполнением теста шаблона пути, следующим этими правилами:

- Все пустые сегменты удаляются из пути, например: `///foo//bar` становится `/foo/bar`.
- Сопоставление пути **чувствительно к регистру**, если только `Router.MatchRoutesIgnoreCase` не установлен в `true`.

Свойства `Query` и `RouteParameters` объекта `HttpRequest` возвращают объект `StringValueCollection`, где каждый индексированный свойство возвращает не-нулевой `StringValue`, который можно использовать как опцию/монаду для преобразования его сырого значения в управляемый объект.

Пример ниже читает параметр маршрута "id" и получает из него `Guid`. Если параметр не является допустимым `Guid`, выбрасывается исключение, и возвращается ошибка 500 клиенту, если сервер не обрабатывает `Router.CallbackErrorHandler`.

```
1 mainRouter.SetRoute(RouteMethod.Get, "/user/<id>", (request) =>
2 {
3     Guid id = request.RouteParameters["id"].GetGuid();
4 });

```

NOTE

Пути имеют игнорируемый завершающий `/` как в запросе, так и в пути маршрута, то есть если вы попытаетесь получить доступ к маршруту, определённому как `/index/page`, вы сможете получить доступ, используя `/index/page/` тоже.

Вы также можете принудительно завершать URL `/`, включив флаг `ForceTrailingSlash`.

Создание маршрутов с помощью экземпляров классов

Вы также можете определять маршруты динамически с помощью рефлексии и атрибута `RouteAttribute`. Таким образом, экземпляр класса, в котором его методы реализуют этот атрибут, будут иметь свои маршруты, определенные в целевом маршрутизаторе.

Чтобы метод был определен как маршрут, он должен быть помечен атрибутом `RouteAttribute`, таким как сам атрибут или `RouteGetAttribute`. Метод может быть статическим, экземпляром, публичным или приватным. Когда метод `SetObject(type)` или `SetObject<TType>()` используется, методы экземпляра игнорируются.

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      // будет соответствовать GET /
4      [RouteGet]
5      HttpResponseMessage Index(HttpServletRequest request)
6      {
7          HttpResponseMessage res = new HttpResponseMessage();
8          res.Content = new StringContent("Index!");
9          return res;
10     }
11
12     // статические методы также работают
13     [RouteGet("/hello")]
14     static HttpResponseMessage Hello(HttpServletRequest request)
15     {
16         HttpResponseMessage res = new HttpResponseMessage();
17         res.Content = new StringContent("Hello world!");
18         return res;
19     }
20 }
```

Строка ниже определит оба метода `Index` и `Hello` класса `MyController` как маршруты, поскольку они помечены как маршруты, и предоставлен экземпляр класса, а не его тип. Если бы вместо этого был предоставлен его тип, были бы определены только статические методы.

```
1  var myController = new MyController();
2  mainRouter.SetObject(myController);
```

С версии Sisk 0.16 возможно включить `AutoScan`, который будет искать пользовательские классы, реализующие `RouterModule`, и автоматически ассоциировать их с маршрутизатором. Это не поддерживается с AOT-компиляцией.

```
mainRouter.AutoScanModules<ApiController>();
```

Вышеуказанная инструкция будет искать все типы, которые реализуют ApiController, но не сам тип. Два необязательных параметра указывают, как метод будет искать эти типы. Первый аргумент подразумевает сборку, где будут искааться типы, а второй указывает, каким образом будут определены типы.

Регулярные маршруты

Вместо использования методов сопоставления пути HTTP по умолчанию вы можете пометить маршрут как интерпретируемый с помощью Regex.

```
1  Route indexRoute = new Route(RouteMethod.Get, @"\/[a-z]+\/", "My route", IndexPage, null);
2  indexRoute.UseRegex = true;
3  mainRouter.SetRoute(indexRoute);
```

Или с помощью класса [RegexRoute](#):

```
1  mainRouter.SetRoute(new RegexRoute(RouteMethod.Get, @"\/[a-z]+\/", request =>
2  {
3      return new HttpResponse("hello, world");
4  }));
5 }
```

Вы также можете захватить группы из шаблона Regex в содержимое [HttpRequest.RouteParameters](#):

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      [RegexRoute(RouteMethod.Get, "/uploads/(?<filename>.*\.(jpeg|jpg|png))")]
4      static HttpResponse RegexRoute(HttpRequest request)
5      {
6          string filename = request.RouteParameters["filename"].GetString();
7          return new HttpResponse().WithContent($"Acessing file {filename}");
8      }
9 }
```

Префиксирование маршрутов

Вы можете префиксировать все маршруты в классе или модуле с помощью атрибута `RoutePrefix` и установить префикс как строку.

Смотрите пример ниже, используя архитектуру BREAD (Browse, Read, Edit, Add и Delete):

Controller/Api/UsersController.cs

C#

```
1  [RoutePrefix("/api/users")]
2  public class UsersController
3  {
4      // GET /api/users/<id>
5      [RouteGet]
6      public async Task<HttpResponse> Browse()
7      {
8          ...
9      }
10
11     // GET /api/users
12     [RouteGet("</id>")]
13     public async Task<HttpResponse> Read()
14     {
15         ...
16     }
17
18     // PATCH /api/users/<id>
19     [RoutePatch("</id>")]
20     public async Task<HttpResponse> Edit()
21     {
22         ...
23     }
24
25     // POST /api/users
26     [RoutePost]
27     public async Task<HttpResponse> Add()
28     {
29         ...
30     }
31
32     // DELETE /api/users/<id>
33     [RouteDelete("</id>")]
34     public async Task<HttpResponse> Delete()
35     {
```

```
36         ...
37     }
38 }
```

В вышеуказанном примере параметр `HttpResponse` опущен в пользу использования через глобальный контекст `HttpContext.Current`. Читайте больше в следующем разделе.

Маршруты без параметра запроса

Маршруты можно определять без параметра `HttpRequest` и все равно можно получить запрос и его компоненты в контексте запроса. Давайте рассмотрим абстракцию `ControllerBase`, которая служит основой для всех контроллеров API, и эта абстракция предоставляет свойство `Request` для получения `HttpRequest`, в настоящее время полученного.

Controller/BaseController.cs

C#

```
1  public abstract class ControllerBase
2  {
3      // получает запрос из текущей нити
4      public HttpRequest Request { get => HttpContext.Current.Request; }
5
6      // строка ниже, когда вызывается, получает базу данных из текущей HTTP-сессии,
7      // или создает новую, если она не существует
8      public DbContext Database { get => HttpContext.Current.RequestBag.GetOrAdd<DbContext>
9          (); }
10 }
```

И для всех его потомков, чтобы они могли использовать синтаксис маршрута без параметра запроса:

Controller/UsersController.cs

C#

```
1  [RoutePrefix("/api/users")]
2  public class UsersController : ControllerBase
3  {
4      [HttpPost]
5      public async Task<HttpResponse> Create()
6      {
7          // читает JSON-данные из текущего запроса
8          UserCreationDto? user = JsonSerializer.DeserializeAsync<UserCreationDto>
```

```
9     (Request.Body);  
10    ...  
11    Database.Users.Add(user);  
12  
13    return new HttpResponseMessage(201);  
14 }  
}
```

Более подробную информацию о текущем контексте и внедрении зависимостей можно найти в учебнике [внедрение зависимостей](#).

Маршруты любого метода

Вы можете определить маршрут, чтобы он соответствовал только его пути и пропустить HTTP-метод. Это может быть полезно для проверки метода внутри маршрута.

```
1 // будет соответствовать / на любом HTTP-методе  
2 mainRouter.SetRoute(RouteMethod.Any, "/", callbackFunction);
```

Маршруты любого пути

Маршруты любого пути тестируют любой путь, полученный HTTP-сервером, подлежащий маршруту метода. Если маршрут метода равен RouteMethod.Any и маршрут использует [Route.AnyPath](#) в его выражении пути, этот маршрут будет слушать все запросы от HTTP-сервера, и не могут быть определены другие маршруты.

```
1 // следующий маршрут будет соответствовать всем запросам POST  
2 mainRouter.SetRoute(RouteMethod.Post, Route.AnyPath, callbackFunction);
```

Игнорирование регистра маршрута

По умолчанию интерпретация маршрутов с запросами чувствительна к регистру. Чтобы сделать ее игнорирующей регистр, включите эту опцию:

```
mainRouter.MatchRoutesIgnoreCase = true;
```

Это также включит опцию `RegexOptions.IgnoreCase` для маршрутов, где используется сопоставление с помощью Regex.

Обработчик не найденного (404) обратного вызова

Вы можете создать пользовательский обратный вызов для случая, когда запрос не соответствует ни одному известному маршруту.

```
1  mainRouter.NotFoundErrorHandler = () =>
2  {
3      return new HttpResponseMessage(404)
4      {
5          // С версии v0.14
6          Content = new HtmlContent("<h1>Not found</h1>")
7          // более ранние версии
8          Content = new StringContent("<h1>Not found</h1>", Encoding.UTF8, "text/html")
9      };
10 };


```

Обработчик метода не допускается (405) обратного вызова

Вы также можете создать пользовательский обратный вызов для случая, когда запрос соответствует его пути, но не соответствует методу.

```
1 mainRouter.MethodNotAllowedErrorHandler = (context) =>
2 {
3     return new HttpResponseMessage(405)
4     {
5         Content = new StringContent($"Method not allowed for this route.")
6     };
7 };
```

Внутренний обработчик ошибок

Обратные вызовы маршрутов могут выбрасывать ошибки во время выполнения сервера. Если они не обрабатываются правильно, общая работа HTTP-сервера может быть прервана. Маршрутизатор имеет обратный вызов для случая, когда обратный вызов маршрута неудачно и предотвращает прерывание службы.

Этот метод доступен только тогда, когда `ThrowExceptions` установлен в `false`.

```
1 mainRouter.CallbackErrorHandler = (ex, context) =>
2 {
3     return new HttpResponseMessage(500)
4     {
5         Content = new StringContent($"Error: {ex.Message}")
6     };
7 };
```

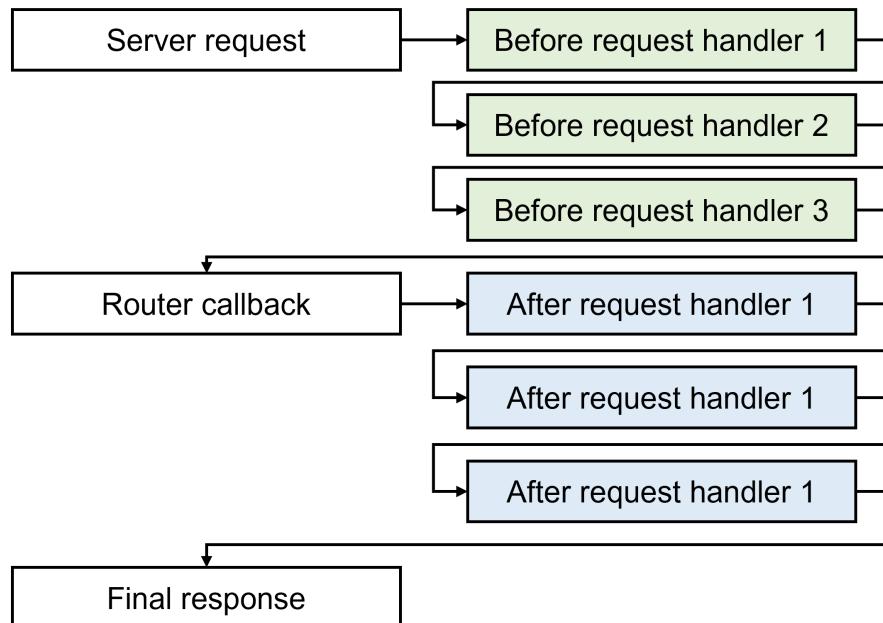
Обработка запросов

Обработчики запросов, также известные как "посредники", являются функциями, которые выполняются до или после выполнения запроса на маршрутизаторе. Они могут быть определены для каждого маршрута или для всего маршрутизатора.

Существует два типа обработчиков запросов:

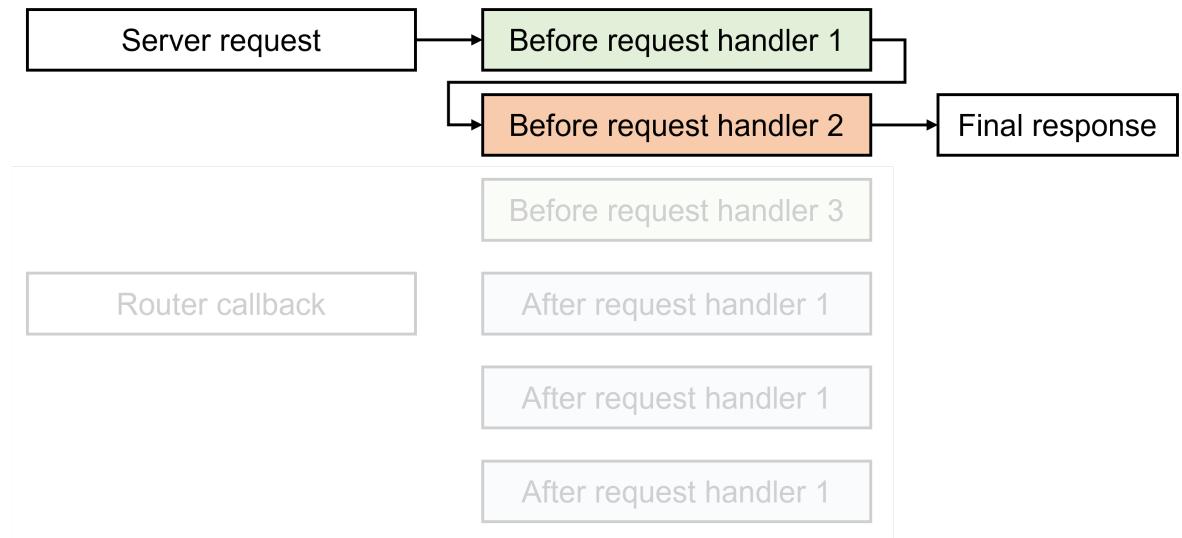
- **BeforeResponse**: определяет, что обработчик запроса будет выполнен до вызова действия маршрутизатора.
- **AfterResponse**: определяет, что обработчик запроса будет выполнен после вызова действия маршрутизатора. Отправка HTTP-ответа в этом контексте перезапишет ответ действия маршрутизатора.

Оба обработчика запросов могут переопределить фактический ответ функции обратного вызова маршрутизатора. Кроме того, обработчики запросов могут быть полезны для проверки запроса, такой как аутентификация, содержимое или любую другую информацию, такую как хранение информации, журналов или других шагов, которые можно выполнить до или после ответа.



Таким образом, обработчик запроса может прервать все это выполнение и вернуть ответ до завершения цикла, отбрасывая все остальное в процессе.

Пример: предположим, что обработчик запроса аутентификации пользователя не аутентифицирует его. Это предотвратит продолжение жизненного цикла запроса и повесит. Если это происходит в обработчике запроса на позиции два, третий и последующие не будут оценены.



Создание обработчика запроса

Чтобы создать обработчик запроса, мы можем создать класс, который наследует интерфейс `IRequestHandler`, в следующем формате:

Middleware/AuthenticateUserRequestHandler.cs

C#

```
1  public class AuthenticateUserRequestHandler : IRequestHandler
2  {
3      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
4          RequestHandlerExecutionMode.BeforeResponse;
5
6      public HttpResponse? Execute(HttpContext context)
7      {
8          if (context.Request.Headers.Authorization != null)
9          {
10              // Возвращение null указывает на то, что запрос может быть продолжен
11          }
12      }
13  }
```

```

11         return null;
12     }
13     else
14     {
15         // Возвращение объекта HttpResponse указывает на то, что этот ответ перезапишет
16         соседние ответы.
17         return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized);
18     }
}

```

В приведенном выше примере мы указали, что если заголовок `Authorization` присутствует в запросе, он должен продолжаться, и следующий обработчик запроса или функция обратного вызова маршрутизатора должна быть вызвана, в зависимости от того, что происходит дальше. Если обработчик запроса выполняется после ответа по свойству `ExecutionMode` и возвращает не-нулевое значение, он перезапишет ответ маршрутизатора.

Когда обработчик запроса возвращает `null`, это указывает на то, что запрос должен продолжаться, и следующий объект должен быть вызван, или цикл должен завершиться ответом маршрутизатора.

Связывание обработчика запроса с одним маршрутом

Вы можете определить один или несколько обработчиков запросов для маршрута.

Router.cs

C#

```

1 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage, "", new IRequestHandler[]
2 {
3     new AuthenticateUserRequestHandler(),      // до запроса обработчик
4     new ValidateJsonContentRequestHandler(), // до запроса обработчик
5     //                                         -- метод IndexPage будет выполнен здесь
6     new WriteToLogRequestHandler()           // после запроса обработчик
7 });

```

Или создавая объект `Route`:

Router.cs

C#

```
1  Route indexRoute = new Route(RouteMethod.Get, "/", "", IndexPage, null);
2  indexRoute.RequestHandlers = new IRequestHandler[]
3  {
4      new AuthenticateUserRequestHandler()
5  };
6  mainRouter.SetRoute(indexRoute);
```

Связывание обработчика запроса с маршрутизатором

Вы можете определить глобальный обработчик запроса, который будет запущен на всех маршрутах маршрутизатора.

Router.cs

C#

```
1  mainRouter.GlobalRequestHandlers = new IRequestHandler[]
2  {
3      new AuthenticateUserRequestHandler()
4  };
```

Связывание обработчика запроса с атрибутом

Вы можете определить обработчик запроса на методе атрибута вместе с атрибутом маршрута.

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      [RouteGet("/")]
4      [RequestHandler<AuthenticateUserRequestHandler>]
5      static HttpResponseMessage Index(HttpContext request)
6      {
7          return new HttpResponseMessage() {
8              Content = new StringContent("Hello world!")
9      };
10 }
```

```
10     }
11 }
```

Обратите внимание, что необходимо передать желаемый тип обработчика запроса, а не экземпляр объекта. Таким образом, обработчик запроса будет создан парсером маршрутизатора. Вы можете передать аргументы в конструктор класса с помощью свойства [ConstructorArguments](#).

Пример:

Controller/MyController.cs

C#

```
1 [RequestHandler<AuthenticateUserRequestHandler>("arg1", 123, ...)]
2 public HttpResponse Index(HttpRequest request)
3 {
4     return res = new HttpResponse() {
5         Content = new StringContent("Hello world!")
6     };
7 }
```

Вы также можете создать собственный атрибут, который реализует RequestHandler:

Middleware/Attributes/AuthenticateAttribute.cs

C#

```
1 public class AuthenticateAttribute : RequestHandlerAttribute
2 {
3     public AuthenticateAttribute() : base(typeof(AuthenticateUserRequestHandler),
4 ConstructorArguments = new object?[] { "arg1", 123, ... })
5     {
6         ;
7     }
}
```

И использовать его как:

Controller/MyController.cs

C#

```
1 [Authenticate]
2 static HttpResponse Index(HttpRequest request)
3 {
4     return res = new HttpResponse() {
5         Content = new StringContent("Hello world!")
6     };
7 }
```

Пропуск глобального обработчика запроса

После определения глобального обработчика запроса на маршруте вы можете игнорировать этот обработчик запроса на конкретных маршрутах.

Router.cs

C#

```
1  var myRequestHandler = new AuthenticateUserRequestHandler();
2  mainRouter.GlobalRequestHandlers = new IRequestHandler[]
3  {
4      myRequestHandler
5  };
6
7  mainRouter.SetRoute(new Route(RouteMethod.Get, "/", "My route", IndexPage, null)
8  {
9      BypassGlobalRequestHandlers = new IRequestHandler[]
10     {
11         myRequestHandler, // ок: тот же экземпляр, что и в глобальных
12         // обработчиках запросов
13         new AuthenticateUserRequestHandler() // неправильно: не пропустит глобальный
14         // обработчик запроса
15     }
16 });
17 );
```

NOTE

Если вы пропускаете обработчик запроса, вы должны использовать тот же ссылку на то, что было создано ранее, чтобы пропустить. Создание другого экземпляра обработчика запроса не пропустит глобальный обработчик запроса, поскольку ссылка изменится. Помните, что необходимо использовать ту же ссылку на обработчик запроса, которая используется как в GlobalRequestHandlers, так и в BypassGlobalRequestHandlers.

Запросы

Запросы — это структуры, представляющие сообщение HTTP-запроса. Объект [HttpRequest](#) содержит полезные функции для обработки HTTP-сообщений во всей вашей программе.

HTTP-запрос формируется методом, путем, версией, заголовками и телом.

В этом документе мы покажем, как получить каждый из этих элементов.

Получение метода запроса

Чтобы получить метод полученного запроса, можно использовать свойство `Method`:

```
1 static HttpResponse Index(HttpServletRequest request)
2 {
3     HttpMethod requestMethod = request.Method;
4     ...
5 }
```

Это свойство возвращает метод запроса, представленный объектом [HttpMethod](#).

NOTE

В отличие от методов маршрута, это свойство не обслуживает элемент `RouteMethod.Any`. Вместо этого оно возвращает реальный метод запроса.

Получение компонентов URL запроса

Вы можете получить различные компоненты из URL через определенные свойства запроса. Для этого примера рассмотрим URL:

```
http://localhost:5000/user/login?email=foo@bar.com
```

| Название компонента | Описание | Значение компонента |
|---------------------|--|--|
| Path | Получает путь запроса. | /user/login |
| FullPath | Получает путь запроса и строку запроса. | /user/login?email=foo@bar.com |
| FullUrl | Получает всю строку URL запроса. | http://localhost:5000/user/login? email=foo@bar.com |
| Host | Получает хост запроса. | localhost |
| Authority | Получает хост и порт запроса. | localhost:5000 |
| QueryString | Получает запрос. | ?email=foo@bar.com |
| Query | Получает запрос в виде коллекции именованных значений. | {StringValueCollection object} |
| IsSecure | Определяет, использует ли запрос SSL (true) или нет (false). | false |

Вы также можете воспользоваться свойством [HttpRequest.Uri](#), которое включает всё вышеупомянутое в одном объекте.

Получение тела запроса

Некоторые запросы включают тело, такое как формы, файлы или транзакции API. Вы можете получить тело запроса из свойства:

```
1 // получает тело запроса как строку, используя кодировку запроса в качестве кодировщика
2 string body = request.Body;
3
4 // или получает его в виде массива байт
5 byte[] bodyBytes = request.RawBody;
6
```

```
7 // или, наоборот, вы можете потоково читать его.  
8 Stream requestStream = request.GetRequestStream();
```

Также возможно определить, есть ли тело в запросе и загружено ли оно, с помощью свойств [HasContents](#), которое определяет, есть ли у запроса содержимое, и [IsContentAvailable](#), которое указывает, что HTTP-сервер полностью получил содержимое от удалённой точки.

Невозможно прочитать содержимое запроса через `GetRequestStream` более одного раза. Если вы прочитаете с помощью этого метода, значения в `RawBody` и `Body` также не будут доступны. Нет необходимости освобождать поток запроса в контексте запроса, так как он освобождается в конце HTTP-сессии, в которой он создан. Кроме того, вы можете использовать свойство [HttpRequest.RequestEncoding](#), чтобы получить лучшую кодировку для ручного декодирования запроса.

Сервер имеет ограничения на чтение содержимого запроса, которые применяются как к [HttpRequest.Body](#), так и к [HttpRequest.RawBody](#). Эти свойства копируют весь входной поток в локальный буфер того же размера, что и [HttpRequest.ContentLength](#).

Ответ со статусом 413 Content Too Large возвращается клиенту, если отправленное содержимое превышает [HttpServerConfiguration.MaximumContentSize](#), определённое в пользовательской конфигурации. Кроме того, если нет настроенного ограничения или оно слишком велико, сервер выбросит [OutOfMemoryException](#), когда содержимое, отправленное клиентом, превышает [Int32.MaxValue](#) (2 ГБ), и если содержимое попытаться получить через одно из упомянутых выше свойств. Вы всё равно можете работать с содержимым через поток.

NOTE

Хотя Sisk позволяет это, всегда полезно следовать HTTP-семантике при создании вашего приложения и не получать или обслуживать контент в методах, которые этого не допускают. Читайте о [RFC 9110 "HTTP Semantics"](#).

Получение контекста запроса

HTTP Context — это эксклюзивный объект Sisk, который хранит информацию о сервере, маршруте, роутере и обработчике запроса. Вы можете использовать его, чтобы организовать себя в среде, где эти объекты трудно организовать.

Объект [RequestBag](#) содержит сохранённую информацию, которая передаётся от обработчика запроса к другому пункту, и может быть использована в конечном пункте. Этот объект также может

использоваться обработчиками запросов, которые выполняются после обратного вызова маршрута.

(i) TIP

Это свойство также доступно через свойство [HttpRequest.Bag](#).

Middleware/AuthenticateUserRequestHandler.cs

C#

```
1  public class AuthenticateUserRequestHandler : IRequestHandler
2  {
3      public string Identifier { get; init; } = Guid.NewGuid().ToString();
4      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
5 RequestHandlerExecutionMode.BeforeResponse;
6
7      public HttpResponse? Execute(HttpRequest request, HttpContext context)
8      {
9          if (request.Headers.Authorization != null)
10         {
11             context.RequestBag.Add("AuthenticatedUser", new User("Bob"));
12             return null;
13         }
14         else
15         {
16             return new HttpResponse(System.Net.HttpStatusCode.Unauthorized);
17         }
18     }
19 }
```

Вышеуказанный обработчик запроса определит `AuthenticatedUser` в мешке запроса и может быть использован позже в конечном обратном вызове:

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      [RouteGet("/")]
4      [RequestHandler<AuthenticateUserRequestHandler>]
5      static HttpResponse Index(HttpRequest request)
6      {
7          User authUser = request.Context.RequestBag["AuthenticatedUser"];
8
9          return new HttpResponse() {
10              Content = new StringContent($"Hello, {authUser.Name}!")
11          };
12 }
```

```
12     }
13 }
```

Вы также можете использовать вспомогательные методы `Bag.Set()` и `Bag.Get()` для получения или установки объектов по их типу-единственникам.

Middleware/Authenticate.cs

C#

```
1 public class Authenticate : RequestHandler
2 {
3     public override HttpResponse? Execute(HttpContext context)
4     {
5         request.Bag.Set<User>(authUser);
6     }
7 }
```

Controller/MyController.cs

C#

```
1 [RouteGet("/")]
2 [RequestHandler<Authenticate>]
3 public static HttpResponse GetUser(HttpContext request)
4 {
5     var user = request.Bag.Get<User>();
6     ...
7 }
```

Получение данных формы

Вы можете получить значения данных формы в [NameValueCollection](#) с примером ниже:

Controller/Auth.cs

C#

```
1 [RoutePost("/auth")]
2 public HttpResponse Index(HttpContext request)
3 {
4     var form = request.GetFormContent();
5
6     string? username = form["username"];
```

```
7     string? password = form["password"];
8
9     if (AttemptLogin(username, password))
10    {
11        ...
12    }
13 }
```

Получение многокомпонентных данных формы

HTTP-запрос Sisk позволяет получить загруженные многокомпонентные содержимое, такие как файлы, поля формы или любой бинарный контент.

Controller/Auth.cs

C#

```
1  [RoutePost("/upload-contents")]
2  public HttpResponse Index(HttpRequest request)
3  {
4      // следующий метод читает весь входной запрос в
5      // массив MultipartObjects
6      var multipartFormDataObjects = request.GetMultipartFormContent();
7
8      foreach (MultipartObject uploadedObject in multipartFormDataObjects)
9      {
10          // Имя файла, предоставленное данными формы Multipart.
11          // Возвращается Null, если объект не является файлом.
12          Console.WriteLine("File name : " + uploadedObject.Filename);
13
14          // Имя поля объекта данных формы Multipart.
15          Console.WriteLine("Field name : " + uploadedObject.Name);
16
17          // Длина содержимого данных формы Multipart.
18          Console.WriteLine("Content length : " + uploadedObject.ContentLength);
19
20          // Определить формат изображения, основанный на заголовке файла для каждого
21          // известного типа содержимого. Если содержимое не является распознанным
22          общим файлом,
23          // этот метод ниже вернёт MultipartObjectCommonFormat.Unknown
24          Console.WriteLine("Common format : " + uploadedObject.GetCommonFileFormat());
25 }
```

```
    }  
}
```

Вы можете прочитать больше о [Multipart form objects](#) и их методах, свойствах и функциональностях.

Обнаружение отключения клиента

Начиная с версии v1.15 Sisk предоставляет CancellationToken, который выбрасывается, когда соединение между клиентом и сервером преждевременно закрывается до получения ответа. Этот токен может быть полезен для обнаружения, когда клиент больше не хочет ответа и отмены длительных операций.

```
1   router.MapGet("/connect", async (HttpRequest req) =>  
2   {  
3       // получает токен отключения из запроса  
4       var dc = req.DisconnectToken;  
5  
6       await LongOperationAsync(dc);  
7  
8       return new HttpResponseMessage();  
9   });
```

Этот токен не совместим со всеми HTTP-движками, и каждый требует реализации.

Поддержка событий, отправляемых сервером

Sisk поддерживает [Server-sent events](#), которые позволяют отправлять куски как поток и держать соединение между сервером и клиентом живым.

Вызов метода `HttpRequest.GetEventSource` поместит `HttpRequest` в его состояние слушателя. Отсюда контекст этого HTTP-запроса не будет ожидать `HttpResponse`, так как он будет перекрывать пакеты, отправляемые серверными событиями.

После отправки всех пакетов, обратный вызов должен вернуть метод `Close`, который отправит окончательный ответ серверу и укажет, что поток завершён.

Невозможно предсказать, какой будет общая длина всех пакетов, которые будут отправлены, поэтому невозможно определить конец соединения с заголовком Content-Length .

По умолчанию большинство браузеров не поддерживают отправку HTTP-заголовков или методов, отличных от GET. Поэтому будьте осторожны при использовании обработчиков запросов с запросами event-source, которые требуют конкретных заголовков в запросе, так как они, вероятно, не будут иметь их.

Кроме того, большинство браузеров перезапускают потоки, если метод [EventSource.close](#) не вызывается на стороне клиента после получения всех пакетов, вызывая бесконечную дополнительную обработку на стороне сервера. Чтобы избежать такой проблемы, обычно отправляется окончательный пакет, указывающий, что источник событий завершил отправку всех пакетов.

Ниже приведён пример, показывающий, как браузер может сообщить серверу, поддерживающему Server-side events.

sse-example.html HTML

```
1  <html>
2      <body>
3          <b>Fruits:</b>
4          <ul></ul>
5      </body>
6      <script>
7          const evtSource = new EventSource('http://localhost:5555/event-source');
8          const eventList = document.querySelector('ul');
9
10         evtSource.onmessage = (e) => {
11             const newElement = document.createElement("li");
12
13             newElement.textContent = `message: ${e.data}`;
14             eventList.appendChild(newElement);
15
16             if (e.data == "Tomato") {
17                 evtSource.close();
18             }
19         }
20     </script>
21 </html>
```

И постепенно отправлять сообщения клиенту:

Controller/MyController.cs C#

```
1  public class MyController
2  {
3      [RouteGet( "/event-source")]
4      public async Task<HttpResponse> ServerEventsResponse( HttpRequest request)
5      {
6          var sse = await request.GetEventSourceAsync();
7
8          string[] fruits = new[] { "Apple", "Banana", "Watermelon", "Tomato" };
9
10         foreach (string fruit in fruits)
11         {
12             await serverEvents.SendAsync(fruit);
13             await Task.Delay(1500);
14         }
15
16         return serverEvents.Close();
17     }
18 }
```

При запуске этого кода ожидается результат, похожий на это:



Разрешение проксируемых IP и хостов

Sisk можно использовать с прокси, и поэтому IP-адреса могут быть заменены прокси-конечной точкой в транзакции от клиента к прокси.

Вы можете определить собственные резолверы в Sisk с [forwarding resolvers](#).

Кодирование заголовков

Кодирование заголовков может быть проблемой для некоторых реализаций. В Windows заголовки UTF-8 не поддерживаются, поэтому используется ASCII. Sisk имеет встроенный конвертер кодировок, который может быть полезен для декодирования некорректно закодированных заголовков.

Эта операция затратна и отключена по умолчанию, но может быть включена под флагом [NormalizeHeadersEncodings](#).

Responses

Ответы представляют собой объекты, которые являются HTTP-ответами на HTTP-запросы. Они отправляются сервером клиенту в качестве указания на запрос ресурса, страницы, документа, файла или другого объекта.

HTTP-ответ состоит из статуса, заголовков и содержимого.

В этом документе мы научим вас, как архитектурировать HTTP-ответы с помощью Sisk.

Установка HTTP-статуса

Список HTTP-статусов одинаков с момента появления HTTP/1.0, и Sisk поддерживает все из них.

```
1  HttpResponse res = new HttpResponse();
2  res.Status = System.Net HttpStatusCode.Accepted; //202
```

Или с помощью Fluent Syntax:

```
1  new HttpResponseMessage()
2  .WithStatus(200) // или
3  .WithStatus(HttpStatusCode.OK) // или
4  .WithStatus(HttpStatusCode.Information.OK);
```

Вы можете просмотреть полный список доступных HttpStatusCode [здесь](#). Вы также можете указать свой собственный код статуса, используя структуру [HttpStatusInformation](#).

Тело и тип содержимого

Sisk поддерживает родные объекты содержимого .NET для отправки тела в ответах. Вы можете использовать класс [StringContent](#) для отправки JSON-ответа, например:

```
1  HttpResponseMessage res = new HttpResponseMessage();
2  res.Content = new StringContent(myJson, Encoding.UTF8, "application/json");
```

Сервер всегда попытается рассчитать Content-Length из того, что вы определили в содержимом, если вы явно не определили его в заголовке. Если сервер не может явно получить заголовок Content-Length из содержимого ответа, ответ будет отправлен с Chunked-Encoding.

Вы также можете передавать ответ, отправляя [StreamContent](#) или используя метод [GetResponseStream](#).

Заголовки ответа

Вы можете добавлять, редактировать или удалять заголовки, которые отправляются в ответе. Пример ниже показывает, как отправить ответ с перенаправлением клиенту.

```
1  HttpResponseMessage res = new HttpResponseMessage();
2  res.StatusCode = HttpStatusCode.Moved;
3  res.Headers.Add(HttpKnownHeaderNames.Location, "/login");
```

Или с помощью Fluent Syntax:

```
1  new HttpResponseMessage(301)
2  .WithHeader("Location", "/login");
```

Когда вы используете метод [Add](#) коллекции [HttpHeaderCollection](#), вы добавляете заголовок к запросу, не изменяя уже отправленные. Метод [Set](#) заменяет заголовки с тем же именем на указанное значение. Индексатор [HttpHeaderCollection](#) внутренне вызывает метод [Set](#) для замены заголовков.

Отправка куки

Sisk имеет методы, которые облегчают определение куки на клиенте. Куки, установленные этим методом, уже закодированы URL и соответствуют стандарту RFC-6265.

```
1  HttpResponseMessage res = new HttpResponseMessage();
2  res.SetCookie("cookie-name", "cookie-value");
```

Или с помощью Fluent Syntax:

```
1  new HttpResponseMessage(301)
2  .WithCookie("cookie-name", "cookie-value", expiresAt:
DateTime.Now.Add(TimeSpan.FromDays(7)));
```

Имеются другие [более полные версии](#) того же метода.

Частичные ответы

Вы можете установить тип кодирования передачи на частичный для отправки больших ответов.

```
1  HttpResponseMessage res = new HttpResponseMessage();
2  res.SendChunked = true;
```

При использовании chunked-encoding заголовок Content-Length автоматически опускается.

Поток ответа

Потоки ответа - это управляемый способ, который позволяет отправлять ответы в сегментированном виде. Это более низкоуровневая операция, чем использование объектов HttpResponseMessage, поскольку они требуют от вас отправки заголовков и содержимого вручную, а затем закрытия соединения.

Этот пример открывает поток только для чтения для файла, копирует поток в выходной поток ответа и не загружает весь файл в память. Это может быть полезно для обслуживания средних или больших файлов.

```

1 // получает выходной поток ответа
2 using var fileStream = File.OpenRead("my-big-file.zip");
3 var responseStream = request.GetResponseStream();
4
5 // устанавливает кодирование ответа для использования chunked-encoding
6 // также не следует отправлять заголовок content-length при использовании
7 // chunked encoding
8 responseStream.SendChunked = true;
9 responseStream.SetStatus(200);
10 responseStream.SetHeader(HttpKnownHeaderNames.ContentType, contentType);
11
12 // копирует поток файла в выходной поток ответа
13 fileStream.CopyTo(responseStream.ResponseStream);
14
15 // закрывает поток
16 return responseStream.Close();

```

Сжатие GZip, Deflate и Brotli

Вы можете отправлять ответы со сжатым содержимым в Sisk с помощью сжатия HTTP-содержимого. Во-первых, инкапсулируйте ваш объект [HttpContent](#) внутри одного из компрессоров ниже, чтобы отправить сжатый ответ клиенту.

```

1 router.MapGet("/hello.html", request => {
2     string myHtml = "...";
3
4     return new HttpResponse () {
5         Content = new GZipContent(new HtmlContent(myHtml)),
6         // или Content = new BrotliContent(new HtmlContent(myHtml)),
7         // или Content = new DeflateContent(new HtmlContent(myHtml)),
8     };
9 });

```

Вы также можете использовать эти сжатые содержимое с потоками.

```

1 router.MapGet("/archive.zip", request => {
2
3     // не применяйте "using" здесь. HttpServer отклонит ваше содержимое
4     // после отправки ответа.

```

```
5 var archive = File.OpenRead("/path/to/big-file.zip");
6
7     return new HttpResponseMessage () {
8         Content = new GZipContent(archive)
9     }
10 } );
```

Заголовки Content-Encoding устанавливаются автоматически при использовании этих содержимостей.

Автоматическое сжатие

Возможно автоматически сжимать HTTP-ответы с помощью свойства [EnableAutomaticResponseCompression](#). Это свойство автоматически инкапсулирует содержимое ответа от маршрутизатора в сжимаемое содержимое, которое принимается запросом, при условии, что ответ не унаследован от [CompressedContent](#).

Только одно сжимаемое содержимое выбирается для запроса, выбранное в соответствии с заголовком Accept-Encoding, который следует порядку:

- [BrotliContent](#) (br)
- [GZipContent](#) (gzip)
- [DeflateContent](#) (deflate)

Если запрос указывает, что он принимает любой из этих методов сжатия, ответ будет автоматически сжат.

Неявные типы ответов

Вы можете использовать другие типы возвращаемых значений, кроме HttpResponseMessage, но необходимо настроить маршрутизатор, чтобы он обрабатывал каждый тип объекта.

Концепция состоит в том, чтобы всегда возвращать ссылочный тип и преобразовывать его в допустимый объект HttpResponseMessage. Маршруты, которые возвращают HttpResponseMessage, не подвергаются никаким преобразованиям.

Типы значений (структуры) не могут быть использованы в качестве типа возвращаемого значения, потому что они не совместимы с [RouterCallback](#), поэтому они должны быть обернуты в [ValueResult](#), чтобы иметь возможность использоваться в обработчиках.

Рассмотрим следующий пример из модуля маршрутизатора, не использующего [HttpResponse](#) в качестве типа возвращаемого значения:

```
1 [RoutePrefix("/users")]
2 public class UsersController : RouterModule
3 {
4     public List<User> Users = new List<User>();
5
6     [RouteGet]
7     public IEnumerable<User> Index(HttpContext request)
8     {
9         return Users.ToArray();
10    }
11
12    [RouteGet("<id>")]
13    public User View(HttpContext request)
14    {
15        int id = request.RouteParameters["id"].GetInteger();
16        User dUser = Users.First(u => u.Id == id);
17
18        return dUser;
19    }
20
21    [RoutePost]
22    public ValueResult<bool> Create(HttpContext request)
23    {
24        User fromBody = JsonSerializer.Deserialize<User>(request.Body)!;
25        Users.Add(fromBody);
26
27        return true;
28    }
29 }
```

При этом теперь необходимо определить в маршрутизаторе, как он будет обрабатывать каждый тип объекта. Объекты всегда являются первым аргументом обработчика, а тип вывода должен быть допустимым объектом [HttpResponse](#). Также выходные объекты маршрута никогда не должны быть null.

Для типов [ValueResult](#) не обязательно указывать, что входной объект является [ValueResult](#), и только T, поскольку [ValueResult](#) является объектом, отраженным от его исходного компонента.

Ассоциация типов не сравнивает то, что было зарегистрировано, с типом объекта, возвращаемого из обратного вызова маршрутизатора. Вместо этого она проверяет, является ли тип результата маршрутизатора присваиваемым зарегистрированному типу.

Регистрация обработчика типа `Object` будет резервным для всех ранее не проверенных типов. Порядок вставки обработчиков значений также имеет значение, поэтому регистрация обработчика `Object` проигнорирует все другие обработчики, специфичные для типов. Всегда регистрируйте обработчики значений сначала, чтобы обеспечить порядок.

```
1 Router r = new Router();
2 r.SetObject(new UsersController());
3
4 r.RegisterValueHandler<ApiResult>(apiResult =>
5 {
6     return new HttpResponseMessage() {
7         Status = apiResult.Success ? HttpStatusCode.OK : HttpStatusCode.BadRequest,
8         Content = apiResult.GetHttpContent(),
9         Headers = apiResult.GetHeaders()
10    };
11 });
12 r.RegisterValueHandler<bool>(bvalue =>
13 {
14     return new HttpResponseMessage() {
15         Status = bvalue ? HttpStatusCode.OK : HttpStatusCode.BadRequest
16     };
17 });
18 r.RegisterValueHandler<IEnumerable<object>>(enumerableValue =>
19 {
20     return new HttpResponseMessage(string.Join("\n", enumerableValue));
21 });
22
23 // регистрация обработчика значений объекта должна быть последней
24 // обработчиком значений, который будет использоваться в качестве резервного
25 r.RegisterValueHandler<object>(fallback =>
26 {
27     return new HttpResponseMessage() {
28         Status = HttpStatusCode.OK,
29         Content = JsonContent.Create(fallback)
30     };
31 });
```

Примечание о перечислимых объектах и массивах

Неявные объекты ответа, реализующие [IEnumerable](#), читаются в память с помощью метода `ToArray()` перед преобразованием с помощью определенного обработчика значений. Чтобы это произошло, объект `IEnumerable` преобразуется в массив объектов, и преобразователь ответа всегда получит `Object[]` вместо исходного типа.

Рассмотрим следующий сценарий:

```
1  using var host = HttpServer.CreateBuilder(12300)
2    .UseRouter(r =>
3    {
4      r.RegisterValueHandler<IEnumerable<string>>(stringEnumerable =>
5      {
6        return new HttpResponseMessage("Массив строк:\n" + string.Join("\n", stringEnumerable));
7      });
8      r.RegisterValueHandler<IEnumerable<object>>(stringEnumerable =>
9      {
10        return new HttpResponseMessage("Массив объектов:\n" + string.Join("\n", stringEnumerable));
11      });
12      r.MapGet("/", request =>
13      {
14        return (IEnumerable<string>)["hello", "world"];
15      });
16    })
17    .Build();
```

В приведенном выше примере преобразователь `IEnumerable<string>` **никогда не будет вызван**, потому что входной объект всегда будет массивом `Object[]` и не может быть преобразован в `IEnumerable<string>`. Однако преобразователь ниже, который получает `IEnumerable<object>`, получит свой вход, поскольку его значение совместимо.

Если вам нужно фактически обрабатывать тип объекта, который будет перечислен, вам нужно использовать отражение, чтобы получить тип элемента коллекции. Все перечислимые объекты (списки, массивы и коллекции) преобразуются в массив объектов преобразователем HTTP-ответа.

Значения, реализующие [IAsyncEnumerable](#), обрабатываются автоматически сервером, если включено свойство `ConvertIAsyncEnumerableIntoEnumerable`, подобно тому, что происходит с `IEnumerable`. Асинхронная перечисляемая последовательность преобразуется в блокирующий перечислитель, а затем преобразуется в синхронный массив объектов.

Logging

Вы можете настроить Sisk для автоматического записи журналов доступа и ошибок. Можно определить ротацию журналов, расширения и частоту.

Класс [LogStream](#) предоставляет асинхронный способ записи журналов и их хранение в ожидаемой очереди записи.

В этой статье мы покажем, как настроить логирование для вашего приложения.

Журналы доступа на основе файлов

Журналы в файлы открывают файл, записывают строку текста, а затем закрывают файл для каждой записанной строки. Эта процедура была принята для поддержания отзывчивости записи в журналах.

Program.cs

C#

```
1  class Program
2  {
3      static async Task Main(string[] args)
4      {
5          using var app = HttpServer.CreateBuilder()
6              .UseConfiguration(config => {
7                  config.AccessLogsStream = new LogStream("logs/access.log");
8              })
9              .Build();
10
11         ...
12
13         await app.StartAsync();
14     }
15 }
```

Вышеуказанный код будет записывать все входящие запросы в файл `logs/access.log`. Обратите внимание, что файл создаётся автоматически, если он не существует, однако папка перед ним не

создаётся. Создавать каталог `logs/` вручную не требуется, так как класс `LogStream` автоматически его создаёт.

Логирование на основе потока

Вы можете записывать журналы в объекты `TextWriter`, такие как `Console.Out`, передав объект `TextWriter` в конструктор:

Program.cs

C#

```
1  using var app = HttpServer.CreateBuilder()
2      .UseConfiguration(config => {
3          config.AccessLogsStream = new LogStream(Console.Out);
4      })
5      .Build();
```

Для каждой записи в потоковом журнале вызывается метод `TextWriter.Flush()`.

Форматирование журнала доступа

Вы можете настроить формат журнала доступа с помощью предопределённых переменных. Рассмотрим следующую строку:

```
config.AccessLogsFormat = "%dd/%dmm/%dy %tH:%ti:%ts %tz %ls %ri %rs://%ra%rz%rq [%sc %sd] %lin -
> %lou in %lmsms [{user-agent}];
```

Она будет записывать сообщение вида:

```
29/mar./2023 15:21:47 -0300 Executed ::1 http://localhost:5555/ [200 OK] 689B → 707B in 84ms
[Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/111.0.0.0 Safari/537.36]
```

Вы можете форматировать ваш файл журнала по описанному в таблице формату:

| Value | Что это представляет | Пример |
|-------|---|--|
| %dd | День месяца (форматирован как два числа) | 05 |
| %dmmm | Полное название месяца | July |
| %dmm | Сокращённое название месяца (три буквы) | Jul |
| %dm | Номер месяца (форматирован как два числа) | 07 |
| %dy | Год (форматирован как четыре числа) | 2023 |
| %th | Час в 12-часовом формате | 03 |
| %tH | Час в 24-часовом формате (HH) | 15 |
| %ti | Минуты (форматирован как два числа) | 30 |
| %ts | Секунды (форматирован как два числа) | 45 |
| %tm | Миллисекунды (форматирован как три числа) | 123 |
| %tz | Смещение часового пояса (полные часы в UTC) | +03:00 |
| %ri | Удалённый IP-адрес клиента | 192.168.1.100 |
| %rm | HTTP-метод (в верхнем регистре) | GET |
| %rs | Схема URI (http/https) | https |
| %ra | Авторитет URI (домен) | example.com |
| %rh | Хост запроса | www.example.com ↗ |
| %rp | Порт запроса | 443 |
| %rz | Путь запроса | /path/to/resource |
| %rq | Строка запроса | ?key=value&another=123 |
| %sc | Код статуса HTTP ответа | 200 |
| %sd | Описание статуса HTTP ответа | OK |
| %lin | Человеко-читабельный размер запроса | 1.2 KB |
| %linr | Сырой размер запроса (байты) | 1234 |

| Value | Что это представляет | Пример |
|----------------|--|---|
| %lou | Человеко-читабельный размер ответа | 2.5 KB |
| %lour | Сырой размер ответа (байты) | 2560 |
| %lms | Время выполнения в миллисекундах | 120 |
| %ls | Статус выполнения | Executed |
| %{header-name} | Представляет заголовок <code>header-name</code> запроса. | Mozilla/5.0 (platform; rv:gecko [...] |
| %{:res-name} | Представляет заголовок <code>res-name</code> ответа. | |

Ротация журналов

Вы можете настроить HTTP-сервер для ротации файлов журналов в сжатый файл .gz, когда они достигают определённого размера. Размер проверяется периодически по порогу, который вы определяете.

```

1 LogStream errorLog = new LogStream("logs/error.log")
2     .ConfigureRotatingPolicy(
3         maximumSize: 64 * SizeHelper.UnitMb,
4         dueTime: TimeSpan.FromHours(6));

```

Вышеуказанный код будет проверять каждые шесть часов, достиг ли файл LogStream своего лимита в 64 МБ. Если да, файл сжимается в .gz файл, а затем access.log очищается.

Во время этого процесса запись в файл блокируется до тех пор, пока файл не будет сжат и очищен. Все строки, которые попадают в запись в этот период, находятся в очереди, ожидающей завершения сжатия.

Эта функция работает только с файловыми LogStreams.

Логирование ошибок

Когда сервер не генерирует ошибки в отладчик, он перенаправляет ошибки в запись журнала, если они есть. Вы можете настроить запись ошибок с помощью:

```
1 config.ThrowExceptions = false;
2 config.ErrorsLogsStream = new LogStream("error.log");
```

Это свойство будет записывать что-то в журнал только если ошибка не захвачена обратным вызовом или свойством [Router.CallbackErrorHandler](#).

Запись ошибки сервером всегда включает дату и время, заголовки запроса (не тело), трассировку ошибки и трассировку внутреннего исключения, если таковые имеются.

Другие экземпляры логирования

Ваше приложение может иметь ноль или несколько LogStreams, нет ограничения на количество каналов журналов. Поэтому возможно перенаправить журнал вашего приложения в файл, отличный от стандартного AccessLog или ErrorLog.

```
1 LogStream appMessages = new LogStream("messages.log");
2 appMessages.WriteLine("Application started at {0}", DateTime.Now);
```

Расширение LogStream

Вы можете расширить класс `LogStream`, чтобы писать пользовательские форматы, совместимые с текущим движком журналов Sisk. Пример ниже позволяет писать цветные сообщения в консоль через библиотеку Spectre.Console:

```
1 public class CustomLogStream : LogStream
2 {
3     protected override void WriteLineInternal(string line)
4     {
5         base.WriteLineInternal($"[{DateTime.Now:g}] {line}");
6     }
7 }
```

Другой способ автоматически писать пользовательские журналы для каждого запроса/ответа – создать `HttpServerHandler`. Пример ниже чуть более полный. Он записывает тело запроса и ответа в JSON в консоль. Это может быть полезно для отладки запросов в целом. В этом примере используется `ContextBag` и `HttpServerHandler`.

Program.cs

C#

```
1 class Program
2 {
3     static async Task Main(string[] args)
4     {
5         var app = HttpServer.CreateBuilder(host =>
6         {
7             host.UseListeningPort(5555);
8             host.UseHandler<JsonMessageHandler>();
9         });
10
11         app.Router += new Route(RouteMethod.Any, "/json", request =>
12         {
13             return new HttpResponseMessage()
14                 .WithContent(JsonContent.Create(new
15                 {
16                     method = request.Method.Method,
17                     path = request.Path,
18                     specialMessage = "Hello, world!!"
19                 }));
20         });
21
22         await app.StartAsync();
23     }
24 }
```

JsonMessageHandler.cs

C#

```
1 class JsonMessageHandler : HttpServerHandler
2 {
```

```

3     protected override void OnHttpRequestOpen(HttpRequest request)
4     {
5         if (request.Method != HttpMethod.Get && request.Headers["Content-
6             Type"]?.Contains("json", StringComparison.InvariantCultureIgnoreCase) = true)
7         {
8             // На этом этапе соединение открыто, и клиент отправил заголовок, указывающий,
9             // что контент является JSON. Ниже строка читает контент и оставляет его
10            // в запросе.
11            //
12            // Если контент не будет прочитан в действии запроса, GC, скорее всего,
13            // соберёт контент после отправки ответа клиенту, поэтому контент
14            // может быть недоступен после закрытия ответа.
15            //
16            _ = request.RawBody;
17
18            // добавляем подсказку в контекст, чтобы указать, что этот запрос
19            // имеет JSON-тело
20            request.Bag.Add("IsJsonRequest", true);
21        }
22    }
23
24    protected override async void OnHttpRequestClose(HttpServerExecutionResult result)
25    {
26        string? requestJson = null,
27                    responseJson = null,
28                    responseMessage;
29
30        if (result.Request.Bag.ContainsKey("IsJsonRequest"))
31        {
32            // переформатирует JSON с помощью библиотеки CypherPotato.LightJson
33            var content = result.Request.Body;
34            requestJson = JsonValue.Deserialize(content, new JsonOptions() { WriteIndented
35 = true }).ToString();
36        }
37
38        if (result.Response is {} response)
39        {
40            var content = response.Content;
41            responseMessage = $"{(int)response.Status}
42 {HttpStatusInformation.GetStatusCodeDescription(response.Status)}";
43
44            if (content is HttpContent httpContent &&
45                // проверяем, является ли ответ JSON
46                httpContent.Headers.ContentType?.MediaType?.Contains("json",
47 StringComparison.InvariantCultureIgnoreCase) = true)
48            {
49                string json = await httpContent.ReadAsStringAsync();
50                responseJson = JsonValue.Deserialize(json, new JsonOptions() {

```

```
51     WriteIndented = true }).ToString();
52 }
53 }
54 else
55 {
56     // получает внутренний статус обработки сервера
57     responseMessage = result.Status.ToString();
58 }
59
60 StringBuilder outputMessage = new StringBuilder();
61
62 if (requestJson != null)
63 {
64     outputMessage.AppendLine("-----");
65     outputMessage.AppendLine($">>> {result.Request.Method} {result.Request.Path}");
66
67     if (requestJson is not null)
68         outputMessage.AppendLine(requestJson);
69 }
70
71 outputMessage.AppendLine($"<<< {responseMessage}");
72
73 if (responseJson is not null)
74     outputMessage.AppendLine(responseJson);
75
76 outputMessage.AppendLine("-----");

    await Console.Out.WriteLineAsync(outputMessage.ToString());
}
}
```

Server Sent Events

Sisk поддерживает отправку сообщений через Server Sent Events из коробки. Вы можете создавать одноразовые и постоянные подключения, получать подключения во время выполнения и использовать их.

Эта функция имеет некоторые ограничения, налагаемые браузерами, такие как отправка только текстовых сообщений и невозможность навсегда закрыть подключение. Закрытое с серверной стороны подключение будет иметь клиент, периодически пытающийся reconnect каждые 5 секунд (3 для некоторых браузеров).

Эти подключения полезны для отправки событий с сервера клиенту без необходимости запроса информации клиентом каждый раз.

Создание подключения SSE

Подключение SSE работает как обычный HTTP-запрос, но вместо отправки ответа и немедленного закрытия подключения, подключение остается открытым для отправки сообщений.

Вызывая метод [HttpRequest.GetEventSource\(\)](#), запрос помещается в состояние ожидания, пока создается экземпляр SSE.

```
1  r += new Route(RouteMethod.Get, "/", (req) =>
2  {
3      using var sse = req.GetEventSource();
4
5      sse.Send("Hello, world!");
6
7      return sse.Close();
8  });
```

В приведенном выше коде мы создаем подключение SSE и отправляем сообщение "Hello, world", затем закрываем подключение SSE с серверной стороны.

NOTE

При закрытии подключения с серверной стороны, по умолчанию клиент будет пытаться подключиться снова и подключение будет перезапущено, выполняя метод снова, навсегда.

Обычно отправляют сообщение о завершении с сервера, когда подключение закрывается с сервера, чтобы предотвратить попытки клиента reconnect снова.

Добавление заголовков

Если вам нужно отправить заголовки, вы можете использовать метод [HttpRequestEventSource.AppendHeader](#) перед отправкой любых сообщений.

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3     using var sse = req.GetEventSource();
4     sse.AppendHeader("Header-Key", "Header-value");
5
6     sse.Send("Hello!");
7
8     return sse.Close();
9   });

```

Обратите внимание, что необходимо отправить заголовки перед отправкой любых сообщений.

Подключения Wait-For-Fail

Подключения обычно завершаются, когда сервер больше не может отправлять сообщения из-за возможного отключения клиента. При этом подключение автоматически завершается и экземпляр класса удаляется.

Даже при reconnect, экземпляр класса не будет работать, поскольку он связан с предыдущим подключением. В некоторых ситуациях вам может понадобиться это подключение позже и вы не хотите управлять им через метод callback маршрута.

Для этого мы можем идентифицировать подключения SSE с помощью идентификатора и получить их позже, даже вне callback маршрута. Кроме того, мы помечаем подключение с помощью [WaitForFail](#), чтобы не завершать маршрут и не завершать подключение автоматически.

Подключение SSE в KeepAlive будет ждать ошибки отправки (вызванной отключением) для возобновления выполнения метода. Также можно установить Timeout для этого. После истечения времени, если не было отправлено никаких сообщений, подключение завершается и выполнение возобновляется.

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3     using var sse = req.GetEventSource("my-index-connection");
4
5     sse.WaitForFail(TimeSpan.FromSeconds(15)); // ждать 15 секунд без каких-либо сообщений
6     перед завершением подключения
7
8     return sse.Close();
9   });
10 }
```

Приведенный выше метод создаст подключение, обработает его и будет ждать отключения или ошибки.

```
1   HttpRequestEventSource? evs = server.EventSources.GetByIdentifier("my-index-connection");
2   if (evs != null)
3   {
4     // подключение все еще живо
5     evs.Send("Hello again!");
6   }
7 }
```

А приведенный выше фрагмент кода попытается найти вновь созданное подключение и если оно существует, отправит ему сообщение.

Все активные подключения сервера, которые идентифицированы, будут доступны в коллекции [HttpServer.EventSources](#). Эта коллекция хранит только активные и идентифицированные подключения. Закрытые подключения удаляются из коллекции.

NOTE

Важно отметить, что keep alive имеет ограничение, установленное компонентами, которые могут быть подключены к Sisk неконтролируемым образом, такими как веб-прокси, HTTP-ядро или сетевой драйвер, и они закрывают неактивные подключения после определенного периода времени.

Поэтому важно поддерживать подключение открытым, отправляя периодические пинги или продлевая максимальное время до закрытия подключения. Читайте следующий раздел, чтобы лучше понять отправку периодических пингов.

Настройка политики пингов подключения

Политика пингов – это автоматизированный способ отправки периодических сообщений клиенту. Эта функция позволяет серверу понимать, когда клиент отключился от этого подключения, не держа подключение открытым бесконечно.

```
1 [RouteGet("/sse")]
2 public HttpResponse Events(HttpRequest request)
3 {
4     using var sse = request.GetEventSource();
5     sse.WithPing(ping =>
6     {
7         ping.DataMessage = "ping-message";
8         ping.Interval = TimeSpan.FromSeconds(5);
9         ping.Start();
10    });
11
12    sse.KeepAlive();
13    return sse.Close();
14 }
```

В приведенном выше коде, каждые 5 секунд, будет отправлено новое сообщение пинга клиенту. Это будет поддерживать подключение TCP в активном состоянии и предотвращать его закрытие из-за неактивности. Кроме того, когда отправка сообщения fails, подключение автоматически закрывается, освобождая ресурсы, используемые подключением.

Запрос подключений

Вы можете искать активные подключения, используя предикат по идентификатору подключения, чтобы иметь возможность广播, например.

```
1  HttpRequestEventSource[] evs = server.EventSources.Find(es => es.StartsWith("my-
2  connection-"));
3  foreach (HttpRequestEventSource e in evs)
4  {
5    e.Send("Broadcasting to all event sources that starts with 'my-connection-'");
6  }
```

Вы также можете использовать метод [All](#), чтобы получить все активные подключения SSE.

Web Sockets

Sisk поддерживает веб-сокеты, например, получение и отправка сообщений клиенту.

Эта функция работает корректно во многих браузерах, но в Sisk она всё ещё экспериментальная. Если вы обнаружите ошибки, сообщите о них на GitHub.

Приём сообщений асинхронно

Сообщения WebSocket принимаются в порядке, ставятся в очередь до обработки `ReceiveMessageAsync`. Этот метод не возвращает сообщение, если истёк таймаут, операция отменена или клиент отключён.

Одновременно может выполняться только одна операция чтения и записи, поэтому, пока вы ждёте сообщение через `ReceiveMessageAsync`, писать в подключённого клиента невозможно.

```
1  router.MapGet("/connect", async (HttpRequest req) =>
2  {
3      using var ws = await req.GetWebSocketAsync();
4
5      while (await ws.ReceiveMessageAsync(timeout: TimeSpan.FromSeconds(30)) is {
6          receivedMessage)
7      {
8          string msgText = receivedMessage.GetString();
9          Console.WriteLine("Received message: " + msgText);
10
11         await ws.SendAsync("Hello!");
12     }
13
14     return await ws.CloseAsync();
15 });

});
```

Приём сообщений синхронно

Ниже приведён пример использования синхронного веб-сокета без асинхронного контекста, где вы получаете сообщения, обрабатываете их и завершаете работу с сокетом.

```
1  router.MapGet("/connect", async (HttpRequest req) =>
2  {
3      using var ws = await req.GetWebSocketAsync();
4      WebSocketMessage? msg;
5
6      askName:
7          await ws.SendAsync("What is your name?");
8          msg = await ws.ReceiveMessageAsync();
9
10     if (msg is null)
11         return await ws.CloseAsync();
12
13     string name = msg.GetString();
14
15     if (string.IsNullOrEmpty(name))
16     {
17         await ws.SendAsync("Please, insert your name!");
18         goto askName;
19     }
20
21     askAge:
22         await ws.SendAsync("And your age?");
23         msg = await ws.ReceiveMessageAsync();
24
25         if (msg is null)
26             return await ws.CloseAsync();
27
28         if (!Int32.TryParse(msg?.GetString(), out int age))
29         {
30             await ws.SendAsync("Please, insert an valid number");
31             goto askAge;
32         }
33
34         await ws.SendAsync($"You're {name}, and you are {age} old.");
35
36         return await ws.CloseAsync();
37     });
38 }
```

Политика Ping

Подобно тому, как работает политика ping в Server Side Events, вы также можете настроить политику ping, чтобы держать TCP-соединение открытым при отсутствии активности.

```
1 ws.PingPolicy.Start(  
2     dataMessage: "ping-message",  
3     interval: TimeSpan.FromSeconds(10));
```

Синтаксис Discard

Веб-сервер HTTP можно использовать для прослушивания запроса обратного вызова из действия, такого как аутентификация OAuth, и можно отбросить после получения этого запроса. Это может быть полезно в случаях, когда вам нужна фоновое действие, но вы не хотите настраивать整个 веб-приложение для этого.

Следующий пример показывает, как создать прослушивающий HTTP-сервер на порту 5555 с помощью [CreateListener](#) и ожидать следующий контекст:

```
1  using (var server = HttpServer.CreateListener\(5555\))
2  {
3      // ожидать следующий HTTP-запрос
4      var context = await server.WaitNextAsync\(\);
5      Console.WriteLine\(\$"Запрошенный путь: {context.Request.Path}"\);
6  }
```

Функция [WaitNext](#) ожидает следующий контекст завершенной обработки запроса. Как только получен результат этой операции, сервер уже полностью обработал запрос и отправил ответ клиенту.

Внедрение зависимостей

Обычно выделяют члены и экземпляры, которые существуют в течение всего времени жизни запроса, такие как соединение с базой данных, аутентифицированный пользователь или токен сессии. Одним из возможностей является использование `HttpContext.RequestBag`, который создает словарь, существующий в течение всего времени жизни запроса.

Этот словарь можно получить у `обработчиков запросов` и определить переменные на протяжении всего запроса. Например, обработчик запроса, который аутентифицирует пользователя, устанавливает этого пользователя в `HttpContext.RequestBag`, и в логике запроса этот пользователь можно получить с помощью `HttpContext.RequestBag.Get<User>()`.

Вот пример:

RequestHandlers/AuthenticateUser.cs

C#

```
1  public class AuthenticateUser : IRequestHandler
2  {
3      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
4          RequestHandlerExecutionMode.BeforeResponse;
5
6      public HttpResponse? Execute(HttpContext context)
7      {
8          User authenticatedUser = AuthenticateUser(context);
9          context.RequestBag.Set(authenticatedUser);
10         return null; // advance to the next request handler or request logic
11     }
12 }
```

Controllers/HelloController.cs

C#

```
1  [RouteGet("/hello")]
2  [RequestHandler<AuthenticateUser>]
3  public static HttpResponse SayHello(HttpContext request)
4  {
5      var authenticatedUser = request.Bag.Get<User>();
6      return new HttpResponseMessage()
7      {
8          Content = new StringContent($"Hello {authenticatedUser.Name}!")
9      }
10 }
```

```
9      };
10 }
```

Это предварительный пример этой операции. Экземпляр `User` был создан в обработчике запроса, посвященном аутентификации, и все маршруты, которые используют этот обработчик запроса, будут иметь гарантию, что в их экземпляре `HttpContext.RequestBag` будет `User`.

Возможно определить логику получения экземпляров, когда они не были предварительно определены в `RequestBag`, через методы типа `GetOrAdd` или `GetOrAddAsync`.

С версии 1.3 был введен статический свойство `HttpContext.Current`, которое позволяет получить доступ к текущему контексту запроса. Это позволяет экспонировать члены `HttpContext` вне текущего запроса и определять экземпляры в объектах маршрутов.

Пример ниже определяет контроллер, который имеет члены, обычно доступные контекстом запроса.

Controllers/Controller.cs

C#

```
1  public abstract class Controller : RouterModule
2  {
3      public DbContext Database
4      {
5          get
6          {
7              // создать DbContext или получить существующий
8              return HttpContext.Current.RequestBag.GetOrAdd(() => new DbContext());
9          }
10     }
11
12     // следующая строка выдаст исключение, если свойство доступно, когда User не
13     // определен в пакете запроса
14     public User AuthenticatedUser { get => HttpContext.Current.RequestBag.Get<User>(); }
15
16     // экспонирование экземпляра HttpRequest также поддерживается
17     public HttpRequest Request { get => HttpContext.Current.Request; }
18 }
```

И определить типы, которые наследуются от контроллера:

Controllers/PostsController.cs

C#

```
1  [RoutePrefix( "/api/posts" )]
2  public class PostsController : Controller
3  {
```

```

4     [RouteGet]
5     public IEnumerable<Blog> ListPosts()
6     {
7         return Database.Posts
8             .Where(post => post.AuthorId = AuthenticatedUser.Id)
9             .ToList();
10    }
11
12    [RouteGet("<id>")]
13    public Post GetPost()
14    {
15        int blogId = Request.RouteParameters["id"].GetInteger();
16
17        Post? post = Database.Posts
18            .FirstOrDefault(post => post.Id = blogId && post.AuthorId
19 = AuthenticatedUser.Id);
20
21        return post ?? new HttpResponse(404);
22    }
}

```

Для примера выше вам необходимо настроить [обработчик значения](#) в вашем маршрутизаторе, чтобы объекты, возвращаемые маршрутизатором, были преобразованы в допустимый [HttpResponse](#).

Обратите внимание, что методы не имеют аргумента `HttpRequest request`, как это присутствует в других методах. Это связано с тем, что с версии 1.3 маршрутизатор поддерживает два типа делегатов для маршрутизации ответов: [RouteAction](#), который является делегатом по умолчанию, получающим аргумент `HttpRequest`, и [ParameterlessRouteAction](#). Экземпляр `HttpRequest` все равно можно получить через свойство [Request](#) статического `HttpContext` в потоке.

В примере выше мы определили объект, подлежащий удалению, `DbContext`, и нам необходимо обеспечить, чтобы все экземпляры, созданные в `DbContext`, были удалены, когда HTTP-сессия завершается. Для этого можно использовать два способа достижения этой цели. Один из них - создать [обработчик запроса](#), который выполняется после действия маршрутизатора, а другой способ - через пользовательский [обработчик сервера](#).

Для первого метода можно создать обработчик запроса `true` в методе [OnSetup](#), унаследованном от `RouterModule`:

Controllers/PostsController.cs

C#

```

1  public abstract class Controller : RouterModule
2  {
3      ...

```

```
4
5     protected override void OnSetup(Router parentRouter)
6     {
7         base.OnSetup(parentRouter);
8
9         HasRequestHandler(RequestHandler.Create(
10             execute: (req, ctx) =>
11             {
12                 // получить один DbContext, определенный в контексте обработчика запроса и
13                 // удалить его
14                 ctx.RequestBag.GetOrDefault<DbContext>()?.Dispose();
15                 return null;
16             },
17             executionMode: RequestHandlerExecutionMode.AfterResponse));
18     }
19 }
```

TIP

С версии Sisk 1.4 свойство `HttpServerConfiguration.DisposeDisposableContextValues` было введено и включено по умолчанию, которое определяет, должен ли HTTP-сервер удалять все значения `IDisposable` в пакете контекста, когда HTTP-сессия закрывается.

Метод выше обеспечит удаление `DbContext`, когда HTTP-сессия завершается. Вы можете сделать это для других членов, которые необходимо удалить в конце ответа.

Для второго метода можно создать пользовательский [обработчик сервера](#), который будет удалять `DbContext`, когда HTTP-сессия завершается.

Server/Handlers/ObjectDisposerHandler.cs

C#

```
1  public class ObjectDisposerHandler : HttpServerHandler
2  {
3      protected override void OnHttpRequestClose(HttpServerExecutionResult result)
4      {
5          result.Context.RequestBag.GetOrDefault<DbContext>()?.Dispose();
6      }
7  }
```

И использовать его в вашем построителе приложения:

Program.cs

C#

```
1  using var host = HttpServer.CreateBuilder()
2      .UseHandler<ObjectDisposerHandler>()
3      .Build();
```

Это один из способов обработки очистки кода и поддержания зависимостей запроса, разделенных по типу модуля, который будет использоваться, уменьшая количество дублирующего кода внутри каждого действия маршрутизатора. Это практика, аналогичная тому, для чего используется внедрение зависимостей в фреймворках типа ASP.NET.

Потоковая передача контента

Sisk поддерживает чтение и отправку потоков контента на клиент и с клиента. Эта функция полезна для удаления нагрузки на память при сериализации и десериализации контента во время жизни запроса.

Поток контента запроса

Маленькие содержимые автоматически загружаются в буфер памяти HTTP-соединения, быстро загружая это содержимое в [HttpRequest.Body](#) и [HttpRequest.RawBody](#). Для более крупных содержимых можно использовать метод [HttpRequest.GetRequestStream](#), чтобы получить поток чтения контента запроса.

Стоит отметить, что метод [HttpRequest.GetMultipartFormContent](#) читает весь контент запроса в память, поэтому он может не быть полезен для чтения крупных содержимых.

Рассмотрим следующий пример:

Controller/UploadDocument.cs

C#

```
1  [RoutePost ( "/api/upload-document/<filename>" )]
2  public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4      var fileName = request.RouteParameters [ "filename" ].GetString ();
5
6      if ( !request.HasContents ) {
7          // запрос не содержит контента
8          return new HttpResponse ( HttpStatusInformation.BadRequest );
9      }
10
11     var contentStream = request.GetRequestStream ();
12     var outputFileName = Path.Combine (
13         AppDomain.CurrentDomain.BaseDirectory,
14         "uploads",
15         fileName );
16
17     using ( var fs = File.Create ( outputFileName ) ) {
18         await contentStream.CopyToAsync ( fs );
```

```
19     }
20
21     return new HttpResponseMessage() {
22         Content = JsonContent.Create(new { message = "Файл отправлен успешно." })
23     };
24 }
```

В примере выше метод `UploadDocument` читает контент запроса и сохраняет контент в файл. Не производится дополнительная алокация памяти, кроме буфера чтения, используемого `Stream.CopyToAsync`. Пример выше удаляет нагрузку на алокацию памяти для очень крупного файла, что может оптимизировать производительность приложения.

Хорошой практикой является всегда использовать [CancellationToken](#) в операции, которая может занять время, такой как отправка файлов, поскольку она зависит от скорости сети между клиентом и сервером.

Настройка с помощью `CancellationToken` может быть выполнена следующим образом:

Controller/UploadDocument.cs

C#

```
1 // токен отмены ниже будет выбрасывать исключение, если истечет 30-секундный таймаут.
2 CancellationTokenSource copyCancellation = new CancellationTokenSource( delay:
3     TimeSpan.FromSeconds( 30 ) );
4
5 try {
6     using ( var fs = File.Create( outputFileName ) ) {
7         await contentStream.CopyToAsync( fs, copyCancellation.Token );
8     }
9 }
10 catch ( OperationCanceledException ) {
11     return new HttpResponseMessage( HttpStatusCode.BadRequest ) {
12         Content = JsonContent.Create( new { Error = "Отправка файла превысила максимальное
13 время отправки (30 секунд)." } )
14     };
15 }
```

Поток контента ответа

Отправка контента ответа также возможна. В настоящее время существует два способа сделать это: через метод [HttpRequest.GetResponseStream](#) и используя контент типа [StreamContent](#).

Рассмотрим сценарий, в котором нам нужно предоставить файл изображения. Для этого можно использовать следующий код:

Controller/ImageController.cs

C#

```
1  [RouteGet ( "/api/profile-picture" )]
2  public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4      // пример метода для получения изображения профиля
5      var profilePictureFilename = "profile-picture.jpg";
6      byte[] profilePicture = await File.ReadAllBytesAsync ( profilePictureFilename );
7
8      return new HttpResponse () {
9          Content = new ByteArrayContent ( profilePicture ),
10         Headers = new () {
11             ContentType = "image/jpeg",
12             ContentDisposition = $"inline; filename={profilePictureFilename}"
13         }
14     };
15 }
```

Метод выше производит аллокацию памяти каждый раз, когда он читает контент изображения. Если изображение большое, это может вызвать проблему производительности, и в пиковых ситуациях даже привести к переполнению памяти и краху сервера. В таких ситуациях кэширование может быть полезным, но оно не устранит проблему, поскольку память все равно будет зарезервирована для этого файла. Кэширование облегчит нагрузку на аллокацию памяти для каждого запроса, но для крупных файлов оно не будет достаточно.

Отправка изображения через поток может быть решением проблемы. Вместо чтения всего контента изображения создается поток чтения файла и копируется на клиент с помощью небольшого буфера.

Отправка через метод GetResponseStream

Метод [HttpRequest.GetResponseStream](#) создает объект, который позволяет отправлять части HTTP-ответа по мере подготовки потока контента. Этот метод более ручной, требующий определения статуса, заголовков и размера контента перед отправкой контента.

Controller/ImageController.cs

C#

```

1  [RouteGet ( "/api/profile-picture" )]
2  public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4      var profilePictureFilename = "profile-picture.jpg";
5
6      // в этой форме отправки необходимо определить статус и заголовки
7      // перед отправкой контента
8      var requestStreamManager = request.GetResponseStream ();
9
10     requestStreamManager.SetStatus ( System.Net.HttpStatusCode.OK );
11     requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentType, "image/jpeg" );
12     requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentDisposition, $"inline;
13     filename={profilePictureFilename}" );
14
15     using (var fs = File.OpenRead ( profilePictureFilename ) ) {
16
17         // в этой форме отправки также необходимо определить размер контента
18         // перед отправкой его.
19         requestStreamManager.SetContentLength ( fs.Length );
20
21         // если вы не знаете размера контента, можно использовать chunked-encoding
22         // для отправки контента
23         requestStreamManager.SendChunked = true;
24
25         // и затем, записать в поток вывода
26         await fs.CopyToAsync ( requestStreamManager.ResponseStream );
27     }
}

```

Отправка контента через StreamContent

Класс [StreamContent](#) позволяет отправлять контент из источника данных в виде потока байтов. Эта форма отправки проще, удаляя предыдущие требования, и даже позволяет использовать [сжатие контента](#), чтобы уменьшить размер контента.

Controller/ImageController.cs

C#

```

1  [RouteGet ( "/api/profile-picture" )]
2  public HttpResponseMessage UploadDocument ( HttpRequest request ) {
3
4      var profilePictureFilename = "profile-picture.jpg";
5
6      return new HttpResponseMessage () {
7          Content = new StreamContent ( File.OpenRead ( profilePictureFilename ) ),

```

```
8     Headers = new () {
9         ContentType = "image/jpeg",
10        ContentDisposition = $"inline; filename=\"{profilePictureFilename}\""
11    }
12 };
13 }
```

⊗ IMPORTANT

В этом типе контента не заключайте поток в блок `using`. Контент будет автоматически удален HTTP-сервером, когда поток контента будет завершен, с ошибками или без них.

Включение CORS (Cross-Origin Resource Sharing) в Sisk

Sisk имеет инструмент, который может быть полезен для обработки Cross-Origin Resource Sharing (CORS) [🔗](#) при открытии вашего сервиса для публичного доступа. Эта функция не является частью протокола HTTP, а специфической особенностью веб-браузеров, определённой W3C. Этот механизм безопасности предотвращает выполнение веб-страницы запросов к другому домену, отличному от того, который предоставил веб-страницу. Поставщик услуг может разрешить доступ определённым доменам, или только одному.

Same Origin

Для того чтобы ресурс был идентифицирован как «same origin», запрос должен содержать заголовок **Origin** [🔗](#) в своём запросе:

```
1 GET /api/users HTTP/1.1
2 Host: example.com
3 Origin: http://example.com
4 ...
```

И удалённый сервер должен ответить заголовком **Access-Control-Allow-Origin** [🔗](#) со значением, совпадающим с запрошенным origin:

```
1 HTTP/1.1 200 OK
2 Access-Control-Allow-Origin: http://example.com
3 ...
```

Эта проверка **явная**: хост, порт и протокол должны совпадать с запрошенными. Проверьте пример:

- Сервер отвечает, что его Access-Control-Allow-Origin равен https://example.com :
 - https://example.net – домен отличается.
 - http://example.com – схема отличается.
 - http://example.com:5555 – порт отличается.

- `https://www.example.com` – хост отличается.

В спецификации разрешён только синтаксис для обоих заголовков, как для запросов, так и для ответов. Путь URL игнорируется. Порт также опускается, если это порт по умолчанию (80 для HTTP и 443 для HTTPS).

```
1  Origin: null
2  Origin: <scheme>://<hostname>
3  Origin: <scheme>://<hostname>:<port>
```

Включение CORS

Нативно у вас есть объект `CrossOriginResourceSharingHeaders` внутри вашего `ListeningHost`.

Вы можете настроить CORS при инициализации сервера:

```
1  static async Task Main(string[] args)
2  {
3      using var app = HttpServer.CreateBuilder()
4          .UseCors(new CrossOriginResourceSharingHeaders(
5              allowOrigin: "http://example.com",
6              allowHeaders: ["Authorization"],
7              exposeHeaders: ["Content-Type"]))
8          .Build();
9
10     await app.StartAsync();
11 }
```

Код выше отправит следующие заголовки для **всех ответов**:

```
1  HTTP/1.1 200 OK
2  Access-Control-Allow-Origin: http://example.com
3  Access-Control-Allow-Headers: Authorization
4  Access-Control-Expose-Headers: Content-Type
```

Эти заголовки необходимо отправлять для всех ответов веб-клиенту, включая ошибки и перенаправления.

Вы можете заметить, что класс `CrossOriginResourceSharingHeaders` имеет два похожих свойства: `AllowOrigin` и `AllowOrigins`. Обратите внимание, что одно во множественном числе, другое — в единственном.

- Свойство `AllowOrigin` статическое: только указанный origin будет отправлен для всех ответов.
- Свойство `AllowOrigins` динамическое: сервер проверяет, содержится ли origin запроса в этом списке. Если найден, он отправляется в ответе для того origin.

Wildcards и автоматические заголовки

В качестве альтернативы вы можете использовать подстановочный символ (*) в origin ответа, чтобы указать, что любой origin разрешён для доступа к ресурсу. Однако это значение не допускается для запросов с учётными данными (заголовки авторизации), и эта операция [приведёт к ошибке](#).

Вы можете обойти эту проблему, явно перечислив, какие origins будут разрешены через свойство `AllowOrigins` или также использовать константу `AutoAllowOrigin` в значении `AllowOrigin`. Это «магическое» свойство определит заголовок `Access-Control-Allow-Origin` со значением, совпадающим с заголовком `Origin` запроса.

Вы также можете использовать `AutoFromRequestMethod` и `AutoFromRequestHeaders` для поведения, аналогичного `AllowOrigin`, которое автоматически отвечает на основе отправленных заголовков.

```
1  using var host = HttpServer.CreateBuilder()
2      .UseCors(new CrossOriginResourceSharingHeaders(
3
4          // Отвечает на основе заголовка Origin запроса
5          allowOrigin: CrossOriginResourceSharingHeaders.AutoAllowOrigin,
6
7          // Отвечает на основе заголовка Access-Control-Request-Method или метода запроса
8          allowMethods: [CrossOriginResourceSharingHeaders.AutoFromRequestMethod],
9
10         // Отвечает на основе заголовка Access-Control-Request-Headers или
11         // отправленных заголовков
12         allowHeaders: [CrossOriginResourceSharingHeaders.AutoFromRequestHeaders]))
```

Другие способы применения CORS

Если вы работаете с [service providers](#), вы можете переопределить значения, определённые в файле конфигурации:

```
1 static async Task Main(string[] args)
2 {
3     using var app = HttpServer.CreateBuilder()
4         .UsePortableConfiguration( ... )
5         .UseCors(cors => {
6             // Переопределит origin, определённый в конфигурации
7             // файле.
8             cors.AllowOrigin = "http://example.com";
9         })
10        .Build();
11
12     await app.StartAsync();
13 }
```

Отключение CORS на конкретных маршрутах

Свойство `UseCors` доступно как для маршрутов, так и для всех атрибутов маршрута и может быть отключено следующим примером:

```
1 [RoutePrefix("api/widgets")]
2 public class WidgetController : Controller {
3
4     // GET /api/widgets/colors
5     [RouteGet("/colors", UseCors = false)]
6     public IEnumerable<string> GetWidgets() {
7         return new[] { "Green widget", "Red widget" };
8     }
9 }
```

Замена значений в ответе

Вы можете явно заменить или удалить значения в действии роутера:

```
1 [RoutePrefix("api/widgets")]
2 public class WidgetController : Controller {
3
4     public IEnumerable<string> GetWidgets(HttpServletRequest request) {
5
6         // Удаляет заголовок Access-Control-Allow-Credentials
7         request.Context.OverrideHeaders.AccessControlAllowCredentials = string.Empty;
8
9         // Заменяет Access-Control-Allow-Origin
10        request.Context.OverrideHeaders.AccessControlAllowOrigin = "https://contorso.com";
11
12        return new[] { "Green widget", "Red widget" };
13    }
14}
```

Предварительные запросы

Предварительный запрос — это запрос метода [OPTIONS](#), который клиент отправляет до фактического запроса.

Сервер Sisk всегда отвечает на запрос `200 OK` и применимыми заголовками CORS, после чего клиент может продолжить с фактическим запросом. Это условие не применяется, когда существует маршрут для запроса с явно настроенным [RouteMethod](#) для `Options`.

Отключение CORS глобально

Это невозможно. Чтобы не использовать CORS, не настраивайте его.

Расширение JSON-RPC

Sisk имеет экспериментальный модуль для API [JSON-RPC 2.0](#), который позволяет создавать еще более простые приложения. Это расширение строго реализует транспортный интерфейс JSON-RPC 2.0 и предлагает транспорт через HTTP GET, POST-запросы и также веб-сокеты с Sisk.

Вы можете установить расширение через Nuget с помощью команды ниже. Обратите внимание, что в экспериментальных/бета-версиях необходимо включить опцию поиска предварительных пакетов в Visual Studio.

```
dotnet add package Sisk.JsonRpc
```

Транспортный Интерфейс

JSON-RPC - это бесстаточный, асинхронный протокол удаленного выполнения процедур (RDP), который использует JSON для односторонней передачи данных. Запрос JSON-RPC обычно идентифицируется по ID, и ответ доставляется с тем же ID, который был отправлен в запросе. Не все запросы требуют ответа, которые называются "уведомлениями".

[Спецификация JSON-RPC 2.0](#) подробно объясняет, как работает транспорт. Этот транспорт независим от того, где он будет использоваться. Sisk реализует этот протокол через HTTP, следуя соответствиям [JSON-RPC over HTTP](#), который частично поддерживает GET-запросы, но полностью поддерживает POST-запросы. Также поддерживаются веб-сокеты, которые обеспечивают асинхронную передачу сообщений.

Запрос JSON-RPC выглядит примерно так:

```
1  {
2      "jsonrpc": "2.0",
3      "method": "Sum",
4      "params": [1, 2, 4],
5      "id": 1
6  }
```

И успешный ответ выглядит примерно так:

```
1  {
2      "jsonrpc": "2.0",
3      "result": 7,
4      "id": 1
5 }
```

Методы JSON-RPC

Следующий пример показывает, как создать API JSON-RPC с помощью Sisk. Класс математических операций выполняет удаленные операции и доставляет сериализованный ответ клиенту.

Program.cs

C#

```
1  using var app = HttpServer.CreateBuilder(port: 5555)
2      .UseJsonRPC((sender, args) =>
3      {
4          // добавляет все методы, помеченные как WebMethod, в обработчик JSON-RPC
5          args.Handler.Methods.AddMethodsFromType(new MathOperations());
6
7          // сопоставляет маршрут /service с обработчиком JSON-RPC POST и GET-запросов
8          args.Router.MapPost("/service", args.Handler.Transport.HttpPost);
9          args.Router.MapGet("/service", args.Handler.Transport.HttpGet);
10
11         // создает обработчик веб-сокета на GET /ws
12         args.Router.MapGet("/ws", request =>
13         {
14             var ws = request.GetWebSocket();
15             ws.OnReceive += args.Handler.Transport.WebSocket;
16
17             ws.WaitForClose(timeout: TimeSpan.FromSeconds(30));
18             return ws.Close();
19         });
20     })
21     .Build();
22
23 await app.StartAsync();
```

```
1  public class MathOperations
2  {
3      [WebMethod]
4      public float Sum(float a, float b)
5      {
6          return a + b;
7      }
8
9      [WebMethod]
10     public double Sqrt(double a)
11     {
12         return Math.Sqrt(a);
13     }
14 }
```

Вышеуказанный пример сопоставит методы `Sum` и `Sqrt` с обработчиком JSON-RPC, и эти методы будут доступны по адресам `GET /service`, `POST /service` и `GET /ws`. Имена методов регистрационезависимы.

Параметры методов автоматически десериализуются в свои конкретные типы. Также поддерживается использование запросов с именованными параметрами. Сериализация JSON выполняется библиотекой [LightJson](#). Если тип не десериализуется правильно, вы можете создать специальный [конвертер JSON](#) для этого типа и связать его с вашими `JsonSerializerOptions` позже.

Вы также можете получить объект `$.params` из запроса JSON-RPC напрямую в вашем методе.

```
1  [WebMethod]
2  public float Sum(JsonArray|JsonObject @params)
3  {
4      ...
5  }
```

Для этого `@params` должен быть единственным параметром в вашем методе, с точным именем `params` (в C#, символ `@` необходим для экранирования этого имени параметра).

Десериализация параметров происходит как для именованных объектов, так и для позиционных массивов. Например, следующий метод можно вызывать удаленно как с помощью запроса с именованными параметрами, так и с помощью запроса с позиционными параметрами.

```
1 [WebMethod]
2 public float AddUserToStore(string apiKey, User user, UserStore store)
3 {
4     ...
5 }
```

Для массива порядок параметров должен быть соблюден.

```
1 {
2     "jsonrpc": "2.0",
3     "method": "AddUserToStore",
4     "params": [
5         "1234567890",
6         {
7             "name": "John Doe",
8             "email": "john@example.com"
9         },
10        {
11            "name": "My Store"
12        }
13    ],
14    "id": 1
15
16 }
```

Настройка сериализатора

Вы можете настроить сериализатор JSON в свойстве `JsonRpcHandler.JsonSerializerOptions`. В этом свойстве вы можете включить использование [JSON5](#) для десериализации сообщений. Хотя это не соответствует спецификации JSON-RPC 2.0, JSON5 является расширением JSON, которое позволяет писать более читаемые и понятные данные.

Program.cs

C#

```
1 using var host = HttpServer.CreateBuilder( 5556 )
2     .UseJsonRPC( ( o, e ) => {
3
4         // использует санитизированный компаратор имен. этот компаратор сравнивает
5         только буквы
```

```
6      // и цифры в имени, и игнорирует другие символы. например:  
7      // foo_bar10 = FooBar10  
8      e.Handler.JsonSerializerOptions.PropertyNameComparer = new  
9      JsonSanitizedComparer ();  
10  
11      // включает JSON5 для интерпретатора JSON. даже активируя это, обычный JSON все  
12      // еще поддерживается  
13      e.Handler.JsonSerializerOptions.SerializationFlags =  
14      LightJson.Serialization.JsonSerializationFlags.Json5;  
15  
16      // сопоставляет маршрут POST /service с обработчиком JSON-RPC  
17      e.Router.MapPost ( "/service", e.Handler.Transport.HttpPost );  
 } )  
.Build ();  
  
host.Start ();
```

SSL Proxy

! WARNING

Эта функция экспериментальная и не должна использоваться в производстве. Пожалуйста, обратитесь к [этому документу](#), если вы хотите сделать Sisk работать с SSL.

Sisk SSL Proxy - это модуль, который предоставляет HTTPS-соединение для [ListeningHost](#) в Sisk и маршрутизирует HTTPS-сообщения в ~~не~~安全ный HTTP-контекст. Модуль был создан для предоставления SSL-соединения для службы, которая использует [HttpListener](#) для запуска, который не поддерживает SSL.

Прокси работает внутри одного и того же приложения и слушает HTTP/1.1-сообщения, пересылая их в том же протоколе в Sisk. В настоящее время эта функция является высокоядерной и может быть достаточно нестабильной, чтобы не использовать ее в производстве.

На данный момент SslProxy поддерживает почти все функции HTTP/1.1, такие как keep-alive, chunked encoding, websockets и т. д. Для открытого соединения с SSL-прокси создается TCP-соединение с целевым сервером, и прокси пересыпается на установленное соединение.

SslProxy можно использовать с [HttpServer.CreateBuilder](#) следующим образом:

```
1  using var app = HttpServer.CreateBuilder(port: 5555)
2      .UseRouter(r =>
3      {
4          r.MapGet("/", request =>
5          {
6              return new HttpResponseMessage("Hello, world!");
7          });
8      })
9      // добавление SSL в проект
10     .UseSsl(
11         sslListeningPort: 5567,
12         new X509Certificate2(@".\ssl.pfx", password: "12345")
13     )
14     .Build();
15
16 app.Start();
```

Вам необходимо предоставить действительный SSL-сертификат для прокси. Чтобы обеспечить принятие сертификата браузерами, не забудьте импортировать его в операционную систему, чтобы он

функционировал правильно.

Базовая Аутентификация

Пакет Basic Auth добавляет обработчик запросов, способный обрабатывать базовую схему аутентификации в вашем приложении Sisk с минимальной конфигурацией и усилиями. Базовая аутентификация HTTP – это минимальная форма аутентификации запросов по идентификатору пользователя и паролю, где сессия контролируется исключительно клиентом, и нет аутентификационных или доступных токенов.



Читайте больше о схеме базовой аутентификации в [спецификации MDN](#).

Установка

Чтобы начать, установите пакет Sisk.BasicAuth в вашем проекте:

```
> dotnet add package Sisk.BasicAuth
```

Вы можете просмотреть больше способов установки его в вашем проекте в [репозитории Nuget](#).

Создание обработчика аутентификации

Вы можете контролировать схему аутентификации для всего модуля или для отдельных маршрутов. Для этого давайте сначала напишем наш первый базовый обработчик аутентификации.

В примере ниже устанавливается соединение с базой данных, проверяется, существует ли пользователь и является ли пароль действительным, и после этого хранит пользователя в контексте.

```
1  public class UserAuthHandler : BasicAuthenticateRequestHandler  
2  {  
3      public UserAuthHandler() : base()  
4      {
```

```

5         Realm = "Чтобы войти на эту страницу, пожалуйста, введите ваши учетные данные.";
6     }
7
8     public override HttpResponse? OnValidating(BasicAuthenticationCredentials credentials,
9     HttpContext context)
10    {
11        DbContext db = new DbContext();
12
13        // в этом случае мы используем электронную почту в качестве идентификатора
14        // пользователя, поэтому мы
15        // ищем пользователя по его электронной почте.
16        User? user = db.Users.FirstOrDefault(u => u.Email == credentials.UserId);
17        if (user == null)
18        {
19            return base.CreateUnauthorizedResponse("Извините! Пользователь с таким
20            электронным адресом не найден.");
21        }
22
23        // проверяет, что пароль учетных данных действителен для этого пользователя.
24        if (!user.ValidatePassword(credentials.Password))
25        {
26            return base.CreateUnauthorizedResponse("Недействительные учетные данные.");
27        }
28
29        // добавляет вошедшего пользователя в контекст HTTP
30        // и продолжает выполнение
31        context.Bag.Add("loggedUser", user);
32        return null;
33    }
34}

```

Итак, просто ассоциируйте этот обработчик запросов с нашим маршрутом или классом.

```

1  public class UsersController
2  {
3      [RouteGet("/")]
4      [RequestHandler(typeof(UserAuthHandler))]
5      public string Index(HttpContext request)
6      {
7          User loggedUser = (User)request.Context.RequestBag["loggedUser"];
8          return "Привет, " + loggedUser.Name + "!";
9      }
10 }

```

Или используя класс [RouterModule](#):

```
1  public class UsersController : RouterModule
2  {
3      public ClientModule()
4      {
5          // теперь все маршруты внутри этого класса будут обрабатываться
6          // UserAuthHandler.
7          base.HasRequestHandler(new UserAuthHandler());
8      }
9
10     [RouteGet("/")]
11     public string Index(HttpRequest request)
12     {
13         User loggedUser = (User)request.Context.RequestBag["loggedUser"];
14         return "Привет, " + loggedUser.Name + "!";
15     }
16 }
```

Примечания

Основная ответственность базовой аутентификации лежит на клиентской стороне. Хранение, кэширование и шифрование обрабатываются локально на клиенте. Сервер получает только учетные данные и проверяет, разрешен ли доступ или нет.

Обратите внимание, что этот метод не является одним из самых безопасных, поскольку он возлагает значительную ответственность на клиента, что может быть трудно отслеживать и поддерживать безопасность его учетных данных. Кроме того, важно передавать пароли в безопасном контексте соединения (SSL), поскольку они не имеют встроенного шифрования. Короткий перехват в заголовках запроса может раскрыть учетные данные доступа вашего пользователя.

Выбирайте более надежные решения аутентификации для приложений в производстве и избегайте использования слишком многих готовых компонентов, поскольку они могут не адаптироваться к потребностям вашего проекта и в конечном итоге подвергать его риску безопасности.

Поставщики услуг

Поставщики услуг - это способ переноса вашего приложения Sisk в разные среды с помощью переносимого файла конфигурации. Эта функция позволяет изменить порт сервера, параметры и другие настройки без необходимости изменения кода приложения для каждой среды. Этот модуль зависит от синтаксиса конструкции Sisk и может быть настроен с помощью метода UsePortableConfiguration.

Поставщик конфигурации реализуется с помощью `IConfigurationProvider`, который предоставляет читатель конфигурации и может получать любую реализацию. По умолчанию, Sisk предоставляет читатель конфигурации JSON, но также есть пакет для файлов INI. Вы также можете создать свой собственный поставщик конфигурации и зарегистрировать его с помощью:

```
1  using var app = HttpServer.CreateBuilder()
2      .UsePortableConfiguration(config =>
3      {
4          config.WithConfigReader<MyConfigurationReader>();
5      })
6      .Build();
```

Как упоминалось ранее, поставщик конфигурации по умолчанию - это файл JSON. По умолчанию, имя файла, которое ищется, - это `service-config.json`, и он ищется в текущем каталоге запускаемого процесса, а не в каталоге исполняемого файла.

Вы можете выбрать изменение имени файла, а также указать, где Sisk должен искать файл конфигурации, с помощью:

```
1  using Sisk.Core.Http;
2  using Sisk.Core.Http.Hosting;
3
4  using var app = HttpServer.CreateBuilder()
5      .UsePortableConfiguration(config =>
6      {
7          config.WithConfigFile("config.toml",
8              createIfDontExists: true,
9              lookupDirectories:
10                 ConfigurationFileLookupDirectory.CurrentDirectory |
11                 ConfigurationFileLookupDirectory.AppDirectory);
12      })
13      .Build();
```

Код выше будет искать файл config.toml в текущем каталоге запускаемого процесса. Если файл не найден, он затем будет искать в каталоге, где находится исполняемый файл. Если файл не существует, параметр `createIfDontExists` будет выполнен, создав файл без содержимого в последнем проверенном пути (на основе `lookupDirectories`), и будет выдано сообщение об ошибке в консоли, предотвращая инициализацию приложения.

TIP

Вы можете посмотреть исходный код поставщика конфигурации INI и поставщика конфигурации JSON, чтобы понять, как реализуется `IConfigurationProvider`.

Чтение конфигураций из файла JSON

По умолчанию, Sisk предоставляет поставщик конфигурации, который читает конфигурации из файла JSON. Этот файл имеет фиксированную структуру и состоит из следующих параметров:

```
1  {
2      "Server": {
3          "DefaultEncoding": "UTF-8",
4          "ThrowExceptions": true,
5          "IncludeRequestIdHeader": true
6      },
7      "ListeningHost": {
8          "Label": "Мое приложение Sisk",
9          "Ports": [
10             "http://localhost:80/",
11             "https://localhost:443/", // Файлы конфигурации также поддерживают комментарии
12         ],
13         "CrossOriginResourceSharingPolicy": {
14             "AllowOrigin": "*",
15             "AllowOrigins": [ "*" ], // новое в 0.14
16             "AllowMethods": [ "*" ],
17             "AllowHeaders": [ "*" ],
18             "MaxAge": 3600
19         },
20         "Parameters": {
21             "MySqlConnection": "server=localhost;user=root;"
22         }
23     }
24 }
```

Параметры, созданные из файла конфигурации, можно получить в конструкторе сервера:

```
1  using var app = HttpServer.CreateBuilder()
2      .UsePortableConfiguration(config =>
3      {
4          config.WithParameters(paramCollection =>
5          {
6              string databaseConnection = paramCollection.GetValueOrDefault("MySqlConnection");
7          });
8      })
9      .Build();
```

Каждый поставщик конфигурации предоставляет способ чтения параметров инициализации сервера. Некоторые свойства указаны для того, чтобы они находились в процессе окружения вместо того, чтобы быть определены в файле конфигурации, такие как чувствительные данные API, ключи API и т. д.

Структура файла конфигурации

Файл конфигурации JSON состоит из следующих свойств:

| Свойство | Обязательное | Описание |
|-------------------------|---------------|--|
| Server | Требуется | Представляет собой сервер с его настройками. |
| Server.AccessLogsStream | Необязательно | По умолчанию - console. Указывает поток вывода журнала доступа. Может быть именем файла, null или console. |
| Server.ErrorsLogsStream | Необязательно | По умолчанию - null. Указывает поток вывода журнала ошибок. Может быть именем |

| Свойство | Обязательное | Описание |
|--|---------------|---|
| | | файла, null или console. |
| Server.MaximumContentLength | Необязательно | |
| Server.MaximumContentLength | Необязательно | По умолчанию - 0 . Указывает максимальную длину содержимого в байтах. Ноль означает бесконечность. |
| Server.IncludeRequestIdHeader | Необязательно | По умолчанию - false . Указывает, должен ли HTTP-сервер отправлять заголовок X-Request-Id . |
| Server.ThrowExceptions | Необязательно | По умолчанию - true . Указывает, должны ли быть выброшены необработанные исключения. Установите значение false при производстве и true при отладке. |
| ListeningHost | Требуется | Представляет собой хост, на котором слушает сервер. |
| ListeningHost.Label | Необязательно | Представляет собой метку приложения. |
| ListeningHost.Ports | Требуется | Представляет собой массив строк, соответствующих синтаксису ListeningPort. |
| ListeningHost.CrossOriginResourceSharingPolicy | Необязательно | Настройка CORS-заголовков для |

| Свойство | Обязательное | Описание |
|---|---------------|---|
| ListeningHost.CrossOriginResourceSharingPolicy.AllowCredentials | Необязательно | Приложения. |
| ListeningHost.CrossOriginResourceSharingPolicy.ExposeHeaders | Необязательно | По умолчанию – <code>false</code> . Указывает заголовок <code>Allow-Credentials</code> . |
| ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigin | Необязательно | По умолчанию – <code>null</code> . Это свойство ожидает массив строк. Указывает заголовок <code>Expose-Headers</code> . |
| ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigins | Необязательно | По умолчанию – <code>null</code> . Это свойство ожидает строку. Указывает заголовок <code>Allow-Origin</code> . |
| ListeningHost.CrossOriginResourceSharingPolicy.AllowMethods | Необязательно | По умолчанию – <code>null</code> . Это свойство ожидает массив строк. Указывает заголовок <code>Allow-Methods</code> . |
| ListeningHost.CrossOriginResourceSharingPolicy.AllowHeaders | Необязательно | По умолчанию – <code>null</code> . Это свойство ожидает массив строк. Указывает |

| Свойство | Обязательное | Описание |
|---|---------------|---|
| | | заголовок Allow-Headers . |
| ListeningHost.CrossOriginResourceSharingPolicy.MaxAge | Необязательно | По умолчанию – null . Это свойство ожидает целое число. Указывает заголовок Max-Age в секундах. |
| ListeningHost.Parameters | Необязательно | Указывает свойства, предоставляемые методу настройки приложения. |

Провайдер конфигурации INI

Sisk имеет метод для получения конфигураций запуска, отличных от JSON. На самом деле, любой конвейер, реализующий [IConfigurationReader](#), может быть использован с [PortableConfigurationBuilder.WithConfigurationPipeline](#), читая конфигурацию сервера из любого типа файла.

Пакет [Sisk.IniConfiguration](#) предоставляет потоковый читатель файлов INI, который не выбрасывает исключения для обычных синтаксических ошибок и имеет простой синтаксис конфигурации. Этот пакет можно использовать вне рамок фреймворка Sisk, предлагая гибкость для проектов, требующих эффективного считывателя документов INI.

Установка

Чтобы установить пакет, можно начать с:

```
$ dotnet add package Sisk.IniConfiguration
```

Также можно установить основной пакет, который не включает в себя INI [IConfigurationReader](#), ни зависимость от Sisk, только сериализаторы INI:

```
$ dotnet add package Sisk.IniConfiguration.Core
```

С основным пакетом можно использовать его в коде, как показано в примере ниже:

```
1  class Program
2  {
3      static HttpContext Host = null!;
4
5      static void Main(string[] args)
6      {
7          Host = HttpServer.CreateBuilder()
8              .UsePortableConfiguration(config =>
9              {
10                  config.WithConfigFile("app.ini", createIfDontExists: true);
```

```

11             // использует конфигурационный читатель IniConfigurationReader
12             config.WithConfigurationPipeline<IniConfigurationReader>();
13         }
14     .UseRouter(r =>
15     {
16         r.MapGet("/", SayHello);
17     })
18     .Build();
19
20     Host.Start();
21 }
22
23
24     static HttpResponse SayHello(HttpContext request)
25     {
26         string? name = Host.Parameters["name"] ?? "world";
27         return new HttpResponse($"Hello, {name}!");
28     }
29 }
```

Код выше будет искать файл app.ini в текущем каталоге процесса (CurrentDirectory). Файл INI выглядит так:

```

1 [Server]
2 # Множественные адреса прослушивания поддерживаются
3 Listen = http://localhost:5552/
4 Listen = http://localhost:5553/
5 ThrowExceptions = false
6 AccessLogsStream = console
7
8 [Cors]
9 AllowMethods = GET, POST
10 AllowHeaders = Content-Type, Authorization
11 AllowOrigin = *
12
13 [Parameters]
14 Name = "Kanye West"
```

Вкус и синтаксис INI

Текущая реализация вкуса:

- Имена свойств и секций **не чувствительны к регистру**.
- Имена свойств и значения **обрезаются**, если значения не заключены в кавычки.
- Значения можно заключать в одинарные или двойные кавычки. Кавычки могут содержать переносы строк внутри себя.
- Комментарии поддерживаются с помощью # и ; . Также **допускаются конечные комментарии**.
- Свойства могут иметь несколько значений.

Подробнее, документация для "вкуса" парсера INI, используемого в Sisk, доступна [в этом документе](#).

Используя следующий код INI в качестве примера:

```
1  One = 1
2  Value = это значение
3  Another value = "это значение
4      имеет перенос строки на нем"
5
6  ; код ниже имеет некоторые цвета
7  [some section]
8  Color = Red
9  Color = Blue
10 Color = Yellow ; не используйте желтый
```

Парсить его с помощью:

```
1 // парсить текст INI из строки
2 IniDocument doc = IniDocument.FromString(iniText);
3
4 // получить одно значение
5 string? one = doc.Global.GetOne("one");
6 string? anotherValue = doc.Global.GetOne("another value");
7
8 // получить несколько значений
9 string[]? colors = doc.GetSection("some section")?.GetMany("color");
```

Параметры конфигурации

| Секция и имя | Разрешить несколько значений | Описание |
|-------------------------------|---|---|
| Server.Listen | Да | Адреса и порты прослушивания сервера. |
| Server.Encoding | Нет | Кодировка сервера по умолчанию. |
| Server.MaximumContentLength | Нет | Максимальный размер содержимого в байтах. |
| Server.IncludeRequestIdHeader | Нет | Указывает, должен ли HTTP-сервер отправлять заголовок X-Request-Id. |
| Server.ThrowExceptions | Нет | Указывает, должны ли быть выброшены необработанные исключения. |
| Server.AccessLogsStream | Нет | Указывает поток вывода журнала доступа. |
| Server.ErrorsLogsStream | Нет | Указывает поток вывода журнала ошибок. |
| Cors.AllowMethods | Нет | Указывает значение заголовка CORS Allow-Methods. |
| Cors.AllowHeaders | Нет | Указывает значение заголовка CORS Allow-Headers. |
| Cors.AllowOrigins | Нет | Указывает несколько заголовков Allow-Origin, разделенных запятыми. AllowOrigins для более подробной информации. |
| Cors.AllowOrigin | Нет | Указывает один заголовок Allow-Origin. |
| Cors.ExposeHeaders | Нет | Указывает значение заголовка CORS Expose-Headers. |
| Cors.AllowCredentials | Нет | Указывает значение заголовка CORS Allow-Credentials. |
| Cors.MaxAge | Нет | Указывает значение заголовка CORS Max-Age. |

Руководство (расширенная) настройка

В этом разделе мы создадим наш HTTP-сервер без каких-либо предопределенных стандартов, совершенно абстрактным способом. Здесь вы можете вручную настроить, как будет функционировать ваш HTTP-сервер. Каждый ListeningHost имеет маршрутизатор, и HTTP-сервер может иметь несколько ListeningHosts, каждый из которых указывает на другой хост на другом порту.

Сначала нам нужно понять концепцию запроса/ответа. Это довольно просто: на каждый запрос должен быть ответ. Sisk также следует этому принципу. Давайте создадим метод, который отвечает сообщением "Hello, World!" в HTML, указывая статусный код и заголовки.

```
1 // Program.cs
2 using Sisk.Core.Http;
3 using Sisk.Core.Routing;
4
5 static HttpResponse IndexPage(HttpServletRequest request)
6 {
7     HttpResponse indexResponse = new HttpResponse
8     {
9         Status = System.Net HttpStatusCode.OK,
10        Content = new HtmlContent(@"
11            <html>
12                <body>
13                    <h1>Привет, мир!</h1>
14                </body>
15            </html>
16        ")
17    };
18
19    return indexResponse;
20 }
```

Следующий шаг - связать этот метод с HTTP-маршрутом.

Маршрутизаторы

Маршрутизаторы являются абстракциями запросов и служат мостом между запросами и ответами для службы. Маршрутизаторы управляют маршрутами службы, функциями и ошибками.

Маршрутизатор может иметь несколько маршрутов, и каждый маршрут может выполнять разные операции на этом пути, такие как выполнение функции, служба страницы или предоставление ресурса с сервера.

Давайте создадим наш первый маршрутизатор и свяжем наш метод `IndexPage` с индексным путем.

```
1 Router mainRouter = new Router();
2
3 // SetRoute будет связывать все индексные маршруты с нашим методом.
4 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage);
```

Теперь наш маршрутизатор может получать запросы и отправлять ответы. Однако `mainRouter` не привязан к хосту или серверу, поэтому он не будет работать самостоятельно. Следующий шаг – создать наш `ListeningHost`.

Listening Hosts и Порты

`ListeningHost` может хостить маршрутизатор и несколько прослушиваемых портов для одного и того же маршрутизатора. `ListeningPort` – это префикс, где HTTP-сервер будет слушать.

Здесь мы можем создать `ListeningHost`, который указывает на два порта для нашего маршрутизатора:

```
1 ListeningHost myHost = new ListeningHost
2 {
3     Router = new Router(),
4     Ports = new ListeningPort[]
5     {
6         new ListeningPort("http://localhost:5000/")
7     }
8 };
```

Теперь наш HTTP-сервер будет слушать указанные порты и перенаправлять свои запросы на наш маршрутизатор.

Настройка Сервера

Настройка сервера отвечает за большую часть поведения HTTP-сервера. В этой конфигурации мы можем связать ListeningHosts с нашим сервером.

```
1  HttpServerConfiguration config = new HttpServerConfiguration();
2  config.ListeningHosts.Add(myHost); // Добавляем наш ListeningHost в эту
    конфигурацию сервера
```

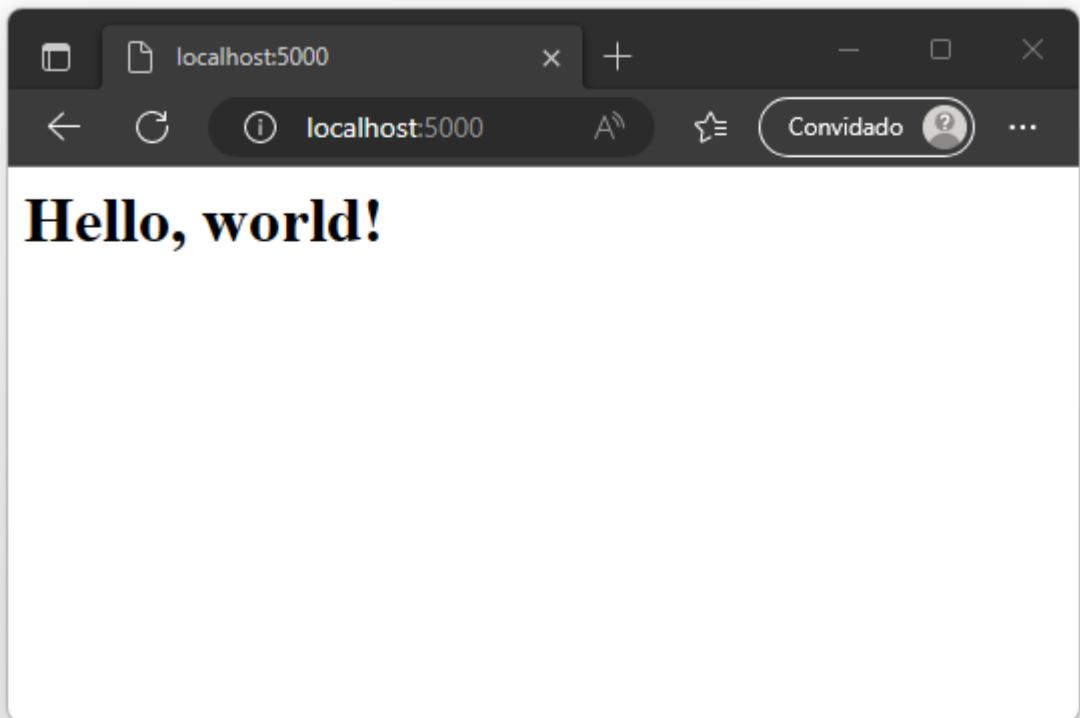
Далее мы можем создать наш HTTP-сервер:

```
1  HttpServer server = new HttpServer(config);
2  server.Start(); // Запускает сервер
3  Console.ReadKey(); // Предотвращает выход из приложения
```

Теперь мы можем скомпилировать наш исполняемый файл и запустить наш HTTP-сервер с командой:

```
dotnet watch
```

Во время выполнения откройте браузер и перейдите по пути сервера, и вы должны увидеть:



Жизненный цикл запроса

Ниже объясняется весь жизненный цикл запроса на примере HTTP-запроса.

- **Приём запроса:** каждый запрос создаёт контекст HTTP между самим запросом и ответом, который будет доставлен клиенту. Этот контекст поступает от встроенного слушателя в Sisk, который может быть [HttpListener](#), [Kestrel](#) или [Cadente](#).
 - Внешняя валидация запроса: выполняется валидация [HttpServerConfiguration.RemoteRequestsAction](#) для запроса.
 - Если запрос внешний и свойство равно `Drop`, соединение закрывается без ответа клиенту с `HttpServerExecutionStatus = RemoteRequestDropped`.
 - Конфигурация разрешения пересылки: если настроен [ForwardingResolver](#), вызывается метод [OnResolveRequestHost](#) для исходного хоста запроса.
 - Сопоставление DNS: с разрешённым хостом и более чем одним настроенным [ListeningHost](#), сервер ищет соответствующий хост для запроса.
 - Если ни один [ListeningHost](#) не соответствует, клиенту возвращается ответ `400 Bad Request` и статус `HttpServerExecutionStatus = DnsUnknownHost` возвращается контексту HTTP.
 - Если [ListeningHost](#) соответствует, но его [Router](#) ещё не инициализирован, клиенту возвращается ответ `503 Service Unavailable` и статус `HttpServerExecutionStatus = ListeningHostNotReady` возвращается контексту HTTP.
 - Связывание маршрутизатора: маршрутизатор соответствующего [ListeningHost](#) связывается с полученным HTTP-сервером.
 - Если маршрутизатор уже связан с другим HTTP-сервером, что не допускается, поскольку маршрутизатор активно использует ресурсы конфигурации сервера, выбрасывается исключение `InvalidOperationException`. Это происходит только во время инициализации HTTP-сервера, а не во время создания контекста HTTP.
 - Предварительное определение заголовков:
 - Предварительно определяет заголовок `X-Request-Id` в ответе, если он настроен.
 - Предварительно определяет заголовок `X-Powered-By` в ответе, если он настроен.
 - Валидация размера содержимого: проверяет, является ли содержимое запроса меньше [HttpServerConfiguration.MaximumContentSize](#), только если оно больше нуля.
 - Если запрос отправляет `Content-Length` больше настроенного, клиенту возвращается ответ `413 Payload Too Large` и статус `HttpServerExecutionStatus = ContentTooLarge` возвращается контексту HTTP.
 - Событие `OnHttpRequestOpen` вызывается для всех настроенных обработчиков HTTP-сервера.
- **Маршрутизация действия:** сервер вызывает маршрутизатор для полученного запроса.
 - Если маршрутизатор не находит маршрут, соответствующий запросу:

- Если свойство [Router.NotFoundErrorHandler](#) настроено, вызывается действие, и ответ действия пересыпается HTTP-клиенту.
 - Если предыдущее свойство равно null, клиенту возвращается ответ 404 Not Found по умолчанию.
- Если маршрутизатор находит соответствующий маршрут, но метод маршрута не соответствует методу запроса:
 - Если свойство [Router.MethodNotAllowedErrorHandler](#) настроено, вызывается действие, и ответ действия пересыпается HTTP-клиенту.
 - Если предыдущее свойство равно null, клиенту возвращается ответ 405 Method Not Allowed по умолчанию.
 - Если запрос имеет метод `OPTIONS` :
 - Маршрутизатор возвращает ответ 200 Ok клиенту только в том случае, если ни один маршрут не соответствует методу запроса (метод маршрута не явно указан как [RouteMethod.Options](#)).
 - Если свойство [HttpServerConfiguration.ForceTrailingSlash](#) включено, соответствующий маршрут не является регулярным выражением, путь запроса не заканчивается на `/`, и метод запроса равен `GET` :
 - Клиенту возвращается ответ 307 Temporary Redirect с заголовком `Location` , содержащим путь и запрос к тому же месту с `/` в конце.
 - Событие `OnContextBagCreated` вызывается для всех настроенных обработчиков HTTP-сервера.
 - Все глобальные экземпляры [IRequestHandler](#) с флагом `BeforeResponse` выполняются.
 - Если любой обработчик возвращает непустой ответ, этот ответ пересыпается HTTP-клиенту, и контекст закрывается.
 - Если во время этого шага возникает ошибка и [HttpServerConfiguration.ThrowExceptions](#) отключено:
 - Если свойство [Router.CallbackErrorHandler](#) включено, оно вызывается, и полученный ответ возвращается клиенту.
 - Если предыдущее свойство не определено, серверу возвращается пустой ответ, который пересыпает ответ в зависимости от типа возбуждённого исключения, обычно 500 Internal Server Error.
 - Все экземпляры [IRequestHandler](#), определённые в маршруте и имеющие флаг `BeforeResponse` , выполняются.
 - Если любой обработчик возвращает непустой ответ, этот ответ пересыпается HTTP-клиенту, и контекст закрывается.
 - Если во время этого шага возникает ошибка и [HttpServerConfiguration.ThrowExceptions](#) отключено:
 - Если свойство [Router.CallbackErrorHandler](#) включено, оно вызывается, и полученный ответ возвращается клиенту.
 - Если предыдущее свойство не определено, серверу возвращается пустой ответ, который пересыпает ответ в зависимости от типа возбуждённого исключения,

обычно 500 Internal Server Error.

- Действие маршрутизатора вызывается и преобразуется в HTTP-ответ.
 - Если во время этого шага возникает ошибка и [HttpServerConfiguration.ThrowExceptions](#) отключено:
 - Если свойство [Router.CallbackErrorHandler](#) включено, оно вызывается, и полученный ответ возвращается клиенту.
 - Если предыдущее свойство не определено, серверу возвращается пустой ответ, который пересыпает ответ в зависимости от типа возбуждённого исключения, обычно 500 Internal Server Error.
 - Все глобальные экземпляры [IRequestHandler](#) с флагом `AfterResponse` выполняются.
 - Если любой обработчик возвращает непустой ответ, ответ обработчика заменяет предыдущий ответ и сразу пересыпается HTTP-клиенту.
 - Если во время этого шага возникает ошибка и [HttpServerConfiguration.ThrowExceptions](#) отключено:
 - Если свойство [Router.CallbackErrorHandler](#) включено, оно вызывается, и полученный ответ возвращается клиенту.
 - Если предыдущее свойство не определено, серверу возвращается пустой ответ, который пересыпает ответ в зависимости от типа возбуждённого исключения, обычно 500 Internal Server Error.
 - Все экземпляры [IRequestHandler](#), определённые в маршруте и имеющие флаг `AfterResponse`, выполняются.
 - Если любой обработчик возвращает непустой ответ, ответ обработчика заменяет предыдущий ответ и сразу пересыпается HTTP-клиенту.
 - Если во время этого шага возникает ошибка и [HttpServerConfiguration.ThrowExceptions](#) отключено:
 - Если свойство [Router.CallbackErrorHandler](#) включено, оно вызывается, и полученный ответ возвращается клиенту.
 - Если предыдущее свойство не определено, серверу возвращается пустой ответ, который пересыпает ответ в зависимости от типа возбуждённого исключения, обычно 500 Internal Server Error.
- **Обработка ответа:** с готовым ответом сервер готовит его для отправки клиенту.
 - Заголовки Cross-Origin Resource Sharing Policy (CORS) определяются в ответе в соответствии с настройками текущего [ListeningHost.CrossOriginResourceSharingPolicy](#).
 - Код состояния и заголовки ответа отправляются клиенту.
 - Содержимое ответа отправляется клиенту:
 - Если содержимое ответа является потомком [ByteArrayContent](#), байты ответа **直接** копируются в поток вывода ответа.
 - Если предыдущее условие не выполнено, ответ сериализуется в поток и копируется в поток вывода ответа.
 - Потоки закрываются, и содержимое ответа удаляется.

- Если [HttpServerConfiguration.DisposeDisposableContextValues](#) включено, все объекты, определённые в контексте запроса и наследующие [IDisposable](#), удаляются.
- Событие [OnHttpRequestClose](#) вызывается для всех настроенных обработчиков HTTP-сервера.
- Если на сервере возникло исключение, событие [OnException](#) вызывается для всех настроенных обработчиков HTTP-сервера.
- Если маршрут позволяет доступ-логирование и [HttpServerConfiguration.AccessLogsStream](#) не равно null, строка лога записывается в выходной лог.
- Если маршрут позволяет ошибочно-логирование, есть исключение, и [HttpServerConfiguration.ErrorsLogsStream](#) не равно null, строка лога записывается в выходной лог ошибок.
- Если сервер ожидает запрос через [HttpServer.WaitNext](#), мьютекс освобождается, и контекст становится доступным пользователю.

Перенаправляющие Резолверы

Перенаправляющий Резолвер – это помощник, который помогает декодировать информацию, идентифицирующую клиента через запрос, прокси, CDN или балансировщики нагрузки. Когда ваш сервис Sisk запускается через обратный или прямой прокси, IP-адрес клиента, хост и протокол могут быть khácными от исходного запроса, поскольку это перенаправление из одного сервиса в другой. Эта функциональность Sisk позволяет вам контролировать и решать эту информацию перед работой с запросом. Эти прокси обычно предоставляют полезные заголовки для идентификации их клиента.

В настоящее время с помощью класса `ForwardingResolver` возможно решить IP-адрес клиента, хост и HTTP-протокол, используемый. После версии 1.0 Sisk сервер больше не имеет стандартной реализации для декодирования этих заголовков по причинам безопасности, которые варьируются от сервиса к сервису.

Например, заголовок `X-Forwarded-For` содержит информацию об IP-адресах, которые перенаправили запрос. Этот заголовок используется прокси для переноса цепочки информации к конечному сервису и включает IP-адрес всех прокси, использованных, включая реальный адрес клиента. Проблема в том, что иногда бывает сложно идентифицировать удаленный IP-адрес клиента, и нет конкретного правила для идентификации этого заголовка. Высоко рекомендуется прочитать документацию для заголовков, которые вы собираетесь реализовать ниже:

- Прочтите о заголовке `X-Forwarded-For` [здесь](#).
- Прочтите о заголовке `X-Forwarded-Host` [здесь](#).
- Прочтите о заголовке `X-Forwarded-Proto` [здесь](#).

Класс ForwardingResolver

Этот класс имеет три виртуальных метода, которые позволяют наиболее подходящую реализацию для каждого сервиса. Каждый метод отвечает за решение информации из запроса через прокси: IP-адрес клиента, хост запроса и протокол безопасности, используемый. По умолчанию Sisk всегда будет использовать информацию из исходного запроса, не решая никаких заголовков.

Пример ниже показывает, как можно использовать эту реализацию. Этот пример решает IP-адрес клиента через заголовок `X-Forwarded-For` и выбрасывает ошибку, когда более одного IP-адреса было

перенаправлено в запросе.

⊗ IMPORTANT

Не используйте этот пример в производственном коде. Всегда проверяйте, является ли реализация подходящей для использования. Прочитайте документацию заголовков перед их реализацией.

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          using var host = HttpServer.CreateBuilder()
6              .UseForwardingResolver<Resolver>()
7              .UseListeningPort(5555)
8              .Build();
9
10         host.Router.SetRoute(RouteMethod.Any, Route.AnyPath, request =>
11             new HttpResponseMessage("Hello, world!!!"));
12
13         host.Start();
14     }
15
16     class Resolver : ForwardingResolver
17     {
18         public override IPAddress OnResolveClientAddress(HttpContext request,
19         IPEndPoint connectingEndpoint)
20         {
21             string? forwardedFor = request.Headers.XForwardedFor;
22             if (forwardedFor is null)
23             {
24                 throw new Exception("Заголовок X-Forwarded-For отсутствует.");
25             }
26             string[] ipAddresses = forwardedFor.Split(',');
27             if (ipAddresses.Length != 1)
28             {
29                 throw new Exception("Слишком много адресов в заголовке X-Forwarded-For.");
30             }
31
32             return IPAddress.Parse(ipAddresses[0]);
33         }
34     }
}
```

Обработчики HTTP-сервера

В версии Sisk 0.16 мы ввели класс `HttpServerHandler`, целью которого является расширение общего поведения Sisk и предоставление дополнительных обработчиков событий для Sisk, таких как обработка запросов HTTP, маршрутизаторы, контекстные сумки и многое другое.

Этот класс концентрирует события, которые происходят во время существования всего HTTP-сервера и каждого запроса. Протокол HTTP не имеет сессий, и поэтому невозможно сохранить информацию от одного запроса к другому. Sisk в настоящее время предоставляет способ реализации сессий, контекстов, подключений к базе данных и других полезных провайдеров, чтобы помочь вашей работе.

Пожалуйста, обратитесь к [этой странице](#), чтобы прочитать, где каждое событие вызывается и какова его цель. Вы также можете просмотреть [жизненный цикл запроса HTTP](#), чтобы понять, что происходит с запросом и где вызываются события. HTTP-сервер позволяет использовать несколько обработчиков одновременно. Каждый вызов события является синхронным, то есть он будет блокировать текущую нить для каждого запроса или контекста, пока все обработчики, связанные с этой функцией, не будут выполнены и завершены.

В отличие от `RequestHandlers`, они не могут быть применены к определенным группам маршрутов или конкретным маршрутам. Вместо этого они применяются к [весь](#) HTTP-серверу. Вы можете применять условия внутри вашего обработчика HTTP-сервера. Кроме того, синглтоны каждого `HttpServerHandler` определяются для каждого приложения Sisk, поэтому существует только один экземпляр на `HttpServerHandler`.

Практическим примером использования `HttpServerHandler` является автоматическое освобождение подключения к базе данных в конце запроса.

```
1 // DatabaseConnectionHandler.cs
2
3 public class DatabaseConnectionHandler : HttpServerHandler
4 {
5     public override void OnHttpRequestClose(HttpServerExecutionResult result)
6     {
7         var requestBag = result.Request.Context.RequestBag;
8
9         // проверяет, определена ли DbContext в контекстной сумке запроса
10        if (requestBag.IsSet<DbContext>())
11        {
12            var db = requestBag.Get<DbContext>();
13            db.Dispose();
```

```

14         }
15     }
16 }
17
18 public static class DatabaseConnectionHandlerExtensions
19 {
20     // позволяет пользователю создать контекст базы данных из запроса HTTP
21     // и сохранить его в контекстной сумке
22     public static DbContext GetDbContext(this HttpRequest request)
23     {
24         var db = new DbContext();
25         return request.SetContextBag<DbContext>(db);
26     }
27 }
```

С помощью кода выше, расширение `GetDbContext` позволяет создать контекст подключения к базе данных [直接](#) из объекта `HttpRequest`. Неправильно освобожденное подключение может вызывать проблемы при работе с базой данных, поэтому оно завершается в `OnHttpRequestClose`.

Вы можете зарегистрировать обработчик на HTTP-сервере в вашем построителе или [直接](#) с помощью `HttpServer.RegisterHandler`.

```

1 // Program.cs
2
3 class Program
4 {
5     static void Main(string[] args)
6     {
7         using var app = HttpServer.CreateBuilder()
8             .UseHandler<DatabaseConnectionHandler>()
9             .Build();
10
11         app.Router.SetObject(new UserController());
12         app.Start();
13     }
14 }
```

С помощью этого, класс `UsersController` может использовать контекст базы данных следующим образом:

```

1 // UserController.cs
2
3 [RoutePrefix("/users")]
4 public class UserController : ApiController
5 {
```

```

6     [RouteGet()]
7     public async Task<HttpResponse> List(HttpContext request)
8     {
9         var db = request.GetDbContext();
10        var users = db.Users.ToArray();
11
12        return JsonOk(users);
13    }
14
15    [RouteGet("<id>")]
16    public async Task<HttpResponse> View(HttpContext request)
17    {
18        var db = request.GetDbContext();
19
20        var userId = request.GetQueryValue<int>("id");
21        var user = db.Users.FirstOrDefault(u => u.Id == userId);
22
23        return JsonOk(user);
24    }
25
26    [RoutePost]
27    public async Task<HttpResponse> Create(HttpContext request)
28    {
29        var db = request.GetDbContext();
30        var user = JsonSerializer.Deserialize<User>(request.Body);
31
32        ArgumentNullException.ThrowIfNull(user);
33
34        db.Users.Add(user);
35        await db.SaveChangesAsync();
36
37        return JsonMessage("Пользователь добавлен.");
38    }
39}

```

Код выше использует методы, такие как `JsonOk` и `JsonMessage`, которые встроены в `ApiController`, который наследуется от `RouterController`:

```

1 // ApiController.cs
2
3 public class ApiController : RouterModule
4 {
5     public HttpResponse JsonOk(object value)
6     {
7         return new HttpResponse(200)
8             .WithContent(JsonContent.Create(value, null, new JsonSerializerOptions()))

```

```
9             {
10                 PropertyNameCaseInsensitive = true
11             }));
12         }
13
14     public HttpResponseMessage JsonMessage(string message, int statusCode = 200)
15     {
16         return new HttpResponseMessage(statusCode)
17             .WithContent(JsonContent.Create(new
18             {
19                 Message = message
20             }));
21     }
22 }
```

Разработчики могут реализовать сессии, контексты и подключения к базе данных, используя этот класс. Предоставленный код демонстрирует практический пример с DatabaseConnectionHandler, автоматизирующий освобождение подключения к базе данных в конце каждого запроса.

Интеграция проста, с обработчиками, зарегистрированными во время настройки сервера. Класс HttpServerHandler предлагает мощный набор инструментов для управления ресурсами и расширения поведения Sisk в HTTP-приложениях.

Несколько прослушивающих хостов на сервере

Фреймворк Sisk всегда поддерживал использование более одного хоста на сервер, то есть один HTTP-сервер может прослушивать несколько портов, и каждый порт имеет свой собственный маршрутизатор и свою службу, работающую на нем.

Таким образом, легко разделить обязанности и управлять службами на одном HTTP-сервере с помощью Sisk. Пример ниже показывает создание двух прослушивающих хостов, каждый из которых прослушивает разный порт, с разными маршрутизаторами и действиями.

Прочтите [создание приложения вручную](#), чтобы понять детали об этом абстрактном классе.

```
1 static void Main(string[] args)
2 {
3     // создаем два прослушивающих хоста, каждый из которых имеет свой собственный
4     // маршрутизатор и
5     // прослушивает свой собственный порт
6     //
7     ListeningHost hostA = new ListeningHost();
8     hostA.Ports = [new ListeningPort(12000)];
9     hostA.Router = new Router();
10    hostA.Router.SetRoute(RouteMethod.Get, "/", request => new
11        HttpResponseMessage().WithContent("Привет от хоста А!"));
12
13    ListeningHost hostB = new ListeningHost();
14    hostB.Ports = [new ListeningPort(12001)];
15    hostB.Router = new Router();
16    hostB.Router.SetRoute(RouteMethod.Get, "/", request => new
17        HttpResponseMessage().WithContent("Привет от хоста Б!"));
18
19    // создаем конфигурацию сервера и добавляем оба
20    // прослушивающих хоста в нее
21    //
22    HttpServerConfiguration configuration = new HttpServerConfiguration();
23    configuration.ListeningHosts.Add(hostA);
24    configuration.ListeningHosts.Add(hostB);
25
26    // создаем HTTP-сервер, который использует указанную
27    // конфигурацию
28    //
```

```
29     HttpServer server = new HttpServer(configuration);
30
31     // запускаем сервер
32     server.Start();
33
34     Console.WriteLine("Попробуйте обратиться к хосту А по адресу
35 {0}", server.ListeningPrefixes[0]);
36     Console.WriteLine("Попробуйте обратиться к хосту Б по адресу
37 {0}", server.ListeningPrefixes[1]);
38
39     Thread.Sleep(-1);
40 }
```