Erste Schritte

Willkommen in der Sisk-Dokumentation!

Schließlich, was ist das Sisk-Framework? Es ist eine Open-Source-Bibliothek, die mit .NET erstellt wurde, und minimalistisch, flexibel und abstrakt konzipiert ist. Sie ermöglicht es Entwicklern, Internetdienste schnell zu erstellen, mit wenig oder keiner notwendigen Konfiguration. Sisk ermöglicht es Ihrer bestehenden Anwendung, ein verwaltetes HTTP-Modul zu haben, das vollständig und entsorgbar ist.

Die Werte von Sisk umfassen Code-Transparenz, Modularität, Leistung und Skalierbarkeit und können verschiedene Arten von Anwendungen verarbeiten, wie z.B. Restful, JSON-RPC, Web-Sockets und mehr.

Die wichtigsten Funktionen umfassen:

Ressource	Beschreibung
Routing	Ein Pfad-Router, der Präfixe, benutzerdefinierte Methoden, Pfadvariablen, Wertkonverter und mehr unterstützt.
Request-Handler	Auch bekannt als *Middleware*, bietet eine Schnittstelle, um eigene Request-Handler zu erstellen, die mit der Anfrage vor oder nach einer Aktion arbeiten.
Komprimierung	Komprimieren Sie den Inhalt Ihrer Antwort einfach mit Sisk.
Web-Sockets	Bietet Routen, die vollständige Web-Sockets akzeptieren, für das Lesen und Schreiben an den Client.
Server-sent Events	Bietet das Senden von Server-Ereignissen an Clients, die das SSE-Protokoll unterstützen.
Protokollierung	Vereinfachte Protokollierung. Protokollieren Sie Fehler, Zugriffe, definieren Sie rotierende Protokolle nach Größe, mehrere Ausgabeströme für das gleiche Protokoll und mehr.
Multi-Host	Haben Sie einen HTTP-Server für mehrere Ports, und jeden Port mit seinem eigenen Router, und jeden Router mit seiner eigenen Anwendung.
Server-Handler	Erweitern Sie Ihre eigene Implementierung des HTTP-Servers. Anpassen Sie mit Erweiterungen, Verbesserungen und neuen Funktionen.

Erste Schritte

Sisk kann in jeder .NET-Umgebung ausgeführt werden. In diesem Leitfaden werden wir Ihnen zeigen, wie Sie eine Sisk-Anwendung mit .NET erstellen. Wenn Sie es noch nicht installiert haben, laden Sie bitte das SDK von hier herunter.

In diesem Tutorial werden wir zeigen, wie Sie eine Projektstruktur erstellen, eine Anfrage empfangen, einen URL-Parameter abrufen und eine Antwort senden. Dieser Leitfaden konzentriert sich auf den Aufbau eines einfachen Servers mit C#. Sie können auch Ihre bevorzugte Programmiersprache verwenden.

> **NOTE**
>
> Sie könnten an einem Quickstart-Projekt interessiert sein. Überprüfen Sie dieses Repository für weitere Informationen.

Projekt erstellen

Nennen wir unser Projekt "Meine Sisk-Anwendung". Sobald Sie .NET eingerichtet haben, können Sie Ihr Projekt mit dem folgenden Befehl erstellen:

```
dotnet new console -n meine-sisk-anwendung
```

Navigieren Sie als Nächstes zu Ihrem Projektverzeichnis und installieren Sie Sisk mit dem .NET-Utility-Tool:

```
1    cd meine-sisk-anwendung
2    dotnet add package Sisk.HttpServer
```

Sie können weitere Möglichkeiten finden, Sisk in Ihrem Projekt zu installieren, hier.

Lassen Sie uns nun eine Instanz unseres HTTP-Servers erstellen. Für dieses Beispiel werden wir es so konfigurieren, dass es auf Port 5000 hört.

HTTP-Server erstellen

Sisk ermöglicht es Ihnen, Ihre Anwendung Schritt für Schritt manuell aufzubauen, da es Routen zum HttpServer-Objekt ermöglicht. Dies kann jedoch für die meisten Projekte nicht sehr praktisch sein. Daher können wir die Builder-Methode verwenden, die es einfacher macht, unsere Anwendung in Betrieb zu nehmen.

Program.cs C#

```
class Program
{
    static async Task Main(string[] args)
    {
        using var app = HttpServer.CreateBuilder()
            .UseListeningPort("http://localhost:5000/")
            .Build();

        app.Router.MapGet("/", request =>
        {
            return new HttpResponse()
            {
                Status = 200,
                Content = new StringContent("Hallo, Welt!")
            };
        });

        await app.StartAsync();
    }
}
```

Es ist wichtig, jedes wichtige Komponente von Sisk zu verstehen. Später in diesem Dokument werden Sie mehr über die Funktionsweise von Sisk erfahren.

Manuelles (erweitertes) Setup

Sie können erfahren, wie jedes Sisk-Mechanismus funktioniert, in diesem Abschnitt der Dokumentation, der das Verhalten und die Beziehungen zwischen dem HttpServer, Router, ListeningPort und anderen Komponenten erklärt.

Installation

Sie können Sisk über Nuget, dotnet cli oder andere Optionen installieren. Sie können Ihre Sisk-Umgebung leicht einrichten, indem Sie diesen Befehl in Ihrer Entwicklerkonsole ausführen:

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

Dieser Befehl installiert die neueste Version von Sisk in Ihrem Projekt.

Native AOT-Unterstützung

.NET Native AOT ermöglicht die Veröffentlichung von nativen .NET-Anwendungen, die selbstständig sind und nicht die .NET-Laufzeit auf dem Zielhost benötigen. Zusätzlich bietet Native AOT Vorteile wie:

- Erheblich kleinere Anwendungen
- Wesentlich schnellere Initialisierung
- Geringeren Speicherbedarf

Das Sisk Framework ermöglicht aufgrund seiner expliziten Natur die Verwendung von Native AOT für fast alle seine Funktionen, ohne dass eine Überarbeitung des Quellcodes erforderlich ist, um es an Native AOT anzupassen.

Nicht unterstützte Funktionen

Allerdings verwendet Sisk Reflexion, wenn auch minimal, für einige Funktionen. Die nachfolgend genannten Funktionen sind möglicherweise teilweise verfügbar oder während der nativen Codeausführung ganz nicht verfügbar:

- Automatisches Scannen von Modulen des Routers: Diese Ressource scannet die im ausführenden Assembly eingebetteten Typen und registriert die Typen, die Router-Module sind. Diese Ressource benötigt Typen, die während des Assembly-Trimming ausgeschlossen werden können.

Alle anderen Funktionen sind mit AOT in Sisk kompatibel. Es ist üblich, eine oder andere Methode zu finden, die eine AOT-Warnung ausgibt, aber dieselbe, wenn sie nicht hier erwähnt wird, hat eine Überladung, die das Übergeben eines Typs, Parameters oder Typinformationen anzeigt, die dem AOT-Compiler helfen, das Objekt zu kompilieren.

Bereitstellung Ihrer Sisk-Anwendung

Der Prozess der Bereitstellung einer Sisk-Anwendung besteht darin, Ihr Projekt in die Produktion zu veröffentlichen. Obwohl der Prozess relativ einfach ist, ist es wichtig, Details zu beachten, die für die Sicherheit und Stabilität der Infrastruktur der Bereitstellung von entscheidender Bedeutung sein können.

Idealerweise sollten Sie bereit sein, Ihre Anwendung in die Cloud zu veröffentlichen, nachdem Sie alle möglichen Tests durchgeführt haben, um Ihre Anwendung bereit zu machen.

Veröffentlichen Ihrer App

Das Veröffentlichen Ihrer Sisk-Anwendung oder eines Dienstes bedeutet, Binärdateien zu generieren, die für die Produktion bereit und optimiert sind. In diesem Beispiel werden wir die Binärdateien für die Produktion kompilieren, um auf einem Computer mit der .NET-Laufzeitumgebung zu laufen.

Sie benötigen die .NET-SDK auf Ihrem Computer, um Ihre App zu erstellen, und die .NET-Laufzeitumgebung auf dem Zielserver, um Ihre App auszuführen. Sie können erfahren, wie Sie die .NET-Laufzeitumgebung auf Ihrem Linux-Server hier, Windows und Mac OS installieren.

Öffnen Sie im Ordner, in dem sich Ihr Projekt befindet, ein Terminal und verwenden Sie den .NET-Veröffentlichungsbefehl:

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

Dies generiert Ihre Binärdateien im Ordner `bin/Release/publish/linux-x64`.

> **NOTE**
>
> Wenn Ihre App mit dem Sisk.ServiceProvider-Paket läuft, sollten Sie Ihre `service-config.json` in den Host-Server zusammen mit allen Binärdateien kopieren, die von `dotnet publish` generiert werden. Sie können die Datei vor konfigurieren, mit Umgebungsvariablen, Lauscher-Ports und -Hosts sowie zusätzlichen Server-Konfigurationen.

Der nächste Schritt besteht darin, diese Dateien auf den Server zu übertragen, auf dem Ihre Anwendung gehostet wird.

Anschließend geben Sie der Binärdatei Ausführungsrechte. In diesem Fall nehmen wir an, dass unser Projektname "my-app" ist:

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

Nachdem Sie Ihre Anwendung gestartet haben, überprüfen Sie, ob sie Fehlermeldungen produziert. Wenn sie keine Fehlermeldungen produziert, bedeutet dies, dass Ihre Anwendung läuft.

An diesem Punkt ist es wahrscheinlich nicht möglich, auf Ihre Anwendung von außerhalb des Servers zuzugreifen, da Zugriffsregeln wie Firewall nicht konfiguriert sind. Wir werden dies in den nächsten Schritten berücksichtigen.

Sie sollten die Adresse des virtuellen Hosts haben, auf dem Ihre Anwendung läuft. Dies wird manuell in der Anwendung festgelegt und hängt davon ab, wie Sie Ihren Sisk-Dienst instanziieren.

Wenn Sie **nicht** das Sisk.ServiceProvider-Paket verwenden, sollten Sie es finden, wo Sie Ihre HttpServer-Instanz definiert haben:

```
HttpServer server = HttpServer.Emit(5000, out HttpServerConfiguration config, out var host,
out var router);
// sisk sollte auf http://localhost:5000/ lauschen
```

Manuelle Zuweisung eines ListeningHost:

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

Oder wenn Sie das Sisk.ServiceProvider-Paket verwenden, in Ihrer `service-config.json`:

```
{
  "Server": { },
  "ListeningHost": {
    "Ports": [
      "http://localhost:5000/"
    ]
  }
}
```

Daraus können wir einen Reverse-Proxy erstellen, um Ihren Dienst zu hören und den Datenverkehr über das offene Netzwerk verfügbar zu machen.

Proxying Ihrer Anwendung

Das Proxying Ihres Dienstes bedeutet, dass Sie Ihren Sisk-Dienst nicht direkt einem externen Netzwerk aussetzen. Diese Praxis ist sehr häufig bei Server-Bereitstellungen, da:

- Sie damit ein SSL-Zertifikat in Ihrer Anwendung verknüpfen können;
- Sie Zugriffsregeln vor dem Zugriff auf den Dienst erstellen und Überlastungen vermeiden können;
- Sie die Bandbreite und Anfragegrenzen kontrollieren können;
- Sie Lastenausgleich für Ihre Anwendung trennen können;
- Sie Sicherheitsschäden an der fehlgeschlagenen Infrastruktur verhindern können.

Sie können Ihre Anwendung durch einen Reverse-Proxy wie Nginx oder Apache bereitstellen, oder Sie können einen http-over-dns-Tunnel wie Cloudflared verwenden.

Außerdem sollten Sie daran denken, die Weiterleitungsheader Ihres Proxys korrekt aufzulösen, um die Informationen Ihres Clients, wie z.B. die IP-Adresse und den Host, über Weiterleitungs-Resolver zu erhalten.

Der nächste Schritt nach der Erstellung Ihres Tunnels, der Firewall-Konfiguration und dem Ausführen Ihrer Anwendung besteht darin, einen Dienst für Ihre Anwendung zu erstellen.

> **NOTE**
>
> Die Verwendung von SSL-Zertifikaten direkt im Sisk-Dienst auf nicht-Windows-Systemen ist nicht möglich. Dies ist ein Punkt der Implementierung von HttpListener, der das zentrale Modul für die HTTP-Warteschlangenverwaltung in Sisk ist, und diese Implementierung variiert von Betriebssystem zu Betriebssystem. Sie können SSL in Ihrem Sisk-Dienst verwenden, wenn Sie ein Zertifikat mit dem virtuellen Host mit IIS verknüpfen. Für andere Systeme wird die Verwendung eines Reverse-Proxys dringend empfohlen.

Erstellen eines Dienstes

Das Erstellen eines Dienstes macht Ihre Anwendung immer verfügbar, auch nach dem Neustart Ihres Servers oder einem nicht wiederherstellbaren Absturz.

In diesem einfachen Tutorial werden wir den Inhalt des vorherigen Tutorials als Showcase verwenden, um Ihren Dienst immer aktiv zu halten.

1. Greifen Sie auf den Ordner zu, in dem sich die Dienstkonfigurationsdateien befinden:

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

2. Erstellen Sie Ihre `my-app.service` -Datei und fügen Sie den Inhalt hinzu:

my-app.service **INI**

```
[Unit]
Description=<Beschreibung über Ihre App>

[Service]
# Setzen Sie den Benutzer, der den Dienst starten wird
User=<Benutzer, der den Dienst starten wird>

# Der ExecStart-Pfad ist nicht relativ zum WorkingDirectory.
# Setzen Sie ihn als vollständigen Pfad zur ausführbaren Datei
WorkingDirectory=/home/htdocs
ExecStart=/home/htdocs/my-app

# Setzen Sie den Dienst so, dass er immer nach einem Absturz neu startet
Restart=always
RestartSec=3

[Install]
WantedBy=multi-user.target
```

3. Starten Sie Ihren Dienst-Manager-Modul neu:

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

4. Starten Sie Ihren neu erstellten Dienst mit dem Namen der Datei, die Sie festgelegt haben, und überprüfen Sie, ob er läuft:

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

5. Wenn Ihre App läuft ("Active: active"), aktivieren Sie Ihren Dienst, um ihn nach einem System-Neustart weiterlaufen zu lassen:

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

Jetzt sind Sie bereit, Ihre Sisk-Anwendung vorzustellen.

Arbeiten mit SSL

Arbeiten mit SSL für die Entwicklung kann notwendig sein, wenn Sie in Kontexten arbeiten, die Sicherheit erfordern, wie z.B. die meisten Webentwicklungsszenarien. Sisk operiert auf Basis von HttpListener, der keine native HTTPS-Unterstützung bietet, sondern nur HTTP. Es gibt jedoch Workarounds, die es Ihnen ermöglichen, mit SSL in Sisk zu arbeiten. Siehe sie unten:

Über IIS auf Windows

- Verfügbar auf: Windows
- Aufwand: mittel

Wenn Sie auf Windows sind, können Sie IIS verwenden, um SSL auf Ihrem HTTP-Server zu aktivieren. Damit dies funktioniert, ist es ratsam, dass Sie diesem Tutorial vorher folgen, wenn Sie möchten, dass Ihre Anwendung auf einem anderen Host als "localhost" hört.

Damit dies funktioniert, müssen Sie IIS über die Windows-Features installieren. IIS ist für Windows- und Windows-Server-Benutzer kostenlos verfügbar. Um SSL in Ihrer Anwendung zu konfigurieren, müssen Sie das SSL-Zertifikat bereit haben, auch wenn es selbstsigniert ist. Als Nächstes können Sie sehen, wie Sie SSL auf IIS 7 oder höher einrichten.

Über mitmproxy

- Verfügbar auf: Linux, macOS, Windows
- Aufwand: einfach

mitmproxy ist ein Interceptions-Proxy-Tool, das es Entwicklern und Sicherheitstestern ermöglicht, HTTP- und HTTPS-Verkehr zwischen einem Client (wie einem Webbrowser) und einem Server zu überwachen, zu modifizieren und aufzuzeichnen. Sie können die **mitmdump**-Utility verwenden, um einen Reverse-SSL-Proxy zwischen Ihrem Client und Ihrer Sisk-Anwendung zu starten.

1. Zuerst installieren Sie mitmproxy auf Ihrem Computer.
2. Starten Sie Ihre Sisk-Anwendung. In diesem Beispiel verwenden wir den Port 8000 als unsicheren HTTP-Port.
3. Starten Sie den mitmproxy-Server, um den sicheren Port 8001 zu hören:

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

Und Sie sind bereit! Sie können Ihre Anwendung bereits über `https://localhost:8001/` aufrufen. Ihre Anwendung muss nicht laufen, damit Sie `mitmdump` starten können.

Alternativ können Sie einen Verweis auf die mitmproxy-Hilfe in Ihrem Projekt hinzufügen. Dies erfordert jedoch, dass mitmproxy auf Ihrem Computer installiert ist.

Über Sisk.SslProxy-Paket

- Verfügbar auf: Linux, macOS, Windows
- Aufwand: einfach

Das Sisk.SslProxy-Paket ist eine einfache Möglichkeit, SSL auf Ihrer Sisk-Anwendung zu aktivieren. Es ist jedoch ein **extrem experimentelles** Paket. Es kann instabil sein, mit diesem Paket zu arbeiten, aber Sie können Teil des kleinen Prozentsatzes von Menschen sein, die dazu beitragen, dieses Paket verwendbar und stabil zu machen. Um loszulegen, können Sie das Sisk.SslProxy-Paket mit installieren:

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

> **NOTE**
>
> Sie müssen "Vorabversionen von Paketen aktivieren" im Visual Studio-Paket-Manager aktivieren, um Sisk.SslProxy zu installieren.

Wiederum ist es ein experimentelles Projekt, also sollten Sie nicht einmal daran denken, es in die Produktion zu übernehmen.

Im Moment kann Sisk.SslProxy die meisten HTTP/1.1-Features verarbeiten, einschließlich HTTP-Continue, Chunked-Encoding, WebSockets und SSE. Lesen Sie mehr über SslProxy hier.

Konfiguration von Namensraumreservierungen auf Windows

Sisk arbeitet mit der HttpListener-Netzwerkschnittstelle, die einen virtuellen Host an das System bindet, um Anfragen zu empfangen.

Unter Windows ist diese Bindung ein bisschen restriktiv und erlaubt nur localhost als gültigen Host. Wenn man versucht, auf einen anderen Host zuzugreifen, wird auf dem Server ein Zugriffsverweigerungsfehler ausgelöst. Dieses Tutorial erklärt, wie man die Autorisierung erteilt, um auf jeden Host auf dem System zuzuhören, den man möchte.

Namespace Setup.bat — BATCH

```batch
@echo off

:: Präfix hier einfügen, ohne Leerzeichen oder Anführungszeichen
SET PREFIX=

SET DOMAIN=%ComputerName%\%USERNAME%
netsh http add urlacl url=%PREFIX% user=%DOMAIN%

pause
```

Wo `PREFIX` das Präfix ("Zuhör-Host→Port") ist, auf das der Server hört. Es muss im URL-Schema, Host, Port und einem Schrägstrich am Ende formatiert sein, Beispiel:

Namespace Setup.bat — BATCH

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

Damit Sie in Ihrer Anwendung über Folgendes zugehört werden können:

Program.cs — C#

```csharp
class Program
{
    static async Task Main(string[] args)
    {
```

```
        using var app = HttpServer.CreateBuilder()
            .UseListeningPort("http://my-anwendung.example.test/")
            .Build();

        app.Router.MapGet("/", request =>
        {
            return new HttpResponse()
            {
                Status = 200,
                Content = new StringContent("Hallo, Welt!")
            };
        });

        await app.StartAsync();
    }
}
```

Changelog

Jeder Änderung an Sisk wird über das Changelog protokolliert. Sie können die Changelogs für alle Sisk-Versionen hier einsehen.

Häufig gestellte Fragen

Häufig gestellte Fragen über Sisk.

Ist Sisk Open-Source?

Vollkommen. Alle Quellcode, die von Sisk verwendet werden, sind veröffentlicht und werden häufig auf GitHub aktualisiert.

Werden Beiträge akzeptiert?

Solange sie mit der Sisk-Philosophie kompatibel sind, sind alle Beiträge sehr willkommen! Beiträge müssen nicht nur Code sein! Sie können auch bei der Dokumentation, Tests, Übersetzungen, Spenden und Beiträgen helfen, zum Beispiel.

Wird Sisk finanziell unterstützt?

Nein. Keine Organisation oder Projekt unterstützt Sisk derzeit finanziell.

Kann ich Sisk in der Produktion verwenden?

Absolut. Das Projekt ist seit über drei Jahren in Entwicklung und wurde in kommerziellen Anwendungen intensiv getestet, die seitdem in Produktion sind. Sisk wird in wichtigen kommerziellen Projekten als Hauptinfrastruktur verwendet.

Ein Leitfaden zur Bereitstellung in verschiedenen Systemen und Umgebungen wurde geschrieben und ist verfügbar.

Hat Sisk Authentifizierung, Überwachung und Datenbankdienste?

Nein. Sisk hat keine davon. Es ist ein Framework für die Entwicklung von HTTP-Webanwendungen, aber es ist immer noch ein minimales Framework, das nur das Nötigste für Ihre Anwendung liefert.

Sie können alle Dienste, die Sie benötigen, mit jeder beliebigen Bibliothek implementieren, die Sie bevorzugen. Sisk wurde so konzipiert, dass es agnostisch, flexibel und mit allem kompatibel ist.

Warum sollte ich Sisk anstelle von verwenden?

Ich weiß nicht. Sie sagen es mir.

Sisk wurde erstellt, um ein generisches Szenario für HTTP-Webanwendungen in .NET zu erfüllen. Etablierte Projekte wie ASP.NET lösen verschiedene Probleme, aber mit unterschiedlichen Vorurteilen. Im Gegensatz zu größeren Frameworks erfordert Sisk, dass der Benutzer weiß, was er tut und baut. Grundlegende Kenntnisse der Webentwicklung und des HTTP-Protokolls sind für die Arbeit mit Sisk unerlässlich.

Sisk ist eher mit Express von Node.js als mit ASP.NET Core vergleichbar. Es ist eine hohe Abstraktion, die es Ihnen ermöglicht, Anwendungen mit der HTTP-Logik zu erstellen, die Sie wollen.

Was muss ich lernen, um Sisk zu verwenden?

Sie benötigen die Grundlagen von:

- Webentwicklung (HTTP, Restful usw.)
- .NET

Das ist alles. Wenn Sie eine Vorstellung von diesen beiden Themen haben, können Sie sich einige Stunden widmen, um eine fortschrittliche Anwendung mit Sisk zu entwickeln.

Kann ich kommerzielle Anwendungen mit Sisk entwickeln?

Absolut.

Sisk wurde unter der MIT-Lizenz erstellt, was bedeutet, dass Sie Sisk in jedem kommerziellen Projekt, kommerziell oder nicht-kommerziell, ohne die Notwendigkeit einer proprietären Lizenz verwenden können.

Was wir bitten, ist, dass Sie irgendwo in Ihrer Anwendung einen Hinweis auf die verwendeten Open-Source-Projekte haben und dass Sisk dabei ist.

Routing

Der Router ist der erste Schritt beim Aufbau des Servers. Er ist verantwortlich für die Unterbringung von Route-Objekten, die Endpunkte sind, die URLs und ihre Methoden mit Aktionen verknüpfen, die vom Server ausgeführt werden. Jede Aktion ist verantwortlich für das Empfangen einer Anfrage und das Liefern einer Antwort an den Client.

Die Routen sind Paare von Pfad-Ausdrücken ("Pfadmuster") und der HTTP-Methode, auf die sie hören können. Wenn eine Anfrage an den Server gestellt wird, versucht er, eine Route zu finden, die der erhaltenen Anfrage entspricht, und ruft dann die Aktion dieser Route auf und liefert die resultierende Antwort an den Client.

Es gibt mehrere Möglichkeiten, Routen in Sisk zu definieren: Sie können statisch, dynamisch oder auto-gescannt, durch Attribute definiert oder direkt im Router-Objekt definiert werden.

```
Router mainRouter = new Router();

// ordnet die GET /-Route der folgenden Aktion zu
mainRouter.MapGet("/", request => {
    return new HttpResponse("Hallo, Welt!");
});
```

Um zu verstehen, was eine Route tun kann, müssen wir verstehen, was eine Anfrage tun kann. Ein HttpRequest enthält alles, was Sie benötigen. Sisk enthält auch einige zusätzliche Funktionen, die die Gesamtentwicklung beschleunigen.

Für jede vom Server empfangene Aktion wird ein Delegat vom Typ RouteAction aufgerufen. Dieser Delegat enthält ein Parameter, das ein HttpRequest mit allen notwendigen Informationen über die vom Server empfangene Anfrage enthält. Das resultierende Objekt aus diesem Delegaten muss ein HttpResponse oder ein Objekt sein, das durch implizite Antworttypen darauf abgebildet werden kann.

Übereinstimmende Routen

Wenn eine Anfrage an den HTTP-Server gestellt wird, sucht Sisk nach einer Route, die den Ausdruck des empfangenen Pfads erfüllt. Der Ausdruck wird immer zwischen der Route und dem Anfragepfad getestet, ohne die Abfragezeichenfolge zu berücksichtigen.

Dieser Test hat keine Priorität und ist exklusiv für eine einzelne Route. Wenn keine Route mit dieser Anfrage übereinstimmt, wird die Router.NotFoundErrorHandler-Antwort an den Client zurückgegeben. Wenn der Pfad-Ausdruck übereinstimmt, aber die HTTP-Methode nicht übereinstimmt, wird die Router.MethodNotAllowedErrorHandler-Antwort an den Client zurückgegeben.

Sisk überprüft die Möglichkeit von Routen-Kollisionen, um diese Probleme zu vermeiden. Wenn Routen definiert werden, sucht Sisk nach möglichen Routen, die mit der definierten Route kollidieren könnten. Dieser Test umfasst die Überprüfung des Pfads und der Methode, die die Route akzeptieren soll.

Erstellen von Routen mit Pfadmustern

Sie können Routen mit verschiedenen `SetRoute`-Methoden definieren.

```
// SetRoute-Methode
mainRouter.SetRoute(RouteMethod.Get, "/hey/<name>", (request) =>
{
    string name = request.RouteParameters["name"].GetString();
    return new HttpResponse($"Hallo, {name}");
});

// Map*-Methode
mainRouter.MapGet("/form", (request) =>
{
    var formData = request.GetFormData();
    return new HttpResponse(); // leerer 200-OK
});

// Route.*-Hilfsmethoden
mainRouter += Route.Get("/image.png", (request) =>
{
    var imageStream = File.OpenRead("image.png");

    return new HttpResponse()
    {
        // der StreamContent-Inner
        // stream wird nach dem Senden
        // der Antwort verworfen.
        Content = new StreamContent(imageStream)
    };
});

// mehrere Parameter
mainRouter.MapGet("/hey/<name>/surname/<surname>", (request) =>
{
```

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

Die RouteParameters-Eigenschaft von HttpResponse enthält alle Informationen über die Pfadvariablen der empfangenen Anfrage.

Jeder vom Server empfangene Pfad wird vor dem Pfad-Ausdruck-Test normalisiert, indem die folgenden Regeln angewendet werden:

- Alle leeren Segmente werden aus dem Pfad entfernt, z. B. `////foo//bar` wird zu `/foo/bar`.
- Der Pfad-Test ist **groß-/kleinschreibungsabhängig**, es sei denn, Router.MatchRoutesIgnoreCase ist auf `true` gesetzt.

Die Query und RouteParameters Eigenschaften von HttpRequest geben ein StringValueCollection-Objekt zurück, bei dem jedes indizierte Eigenschaft ein nicht-Null-StringValue zurückgibt, das als Option/Monad verwendet werden kann, um seinen Rohwert in ein verwaltetes Objekt umzuwandeln.

Das folgende Beispiel liest den Routen-Parameter "id" und erhält ein `Guid` daraus. Wenn der Parameter kein gültiges Guid ist, wird eine Ausnahme ausgelöst und ein 500-Fehler an den Client zurückgegeben, wenn der Server Router.CallbackErrorHandler nicht behandelt.

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

> [!HINWEIS] Pfade haben ihre abschließenden `/` ignoriert, sowohl in der Anfrage als auch in der Routen-Pfad, d. h., wenn Sie versuchen, auf eine Route zuzugreifen, die als `/index/page` definiert ist, können Sie auch auf `/index/page/` zugreifen.
>
> Sie können auch URLs zwingen, mit `/` zu enden, indem Sie die ForceTrailingSlash-Flag setzen.

Erstellen von Routen mit Klasseninstanzen

Sie können auch Routen dynamisch mit Reflexion und dem RouteAttribute definieren. Auf diese Weise werden die Instanzen einer Klasse, deren Methoden dieses Attribut implementieren, ihre Routen im Ziel-Router definiert.

Für eine Methode, die als Route definiert werden soll, muss sie mit einem RouteAttribute markiert werden, wie z. B. dem Attribut selbst oder einem RouteGetAttribute. Die Methode kann statisch, instanziell, öffentlich oder

privat sein. Wenn die Methode `SetObject(type)` oder `SetObject<TType>()` verwendet wird, werden Instanzmethoden ignoriert.

Controller/MyController.cs C#

```
public class MyController
{
    // wird mit GET / übereinstimmen
    [RouteGet]
    HttpResponse Index(HttpRequest request)
    {
        HttpResponse res = new HttpResponse();
        res.Content = new StringContent("Index!");
        return res;
    }

    // statische Methoden funktionieren auch
    [RouteGet("/hello")]
    static HttpResponse Hello(HttpRequest request)
    {
        HttpResponse res = new HttpResponse();
        res.Content = new StringContent("Hallo Welt!");
        return res;
    }
}
```

Die folgende Zeile wird beide Methoden `Index` und `Hello` von `MyController` als Routen definieren, da beide als Routen markiert sind und eine Instanz der Klasse bereitgestellt wurde, nicht deren Typ. Wenn deren Typ stattdessen bereitgestellt worden wäre, würden nur die statischen Methoden definiert.

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

Seit Sisk-Version 0.16 ist es möglich, AutoScan zu aktivieren, das nach benutzerdefinierten Klassen sucht, die `RouterModule` implementieren, und diese automatisch mit dem Router verknüpft. Dies wird nicht mit AOT-Kompilierung unterstützt.

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

Der obige Befehl sucht nach allen Typen, die `ApiController` implementieren, aber nicht den Typ selbst. Die beiden optionalen Parameter geben an, wie die Methoden nach diesen Typen suchen. Der erste Argument impliziert die Assembly, in der die Typen gesucht werden, und der zweite gibt an, wie die Typen definiert werden.

Regex-Routen

Anstelle der Verwendung der Standard-HTTP-Pfad-Übereinstimmungsmethode können Sie eine Route markieren, um sie mit Regex zu interpretieren.

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

Oder mit der RegexRoute-Klasse:

```
mainRouter.SetRoute(new RegexRoute(RouteMethod.Get, @"\/[a-z]+\/", request ⇒
{
    return new HttpResponse("hallo, Welt");
}));
```

Sie können auch Gruppen aus dem Regex-Muster in die HttpRequest.RouteParameters-Inhalte erfassen:

Controller/MyController.cs **C#**

```
public class MyController
{
    [RegexRoute(RouteMethod.Get, @"/uploads/(?<filename>.*\.(jpeg|jpg|png))")]
    static HttpResponse RegexRoute(HttpRequest request)
    {
        string filename = request.RouteParameters["filename"].GetString();
        return new HttpResponse().WithContent($"Zugriff auf Datei {filename}");
    }
}
```

Routen-Präfixe

Sie können alle Routen in einer Klasse oder einem Modul mit dem RoutePrefix-Attribut vordefinieren und den Präfix als Zeichenfolge setzen.

Siehe das folgende Beispiel mit der BREAD-Architektur (Browse, Read, Edit, Add und Delete):

Controller/Api/UsersController.cs C#

```
[RoutePrefix("/api/users")]
public class UsersController
{
    // GET /api/users/<id>
    [RouteGet]
    public async Task<HttpResponse> Browse()
    {
        ...
    }

    // GET /api/users
    [RouteGet("/<id>")]
    public async Task<HttpResponse> Read()
    {
        ...
    }

    // PATCH /api/users/<id>
    [RoutePatch("/<id>")]
    public async Task<HttpResponse> Edit()
    {
        ...
    }

    // POST /api/users
    [RoutePost]
    public async Task<HttpResponse> Add()
    {
        ...
    }

    // DELETE /api/users/<id>
    [RouteDelete("/<id>")]
    public async Task<HttpResponse> Delete()
    {
        ...
    }
}
```

Im obigen Beispiel wird der HttpResponse-Parameter weggelassen, um durch den globalen Kontext HttpContext.Current verwendet zu werden. Weitere Informationen finden Sie im folgenden Abschnitt.

Routen ohne Anfrageparameter

Routen können ohne den HttpRequest-Parameter definiert werden und es ist dennoch möglich, die Anfrage und ihre Komponenten im Anfragekontext zu erhalten. Betrachten wir eine Abstraktion `ControllerBase`, die als Grundlage für alle Controller einer API dient und die `Request`-Eigenschaft bereitstellt, um die HttpRequest zu erhalten, die derzeit im Kontext ist.

Controller/ControllerBase.cs — C#

```
public abstract class ControllerBase
{
    // erhält die Anfrage aus dem aktuellen Thread
    public HttpRequest Request { get => HttpContext.Current.Request; }

    // die folgende Zeile erhält die Datenbank aus der aktuellen HTTP-Sitzung,
    // oder erstellt eine neue, wenn sie nicht existiert
    public DbContext Database { get => HttpContext.Current.RequestBag.GetOrAdd<DbContext>
(); }
}
```

Und für alle seine Nachkommen, um die Routen-Syntax ohne den Anfrageparameter zu verwenden:

Controller/UsersController.cs — C#

```
[RoutePrefix("/api/users")]
public class UsersController : ControllerBase
{
    [RoutePost]
    public async Task<HttpResponse> Create()
    {
        // liest die JSON-Daten aus der aktuellen Anfrage
        UserCreationDto? user = JsonSerializer.DeserializeAsync<UserCreationDto>
(Request.Body);
         ...
        Database.Users.Add(user);

        return new HttpResponse(201);
    }
}
```

Weitere Details zum aktuellen Kontext und zur Abhängigkeitsinjektion finden Sie im Abhängigkeitsinjektion-Tutorial.

Routen für jede Methode

Sie können eine Route definieren, die nur nach ihrem Pfad übereinstimmt und die HTTP-Methode ignoriert. Dies kann nützlich sein, um die Methode innerhalb der Routen-Aktion zu validieren.

```
1    // wird mit / auf jede HTTP-Methode übereinstimmen
2    mainRouter.SetRoute(RouteMethod.Any, "/", callbackFunction);
```

Routen für jeden Pfad

Routen für jeden Pfad testen jeden Pfad, der vom HTTP-Server empfangen wird, unter Vorbehalt der Route-Methode, die getestet wird. Wenn die Route-Methode RouteMethod.Any ist und die Route Route.AnyPath in ihrem Pfad-Ausdruck verwendet, wird diese Route auf alle Anfragen vom HTTP-Server hören, und keine anderen Routen können definiert werden.

```
1    // die folgende Route wird mit allen POST-Anfragen übereinstimmen
2    mainRouter.SetRoute(RouteMethod.Post, Route.AnyPath, callbackFunction);
```

Groß-/Kleinschreibung ignorierende Routen-Übereinstimmung

Standardmäßig ist die Interpretation von Routen mit Anfragen groß-/kleinschreibungsabhängig. Um dies zu ignorieren, aktivieren Sie diese Option:

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

Dies aktiviert auch die Option `RegexOptions.IgnoreCase` für Routen, bei denen es sich um Regex-Übereinstimmung handelt.

Nicht gefunden (404)-Rückruf-Handler

Sie können einen benutzerdefinierten Rückruf für den Fall erstellen, dass eine Anfrage keine bekannte Route entspricht.

```
mainRouter.NotFoundErrorHandler = () =>
{
    return new HttpResponse(404)
    {
        // Seit v0.14
        Content = new HtmlContent("<h1>Nicht gefunden</h1>")
        // ältere Versionen
        Content = new StringContent("<h1>Nicht gefunden</h1>", Encoding.UTF8, "text/html")
    };
};
```

Methode nicht zulässig (405)-Rückruf-Handler

Sie können auch einen benutzerdefinierten Rückruf für den Fall erstellen, dass eine Anfrage ihren Pfad entspricht, aber nicht die Methode.

```
mainRouter.MethodNotAllowedErrorHandler = (context) =>
{
    return new HttpResponse(405)
    {
        Content = new StringContent($"Methode nicht zulässig für diese Route.")
    };
};
```

Interne Fehlerbehandlung

Routen-Rückrufe können während der Serverausführung Fehler auslösen. Wenn diese nicht richtig behandelt werden, kann die Gesamtfunktion des HTTP-Servers unterbrochen werden. Der Router hat einen Rückruf für den Fall, dass ein Routen-Rückruf fehlschlägt und die Serviceunterbrechung verhindert.

Diese Methode ist nur erreichbar, wenn ThrowExceptions auf `false` gesetzt ist.

```
mainRouter.CallbackErrorHandler = (ex, context) =>
{
    return new HttpResponse(500)
    {
        Content = new StringContent($"Fehler: {ex.Message}")
    };
};
```

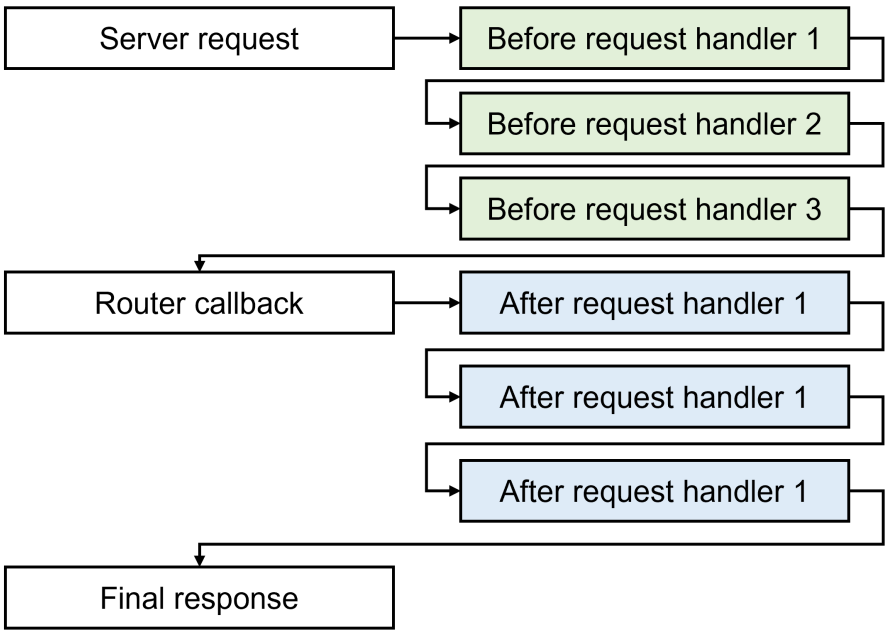
Anfragebehandlung

Anfragebehandler, auch bekannt als "Middleware", sind Funktionen, die vor oder nach der Ausführung einer Anfrage auf dem Router ausgeführt werden. Sie können pro Route oder pro Router definiert werden.

Es gibt zwei Arten von Anfragebehandlern:

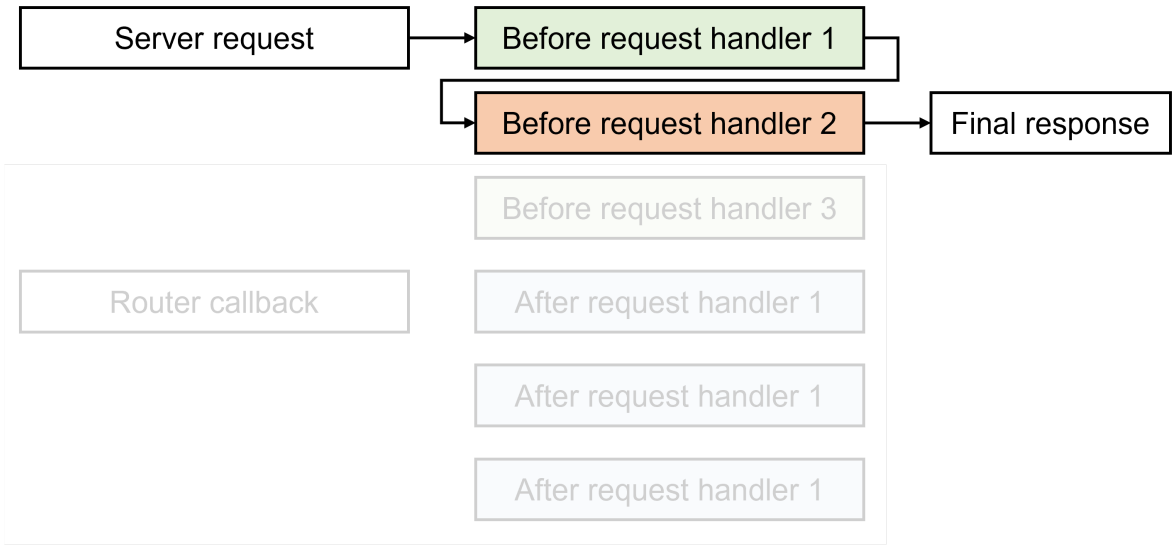
- **BeforeResponse**: definiert, dass der Anfragebehandler vor dem Aufruf der Router-Aktion ausgeführt wird.
- **AfterResponse**: definiert, dass der Anfragebehandler nach dem Aufruf der Router-Aktion ausgeführt wird. Das Senden einer HTTP-Antwort in diesem Kontext überschreibt die Router-Aktionsantwort.

Beide Anfragebehandler können die tatsächliche Router-Rückruf-Funktion überschreiben. Darüber hinaus können Anfragebehandler nützlich sein, um eine Anfrage zu validieren, wie z.B. Authentifizierung, Inhalt oder andere Informationen, wie z.B. das Speichern von Informationen, Protokollen oder anderen Schritten, die vor oder nach einer Antwort ausgeführt werden können.



Auf diese Weise kann ein Anfragebehandler die gesamte Ausführung unterbrechen und eine Antwort zurückgeben, bevor der Zyklus beendet ist, und alles andere im Prozess verwerfen.

Beispiel: Nehmen wir an, dass ein Benutzer-Authentifizierungsanfragebehandler den Benutzer nicht authentifiziert. Er wird den Anfragelebenszyklus nicht fortsetzen und hängen bleiben. Wenn dies im Anfragebehandler an Position zwei passiert, werden die dritte und folgenden nicht ausgewertet.



Erstellen eines Anfragebehandlers

Um einen Anfragebehandler zu erstellen, können wir eine Klasse erstellen, die die IRequestHandler-Schnittstelle erbt, in diesem Format:

Middleware/AuthenticateUserRequestHandler.cs C#

```
public class AuthenticateUserRequestHandler : IRequestHandler
{
    public RequestHandlerExecutionMode ExecutionMode { get; init; } =
RequestHandlerExecutionMode.BeforeResponse;

    public HttpResponse? Execute(HttpRequest request, HttpContext context)
    {
        if (request.Headers.Authorization == null)
        {
            // Rückgabe von null bedeutet, dass der Anfragezyklus fortgesetzt werden kann
            return null;
        }
        else
        {
```

```
            // Rückgabe eines HttpResponse-Objekts bedeutet, dass diese Antwort die
benachbarten Antworten überschreibt.
            return new HttpResponse(System.Net.HttpStatusCode.Unauthorized);
        }
    }
}
```

Im obigen Beispiel haben wir angegeben, dass, wenn die `Authorization` -Header in der Anfrage vorhanden ist, sie fortgesetzt werden sollte und der nächste Anfragebehandler oder die Router-Rückruf-Funktion aufgerufen werden sollte, je nachdem, was zuerst kommt. Wenn ein Anfragebehandler nach der Antwort durch ihre Eigenschaft ExecutionMode ausgeführt wird und einen nicht-leeren Wert zurückgibt, wird er die Router-Antwort überschreiben.

Wenn ein Anfragebehandler `null` zurückgibt, bedeutet dies, dass die Anfrage fortgesetzt werden muss und das nächste Objekt aufgerufen werden muss oder der Zyklus mit der Router-Antwort enden muss.

Zuordnen eines Anfragebehandlers zu einer einzelnen Route

Sie können einen oder mehrere Anfragebehandler für eine Route definieren.

Router.cs C#

```
mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage, "", new IRequestHandler[]
{
    new AuthenticateUserRequestHandler(),     // vorheriger Anfragebehandler
    new ValidateJsonContentRequestHandler(),  // vorheriger Anfragebehandler
    //                                        -- Methode IndexPage wird hier ausgeführt
    new WriteToLogRequestHandler()            // nachfolgender Anfragebehandler
});
```

Oder durch Erstellen eines Route-Objekts:

Router.cs C#

```
Route indexRoute = new Route(RouteMethod.Get, "/", "", IndexPage, null);
indexRoute.RequestHandlers = new IRequestHandler[]
{
    new AuthenticateUserRequestHandler()
```

```
};
mainRouter.SetRoute(indexRoute);
```

Zuordnen eines Anfragebehandlers zu einem Router

Sie können einen globalen Anfragebehandler definieren, der auf allen Routen auf einem Router ausgeführt wird.

Router.cs C#

```
mainRouter.GlobalRequestHandlers = new IRequestHandler[]
{
    new AuthenticateUserRequestHandler()
};
```

Zuordnen eines Anfragebehandlers zu einem Attribut

Sie können einen Anfragebehandler auf einem Methoden-Attribut zusammen mit einem Route-Attribut definieren.

Controller/MyController.cs C#

```
public class MyController
{
    [RouteGet("/")]
    [RequestHandler<AuthenticateUserRequestHandler>]
    static HttpResponse Index(HttpRequest request)
    {
        return new HttpResponse() {
            Content = new StringContent("Hallo Welt!")
        };
    }
}
```

Beachten Sie, dass es notwendig ist, den gewünschten Anfragebehandlertyp und nicht eine Objektinstanz zu übergeben. Auf diese Weise wird der Anfragebehandler vom Router-Parser instanziert. Sie können Argumente im Klassenkonstruktor mit der ConstructorArguments-Eigenschaft übergeben.

Beispiel:

Controller/MyController.cs C#

```
[RequestHandler<AuthenticateUserRequestHandler>("arg1", 123, ...)]
public HttpResponse Index(HttpRequest request)
{
    return res = new HttpResponse() {
        Content = new StringContent("Hallo Welt!")
    };
}
```

Sie können auch Ihr eigenes Attribut erstellen, das RequestHandler implementiert:

Middleware/Attributes/AuthenticateAttribute.cs C#

```
public class AuthenticateAttribute : RequestHandlerAttribute
{
    public AuthenticateAttribute() : base(typeof(AuthenticateUserRequestHandler),
ConstructorArguments = new object?[] { "arg1", 123, ... })
    {
        ;
    }
}
```

Und es wie folgt verwenden:

Controller/MyController.cs C#

```
[Authenticate]
static HttpResponse Index(HttpRequest request)
{
    return res = new HttpResponse() {
        Content = new StringContent("Hallo Welt!")
    };
}
```

Umgehen eines globalen Anfragebehandlers

Nachdem Sie einen globalen Anfragebehandler auf einer Route definiert haben, können Sie diesen Anfragebehandler auf bestimmten Routen ignorieren.

Router.cs C#

```
var myRequestHandler = new AuthenticateUserRequestHandler();
mainRouter.GlobalRequestHandlers = new IRequestHandler[]
{
    myRequestHandler
};

mainRouter.SetRoute(new Route(RouteMethod.Get, "/", "Meine Route", IndexPage, null)
{
    BypassGlobalRequestHandlers = new IRequestHandler[]
    {
        myRequestHandler,                    // ok: dieselbe Instanz wie in den
globalen Anfragebehandlern
        new AuthenticateUserRequestHandler() // falsch: wird den globalen Anfragebehandler
nicht überspringen
    }
});
```

> [!HINWEIS] Wenn Sie einen Anfragebehandler umgehen, müssen Sie dieselbe Referenz verwenden, die Sie zuvor instanziert haben, um den globalen Anfragebehandler zu überspringen. Das Erstellen einer anderen Anfragebehandler-Instanz wird den globalen Anfragebehandler nicht überspringen, da die Referenz geändert wird. Beachten Sie, dass Sie dieselbe Anfragebehandler-Referenz verwenden müssen, die in beiden GlobalRequestHandlers und BypassGlobalRequestHandlers verwendet wird.

Requests

Requests sind Strukturen, die eine HTTP-Anforderungsnachricht darstellen. Das HttpRequest-Objekt enthält nützliche Funktionen zur Handhabung von HTTP-Nachrichten in Ihrer gesamten Anwendung.

Eine HTTP-Anforderung besteht aus Methode, Pfad, Version, Headern und Körper.

In diesem Dokument zeigen wir Ihnen, wie Sie jedes dieser Elemente erhalten.

Erhalten der Anforderungsmethode

Um die Methode der empfangenen Anfrage zu erhalten, können Sie die Eigenschaft Method verwenden:

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

Diese Eigenschaft gibt die Methode der Anfrage als ein HttpMethod-Objekt zurück.

> **NOTE**
>
> Im Gegensatz zu Routemethoden dient diese Eigenschaft nicht dem RouteMethod.Any-Element. Stattdessen gibt sie die echte Anforderungsmethode zurück.

Erhalten der URL-Komponenten der Anfrage

Sie können verschiedene Komponenten einer URL über bestimmte Eigenschaften einer Anfrage abrufen. Für dieses Beispiel betrachten wir die URL:

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

Component name	Description	Component value
Path	Gibt den Pfad der Anfrage zurück.	`/user/login`
FullPath	Gibt den Pfad der Anfrage und die Abfragezeichenfolge zurück.	`/user/login?email=foo@bar.com`
FullUrl	Gibt die gesamte URL-Anforderungszeichenfolge zurück.	`http://localhost:5000/user/login?email=foo@bar.com`
Host	Gibt den Host der Anfrage zurück.	`localhost`
Authority	Gibt den Host und Port der Anfrage zurück.	`localhost:5000`
QueryString	Gibt die Abfrage der Anfrage zurück.	`?email=foo@bar.com`
Query	Gibt die Abfrage der Anfrage in einer benannten Wertsammlung zurück.	`{StringValueCollection object}`
IsSecure	Bestimmt, ob die Anfrage SSL verwendet (true) oder nicht (false).	`false`

Sie können auch die Eigenschaft HttpRequest.Uri verwenden, die alles oben Genannte in einem Objekt enthält.

Erhalten des Anfragekörpers

Einige Anfragen enthalten einen Körper wie Formulare, Dateien oder API-Transaktionen. Sie können den Körper einer Anfrage über die Eigenschaft erhalten:

```
1   // erhält den Anfragekörper als Zeichenkette, unter Verwendung der Anfragekodierung
2   als Encoder
3   string body = request.Body;
4
5   // oder erhält ihn in einem Byte-Array
6   byte[] bodyBytes = request.RawBody;
7
8
```

```
    // oder sonst, können Sie ihn streamen.
    Stream requestStream = request.GetRequestStream();
```

Es ist auch möglich zu bestimmen, ob ein Körper in der Anfrage vorhanden ist und ob er geladen ist, mit den Eigenschaften HasContents, die bestimmt, ob die Anfrage Inhalte hat, und IsContentAvailable, die anzeigt, dass der HTTP-Server den Inhalt vollständig vom entfernten Punkt empfangen hat.

Es ist nicht möglich, den Anfrageinhalt mehr als einmal über `GetRequestStream` zu lesen. Wenn Sie mit dieser Methode lesen, sind die Werte in `RawBody` und `Body` ebenfalls nicht verfügbar. Es ist nicht erforderlich, den Anfrage-Stream im Kontext der Anfrage zu entsorgen, da er am Ende der HTTP-Sitzung, in der er erstellt wurde, entsorgt wird. Außerdem können Sie die Eigenschaft HttpRequest.RequestEncoding verwenden, um die beste Kodierung zum manuellen Dekodieren der Anfrage zu erhalten.

Der Server hat Grenzwerte für das Lesen des Anfrageinhalts, die sowohl für HttpRequest.Body als auch für HttpRequest.RawBody gelten. Diese Eigenschaften kopieren den gesamten Eingabestream in einen lokalen Puffer der gleichen Größe wie HttpRequest.ContentLength.

Eine Antwort mit Status 413 Content Too Large wird an den Client zurückgegeben, wenn der gesendete Inhalt größer ist als HttpServerConfiguration.MaximumContentLength, der in der Benutzereinstellung definiert ist. Zusätzlich, wenn kein konfigurierter Grenzwert vorhanden ist oder wenn er zu groß ist, wirft der Server eine OutOfMemoryException, wenn der vom Client gesendete Inhalt Int32.MaxValue (2 GB) überschreitet und wenn der Inhalt versucht wird, über eine der oben genannten Eigenschaften zuzugreifen. Sie können den Inhalt weiterhin über Streaming behandeln.

> **NOTE**
>
> Obwohl Sisk es zulässt, ist es immer eine gute Idee, die HTTP-Semantik zu befolgen, um Ihre Anwendung zu erstellen und keine Inhalte in Methoden zu erhalten oder bereitzustellen, die dies nicht zulassen. Lesen Sie über RFC 9110 "HTTP Semantics".

Erhalten des Anfragekontexts

Der HTTP Context ist ein exklusives Sisk-Objekt, das Informationen zum HTTP-Server, zur Route, zum Router und zum Anfragehandler speichert. Sie können es verwenden, um sich in einer Umgebung zu organisieren, in der diese Objekte schwer zu organisieren sind.

Das RequestBag-Objekt enthält gespeicherte Informationen, die von einem Anfragehandler an einen anderen Punkt übergeben werden und am endgültigen Ziel verwendet werden können. Dieses Objekt kann auch von

Anfragehandlern verwendet werden, die nach dem Routencallback ausgeführt werden.

> **TIP**
>
> Diese Eigenschaft ist auch über die Eigenschaft HttpRequest.Bag zugänglich.

Middleware/AuthenticateUserRequestHandler.cs **C#**

```
public class AuthenticateUserRequestHandler : IRequestHandler
{
    public string Identifier { get; init; } = Guid.NewGuid().ToString();
    public RequestHandlerExecutionMode ExecutionMode { get; init; } =
RequestHandlerExecutionMode.BeforeResponse;

    public HttpResponse? Execute(HttpRequest request, HttpContext context)
    {
        if (request.Headers.Authorization == null)
        {
            context.RequestBag.Add("AuthenticatedUser", new User("Bob"));
            return null;
        }
        else
        {
            return new HttpResponse(System.Net.HttpStatusCode.Unauthorized);
        }
    }
}
```

Der obenstehende Anfragehandler definiert `AuthenticatedUser` im Anfragebag und kann später im endgültigen Callback verwendet werden:

Controller/MyController.cs **C#**

```
public class MyController
{
    [RouteGet("/")]
    [RequestHandler<AuthenticateUserRequestHandler>]
    static HttpResponse Index(HttpRequest request)
    {
        User authUser = request.Context.RequestBag["AuthenticatedUser"];

        return new HttpResponse() {
            Content = new StringContent($"Hello, {authUser.Name}!")
        };
```

```
    }
}
```

Sie können auch die Hilfsmethoden `Bag.Set()` und `Bag.Get()` verwenden, um Objekte nach ihrem Typ-Singleton zu erhalten oder zu setzen.

Middleware/Authenticate.cs — C#

```
public class Authenticate : RequestHandler
{
    public override HttpResponse? Execute(HttpRequest request, HttpContext context)
    {
        request.Bag.Set<User>(authUser);
    }
}
```

Controller/MyController.cs — C#

```
[RouteGet("/")]
[RequestHandler<Authenticate>]
public static HttpResponse GetUser(HttpRequest request)
{
    var user = request.Bag.Get<User>();
    ...
}
```

Erhalten von Formulardaten

Sie können die Werte von Formulardaten in einer NameValueCollection mit dem folgenden Beispiel erhalten:

Controller/Auth.cs — C#

```
[RoutePost("/auth")]
public HttpResponse Index(HttpRequest request)
{
    var form = request.GetFormContent();

    string? username = form["username"];
```

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

Erhalten von multipart Formulardaten

Die HTTP-Anforderung von Sisk ermöglicht es, hochgeladene multipart Inhalte zu erhalten, wie Dateien, Formularfelder oder beliebigen Binärinhalt.

Controller/Auth.cs — C#

```
1   [RoutePost("/upload-contents")]
2   public HttpResponse Index(HttpRequest request)
3   {
4       // die folgende Methode liest die gesamte Eingabe der Anfrage in ein
5       // Array von MultipartObjects
6       var multipartFormDataObjects = request.GetMultipartFormContent();
7
8       foreach (MultipartObject uploadedObject in multipartFormDataObjects)
9       {
10          // Der Name der Datei, die von Multipart Formulardaten bereitgestellt wird.
11          // Null wird zurückgegeben, wenn das Objekt keine Datei ist.
12          Console.WriteLine("File name       : " + uploadedObject.Filename);
13
14          // Der Feldname des Multipart Formulardatenobjekts.
15          Console.WriteLine("Field name      : " + uploadedObject.Name);
16
17          // Die Länge des Multipart Formulardateninhalts.
18          Console.WriteLine("Content length  : " + uploadedObject.ContentLength);
19
20          // Bestimmt das Bildformat basierend auf der Dateikopfzeile für jeden
21          // bekannten Inhaltstyp. Wenn der Inhalt kein erkanntes übliches Dateiformat ist,
22          // gibt diese Methode unten MultipartObjectCommonFormat.Unknown zurück
23          Console.WriteLine("Common format   : " + uploadedObject.GetCommonFileFormat());
24      }
25  }
```

Sie können mehr über Sisk Multipart form objects und seine Methoden, Eigenschaften und Funktionalitäten lesen.

Erkennen von Client-Abschaltungen

Seit Version v1.15 von Sisk stellt das Framework ein CancellationToken bereit, das ausgelöst wird, wenn die Verbindung zwischen Client und Server vorzeitig geschlossen wird, bevor die Antwort empfangen wird. Dieses Token kann nützlich sein, um zu erkennen, wann der Client die Antwort nicht mehr möchte und lang laufende Operationen abzubrechen.

```
router.MapGet("/connect", async (HttpRequest req) ⇒
{
    // erhält das Abbruchtoken von der Anfrage
    var dc = req.DisconnectToken;

    await LongOperationAsync(dc);

    return new HttpResponse();
});
```

Dieses Token ist nicht mit allen HTTP-Engines kompatibel, und jede erfordert eine Implementierung.

Unterstützung von Server-sent events

Sisk unterstützt Server-sent events, die das Senden von Daten als Stream ermöglichen und die Verbindung zwischen Server und Client aufrechterhalten.

Das Aufrufen der Methode HttpRequest.GetEventSource setzt die HttpRequest in ihren Listener-Zustand. Von dort aus erwartet der Kontext dieser HTTP-Anforderung keine HttpResponse, da er die vom Server gesendeten Pakete überlappt.

Nach dem Senden aller Pakete muss der Callback die Methode Close zurückgeben, die die endgültige Antwort an den Server sendet und anzeigt, dass das Streaming beendet ist.

Es ist nicht möglich vorherzusagen, welche Gesamtlänge aller Pakete gesendet wird, daher ist es nicht möglich, das Ende der Verbindung mit dem Header `Content-Length` zu bestimmen.

Durch die meisten Browser-Standardeinstellungen unterstützen serverseitige Ereignisse keine HTTP-Header oder Methoden außer der GET-Methode. Daher sollten Sie vorsichtig sein, wenn Sie Anfragehandler mit event-source-Anfragen verwenden, die spezifische Header in der Anfrage erfordern, da sie wahrscheinlich nicht vorhanden sind.

Außerdem starten die meisten Browser Streams neu, wenn die Methode EventSource.close auf der Clientseite nicht aufgerufen wird, nachdem alle Pakete empfangen wurden, was zu unendlicher zusätzlicher Verarbeitung auf der Serverseite führt. Um dieses Problem zu vermeiden, ist es üblich, ein finales Paket zu senden, das anzeigt, dass die Ereignisquelle alle Pakete gesendet hat.

Das folgende Beispiel zeigt, wie der Browser mit dem Server kommunizieren kann, der Server-sent events unterstützt.

sse-example.html **HTML**

```html
<html>
    <body>
        <b>Fruits:</b>
        <ul></ul>
    </body>
    <script>
        const evtSource = new EventSource('http://localhost:5555/event-source');
        const eventList = document.querySelector('ul');

        evtSource.onmessage = (e) => {
            const newElement = document.createElement("li");

            newElement.textContent = `message: ${e.data}`;
            eventList.appendChild(newElement);

            if (e.data == "Tomato") {
                evtSource.close();
            }
        }
    </script>
</html>
```

Und senden Sie die Nachrichten schrittweise an den Client:

Controller/MyController.cs **C#**

```
public class MyController
{
    [RouteGet("/event-source")]
    public async Task<HttpResponse> ServerEventsResponse(HttpRequest request)
    {
        var sse = await request.GetEventSourceAsync ();

        string[] fruits = new[] { "Apple", "Banana", "Watermelon", "Tomato" };

        foreach (string fruit in fruits)
        {
            await serverEvents.SendAsync(fruit);
            await Task.Delay(1500);
        }

        return serverEvents.Close();
    }
}
```

Wenn Sie diesen Code ausführen, erwarten wir ein Ergebnis ähnlich dem folgenden:

Auflösen von proxied IPs und Hosts

Sisk kann mit Proxies verwendet werden, und daher können IP-Adressen durch den Proxy-Endpunkt in der Transaktion von einem Client zum Proxy ersetzt werden.

Sie können Ihre eigenen Auflösungen in Sisk mit forwarding resolvers definieren.

Header-Kodierung

Header-Kodierung kann ein Problem für einige Implementierungen sein. Unter Windows werden UTF-8-Header nicht unterstützt, daher wird ASCII verwendet. Sisk verfügt über einen eingebauten Kodierungsumwandler, der nützlich sein kann, um falsch kodierte Header zu dekodieren.

Diese Operation ist kostenintensiv und standardmäßig deaktiviert, kann aber unter dem Flag NormalizeHeadersEncodings aktiviert werden.

Antworten

Antworten stellen Objekte dar, die HTTP-Antworten auf HTTP-Anfragen sind. Sie werden vom Server an den Client gesendet, um die Anfrage nach einer Ressource, Seite, Dokument, Datei oder einem anderen Objekt anzuzeigen.

Eine HTTP-Antwort besteht aus Status, Headern und Inhalt.

In diesem Dokument erfahren Sie, wie Sie HTTP-Antworten mit Sisk entwerfen.

Festlegen eines HTTP-Status

Die Liste der HTTP-Statuscodes ist seit HTTP/1.0 gleich und Sisk unterstützt alle davon.

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

Oder mit Fluent-Syntax:

```
1    new HttpResponse()
2     .WithStatus(200) // oder
3     .WithStatus(HttpStatusCode.Ok) // oder
4     .WithStatus(HttpStatusInformation.Ok);
```

Sie können die vollständige Liste der verfügbaren HttpStatusCode hier sehen. Sie können auch Ihren eigenen Statuscode mithilfe der HttpStatusInformation-Struktur bereitstellen.

Body und Content-Type

Sisk unterstützt .NET-Inhalte Objekte, um den Body in Antworten zu senden. Sie können die StringContent-Klasse verwenden, um beispielsweise eine JSON-Antwort zu senden:

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

Der Server versucht immer, die `Content-Length` aus dem zu definieren, was Sie im Inhalt definiert haben, wenn Sie es nicht explizit in einem Header definiert haben. Wenn der Server den Content-Length-Header nicht implizit aus dem Antwortinhalt abrufen kann, wird die Antwort mit Chunked-Encoding gesendet.

Sie können die Antwort auch streamen, indem Sie einen StreamContent senden oder die Methode GetResponseStream verwenden.

Antwort-Header

Sie können Header hinzufügen, bearbeiten oder entfernen, die Sie in der Antwort senden. Das folgende Beispiel zeigt, wie Sie eine Umleitungsantwort an den Client senden.

```
HttpResponse res = new HttpResponse();
res.Status = HttpStatusCode.Moved;
res.Headers.Add(HttpKnownHeaderNames.Location, "/login");
```

Oder mit Fluent-Syntax:

```
new HttpResponse(301)
 .WithHeader("Location", "/login");
```

Wenn Sie die Add-Methode von HttpHeaderCollection verwenden, fügen Sie einen Header zur Anfrage hinzu, ohne die bereits gesendeten zu ändern. Die Set-Methode ersetzt die Header mit demselben Namen durch den angegebenen Wert. Der Index von HttpHeaderCollection ruft intern die Set-Methode auf, um die Header zu ersetzen.

Senden von Cookies

Sisk verfügt über Methoden, die die Definition von Cookies auf dem Client erleichtern. Cookies, die mit dieser Methode gesetzt werden, sind bereits URL-kodiert und entsprechen dem RFC-6265-Standard.

```
HttpResponse res = new HttpResponse();
res.SetCookie("cookie-name", "cookie-value");
```

Oder mit Fluent-Syntax:

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

Es gibt andere vollständigere Versionen derselben Methode.

Chunked-Antworten

Sie können die Übertragungskodierung auf chunked setzen, um große Antworten zu senden.

```
HttpResponse res = new HttpResponse();
res.SendChunked = true;
```

Bei Verwendung von Chunked- Encoding wird der Content-Length-Header automatisch weggelassen.

Antwortstrom

Antwortströme sind eine verwaltete Möglichkeit, die es Ihnen ermöglicht, Antworten auf eine segmentierte Weise zu senden. Es handelt sich um eine Ebene tiefer als die Verwendung von HttpResponse-Objekten, da sie erfordern, dass Sie die Header und Inhalte manuell senden und dann die Verbindung schließen.

Dieses Beispiel öffnet einen schreibgeschützten Strom für die Datei, kopiert den Strom in den Antwortausgabestrom und lädt die gesamte Datei nicht in den Speicher. Dies kann nützlich sein, um mittelgroße oder große Dateien zu bedienen.

```
// erhält den Antwortausgabestrom
using var fileStream = File.OpenRead("my-big-file.zip");
var responseStream = request.GetResponseStream();

// setzt die Antwortkodierung auf chunked-encoding
// auch sollten Sie den Content-Length-Header nicht senden, wenn Sie chunked-
encoding verwenden
responseStream.SendChunked = true;
responseStream.SetStatus(200);
responseStream.SetHeader(HttpKnownHeaderNames.ContentType, contentType);

// kopiert den Dateistream in den Antwortausgabestrom
fileStream.CopyTo(responseStream.ResponseStream);

// schließt den Strom
return responseStream.Close();
```

GZip-, Deflate- und Brotli-Komprimierung

Sie können Antworten mit komprimierten Inhalten in Sisk senden, indem Sie HTTP-Inhalte komprimieren. Zuerst kapseln Sie Ihr HttpContent-Objekt in eines der untenstehenden Kompressoren, um die komprimierte Antwort an den Client zu senden.

```
router.MapGet("/hello.html", request => {
 string myHtml = "...";

 return new HttpResponse () {
 Content = new GZipContent(new HtmlContent(myHtml)),
 // oder Content = new BrotliContent(new HtmlContent(myHtml)),
 // oder Content = new DeflateContent(new HtmlContent(myHtml)),
 };
});
```

Sie können diese komprimierten Inhalte auch mit Strömen verwenden.

```
router.MapGet("/archive.zip", request => {

 // verwenden Sie hier nicht "using". Der HttpServer verwirft Ihren Inhalt
 // nachdem er die Antwort gesendet hat.
```

```
    var archive = File.OpenRead("/path/to/big-file.zip");

    return new HttpResponse () {
    Content = new GZipContent(archive)
    }
});
```

Die Content-Encoding-Header werden automatisch gesetzt, wenn diese Inhalte verwendet werden.

Automatische Komprimierung

Es ist möglich, HTTP-Antworten mit der EnableAutomaticResponseCompression-Eigenschaft automatisch zu komprimieren. Diese Eigenschaft kapselt den Antwortinhalt vom Router automatisch in einen komprimierbaren Inhalt, der von der Anfrage akzeptiert wird, vorausgesetzt, die Antwort wird nicht von einem CompressedContent geerbt.

Nur ein komprimierbarer Inhalt wird für eine Anfrage ausgewählt, die gemäß der Accept-Encoding-Header, die der Reihe nach folgt:

- BrotliContent (br)
- GZipContent (gzip)
- DeflateContent (deflate)

Wenn die Anfrage angibt, dass sie eine dieser Komprimierungsmethoden akzeptiert, wird die Antwort automatisch komprimiert.

Implizite Antworttypen

Sie können andere Rückgabetypen als HttpResponse verwenden, aber es ist notwendig, den Router zu konfigurieren, wie er jeden Objekttyp behandelt.

Das Konzept besteht darin, immer einen Referenztyp zurückzugeben und ihn in ein gültiges HttpResponse-Objekt umzuwandeln. Routen, die HttpResponse zurückgeben, unterliegen keiner Umwandlung.

Wertetypen (Strukturen) können nicht als Rückgabetyp verwendet werden, da sie nicht mit dem RouterCallback kompatibel sind. Daher müssen sie in ein ValueResult gekapselt werden, um in Handhabern verwendet werden zu können.

Betrachten Sie das folgende Beispiel eines Router-Moduls, das nicht HttpResponse im Rückgabetyp verwendet:

```
[RoutePrefix("/users")]
public class UsersController : RouterModule
{
 public List<User> Users = new List<User>();

 [RouteGet]
 public IEnumerable<User> Index(HttpRequest request)
 {
 return Users.ToArray();
 }

 [RouteGet("<id>")]
 public User View(HttpRequest request)
 {
 int id = request.RouteParameters["id"].GetInteger();
 User dUser = Users.First(u => u.Id == id);

 return dUser;
 }

 [RoutePost]
 public ValueResult<bool> Create(HttpRequest request)
 {
 User fromBody = JsonSerializer.Deserialize<User>(request.Body)!;
 Users.Add(fromBody);

 return true;
 }
}
```

Damit muss nun im Router definiert werden, wie er mit jedem Objekttyp umgeht. Objekte sind immer das erste Argument des Handlers und der Ausgabetyp muss ein gültiges HttpResponse sein. Außerdem sollten die Ausgabeobjekte einer Route niemals null sein.

Für ValueResult-Typen ist es nicht notwendig, anzugeben, dass das Eingabeobjekt ein ValueResult ist und nur T, da ValueResult ein Objekt ist, das von seiner ursprünglichen Komponente reflektiert wird.

Die Zuordnung von Typen vergleicht nicht, was registriert wurde, mit dem Typ des Objekts, das vom Router-Callback zurückgegeben wird. Stattdessen prüft es, ob der Typ des Router-Ergebnisses dem registrierten Typ

zuweisbar ist.

Das Registrieren eines Handlers vom Typ Object wird auf alle zuvor nicht validierten Typen zurückgeführt. Die Einfügereihenfolge der Wert-Handler spielt auch eine Rolle, daher sollten Sie zuerst bestimmte Wert-Handler registrieren, um die Reihenfolge sicherzustellen.

```
Router r = new Router();
r.SetObject(new UsersController());

r.RegisterValueHandler<ApiResult>(apiResult ⇒
{
 return new HttpResponse() {
 Status = apiResult.Success ? HttpStatusCode.OK : HttpStatusCode.BadRequest,
 Content = apiResult.GetHttpContent(),
 Headers = apiResult.GetHeaders()
 };
});
r.RegisterValueHandler<bool>(bvalue ⇒
{
 return new HttpResponse() {
 Status = bvalue ? HttpStatusCode.OK : HttpStatusCode.BadRequest
 };
});
r.RegisterValueHandler<IEnumerable<object>>(enumerableValue ⇒
{
 return new HttpResponse(string.Join("\n", enumerableValue));
});

// das Registrieren eines Wert-Handlers vom Typ Object muss das letzte sein
// Wert-Handler, der als Fallback verwendet wird
r.RegisterValueHandler<object>(fallback ⇒
{
 return new HttpResponse() {
 Status = HttpStatusCode.OK,
 Content = JsonContent.Create(fallback)
 };
});
```

Hinweis zu enumerable Objekten und Arrays

Implizite Antwortobjekte, die IEnumerable implementieren, werden durch die `ToArray()`-Methode in den Speicher gelesen, bevor sie durch einen definierten Wert-Handler umgewandelt werden. Damit dies geschieht, wird das `IEnumerable`-Objekt in ein Array von Objekten umgewandelt, und der Antwortkonverter empfängt immer ein `Object[]` anstelle des ursprünglichen Typs.

Betrachten Sie das folgende Szenario:

```
using var host = HttpServer.CreateBuilder(12300)
 .UseRouter(r =>
 {
 r.RegisterValueHandler<IEnumerable<string>>(stringEnumerable =>
 {
 return new HttpResponse("String-Array:\n" + string.Join("\n", stringEnumerable));
 });
 r.RegisterValueHandler<IEnumerable<object>>(stringEnumerable =>
 {
 return new HttpResponse("Objekt-Array:\n" + string.Join("\n", stringEnumerable));
 });
 r.MapGet("/", request =>
 {
 return (IEnumerable<string>)["hello", "world"];
 });
 })
 .Build();
```

Im obigen Beispiel wird der `IEnumerable<string>`-Konverter **nie aufgerufen**, da das Eingabeobjekt immer ein `Object[]` ist und nicht in ein `IEnumerable<string>` umgewandelt werden kann. Der Konverter unten, der ein `IEnumerable<object>` empfängt, empfängt jedoch seine Eingabe, da sein Wert kompatibel ist.

Wenn Sie den Typ des Objekts, das enumeriert wird, tatsächlich verarbeiten müssen, müssen Sie die Reflexion verwenden, um den Typ des SammlungsElements zu erhalten. Alle enumerable Objekte (Listen, Arrays und Sammlungen) werden durch den HTTP-Antwortkonverter in ein Array von Objekten umgewandelt.

Werte, die IAsyncEnumerable implementieren, werden automatisch vom Server behandelt, wenn die ConvertIAsyncEnumerableIntoEnumerable-Eigenschaft aktiviert ist, ähnlich wie bei `IEnumerable`. Eine asynchrone Enumeration wird in einen blockierenden Enumerator umgewandelt und dann in ein synchrones Array von Objekten umgewandelt.

Logging

Sie können Sisk so konfigurieren, dass Zugriffs- und Fehlerprotokolle automatisch geschrieben werden. Es ist möglich, Logrotation, Erweiterungen und Frequenz zu definieren.

Die Klasse LogStream bietet einen asynchronen Weg, Protokolle zu schreiben und sie in einer wartbaren Schreibwarteschlange zu halten.

In diesem Artikel zeigen wir Ihnen, wie Sie das Logging für Ihre Anwendung konfigurieren.

Dateibasierte Zugriffsprotokolle

Protokolle zu Dateien öffnen die Datei, schreiben die Zeilentext und schließen die Datei anschließend für jede geschriebene Zeile. Dieses Verfahren wurde übernommen, um die Schreibreaktivität in den Protokollen zu erhalten.

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  }
```

Der obige Code schreibt alle eingehenden Anfragen in die Datei `logs/access.log`. Beachten Sie, dass die Datei automatisch erstellt wird, wenn sie nicht existiert, jedoch der Ordner davor nicht. Es ist nicht erforderlich, das

Verzeichnis `logs/` zu erstellen, da die LogStream-Klasse es automatisch erstellt.

Stream-basiertes Logging

Sie können Logdateien in TextWriter-Objekte schreiben, wie z. B. `Console.Out` , indem Sie ein TextWriter-Objekt im Konstruktor übergeben:

Program.cs **C#**

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

Für jede Nachricht, die im stream-basierten Log geschrieben wird, wird die Methode `TextWriter.Flush()` aufgerufen.

Formatierung des Zugriffsprotokolls

Sie können das Format des Zugriffsprotokolls mit vordefinierten Variablen anpassen. Betrachten Sie die folgende Zeile:

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

Es schreibt eine Nachricht wie:

```
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]
```

Sie können Ihre Logdatei nach dem beschriebenen Format formatieren:

Value	Was es repräsentiert	Beispiel
%dd	Tag des Monats (formatiert als zwei Ziffern)	05
%dmmm	Vollständiger Name des Monats	July
%dmm	Abgekürzter Name des Monats (drei Buchstaben)	Jul
%dm	Monatsnummer (formatiert als zwei Ziffern)	07
%dy	Jahr (formatiert als vier Ziffern)	2023
%th	Stunde im 12-Stunden-Format	03
%tH	Stunde im 24-Stunden-Format (HH)	15
%ti	Minuten (formatiert als zwei Ziffern)	30
%ts	Sekunden (formatiert als zwei Ziffern)	45
%tm	Millisekunden (formatiert als drei Ziffern)	123
%tz	Zeitzonenoffset (Gesamtstunden in UTC)	+03:00
%ri	Remote IP-Adresse des Clients	192.168.1.100
%rm	HTTP-Methode (Großbuchstaben)	GET
%rs	URI-Schema (http/https)	https
%ra	URI-Authority (Domain)	example.com
%rh	Host der Anfrage	www.example.com
%rp	Port der Anfrage	443
%rz	Pfad der Anfrage	/path/to/resource
%rq	Abfragezeichenkette	?key=value&another=123
%sc	HTTP-Antwortstatuscode	200
%sd	HTTP-Antwortstatusbeschreibung	OK
%lin	Menschlich lesbare Größe der Anfrage	1.2 KB
%linr	Rohgröße der Anfrage (Bytes)	1234
%lou	Menschlich lesbare Größe der Antwort	2.5 KB

Value	Was es repräsentiert	Beispiel
%lour	Rohgröße der Antwort (Bytes)	2560
%lms	Verstrichene Zeit in Millisekunden	120
%ls	Ausführungsstatus	Executed
%{header-name}	Repräsentiert den Header `header-name` der Anfrage.	`Mozilla/5.0 (platform; rv:gecko [...]`
%{:res-name}	Repräsentiert den Header `res-name` der Antwort.	

Rotierende Protokolle

Sie können den HTTP-Server so konfigurieren, dass die Logdateien in eine komprimierte .gz-Datei umgewandelt werden, wenn sie eine bestimmte Größe erreichen. Die Größe wird periodisch anhand des von Ihnen definierten Schwellenwerts überprüft.

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

Der obige Code prüft alle sechs Stunden, ob die Datei des LogStreams sein 64 MB-Limit erreicht hat. Wenn ja, wird die Datei in eine .gz-Datei komprimiert und anschließend wird `access.log` bereinigt.

Während dieses Prozesses ist das Schreiben in die Datei gesperrt, bis die Datei komprimiert und bereinigt ist. Alle Zeilen, die in diesem Zeitraum geschrieben werden sollen, befinden sich in einer Warteschlange, die auf das Ende der Komprimierung wartet.

Diese Funktion funktioniert nur mit dateibasierten LogStreams.

Fehlerprotokollierung

Wenn ein Server keine Fehler an den Debugger weiterleitet, leitet er die Fehler beim Schreiben von Protokollen weiter, wenn welche vorhanden sind. Sie können das Schreiben von Fehlern mit folgendem Code konfigurieren:

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

Diese Eigenschaft schreibt nur etwas in das Protokoll, wenn der Fehler nicht von der Callback-Funktion oder der Eigenschaft Router.CallbackErrorHandler erfasst wird.

Der vom Server geschriebene Fehler enthält immer Datum und Uhrzeit, die Anfrage-Header (nicht den Körper), die Fehlerspur und die Spur der inneren Ausnahme, falls vorhanden.

Weitere Logging-Instanzen

Ihre Anwendung kann null oder mehrere LogStreams haben, es gibt keine Begrenzung, wie viele Logkanäle sie haben kann. Daher ist es möglich, das Log Ihrer Anwendung auf eine Datei zu leiten, die nicht das Standardzugriffs- oder Fehlerprotokoll ist.

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

Erweiterung von LogStream

Sie können die Klasse `LogStream` erweitern, um benutzerdefinierte Formate zu schreiben, die mit der aktuellen Sisk-Log-Engine kompatibel sind. Das folgende Beispiel ermöglicht das Schreiben farbiger Nachrichten in die Konsole über die Bibliothek Spectre.Console:

CustomLogStream.cs **C#**

```
public class CustomLogStream : LogStream
{
    protected override void WriteLineInternal(string line)
    {
```

```
            base.WriteLineInternal($"[{DateTime.Now:g}] {line}");
        }
    }
```

Eine weitere Möglichkeit, automatisch benutzerdefinierte Protokolle für jede Anfrage/Antwort zu schreiben, besteht darin, einen HttpServerHandler zu erstellen. Das folgende Beispiel ist etwas ausführlicher. Es schreibt den Körper der Anfrage und Antwort in JSON in die Konsole. Es kann nützlich sein, Anfragen im Allgemeinen zu debuggen. Dieses Beispiel nutzt ContextBag und HttpServerHandler.

Program.cs C#

```
class Program
{
    static async Task Main(string[] args)
    {
        var app = HttpServer.CreateBuilder(host =>
        {
            host.UseListeningPort(5555);
            host.UseHandler<JsonMessageHandler>();
        });

        app.Router += new Route(RouteMethod.Any, "/json", request =>
        {
            return new HttpResponse()
                .WithContent(JsonContent.Create(new
                {
                    method = request.Method.Method,
                    path = request.Path,
                    specialMessage = "Hello, world!!"
                }));
        });

        await app.StartAsync();
    }
}
```

JsonMessageHandler.cs C#

```
class JsonMessageHandler : HttpServerHandler
{
    protected override void OnHttpRequestOpen(HttpRequest request)
    {
        if (request.Method != HttpMethod.Get && request.Headers["Content-
Type"]?.Contains("json", StringComparison.InvariantCultureIgnoreCase) == true)
```

```
        {
            // At this point, the connection is open and the client has sent the
header specifying
            // that the content is JSON.The line below reads the content and leaves it
stored in the request.
            //
            // If the content is not read in the request action, the GC is likely to
collect the content
            // after sending the response to the client, so the content may not be
available after the response is closed.
            //
            _ = request.RawBody;

            // add hint in the context to tell that this request has an json body on it
            request.Bag.Add("IsJsonRequest", true);
        }
    }

    protected override async void OnHttpRequestClose(HttpServerExecutionResult result)
    {
        string? requestJson = null,
                responseJson = null,
                responseMessage;

        if (result.Request.Bag.ContainsKey("IsJsonRequest"))
        {
            // reformats the JSON using the CypherPotato.LightJson library
            var content = result.Request.Body;
            requestJson = JsonValue.Deserialize(content, new JsonOptions() { WriteIndented
= true }).ToString();
        }

        if (result.Response is { } response)
        {
            var content = response.Content;
            responseMessage = $"{(int)response.Status}
{HttpStatusInformation.GetStatusCodeDescription(response.Status)}";

            if (content is HttpContent httpContent &&
                // check if the response is JSON
                httpContent.Headers.ContentType?.MediaType?.Contains("json",
StringComparison.InvariantCultureIgnoreCase) == true)
            {
                string json = await httpContent.ReadAsStringAsync();
                responseJson = JsonValue.Deserialize(json, new JsonOptions() {
WriteIndented = true }).ToString();
            }
        }
```

```
        else
        {
            // gets the internal server handling status
            responseMessage = result.Status.ToString();
        }

        StringBuilder outputMessage = new StringBuilder();

        if (requestJson =/= null)
        {
            outputMessage.AppendLine("-----");
            outputMessage.AppendLine($">>> {result.Request.Method} {result.Request.Path}");

            if (requestJson is not null)
                outputMessage.AppendLine(requestJson);
        }

        outputMessage.AppendLine($"<<< {responseMessage}");

        if (responseJson is not null)
            outputMessage.AppendLine(responseJson);

        outputMessage.AppendLine("-----");

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

Server Sent Events

Sisk unterstützt das Senden von Nachrichten über Server Sent Events out of the box. Sie können disposible und persistente Verbindungen erstellen, die Verbindungen während der Laufzeit abrufen und verwenden.

Diese Funktion hat einige Einschränkungen, die von Browsern auferlegt werden, wie z.B. das Senden von nur Textnachrichten und das Nicht-Schließen einer Verbindung dauerhaft. Eine serverseitig geschlossene Verbindung wird von einem Client alle 5 Sekunden (3 für einige Browser) versuchen, erneut zu verbinden.

Diese Verbindungen sind nützlich, um Ereignisse vom Server an den Client zu senden, ohne dass der Client die Informationen jedes Mal anfordern muss.

Erstellen einer SSE-Verbindung

Eine SSE-Verbindung funktioniert wie eine reguläre HTTP-Anfrage, aber anstatt eine Antwort zu senden und die Verbindung sofort zu schließen, wird die Verbindung offen gehalten, um Nachrichten zu senden.

Durch Aufrufen der HttpRequest.GetEventSource()-Methode wird die Anfrage in einen Wartezustand versetzt, während die SSE-Instanz erstellt wird.

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

Im obigen Code erstellen wir eine SSE-Verbindung und senden eine "Hallo, Welt!"-Nachricht, dann schließen wir die SSE-Verbindung von der Serverseite.

> **NOTE**
>
> Wenn eine serverseitige Verbindung geschlossen wird, versucht der Client standardmäßig, die Verbindung erneut herzustellen, und die Verbindung wird neu gestartet, die Methode wird erneut ausgeführt, für immer.
>
> Es ist üblich, eine Terminationsnachricht vom Server zu senden, wenn die Verbindung vom Server geschlossen wird, um zu verhindern, dass der Client versucht, erneut zu verbinden.

Anhängen von Headern

Wenn Sie Header senden müssen, können Sie die HttpRequestEventSource.AppendHeader-Methode verwenden, bevor Sie Nachrichten senden.

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

Beachten Sie, dass es notwendig ist, die Header zu senden, bevor Sie Nachrichten senden.

Wait-For-Fail-Verbindungen

Verbindungen werden normalerweise beendet, wenn der Server nicht mehr in der Lage ist, Nachrichten zu senden, aufgrund einer möglichen clientseitigen Trennung. Damit wird die Verbindung automatisch beendet und die Instanz der Klasse verworfen.

Selbst bei einer erneuten Verbindung funktioniert die Instanz der Klasse nicht, da sie an die vorherige Verbindung gekoppelt ist. In einigen Situationen benötigen Sie diese Verbindung später und möchten sie nicht über die Callback-Methode der Route verwalten.

Dafür können wir die SSE-Verbindungen mit einer Kennung identifizieren und sie später mithilfe dieser Kennung abrufen, auch außerhalb des Callbacks der Route. Darüber hinaus markieren wir die Verbindung mit WaitForFail, um die Route nicht zu beenden und die Verbindung automatisch zu beenden.

Eine SSE-Verbindung in KeepAlive wartet auf einen Sendefehler (verursacht durch Trennung), um die Methoden Ausführung fortzusetzen. Es ist auch möglich, ein Timeout für dies festzulegen. Nach Ablauf der Zeit wird die Verbindung beendet, wenn keine Nachricht gesendet wurde, und die Ausführung wird fortgesetzt.

```
r += new Route(RouteMethod.Get, "/", (req) =>
{
 using var sse = req.GetEventSource("meine-index-verbindung");

 sse.WaitForFail(TimeSpan.FromSeconds(15)); // warten Sie 15 Sekunden ohne Nachrichten,
bevor Sie die Verbindung beenden

 return sse.Close();
});
```

Die obige Methode erstellt die Verbindung, behandelt sie und wartet auf eine Trennung oder einen Fehler.

```
HttpRequestEventSource? evs = server.EventSources.GetByIdentifier("meine-
index-verbindung");
if (evs =/= null)
{
 // die Verbindung ist noch aktiv
 evs.Send("Hallo wieder!");
}
```

Und das obige Code-Snippet versucht, die neu erstellte Verbindung zu suchen, und wenn sie existiert, sendet es eine Nachricht an sie.

Alle aktiven Serververbindungen, die identifiziert werden, stehen in der Sammlung HttpServer.EventSources zur Verfügung. Diese Sammlung speichert nur aktive und identifizierte Verbindungen. Geschlossene Verbindungen werden aus der Sammlung entfernt.

> **NOTE**
>
> Es ist wichtig zu beachten, dass Keep-Alive eine Grenze hat, die von Komponenten festgelegt wird, die möglicherweise auf unkontrollierbare Weise mit Sisk verbunden sind, wie z.B. ein Web-Proxy, ein HTTP-Kernel oder ein Netzwerktreiber, und diese schließen inaktive Verbindungen nach einer bestimmten Zeit.
>
> Daher ist es wichtig, die Verbindung offen zu halten, indem Sie periodische Pings senden oder die maximale Zeit verlängern, bevor die Verbindung geschlossen wird. Lesen Sie den nächsten Abschnitt, um besser zu verstehen, wie periodische Pings gesendet werden.

Einrichtung von Ping-Richtlinien für Verbindungen

Ping-Richtlinien sind eine automatisierte Möglichkeit, periodische Nachrichten an Ihren Client zu senden. Diese Funktion ermöglicht es dem Server, zu verstehen, wann der Client die Verbindung getrennt hat, ohne die Verbindung auf unbestimmte Zeit offen halten zu müssen.

```
[RouteGet("/sse")]
public HttpResponse Events(HttpRequest request)
{
 using var sse = request.GetEventSource();
 sse.WithPing(ping ⇒
 {
 ping.DataMessage = "Ping-Nachricht";
 ping.Interval = TimeSpan.FromSeconds(5);
 ping.Start();
 });

 sse.KeepAlive();
 return sse.Close();
}
```

Im obigen Code wird alle 5 Sekunden eine neue Ping-Nachricht an den Client gesendet. Dies hält die TCP-Verbindung aktiv und verhindert, dass sie aufgrund von Inaktivität geschlossen wird. Wenn eine Nachricht nicht gesendet werden kann, wird die Verbindung automatisch geschlossen, wodurch die von der Verbindung verwendeten Ressourcen freigegeben werden.

Abfragen von Verbindungen

Sie können aktive Verbindungen mithilfe eines Prädikats für die Verbindungs-ID suchen, um beispielsweise zu broadcasten.

```
HttpRequestEventSource[] evs = server.EventSources.Find(es ⇒ es.StartsWith("meine-
verbindung-"));
foreach (HttpRequestEventSource e in evs)
{
 e.Send("Broadcast an alle Ereignisquellen, die mit 'meine-verbindung-' beginnen");
}
```

Sie können auch die All-Methode verwenden, um alle aktiven SSE-Verbindungen zu erhalten.

Web Sockets

Sisk unterstützt WebSockets ebenfalls, zum Beispiel das Empfangen und Senden von Nachrichten an den Client.

Diese Funktion arbeitet in den meisten Browsern einwandfrei, in Sisk ist sie jedoch noch experimentell. Bitte melden Sie etwaige Fehler auf GitHub.

Nachrichten asynchron akzeptieren

WebSocket-Nachrichten werden in Reihenfolge empfangen, bis sie von `ReceiveMessageAsync` verarbeitet werden. Diese Methode liefert keine Nachricht, wenn das Timeout erreicht ist, die Operation abgebrochen wird oder der Client getrennt ist.

Nur eine Lese- und Schreiboperation kann gleichzeitig stattfinden, daher ist es nicht möglich, während des Wartens auf eine Nachricht mit `ReceiveMessageAsync` etwas an den verbundenen Client zu schreiben.

```
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();
     });
```

Nachrichten synchron akzeptieren

Das folgende Beispiel zeigt, wie man einen synchronen WebSocket verwendet, ohne einen asynchronen Kontext, wobei man die Nachrichten empfängt, verarbeitet und den Socket anschließend beendet.

```
router.MapGet("/connect", async (HttpRequest req) =>
{
    using var ws = await req.GetWebSocketAsync();
    WebSocketMessage? msg;

askName:
    await ws.SendAsync("What is your name?");
    msg = await ws.ReceiveMessageAsync();

    if (msg is null)
        return await ws.CloseAsync();

    string name = msg.GetString();

    if (string.IsNullOrEmpty(name))
    {
        await ws.SendAsync("Please, insert your name!");
        goto askName;
    }

askAge:
    await ws.SendAsync("And your age?");
    msg = await ws.ReceiveMessageAsync();

    if (msg is null)
        return await ws.CloseAsync();

    if (!Int32.TryParse(msg?.GetString(), out int age))
    {
        await ws.SendAsync("Please, insert an valid number");
        goto askAge;
    }

    await ws.SendAsync($"You're {name}, and you are {age} old.");

    return await ws.CloseAsync();
});
```

Ping-Policy

Ähnlich wie die Ping-Policy bei Server Side Events kann auch hier eine Ping-Policy konfiguriert werden, um die TCP-Verbindung offen zu halten, wenn dort Inaktivität besteht.

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

Discard-Syntax

Der HTTP-Server kann verwendet werden, um auf eine Callback-Anfrage von einer Aktion, wie z.B. OAuth-Authentifizierung, zu hören und kann nach Erhalt dieser Anfrage verworfen werden. Dies kann in Fällen nützlich sein, in denen Sie eine Hintergrundaktion benötigen, aber keine gesamte HTTP-Anwendung dafür einrichten möchten.

Das folgende Beispiel zeigt, wie man einen lauschenden HTTP-Server auf Port 5555 mit CreateListener erstellt und auf den nächsten Kontext wartet:

```
using (var server = HttpServer.CreateListener(5555))
{
    // warte auf die nächste HTTP-Anfrage
    var context = await server.WaitNextAsync();
    Console.WriteLine($"Angeforderter Pfad: {context.Request.Path}");
}
```

Die WaitNext-Funktion wartet auf den nächsten Kontext einer abgeschlossenen Anfrageverarbeitung. Sobald das Ergebnis dieser Operation erhalten ist, hat der Server die Anfrage bereits vollständig bearbeitet und die Antwort an den Client gesendet.

Abhängigkeitsinjektion

Es ist üblich, Mitglieder und Instanzen zu widmen, die für die gesamte Lebensdauer eines Anfrages gültig sind, wie z.B. eine Datenbankverbindung, einen authentifizierten Benutzer oder ein Sitzungstoken. Eine der Möglichkeiten besteht darin, den HttpContext.RequestBag zu verwenden, der ein Dictionary erstellt, das für die gesamte Lebensdauer eines Anfrages gültig ist.

Dieses Dictionary kann von Anfragebehandlern zugreifen und Variablen während der gesamten Anfrage definieren. Zum Beispiel kann ein Anfragebehandler, der einen Benutzer authentifiziert, diesen Benutzer im `HttpContext.RequestBag` setzen, und innerhalb der Anfrage-Logik kann dieser Benutzer mit `HttpContext.RequestBag.Get<User>()` abgerufen werden.

Hier ist ein Beispiel:

RequestHandlers/AuthenticateUser.cs C#

```
public class AuthenticateUser : IRequestHandler
{
    public RequestHandlerExecutionMode ExecutionMode { get; init; } =
RequestHandlerExecutionMode.BeforeResponse;

    public HttpResponse? Execute(HttpRequest request, HttpContext context)
    {
        User authenticatedUser = AuthenticateUser(request);
        context.RequestBag.Set(authenticatedUser);
        return null; // advance to the next request handler or request logic
    }
}
```

Controllers/HelloController.cs C#

```
[RouteGet("/hello")]
[RequestHandler<AuthenticateUser>]
public static HttpResponse SayHello(HttpRequest request)
{
    var authenticatedUser = request.Bag.Get<User>();
    return new HttpResponse()
    {
        Content = new StringContent($"Hallo {authenticatedUser.Name}!")
```

```
        };
    }
```

Dies ist ein vorläufiges Beispiel für diese Operation. Die Instanz von `User` wurde innerhalb des Anfragebehandlers für die Authentifizierung erstellt, und alle Routen, die diesen Anfragebehandler verwenden, haben die Garantie, dass es eine `User` -Instanz in ihrem `HttpContext.RequestBag` gibt.

Es ist möglich, Logik zu definieren, um Instanzen zu erhalten, wenn sie nicht zuvor im `RequestBag` definiert wurden, durch Methoden wie GetOrAdd oder GetOrAddAsync.

Seit Version 1.3 wurde die statische Eigenschaft HttpContext.Current eingeführt, die den Zugriff auf den aktuellen `HttpContext` des Anfragekontexts ermöglicht. Dies ermöglicht es, Mitglieder des `HttpContext` außerhalb der aktuellen Anfrage zu exponieren und Instanzen in Routen-Objekten zu definieren.

Das folgende Beispiel definiert einen Controller, der Mitglieder enthält, die häufig im Kontext einer Anfrage zugegriffen werden.

Controllers/Controller.cs **C#**

```
public abstract class Controller : RouterModule
{
    public DbContext Database
    {
        get
        {
            // erstelle einen DbContext oder hole den bestehenden
            return HttpContext.Current.RequestBag.GetOrAdd(() => new DbContext());
        }
    }

    // die folgende Zeile wird einen Fehler werfen, wenn die Eigenschaft zugegriffen wird, wenn der Benutzer nicht
    // im RequestBag definiert ist
    public User AuthentifizierterBenutzer { get => HttpContext.Current.RequestBag.Get<User>(); }

    // der Zugriff auf die HttpRequest-Instanz wird auch unterstützt
    public HttpRequest Request { get => HttpContext.Current.Request; }
}
```

Und definiere Typen, die von diesem Controller erben:

Controllers/PostsController.cs **C#**

```
[RoutePrefix("/api/posts")]
public class PostsController : Controller
{
    [RouteGet]
    public IEnumerable<Blog> ListPosts()
    {
        return Database.Posts
            .Where(post => post.AuthorId == AuthentifizierterBenutzer.Id)
            .ToList();
    }

    [RouteGet("<id>")]
    public Post GetPost()
    {
        int blogId = Request.RouteParameters["id"].GetInteger();

        Post? post = Database.Posts
            .FirstOrDefault(post => post.Id == blogId && post.AuthorId
== AuthentifizierterBenutzer.Id);

        return post ?? new HttpResponse(404);
    }
}
```

Für das obige Beispiel müssen Sie einen Wert-Handler in Ihrem Router konfigurieren, damit die von Ihrem Router zurückgegebenen Objekte in ein gültiges HttpResponse umgewandelt werden.

Beachten Sie, dass die Methoden kein `HttpRequest request` -Argument haben, wie es in anderen Methoden der Fall ist. Dies liegt daran, dass der Router seit Version 1.3 zwei Arten von Delegaten für Routing-Antworten unterstützt: RouteAction, der standardmäßig ein `HttpRequest` -Argument erhält, und ParameterlessRouteAction. Das `HttpRequest` -Objekt kann immer noch über die Request-Eigenschaft des statischen `HttpContext` auf dem Thread zugegriffen werden.

Im obigen Beispiel haben wir ein disposable-Objekt, den `DbContext` , definiert, und wir müssen sicherstellen, dass alle im `DbContext` erstellten Instanzen verworfen werden, wenn die HTTP-Sitzung endet. Dazu können wir zwei Methoden verwenden. Eine Möglichkeit besteht darin, einen Anfragebehandler zu erstellen, der nach der Aktion des Routers ausgeführt wird, und die andere Möglichkeit besteht darin, einen benutzerdefinierten Server-Handler zu verwenden.

Für die erste Methode können wir den Anfragebehandler inline direkt im OnSetup-Methoden des `RouterModule` erstellen:

Controllers/PostsController.cs **C#**

```
public abstract class Controller : RouterModule
{
    // ...

    protected override void OnSetup(Router parentRouter)
    {
        base.OnSetup(parentRouter);

        HasRequestHandler(RequestHandler.Create(
            execute: (req, ctx) =>
            {
                // hole einen im Request-Handler-Kontext definierten DbContext und
                // verwerfe ihn
                ctx.RequestBag.GetOrDefault<DbContext>()?.Dispose();
                return null;
            },
            executionMode: RequestHandlerExecutionMode.AfterResponse));
    }
}
```

> **TIP**
>
> Seit Sisk-Version 1.4 ist die Eigenschaft HttpServerConfiguration.DisposeDisposableContextValues eingeführt und standardmäßig aktiviert, die bestimmt, ob der HTTP-Server alle `IDisposable` -Werte im Kontext-Beutel verwerfen soll, wenn eine HTTP-Sitzung geschlossen wird.

Die obige Methode stellt sicher, dass der `DbContext` verworfen wird, wenn die HTTP-Sitzung abgeschlossen ist. Sie können dies für weitere Mitglieder tun, die am Ende einer Antwort verworfen werden müssen.

Für die zweite Methode können Sie einen benutzerdefinierten Server-Handler erstellen, der den `DbContext` verwerfen wird, wenn die HTTP-Sitzung abgeschlossen ist.

Server/Handlers/ObjectDisposerHandler.cs **C#**

```
public class ObjectDisposerHandler : HttpServerHandler
{
    protected override void OnHttpRequestClose(HttpServerExecutionResult result)
    {
        result.Context.RequestBag.GetOrDefault<DbContext>()?.Dispose();
    }
}
```

Und verwenden Sie ihn in Ihrem App-Builder:

Program.cs C#

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

Dies ist eine Möglichkeit, Code-Reinigung und Abhängigkeitsinjektion zu handhaben und die Abhängigkeiten einer Anfrage getrennt von der Art des Moduls zu halten, das verwendet wird, um die Menge an dupliziertem Code innerhalb jeder Aktion eines Routers zu reduzieren. Es ist eine Praxis, die ähnlich ist wie die, die in Frameworks wie ASP.NET verwendet wird.

Streaming-Inhalt

Das Sisk unterstützt das Lesen und Senden von Inhalten als Streams zwischen Client und Server. Diese Funktion ist nützlich, um den Speicherüberkopft für die Serialisierung und Deserialisierung von Inhalten während der Lebensdauer einer Anfrage zu reduzieren.

Anfrage-Inhalt-Stream

Kleine Inhalte werden automatisch in den HTTP-Verbindungspuffer-Speicher geladen, sodass dieser Inhalt schnell in HttpRequest.Body und HttpRequest.RawBody geladen wird. Für größere Inhalte kann die HttpRequest.GetRequestStream-Methode verwendet werden, um den Anfrage-Inhalt-Stream zu erhalten.

Es ist wichtig zu beachten, dass die HttpRequest.GetMultipartFormContent-Methode den gesamten Anfrage-Inhalt in den Speicher lädt, sodass sie für das Lesen großer Inhalte nicht geeignet ist.

Betrachten Sie das folgende Beispiel:

Controller/UploadDocument.cs **C#**

```
[RoutePost ( "/api/upload-document/<filename>" )]
public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {

    var fileName = request.RouteParameters [ "filename" ].GetString ();

    if (!request.HasContents) {
        // Anfrage hat keinen Inhalt
        return new HttpResponse ( HttpStatusInformation.BadRequest );
    }

    var contentStream = request.GetRequestStream ();
    var outputFileName = Path.Combine (
        AppDomain.CurrentDomain.BaseDirectory,
        "uploads",
        fileName );

    using (var fs = File.Create ( outputFileName )) {
        await contentStream.CopyToAsync ( fs );
```

```
    }

    return new HttpResponse () {
        Content = JsonContent.Create ( new { message = "Datei erfolgreich gesendet." } )
    };
}
```

In dem obigen Beispiel liest die `UploadDocument` -Methode den Anfrage-Inhalt und speichert den Inhalt in einer Datei. Es wird keine zusätzliche Speicherzuweisung vorgenommen, außer für den Lese-Puffer, der von `Stream.CopyToAsync` verwendet wird. Das obige Beispiel reduziert den Druck der Speicherzuweisung für sehr große Dateien, was die Anwendungsleistung optimieren kann.

Eine gute Praxis ist es, immer ein CancellationToken in einer Operation zu verwenden, die zeitaufwändig sein kann, wie z.B. das Senden von Dateien, da es von der Netzwerkgeschwindigkeit zwischen Client und Server abhängt.

Die Anpassung mit einem CancellationToken kann wie folgt vorgenommen werden:

Controller/UploadDocument.cs — C#

```
// Der CancellationToken unten wird eine Ausnahme auslösen, wenn die 30-Sekunden-Frist erreicht ist.
CancellationTokenSource copyCancellation = new CancellationTokenSource ( delay: TimeSpan.FromSeconds ( 30 ) );

try {
    using (var fs = File.Create ( outputFileName )) {
        await contentStream.CopyToAsync ( fs, copyCancellation.Token );
    }
}
catch (OperationCanceledException) {
    return new HttpResponse ( HttpStatusInformation.BadRequest ) {
        Content = JsonContent.Create ( new { Error = "Der Upload hat die maximale Upload-Zeit (30 Sekunden) überschritten." } )
    };
}
```

Antwort-Inhalt-Stream

Das Senden von Antwort-Inhalten ist auch möglich. Derzeit gibt es zwei Möglichkeiten, dies zu tun: über die HttpRequest.GetResponseStream-Methode und mit einem Inhalt vom Typ StreamContent.

Betrachten Sie ein Szenario, in dem wir ein Bild senden müssen. Dazu können wir den folgenden Code verwenden:

Controller/ImageController.cs C#

```
[RouteGet ( "/api/profile-picture" )]
public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {

    // Beispiel-Methode, um ein Profilbild zu erhalten
    var profilePictureFilename = "profile-picture.jpg";
    byte[] profilePicture = await File.ReadAllBytesAsync ( profilePictureFilename );

    return new HttpResponse () {
        Content = new ByteArrayContent ( profilePicture ),
        Headers = new () {
            ContentType = "image/jpeg",
            ContentDisposition = $"inline; filename={profilePictureFilename}"
        }
    };
}
```

Die obige Methode führt eine Speicherzuweisung durch, wenn sie den Bild-Inhalt liest. Wenn das Bild groß ist, kann dies ein Leistungsproblem verursachen und in Spitzenzeiten sogar einen Speicherüberlauf und einen Server-Absturz verursachen. In diesen Situationen kann Caching nützlich sein, aber es wird das Problem nicht eliminieren, da Speicher immer noch für diese Datei reserviert wird. Caching kann den Druck der Speicherzuweisung für jede Anfrage lindern, aber für große Dateien wird es nicht ausreichen.

Das Senden des Bildes über einen Stream kann eine Lösung für das Problem sein. Anstatt den gesamten Bild-Inhalt zu lesen, wird ein Lese-Stream auf der Datei erstellt und mit einem kleinen Puffer an den Client kopiert.

Senden über die GetResponseStream-Methode

Die HttpRequest.GetResponseStream-Methode erstellt ein Objekt, das das Senden von Teilen der HTTP-Antwort ermöglicht, während der Inhalt-Fluss vorbereitet wird. Diese Methode ist manuell und erfordert, dass Sie den Status, die Header und die Inhaltsgröße vor dem Senden des Inhalts definieren.

Controller/ImageController.cs C#

```
[RouteGet ( "/api/profile-picture" )]
public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
```

```
3
4       var profilePictureFilename = "profile-picture.jpg";
5
6       // in dieser Form des Sendens müssen der Status und der Header definiert werden
7       // bevor der Inhalt gesendet wird
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          // in dieser Form des Sendens muss auch die Inhaltsgröße definiert werden
18          // bevor sie gesendet wird.
19          requestStreamManager.SetContentLength ( fs.Length );
20
21          // wenn Sie die Inhaltsgröße nicht kennen, können Sie chunked-encoding
22          // verwenden, um den Inhalt zu senden
23          requestStreamManager.SendChunked = true;
24
25          // und dann schreiben Sie in den Ausgabe-Stream
26          await fs.CopyToAsync ( requestStreamManager.ResponseStream );
27      }
    }
```

Senden von Inhalten über einen StreamContent

Die StreamContent-Klasse ermöglicht das Senden von Inhalten aus einer Datenquelle als Byte-Stream. Diese Form des Sendens ist einfacher und entfernt die vorherigen Anforderungen und ermöglicht sogar die Verwendung von Komprimierungs-Codierung, um die Inhaltsgröße zu reduzieren.

Controller/ImageController.cs **C#**

```
1   [RouteGet ( "/api/profile-picture" )]
2   public HttpResponse UploadDocument ( HttpRequest request ) {
3
4       var profilePictureFilename = "profile-picture.jpg";
5
6       return new HttpResponse () {
7           Content = new StreamContent ( File.OpenRead ( profilePictureFilename ) ),
8           Headers = new () {
9               ContentType = "image/jpeg",
10              ContentDisposition = $"inline; filename=\"{profilePictureFilename}\""
```

```
11            }
12        };
13    }
```

> [!WICHTIG]
>
> Bei dieser Art von Inhalten sollten Sie den Stream nicht in einem `using` -Block einwickeln. Der Inhalt wird automatisch vom HTTP-Server verworfen, wenn der Inhalts-Fluss abgeschlossen ist, mit oder ohne Fehler.

Aktivierung von CORS (Cross-Origin Resource Sharing) in Sisk

Sisk verfügt über ein Tool, das bei der Handhabung von Cross-Origin Resource Sharing (CORS) hilfreich sein kann, wenn Sie Ihren Dienst öffentlich zugänglich machen. Diese Funktion ist kein Bestandteil des HTTP-Protokolls, sondern eine spezifische Funktion von Webbrowsern, die vom W3C definiert wurde. Dieser Sicherheitsmechanismus verhindert, dass eine Webseite Anfragen an eine andere Domain als die, die die Webseite bereitgestellt hat, stellt. Ein Dienstanbieter kann bestimmten Domains den Zugriff auf seine Ressourcen erlauben, oder nur einer Domain.

Same Origin

Damit eine Ressource als „same origin“ identifiziert wird, muss eine Anfrage den Origin-Header in ihrer Anfrage angeben:

```
GET /api/users HTTP/1.1
Host: example.com
Origin: http://example.com
...
```

Und der Remote-Server muss mit einem Access-Control-Allow-Origin-Header mit demselben Wert wie die angeforderte Origin antworten:

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

Diese Überprüfung ist **explizit**: Host, Port und Protokoll müssen exakt mit der angeforderten Origin übereinstimmen. Prüfen Sie das Beispiel:

- Ein Server antwortet, dass sein `Access-Control-Allow-Origin` `https://example.com` ist:
  - `https://example.net` – die Domain ist anders.
  - `http://example.com` – das Schema ist anders.

- `http://example.com:5555` – der Port ist anders.
- `https://www.example.com` – der Host ist anders.

In der Spezifikation ist nur die Syntax für beide Header erlaubt, sowohl für Anfragen als auch für Antworten. Der URL-Pfad wird ignoriert. Der Port wird ebenfalls ausgelassen, wenn es sich um einen Standardport handelt (80 für HTTP und 443 für HTTPS).

```
Origin: null
Origin: <scheme>://<hostname>
Origin: <scheme>://<hostname>:<port>
```

Aktivierung von CORS

Standardmäßig haben Sie das Objekt CrossOriginResourceSharingHeaders innerhalb Ihres ListeningHost.

Sie können CORS beim Initialisieren des Servers konfigurieren:

```
static async Task Main(string[] args)
{
    using var app = HttpServer.CreateBuilder()
        .UseCors(new CrossOriginResourceSharingHeaders(
            allowOrigin: "http://example.com",
            allowHeaders: ["Authorization"],
            exposeHeaders: ["Content-Type"]))
        .Build();

    await app.StartAsync();
}
```

Der obige Code sendet die folgenden Header für **alle Antworten**:

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

Diese Header müssen für alle Antworten an einen Webclient gesendet werden, einschließlich Fehler und Weiterleitungen.

Sie werden feststellen, dass die Klasse CrossOriginResourceSharingHeaders zwei ähnliche Eigenschaften hat: AllowOrigin und AllowOrigins. Beachten Sie, dass eine die Mehrzahl und die andere die Einzahl ist.

- Die **AllowOrigin**-Eigenschaft ist statisch: nur die von Ihnen angegebene Origin wird für alle Antworten gesendet.
- Die **AllowOrigins**-Eigenschaft ist dynamisch: der Server prüft, ob die Origin der Anfrage in dieser Liste enthalten ist. Wenn sie gefunden wird, wird sie für die Antwort dieser Origin gesendet.

Wildcards und automatische Header

Alternativ können Sie ein Wildcard (`*`) in der Origin der Antwort verwenden, um anzugeben, dass jede Origin auf die Ressource zugreifen darf. Dieser Wert ist jedoch nicht für Anfragen mit Credentials (Authorisierungsheader) erlaubt und diese Operation führt zu einem Fehler.

Sie können dieses Problem umgehen, indem Sie explizit auflisten, welche Origins über die Eigenschaft AllowOrigins erlaubt sind, oder auch die Konstante AutoAllowOrigin im Wert von AllowOrigin verwenden. Diese magische Eigenschaft definiert den `Access-Control-Allow-Origin` -Header für denselben Wert wie der `Origin` -Header der Anfrage.

Sie können auch AutoFromRequestMethod und AutoFromRequestHeaders für ein Verhalten ähnlich wie `AllowOrigin` verwenden, das automatisch auf Basis der gesendeten Header antwortet.

```
using var host = HttpServer.CreateBuilder()
    .UseCors(new CrossOriginResourceSharingHeaders(

        // Antwortet basierend auf dem Origin-Header der Anfrage
        allowOrigin: CrossOriginResourceSharingHeaders.AutoAllowOrigin,

        // Antwortet basierend auf dem Access-Control-Request-Method-Header oder der
Anfrage-Methode
        allowMethods: [CrossOriginResourceSharingHeaders.AutoFromRequestMethod],

        // Antwortet basierend auf dem Access-Control-Request-Headers-Header oder den
gesendeten Headern
        allowHeaders: [CrossOriginResourceSharingHeaders.AutoFromRequestHeaders]))
```

Andere Wege, CORS anzuwenden

Wenn Sie mit service providers arbeiten, können Sie Werte, die in der Konfigurationsdatei definiert sind, überschreiben:

```
static async Task Main(string[] args)
{
    using var app = HttpServer.CreateBuilder()
        .UsePortableConfiguration( ... )
        .UseCors(cors => {
            // Überschreibt die in der Konfiguration definierte Origin
            // Datei.
            cors.AllowOrigin = "http://example.com";
        })
        .Build();

    await app.StartAsync();
}
```

Deaktivieren von CORS auf spezifischen Routen

Die Eigenschaft `UseCors` ist sowohl für Routen als auch für alle Routenattribute verfügbar und kann mit folgendem Beispiel deaktiviert werden:

```
[RoutePrefix("api/widgets")]
public class WidgetController : Controller {

    // GET /api/widgets/colors
    [RouteGet("/colors", UseCors = false)]
    public IEnumerable<string> GetWidgets() {
        return new[] { "Green widget", "Red widget" };
    }
}
```

Ersetzen von Werten in der Antwort

Sie können Werte explizit in einer Router-Aktion ersetzen oder entfernen:

```
[RoutePrefix("api/widgets")]
public class WidgetController : Controller {

    public IEnumerable<string> GetWidgets(HttpRequest request) {

        // Entfernt den Access-Control-Allow-Credentials-Header
        request.Context.OverrideHeaders.AccessControlAllowCredentials = string.Empty;

        // Ersetzt den Access-Control-Allow-Origin
        request.Context.OverrideHeaders.AccessControlAllowOrigin = "https://contorso.com";

        return new[] { "Green widget", "Red widget" };
    }
}
```

Preflight-Anfragen

Eine Preflight-Anfrage ist eine OPTIONS-Methode, die der Client vor der eigentlichen Anfrage sendet.

Der Sisk-Server antwortet immer mit einem `200 OK` und den entsprechenden CORS-Headern, und dann kann der Client mit der eigentlichen Anfrage fortfahren. Diese Bedingung gilt nicht, wenn eine Route für die Anfrage mit dem RouteMethod explizit für `Options` konfiguriert ist.

Globale Deaktivierung von CORS

Es ist nicht möglich, dies zu tun. Um CORS nicht zu verwenden, konfigurieren Sie es nicht.

JSON-RPC-Erweiterung

Sisk hat ein experimentelles Modul für eine JSON-RPC 2.0-API, mit der Sie noch einfachere Anwendungen erstellen können. Diese Erweiterung implementiert die JSON-RPC 2.0-Transport-Schnittstelle strikt und bietet Transport über HTTP-GET-, POST-Anfragen und auch Web-Sockets mit Sisk.

Sie können die Erweiterung über Nuget mit dem folgenden Befehl installieren. Beachten Sie, dass Sie in experimentellen/Beta-Versionen die Option zum Suchen nach Vorabversionen in Visual Studio aktivieren müssen.

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

Transport-Schnittstelle

JSON-RPC ist ein Zustandsloses, asynchrones Remote-Verfahren-Ausführungsprotokoll (RDP), das JSON für einseitige Datenkommunikation verwendet. Eine JSON-RPC-Anfrage wird normalerweise durch eine ID identifiziert, und eine Antwort wird durch die gleiche ID geliefert, die in der Anfrage gesendet wurde. Nicht alle Anfragen erfordern eine Antwort, die als "Benachrichtigungen" bezeichnet werden.

Die JSON-RPC 2.0-Spezifikation erklärt im Detail, wie der Transport funktioniert. Dieser Transport ist unabhängig davon, wo er verwendet wird. Sisk implementiert dieses Protokoll über HTTP, indem es die Konformitäten von JSON-RPC über HTTP befolgt, die teilweise GET-Anfragen unterstützt, aber vollständig POST-Anfragen unterstützt. Web-Sockets werden auch unterstützt, wodurch asynchrone Nachrichtenkommunikation ermöglicht wird.

Eine JSON-RPC-Anfrage sieht ähnlich aus wie:

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

Und eine erfolgreiche Antwort sieht ähnlich aus wie:

```
{
    "jsonrpc": "2.0",
    "result": 7,
    "id": 1
}
```

JSON-RPC-Methoden

Das folgende Beispiel zeigt, wie Sie eine JSON-RPC-API mit Sisk erstellen können. Eine mathematische Operationenklasse führt die Remote-Operationen aus und liefert die serialisierte Antwort an den Client.

Program.cs **C#**

```
using var app = HttpServer.CreateBuilder(port: 5555)
    .UseJsonRPC((sender, args) =>
    {
        // fügt alle Methoden mit dem Attribut WebMethod zum JSON-RPC-Handler hinzu
        args.Handler.Methods.AddMethodsFromType(new MathOperations());

        // ordnet die Route /service zum JSON-RPC-Handler für POST- und GET-Anfragen zu
        args.Router.MapPost("/service", args.Handler.Transport.HttpPost);
        args.Router.MapGet("/service", args.Handler.Transport.HttpGet);

        // erstellt einen WebSocket-Handler für GET /ws
        args.Router.MapGet("/ws", request =>
        {
            var ws = request.GetWebSocket();
            ws.OnReceive += args.Handler.Transport.WebSocket;

            ws.WaitForClose(timeout: TimeSpan.FromSeconds(30));
            return ws.Close();
        });
    })
    .Build();

await app.StartAsync();
```

MathOperations.cs C#

```
public class MathOperations
{
    [WebMethod]
    public float Sum(float a, float b)
    {
        return a + b;
    }

    [WebMethod]
    public double Sqrt(float a)
    {
        return Math.Sqrt(a);
    }
}
```

Das obige Beispiel ordnet die Methoden `Sum` und `Sqrt` dem JSON-RPC-Handler zu und macht diese Methoden über `GET /service`, `POST /service` und `GET /ws` verfügbar. Methodennamen sind nicht case-sensitiv.

Methodeparameter werden automatisch in ihre spezifischen Typen deserialisiert. Die Verwendung von Anfragen mit benannten Parametern wird auch unterstützt. Die JSON-Serialisierung wird von der LightJson-Bibliothek durchgeführt. Wenn ein Typ nicht korrekt deserialisiert wird, können Sie einen spezifischen JSON-Konverter für diesen Typ erstellen und ihn mit Ihren JsonSerializerOptions verknüpfen.

Sie können auch das `$.params`-Objekt direkt aus der JSON-RPC-Anfrage in Ihrer Methode abrufen.

MathOperations.cs C#

```
[WebMethod]
public float Sum(JsonArray|JsonObject @params)
{
    ...
}
```

Damit dies geschieht, muss `@params` der **einzige** Parameter in Ihrer Methode sein, mit genau dem Namen `params` (in C# ist das `@` erforderlich, um diesen Parameter-Namen zu entkommen).

Die Deserialisierung von Parametern erfolgt sowohl für benannte Objekte als auch für positionale Arrays. Zum Beispiel kann die folgende Methode durch beide Anfragen aufgerufen werden:

```
[WebMethod]
public float AddUserToStore(string apiKey, User user, UserStore store)
```

```
{
    ...
}
```

Für ein Array muss die Reihenfolge der Parameter befolgt werden.

```
{
    "jsonrpc": "2.0",
    "method": "AddUserToStore",
    "params": [
        "1234567890",
        {
            "name": "John Doe",
            "email": "john@example.com"
        },
        {
            "name": "My Store"
        }
    ],
    "id": 1

}
```

Anpassen des Serialisierungsprogramms

Sie können den JSON-Serialisierer in der JsonRpcHandler.JsonSerializerOptions-Eigenschaft anpassen. In dieser Eigenschaft können Sie die Verwendung von JSON5 für die Deserialisierung von Nachrichten aktivieren. Obwohl dies nicht konform mit JSON-RPC 2.0 ist, ist JSON5 eine Erweiterung von JSON, die ein menschenlesbareres und lesbareres Schreiben ermöglicht.

Program.cs C#

```
using var host = HttpServer.CreateBuilder ( 5556 )
    .UseJsonRPC ( ( o, e ) => {

        // verwendet einen gereinigten Namen-Vergleicher. Dieser Vergleicher vergleicht
nur Buchstaben
        // und Ziffern in einem Namen und ignoriert andere Symbole. Zum Beispiel:
        // foo_bar10 == FooBar10
```

```
        e.Handler.JsonSerializerOptions.PropertyNameComparer = new
JsonSanitizedComparer ();

        // aktiviert JSON5 für den JSON-Interpreter. Selbst wenn dies aktiviert ist, ist
Plain-JSON immer noch erlaubt
        e.Handler.JsonSerializerOptions.SerializationFlags =
LightJson.Serialization.JsonSerializationFlags.Json5;

        // ordnet die POST /service-Route zum JSON-RPC-Handler zu
        e.Router.MapPost ( "/service", e.Handler.Transport.HttpPost );
    } )
    .Build ();

host.Start ();
```

SSL-Proxy

> **WARNING**
>
> Diese Funktion ist experimentell und sollte nicht in der Produktion verwendet werden. Bitte beachten Sie dieses Dokument, wenn Sie Sisk mit SSL verwenden möchten.

Der Sisk SSL-Proxy ist ein Modul, das eine HTTPS-Verbindung für einen ListeningHost in Sisk bereitstellt und HTTPS-Nachrichten an einen unsicheren HTTP-Kontext weiterleitet. Das Modul wurde entwickelt, um eine SSL-Verbindung für einen Dienst bereitzustellen, der HttpListener verwendet, um zu laufen, was keine SSL-Unterstützung bietet.

Der Proxy läuft innerhalb der gleichen Anwendung und hört auf HTTP/1.1-Nachrichten, die im gleichen Protokoll an Sisk weitergeleitet werden. Derzeit ist diese Funktion sehr experimentell und möglicherweise instabil genug, um nicht in der Produktion verwendet zu werden.

Derzeit unterstützt der SslProxy fast alle HTTP/1.1-Features, wie z.B. Keep-Alive, Chunked-Encoding, WebSockets usw. Für eine offene Verbindung zum SSL-Proxy wird eine TCP-Verbindung zum Zielserver erstellt und der Proxy wird an die etablierte Verbindung weitergeleitet.

Der SslProxy kann mit HttpServer.CreateBuilder wie folgt verwendet werden:

```
using var app = HttpServer.CreateBuilder(port: 5555)
    .UseRouter(r ⇒
    {
        r.MapGet("/", request ⇒
        {
            return new HttpResponse("Hallo, Welt!");
        });
    })
    // SSL zum Projekt hinzufügen
    .UseSsl(
        sslListeningPort: 5567,
        new X509Certificate2(@".\ssl.pfx", password: "12345")
    )
    .Build();

app.Start();
```

Sie müssen ein gültiges SSL-Zertifikat für den Proxy bereitstellen. Um sicherzustellen, dass das Zertifikat von Browsern akzeptiert wird, importieren Sie es in das Betriebssystem, damit es ordnungsgemäß funktioniert.

Basic Auth

Das Basic-Auth-Paket fügt einen Anfrage-Handler hinzu, der in der Lage ist, das Basic-Authentifizierungsschema in Ihrer Sisk-Anwendung mit sehr wenig Konfiguration und Aufwand zu handhaben. Basic-HTTP-Authentifizierung ist eine minimale Eingabeform der Authentifizierung von Anfragen durch eine Benutzer-ID und ein Passwort, wobei die Sitzung ausschließlich vom Client gesteuert wird und es keine Authentifizierungs- oder Zugriffstoken gibt.

Erfahren Sie mehr über das Basic-Authentifizierungsschema in der MDN-Spezifikation.

Installation

Um loszulegen, installieren Sie das Sisk.BasicAuth-Paket in Ihrem Projekt:

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

Sie können weitere Möglichkeiten zur Installation in Ihrem Projekt im Nuget-Repository anzeigen.

Erstellen Ihres Auth-Handlers

Sie können das Authentifizierungsschema für ein ganzes Modul oder für einzelne Routen steuern. Dazu müssen wir zunächst unseren ersten Basic-Authentifizierungs-Handler schreiben.

Im folgenden Beispiel wird eine Verbindung zur Datenbank hergestellt, es wird überprüft, ob der Benutzer existiert und ob das Passwort gültig ist, und anschließend wird der Benutzer im Kontext-Beutel gespeichert.

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

```
        Realm = "Um diese Seite zu betreten, geben Sie bitte Ihre
Anmeldeinformationen ein.";
    }

    public override HttpResponse? OnValidating(BasicAuthenticationCredentials credentials,
HttpContext context)
    {
        DbContext db = new DbContext();

        // In diesem Fall verwenden wir die E-Mail-Adresse als Benutzer-ID-Feld, also
suchen wir nach einem Benutzer mit seiner E-Mail-Adresse.
        User? user = db.Users.FirstOrDefault(u => u.Email == credentials.UserId);
        if (user == null)
        {
            return base.CreateUnauthorizedResponse("Entschuldigung! Kein Benutzer mit
dieser E-Mail-Adresse gefunden.");
        }

        // Überprüft, ob das Passwort für diesen Benutzer gültig ist.
        if (!user.ValidatePassword(credentials.Password))
        {
            return base.CreateUnauthorizedResponse("Ungültige Anmeldeinformationen.");
        }

        // Fügt den angemeldeten Benutzer zum HTTP-Kontext hinzu
        // und setzt die Ausführung fort
        context.Bag.Add("loggedUser", user);
        return null;
    }
}
```

Assoziieren Sie einfach diesen Anfrage-Handler mit unserer Route oder Klasse.

```
public class UsersController
{
    [RouteGet("/")]
    [RequestHandler(typeof(UserAuthHandler))]
    public string Index(HttpRequest request)
    {
        User loggedUser = (User)request.Context.RequestBag["loggedUser"];
        return "Hallo, " + loggedUser.Name + "!";
    }
}
```

Oder mit der RouterModule-Klasse:

```
public class UsersController : RouterModule
{
    public ClientModule()
    {
        // Jetzt werden alle Routen in dieser Klasse vom UserAuthHandler gehandhabt.
        base.HasRequestHandler(new UserAuthHandler());
    }

    [RouteGet("/")]
    public string Index(HttpRequest request)
    {
        User loggedUser = (User)request.Context.RequestBag["loggedUser"];
        return "Hallo, " + loggedUser.Name + "!";
    }
}
```

Hinweise

Die primäre Verantwortung für die Basic-Authentifizierung liegt auf der Client-Seite. Speicherung, Cache-Steuerung und Verschlüsselung werden alle lokal auf dem Client gehandhabt. Der Server erhält nur die Anmeldeinformationen und überprüft, ob der Zugriff erlaubt ist oder nicht.

Beachten Sie, dass diese Methode nicht eine der sichersten ist, da sie eine erhebliche Verantwortung auf den Client legt, der schwierig zu verfolgen und die Sicherheit seiner Anmeldeinformationen zu gewährleisten ist. Darüber hinaus ist es wichtig, dass Passwörter in einem sicheren Verbindungskontext (SSL) übertragen werden, da sie keine inhärente Verschlüsselung haben. Eine kurze Abfangung der Header einer Anfrage kann die Zugangsdaten Ihres Benutzers offenlegen.

Wählen Sie für Produktionsanwendungen robustere Authentifizierungslösungen und vermeiden Sie die Verwendung von zu vielen vorgefertigten Komponenten, da diese möglicherweise nicht an die Bedürfnisse Ihres Projekts angepasst sind und es letztendlich Sicherheitsrisiken aussetzen.

Diensteanbieter

Diensteanbieter sind eine Möglichkeit, Ihre Sisk-Anwendung auf verschiedene Umgebungen mit einer portablen Konfigurationsdatei zu übertragen. Diese Funktion ermöglicht es Ihnen, den Serverport, Parameter und andere Optionen ohne Änderung des Anwendungscode für jede Umgebung zu ändern. Dieses Modul hängt von der Sisk-Konstruktionsyntax ab und kann über die Methode `UsePortableConfiguration` konfiguriert werden.

Ein Konfigurationsanbieter wird mit `IConfigurationProvider` implementiert, der einen Konfigurationsleser bereitstellt und jede Implementierung erhalten kann. Standardmäßig bietet Sisk einen JSON-Konfigurationsleser, aber es gibt auch ein Paket für INI-Dateien. Sie können auch Ihren eigenen Konfigurationsanbieter erstellen und ihn mit:

```
using var app = HttpServer.CreateBuilder()
    .UsePortableConfiguration(config =>
    {
        config.WithConfigReader<MyConfigurationReader>();
    })
    .Build();
```

Wie bereits erwähnt, ist der Standardanbieter eine JSON-Datei. Standardmäßig wird nach einer Datei mit dem Namen `service-config.json` gesucht, und diese wird im aktuellen Verzeichnis des laufenden Prozesses und nicht im Verzeichnis der ausführbaren Datei gesucht.

Sie können den Dateinamen sowie das Verzeichnis, in dem Sisk nach der Konfigurationsdatei suchen soll, mit:

```
using Sisk.Core.Http;
using Sisk.Core.Http.Hosting;

using var app = HttpServer.CreateBuilder()
    .UsePortableConfiguration(config =>
    {
        config.WithConfigFile("config.toml",
            createIfDontExists: true,
            lookupDirectories:
                ConfigurationFileLookupDirectory.CurrentDirectory |
                ConfigurationFileLookupDirectory.AppDirectory);
    })
    .Build();
```

Der obige Code sucht nach der Datei `config.toml` im aktuellen Verzeichnis des laufenden Prozesses. Wenn sie nicht gefunden wird, sucht er dann im Verzeichnis, in dem die ausführbare Datei liegt. Wenn die Datei nicht existiert, wird der Parameter `createIfDontExists` beachtet, und die Datei wird ohne Inhalt im letzten getesteten Pfad (basierend auf `lookupDirectories`) erstellt, und ein Fehler wird in der Konsole ausgegeben, was die Anwendung verhindert, sich zu initialisieren.

> **TIP**
>
> Sie können den Quellcode des INI-Konfigurationslesers und des JSON-Konfigurationslesers betrachten, um zu verstehen, wie ein `IConfigurationProvider` implementiert wird.

Lesen von Konfigurationen aus einer JSON-Datei

Standardmäßig bietet Sisk einen Konfigurationsanbieter, der Konfigurationen aus einer JSON-Datei liest. Diese Datei folgt einer festen Struktur und besteht aus den folgenden Parametern:

```
{
    "Server": {
        "DefaultEncoding": "UTF-8",
        "ThrowExceptions": true,
        "IncludeRequestIdHeader": true
    },
    "ListeningHost": {
        "Label": "Meine Sisk-Anwendung",
        "Ports": [
            "http://localhost:80/",
            "https://localhost:443/",  // Konfigurationsdateien unterstützen
auch Kommentare
        ],
        "CrossOriginResourceSharingPolicy": {
            "AllowOrigin": "*",
            "AllowOrigins": [ "*" ],   // neu in 0.14
            "AllowMethods": [ "*" ],
            "AllowHeaders": [ "*" ],
            "MaxAge": 3600
        },
        "Parameters": {
            "MySqlConnection": "server=localhost;user=root;"
        }

```

```
        }
    }
```

Die aus einer Konfigurationsdatei erstellten Parameter können im Serverkonstruktor abgerufen werden:

```
using var app = HttpServer.CreateBuilder()
    .UsePortableConfiguration(config ⇒
    {
        config.WithParameters(paramCollection ⇒
        {
            string databaseConnection = paramCollection.GetValueOrThrow("MySqlConnection");
        });
    })
    .Build();
```

Jeder Konfigurationsleser bietet eine Möglichkeit, die Serverinitialisierungsparameter zu lesen. Einige Eigenschaften sind so konzipiert, dass sie in der Prozessumgebung anstelle der Konfigurationsdatei definiert werden, wie z. B. sensible API-Daten, API-Schlüssel usw.

Konfigurationsdateistruktur

Die JSON-Konfigurationsdatei besteht aus den folgenden Eigenschaften:

Eigenschaft	Pflichtfeld	Beschreibung
Server	Erforderlich	Stellt den Server selbst mit s Einstellungen dar.
Server.AccessLogsStream	Optional	Standardmäßig `console` . Gib Ausgabestream für die Zugriffsprotokolle an. Kann e Dateiname, `null` oder `cons` sein.
Server.ErrorsLogsStream	Optional	Standardmäßig `null` . Gibt d Ausgabestream für die Fehlerprotokolle an. Kann ein Dateiname, `null` oder `cons` sein.

Eigenschaft	Pflichtfeld	Beschreibung
Server.MaximumContentLength	Optional	
Server.MaximumContentLength	Optional	Standardmäßig `0`. Gibt die maximale Inhaltslänge in Byt Null bedeutet unendlich.
Server.IncludeRequestIdHeader	Optional	Standardmäßig `false`. Gibt der HTTP-Server den `X-Requ Id` -Header senden soll.
Server.ThrowExceptions	Optional	Standardmäßig `true`. Gibt a unbehandelte Ausnahmen ausgelöst werden sollen. Auf setzen, wenn die Anwendung Produktion ist, und auf `true` die Anwendung debuggt wird
ListeningHost	Erforderlich	Stellt den Server-Host dar, d zugehört.
ListeningHost.Label	Optional	Stellt die Anwendungsbezeic dar.
ListeningHost.Ports	Erforderlich	Stellt ein Array von Zeichenfo dar, die der Syntax von ListeningPort entsprechen.
ListeningHost.CrossOriginResourceSharingPolicy	Optional	Konfiguriert die CORS-Heade die Anwendung.
ListeningHost.CrossOriginResourceSharingPolicy.AllowCredentials	Optional	Standardmäßig `false`. Gibt `Allow-Credentials` -Header
ListeningHost.CrossOriginResourceSharingPolicy.ExposeHeaders	Optional	Standardmäßig `null`. Erwar Array von Zeichenfolgen. Gibt `Expose-Headers` -Header an.
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigin	Optional	Standardmäßig `null`. Erwar Zeichenfolge. Gibt den `Allow Origin` -Header an.
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigins	Optional	Standardmäßig `null`. Erwar Array von Zeichenfolgen. Gibt mehrere `Allow-Origin` -Hea

Eigenschaft	Pflichtfeld	Beschreibung
		Siehe AllowOrigins für weite Informationen.
ListeningHost.CrossOriginResourceSharingPolicy.AllowMethods	Optional	Standardmäßig `null` . Erwar Array von Zeichenfolgen. Gibt `Allow-Methods` -Header an.
ListeningHost.CrossOriginResourceSharingPolicy.AllowHeaders	Optional	Standardmäßig `null` . Erwar Array von Zeichenfolgen. Gibt `Allow-Headers` -Header an.
ListeningHost.CrossOriginResourceSharingPolicy.MaxAge	Optional	Standardmäßig `null` . Erwar Ganzzahl. Gibt den `Max-Age` Header in Sekunden an.
ListeningHost.Parameters	Optional	Gibt die Eigenschaften an, di Anwendungskonfigurationsm bereitgestellt werden.

INI-Konfiguration

Sisk hat eine Methode für das Abrufen von Startkonfigurationen, die nicht JSON sind. Tatsächlich kann jede Pipeline, die IConfigurationReader implementiert, mit PortableConfigurationBuilder.WithConfigurationPipeline verwendet werden, um die Serverkonfiguration aus jeder Dateityp zu lesen.

Das Sisk.IniConfiguration-Paket bietet einen streambasierten INI-Dateileser, der keine Ausnahmen für häufige Syntaxfehler auslöst und eine einfache Konfigurationssyntax hat. Dieses Paket kann außerhalb des Sisk-Frameworks verwendet werden und bietet Flexibilität für Projekte, die einen effizienten INI-Dokumentleser benötigen.

Installation

Um das Paket zu installieren, können Sie mit folgendem Befehl beginnen:

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

Sie können auch das Core-Paket installieren, das weder den INI-IConfigurationReader noch die Sisk-Abhängigkeit enthält, sondern nur die INI-Serialisierer:

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

Mit dem Hauptpaket können Sie es in Ihrem Code wie im folgenden Beispiel verwenden:

```
1   class Program
2   {
3       static HttpServerHostContext 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
12                  // verwendet den IniConfigurationReader-Konfigurationsleser
13                  config.WithConfigurationPipeline<IniConfigurationReader>();
14              })
15              .UseRouter(r =>
16              {
17                  r.MapGet("/", SayHello);
18              })
19              .Build();
20
21          Host.Start();
22      }
23
24      static HttpResponse SayHello(HttpRequest request)
25      {
26          string? name = Host.Parameters["name"] ?? "world";
27          return new HttpResponse($"Hallo, {name}!");
28      }
29  }
```

Der obige Code sucht nach einer app.ini-Datei im aktuellen Verzeichnis des Prozesses (CurrentDirectory). Die INI-Datei sieht wie folgt aus:

```
1   [Server]
2   # Mehrere Zuhöradressen werden unterstützt
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"
```

INI-Geschmack und Syntax

Aktuelle Implementierung des Geschmacks:

- Eigenschaften- und Sektionsnamen sind **groß-/kleinschreibungsunabhängig**.
- Eigenschaftsnamen und Werte sind **gekürzt**, sofern Werte nicht in Anführungszeichen gesetzt sind.
- Werte können mit einfachen oder doppelten Anführungszeichen in Anführungszeichen gesetzt werden. Anführungszeichen können Zeilenumbrüche enthalten.
- Kommentare werden mit `#` und `;` unterstützt. **Nachgestellte Kommentare sind ebenfalls erlaubt**.
- Eigenschaften können mehrere Werte haben.

Im Detail finden Sie die Dokumentation für den "Geschmack" des INI-Parsers, der in Sisk verwendet wird, in diesem Dokument.

Verwenden Sie beispielsweise den folgenden INI-Code:

```
One = 1
Value = dies ist ein Wert
Another value = "dieser Wert
    hat einen Zeilenumbruch darin"

; der Code unten hat einige Farben
[some section]
Color = Red
Color = Blue
Color = Yellow ; verwenden Sie nicht gelb
```

Analysieren Sie ihn mit:

```
// analysieren Sie den INI-Text aus der Zeichenfolge
IniDocument doc = IniDocument.FromString(iniText);

// erhalten Sie einen Wert
string? one = doc.Global.GetOne("one");
string? anotherValue = doc.Global.GetOne("another value");

// erhalten Sie mehrere Werte
string[]? colors = doc.GetSection("some section")?.GetMany("color");
```

Konfigurationsparameter

Sektion und Name	Mehrere Werte zulassen	Beschreibung
`Server.Listen`	Ja	Die Zuhöradressen/Ports des Servers.
`Server.Encoding`	Nein	Die Standardcodierung des Servers.
`Server.MaximumContentLength`	Nein	Die maximale Inhaltslänge des Servers in Bytes.
`Server.IncludeRequestIdHeader`	Nein	Gibt an, ob der HTTP-Server den X-Request-Id-Header senden soll.
`Server.ThrowExceptions`	Nein	Gibt an, ob unbehandelte Ausnahmen ausgelöst werden sollen.
`Server.AccessLogsStream`	Nein	Gibt den Ausgabestream für den Zugriff auf die Protokolle an.
`Server.ErrorsLogsStream`	Nein	Gibt den Ausgabestream für die Fehlerprotokolle an.
`Cors.AllowMethods`	Nein	Gibt den Wert des CORS-Allow-Methods-Headers an.
`Cors.AllowHeaders`	Nein	Gibt den Wert des CORS-Allow-Headers-Headers an.
`Cors.AllowOrigins`	Nein	Gibt mehrere Allow-Origin-Header, getrennt durch Kommata, an. AllowOrigins für weitere Informationen.
`Cors.AllowOrigin`	Nein	Gibt einen Allow-Origin-Header an.
`Cors.ExposeHeaders`	Nein	Gibt den Wert des CORS-Expose-Headers-Headers an.
`Cors.AllowCredentials`	Nein	Gibt den Wert des CORS-Allow-Credentials-Headers an.
`Cors.MaxAge`	Nein	Gibt den Wert des CORS-Max-Age-Headers an.

Manuel (erweitert) Setup

In diesem Abschnitt erstellen wir unseren HTTP-Server ohne vordefinierte Standards, auf eine völlig abstrakte Weise. Hier können Sie manuell aufbauen, wie Ihr HTTP-Server funktionieren wird. Jeder ListeningHost hat einen Router und ein HTTP-Server kann mehrere ListeningHosts haben, die jeweils auf einen anderen Host auf einem anderen Port zeigen.

Zunächst müssen wir das Konzept von Anfrage/Antwort verstehen. Es ist ziemlich einfach: für jede Anfrage muss es eine Antwort geben. Sisk folgt diesem Prinzip auch. Lassen Sie uns eine Methode erstellen, die mit einer "Hallo, Welt!"-Nachricht in HTML antwortet, wobei der Statuscode und die Header angegeben werden.

```
// Program.cs
using Sisk.Core.Http;
using Sisk.Core.Routing;

static HttpResponse IndexPage(HttpRequest request)
{
    HttpResponse indexResponse = new HttpResponse
    {
        Status = System.Net.HttpStatusCode.OK,
        Content = new HtmlContent(@"
            <html>
                <body>
                    <h1>Hallo, Welt!</h1>
                </body>
            </html>
        ")
    };

    return indexResponse;
}
```

Der nächste Schritt ist, diese Methode mit einer HTTP-Route zu verknüpfen.

Router

Router sind Abstraktionen von Anfrage-Routen und dienen als Brücke zwischen Anfragen und Antworten für den Dienst. Router verwalten Dienst-Routen, Funktionen und Fehler.

Ein Router kann mehrere Routen haben und jede Route kann unterschiedliche Operationen auf diesem Pfad ausführen, wie z.B. die Ausführung einer Funktion, das Servieren einer Seite oder die Bereitstellung einer Ressource vom Server.

Lassen Sie uns unseren ersten Router erstellen und unsere `IndexPage` -Methode mit dem Index-Pfad verknüpfen.

```
Router mainRouter = new Router();

// SetRoute wird alle Index-Routen mit unserer Methode verknüpfen.
mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage);
```

Jetzt kann unser Router Anfragen empfangen und Antworten senden. Allerdings ist `mainRouter` nicht an einen Host oder einen Server gebunden, daher wird er nicht alleine funktionieren. Der nächste Schritt ist, unseren ListeningHost zu erstellen.

Listening Hosts und Ports

Ein ListeningHost kann einen Router und mehrere Listening-Ports für denselben Router hosten. Ein ListeningPort ist ein Präfix, an dem der HTTP-Server zuhört.

Hier können wir einen `ListeningHost` erstellen, der auf zwei Endpunkte für unseren Router zeigt:

```
ListeningHost myHost = new ListeningHost
{
    Router = new Router(),
    Ports = new ListeningPort[]
    {
        new ListeningPort("http://localhost:5000/")
    }
};
```

Jetzt wird unser HTTP-Server auf den angegebenen Endpunkten zuhören und seine Anfragen an unseren Router weiterleiten.

Server-Konfiguration

Die Server-Konfiguration ist für das meiste Verhalten des HTTP-Servers selbst verantwortlich. In dieser Konfiguration können wir `ListeningHosts` mit unserem Server verknüpfen.

```
1    HttpServerConfiguration config = new HttpServerConfiguration();
2    config.ListeningHosts.Add(myHost); // Fügen Sie unseren ListeningHost zu dieser Server-
     Konfiguration hinzu
```

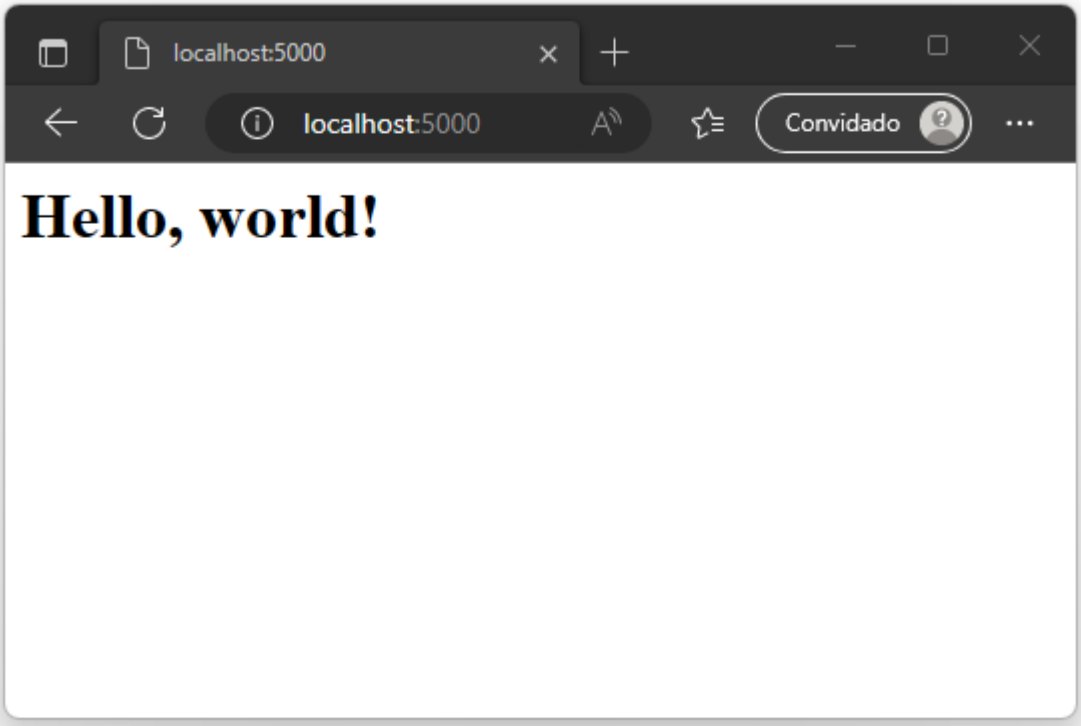
Als Nächstes können wir unseren HTTP-Server erstellen:

```
1    HttpServer server = new HttpServer(config);
2    server.Start();    // Startet den Server
3    Console.ReadKey(); // Verhindert, dass die Anwendung beendet wird
```

Jetzt können wir unsere ausführbare Datei kompilieren und unseren HTTP-Server mit dem Befehl starten:

```
dotnet watch
```

Bei der Laufzeit öffnen Sie Ihren Browser und navigieren zum Server-Pfad, und Sie sollten sehen:


localhost:5000
localhost:5000
Convidado
Hello, world!

Anfragelebenszyklus

Im Folgenden wird der gesamte Lebenszyklus einer Anfrage anhand eines Beispiels einer HTTP-Anfrage erläutert.

- **Empfangen der Anfrage:** Jede Anfrage erstellt einen HTTP-Kontext zwischen der Anfrage selbst und der Antwort, die an den Client geliefert wird. Dieser Kontext stammt vom integrierten Listener in Sisk, der entweder HttpListener, Kestrel oder Cadente sein kann.
  - Externe Anfragevalidierung: Die Validierung von HttpServerConfiguration.RemoteRequestsAction wird für die Anfrage überprüft.
    - Wenn die Anfrage extern ist und die Eigenschaft `Drop` ist, wird die Verbindung ohne Antwort an den Client geschlossen und ein `HttpServerExecutionStatus = RemoteRequestDropped` zurückgegeben.
  - Weiterleitungs-Resolver-Konfiguration: Wenn ein ForwardingResolver konfiguriert ist, wird die Methode OnResolveRequestHost auf dem ursprünglichen Host der Anfrage aufgerufen.
  - DNS-Matching: Mit dem aufgelösten Host und mehr als einem konfigurierten ListeningHost sucht der Server nach dem entsprechenden Host für die Anfrage.
    - Wenn kein ListeningHost übereinstimmt, wird eine 400 Bad Request-Antwort an den Client zurückgegeben und ein `HttpServerExecutionStatus = DnsUnknownHost` -Status an den HTTP-Kontext zurückgegeben.
    - Wenn ein ListeningHost übereinstimmt, aber sein Router noch nicht initialisiert ist, wird eine 503 Service Unavailable-Antwort an den Client zurückgegeben und ein `HttpServerExecutionStatus = ListeningHostNotReady` -Status an den HTTP-Kontext zurückgegeben.
  - Router-Bindung: Der Router des entsprechenden ListeningHost wird mit dem empfangenen HTTP-Server verknüpft.
    - Wenn der Router bereits mit einem anderen HTTP-Server verknüpft ist, was nicht zulässig ist, da der Router die Konfigurationsressourcen des Servers aktiv verwendet, wird eine `InvalidOperationException` ausgelöst. Dies tritt nur während der Initialisierung des HTTP-Servers auf, nicht während der Erstellung des HTTP-Kontexts.
  - Vordefinition von Headern:
    - Definiert den `X-Request-Id` -Header in der Antwort, wenn dies konfiguriert ist.
    - Definiert den `X-Powered-By` -Header in der Antwort, wenn dies konfiguriert ist.
  - Inhaltsgrößenvalidierung: Überprüft, ob der Anfrageinhalt kleiner als HttpServerConfiguration.MaximumContentLength ist, nur wenn dieser größer als Null ist.
    - Wenn die Anfrage eine `Content-Length` größer als die konfigurierte sendet, wird eine 413 Payload Too Large-Antwort an den Client zurückgegeben und ein `HttpServerExecutionStatus = ContentTooLarge` -Status an den HTTP-Kontext zurückgegeben.

    - Das `OnHttpRequestOpen` -Ereignis wird für alle konfigurierten HTTP-Server-Handler aufgerufen.
- **Weiterleiten der Aktion:** Der Server ruft den Router für die empfangene Anfrage auf.
    - Wenn der Router keine Route findet, die der Anfrage entspricht:
        - Wenn die Router.NotFoundErrorHandler-Eigenschaft konfiguriert ist, wird die Aktion aufgerufen und die Antwort der Aktion an den HTTP-Client weitergeleitet.
        - Wenn die vorherige Eigenschaft Null ist, wird eine Standard-404 Not Found-Antwort an den Client zurückgegeben.
    - Wenn der Router eine passende Route findet, aber die Methode der Route nicht der Anfragemethode entspricht:
        - Wenn die Router.MethodNotAllowedErrorHandler-Eigenschaft konfiguriert ist, wird die Aktion aufgerufen und die Antwort der Aktion an den HTTP-Client weitergeleitet.
        - Wenn die vorherige Eigenschaft Null ist, wird eine Standard-405 Method Not Allowed-Antwort an den Client zurückgegeben.
    - Wenn die Anfrage die `OPTIONS` -Methode ist:
        - Der Router gibt eine 200 Ok-Antwort an den Client zurück, nur wenn keine Route der Anfrage entspricht (die Route ist nicht explizit RouteMethod.Options).
    - Wenn die HttpServerConfiguration.ForceTrailingSlash-Eigenschaft aktiviert ist, die passende Route keine Regex ist, der Anfragepfad nicht mit `/` endet und die Anfragemethode `GET` ist:
        - Eine 307 Temporary Redirect-HTTP-Antwort mit dem `Location` -Header mit dem Pfad und der Abfrage an dieselbe Position mit einem `/` am Ende wird an den Client zurückgegeben.
    - Das `OnContextBagCreated` -Ereignis wird für alle konfigurierten HTTP-Server-Handler aufgerufen.
    - Alle globalen IRequestHandler-Instanzen mit der `BeforeResponse` -Flag werden ausgeführt.
        - Wenn ein Handler eine nicht-Null-Antwort zurückgibt, wird diese Antwort an den HTTP-Client weitergeleitet und der Kontext geschlossen.
        - Wenn ein Fehler in diesem Schritt auftritt und HttpServerConfiguration.ThrowExceptions deaktiviert ist:
            - Wenn die Router.CallbackErrorHandler-Eigenschaft aktiviert ist, wird sie aufgerufen und die resultierende Antwort an den Client zurückgegeben.
            - Wenn die vorherige Eigenschaft nicht definiert ist, wird eine leere Antwort an den Server zurückgegeben, der eine Antwort entsprechend dem Typ des ausgelösten Fehlers zurückgibt, der normalerweise 500 Internal Server Error ist.
    - Alle IRequestHandler-Instanzen, die in der Route definiert sind und die `BeforeResponse` -Flag haben, werden ausgeführt.
        - Wenn ein Handler eine nicht-Null-Antwort zurückgibt, wird diese Antwort an den HTTP-Client weitergeleitet und der Kontext geschlossen.
        - Wenn ein Fehler in diesem Schritt auftritt und HttpServerConfiguration.ThrowExceptions deaktiviert ist:
            - Wenn die Router.CallbackErrorHandler-Eigenschaft aktiviert ist, wird sie aufgerufen und die resultierende Antwort an den Client zurückgegeben.

        - Wenn die vorherige Eigenschaft nicht definiert ist, wird eine leere Antwort an den Server zurückgegeben, der eine Antwort entsprechend dem Typ des ausgelösten Fehlers zurückgibt, der normalerweise 500 Internal Server Error ist.
    - Die Aktion des Routers wird aufgerufen und in eine HTTP-Antwort umgewandelt.
        - Wenn ein Fehler in diesem Schritt auftritt und HttpServerConfiguration.ThrowExceptions deaktiviert ist:
            - Wenn die Router.CallbackErrorHandler-Eigenschaft aktiviert ist, wird sie aufgerufen und die resultierende Antwort an den Client zurückgegeben.
            - Wenn die vorherige Eigenschaft nicht definiert ist, wird eine leere Antwort an den Server zurückgegeben, der eine Antwort entsprechend dem Typ des ausgelösten Fehlers zurückgibt, der normalerweise 500 Internal Server Error ist.
    - Alle globalen IRequestHandler-Instanzen mit der `AfterResponse` -Flag werden ausgeführt.
        - Wenn ein Handler eine nicht-Null-Antwort zurückgibt, ersetzt die Antwort des Handlers die vorherige Antwort und wird sofort an den HTTP-Client weitergeleitet.
        - Wenn ein Fehler in diesem Schritt auftritt und HttpServerConfiguration.ThrowExceptions deaktiviert ist:
            - Wenn die Router.CallbackErrorHandler-Eigenschaft aktiviert ist, wird sie aufgerufen und die resultierende Antwort an den Client zurückgegeben.
            - Wenn die vorherige Eigenschaft nicht definiert ist, wird eine leere Antwort an den Server zurückgegeben, der eine Antwort entsprechend dem Typ des ausgelösten Fehlers zurückgibt, der normalerweise 500 Internal Server Error ist.
    - Alle IRequestHandler-Instanzen, die in der Route definiert sind und die `AfterResponse` -Flag haben, werden ausgeführt.
        - Wenn ein Handler eine nicht-Null-Antwort zurückgibt, ersetzt die Antwort des Handlers die vorherige Antwort und wird sofort an den HTTP-Client weitergeleitet.
        - Wenn ein Fehler in diesem Schritt auftritt und HttpServerConfiguration.ThrowExceptions deaktiviert ist:
            - Wenn die Router.CallbackErrorHandler-Eigenschaft aktiviert ist, wird sie aufgerufen und die resultierende Antwort an den Client zurückgegeben.
            - Wenn die vorherige Eigenschaft nicht definiert ist, wird eine leere Antwort an den Server zurückgegeben, der eine Antwort entsprechend dem Typ des ausgelösten Fehlers zurückgibt, der normalerweise 500 Internal Server Error ist.
- **Verarbeiten der Antwort:** Mit der Antwort bereit, bereitet der Server sie für den Versand an den Client vor.
    - Die Cross-Origin Resource Sharing Policy (CORS)-Header werden in der Antwort definiert, entsprechend der Konfiguration in der aktuellen ListeningHost.CrossOriginResourceSharingPolicy.
    - Der Statuscode und die Header der Antwort werden an den Client gesendet.
    - Der Inhalt der Antwort wird an den Client gesendet:
        - Wenn der Inhalt der Antwort ein Nachfahre von ByteArrayContent ist, werden die Antwortbytes direkt in den Ausgabestream der Antwort kopiert.

        - Wenn die vorherige Bedingung nicht erfüllt ist, wird die Antwort in einen Stream serialisiert und in den Ausgabestream der Antwort kopiert.
    - Die Streams werden geschlossen und der Inhalt der Antwort wird verworfen.
    - Wenn HttpServerConfiguration.DisposeDisposableContextValues aktiviert ist, werden alle Objekte, die im Anfragekontext definiert sind und von IDisposable erben, verworfen.
    - Das `OnHttpRequestClose` -Ereignis wird für alle konfigurierten HTTP-Server-Handler aufgerufen.
    - Wenn auf dem Server eine Ausnahme ausgelöst wurde, wird das `OnException` -Ereignis für alle konfigurierten HTTP-Server-Handler aufgerufen.
    - Wenn die Route Zugriffsprotokollierung zulässt und HttpServerConfiguration.AccessLogsStream nicht Null ist, wird eine Protokollzeile in die Protokollausgabe geschrieben.
    - Wenn die Route Fehlerprotokollierung zulässt, eine Ausnahme vorliegt und HttpServerConfiguration.ErrorsLogsStream nicht Null ist, wird eine Protokollzeile in die Fehlerprotokollausgabe geschrieben.
    - Wenn der Server auf eine Anfrage durch HttpServer.WaitNext wartet, wird die Sperre freigegeben und der Kontext wird dem Benutzer zur Verfügung gestellt.

Weiterleitungsauflöser

Ein Weiterleitungsauflöser ist ein Helfer, der hilft, Informationen zu decodieren, die den Client durch eine Anfrage, Proxy, CDN oder Lastenausgleich identifizieren. Wenn Ihr Sisk-Dienst über einen Reverse- oder Forward-Proxy läuft, kann die IP-Adresse, der Host und das Protokoll des Clients von der ursprünglichen Anfrage abweichen, da es sich um eine Weiterleitung von einem Dienst zum anderen handelt. Diese Sisk-Funktion ermöglicht es Ihnen, diese Informationen zu kontrollieren und aufzulösen, bevor Sie mit der Anfrage arbeiten. Diese Proxys liefern normalerweise nützliche Header, um den Client zu identifizieren.

Derzeit ist es mit der ForwardingResolver-Klasse möglich, die IP-Adresse, den Host und das HTTP-Protokoll des Clients aufzulösen. Nach Version 1.0 von Sisk hat der Server keine Standardimplementierung mehr, um diese Header aus Sicherheitsgründen, die von Dienst zu Dienst variieren, zu decodieren.

Zum Beispiel enthält der `X-Forwarded-For` -Header Informationen über die IP-Adressen, die die Anfrage weitergeleitet haben. Dieser Header wird von Proxys verwendet, um eine Kette von Informationen an den Enddienst zu übermitteln und enthält die IP-Adresse aller verwendeten Proxys, einschließlich der tatsächlichen Adresse des Clients. Das Problem ist: manchmal ist es schwierig, die IP-Adresse des Clients zu identifizieren, und es gibt keine spezifische Regel, um diesen Header zu identifizieren. Es wird dringend empfohlen, die Dokumentation für die Header zu lesen, die Sie unten implementieren werden:

- Lesen Sie über den `X-Forwarded-For` -Header hier.
- Lesen Sie über den `X-Forwarded-Host` -Header hier.
- Lesen Sie über den `X-Forwarded-Proto` -Header hier.

Die ForwardingResolver-Klasse

Diese Klasse hat drei virtuelle Methoden, die die am besten geeignete Implementierung für jeden Dienst ermöglichen. Jede Methode ist für die Auflösung von Informationen aus der Anfrage über einen Proxy verantwortlich: die IP-Adresse des Clients, den Host der Anfrage und das Sicherheitsprotokoll, das verwendet wird. Standardmäßig verwendet Sisk immer die Informationen aus der ursprünglichen Anfrage, ohne Header aufzulösen.

Das folgende Beispiel zeigt, wie diese Implementierung verwendet werden kann. Dieses Beispiel löst die IP-Adresse des Clients über den `X-Forwarded-For` -Header auf und wirft einen Fehler, wenn mehr als eine IP-

Adresse in der Anfrage weitergeleitet wurde.

> **IMPORTANT**
>
> Verwenden Sie dieses Beispiel nicht in Produktionscode. Überprüfen Sie immer, ob die Implementierung für die Verwendung geeignet ist. Lesen Sie die Header-Dokumentation, bevor Sie sie implementieren.

```
class Program
{
    static void Main(string[] args)
    {
        using var host = HttpServer.CreateBuilder()
            .UseForwardingResolver<Resolver>()
            .UseListeningPort(5555)
            .Build();

        host.Router.SetRoute(RouteMethod.Any, Route.AnyPath, request =>
            new HttpResponse("Hallo, Welt!!!"));

        host.Start();
    }

    class Resolver : ForwardingResolver
    {
        public override IPAddress OnResolveClientAddress(HttpRequest request,
IPEndPoint connectingEndpoint)
        {
            string? forwardedFor = request.Headers.XForwardedFor;
            if (forwardedFor is null)
            {
                throw new Exception("Der X-Forwarded-For-Header fehlt.");
            }
            string[] ipAddresses = forwardedFor.Split(',');
            if (ipAddresses.Length != 1)
            {
                throw new Exception("Zu viele Adressen im X-Forwarded-For-Header.");
            }

            return IPAddress.Parse(ipAddresses[0]);
        }
    }
}
```

Http-Server-Handler

In Sisk-Version 0.16 haben wir die `HttpServerHandler` -Klasse eingeführt, die darauf abzielt, das Gesamtverhalten von Sisk zu erweitern und zusätzliche Ereignishandler für Sisk bereitzustellen, wie z. B. das Handling von Http-Anfragen, Routern, Kontextbeuteln und mehr.

Die Klasse konzentriert sich auf Ereignisse, die während der Lebensdauer des gesamten HTTP-Servers und auch einer Anfrage auftreten. Das Http-Protokoll hat keine Sitzungen, und daher ist es nicht möglich, Informationen von einer Anfrage zur nächsten zu erhalten. Sisk bietet derzeit eine Möglichkeit, Sitzungen, Kontexte, Datenbankverbindungen und andere nützliche Anbieter zu implementieren, um Ihre Arbeit zu unterstützen.

Bitte besuchen Sie diese Seite, um zu lesen, wo jedes Ereignis ausgelöst wird und welchen Zweck es hat. Sie können auch den Lebenszyklus einer HTTP-Anfrage anzeigen, um zu verstehen, was mit einer Anfrage passiert und wo Ereignisse ausgelöst werden. Der HTTP-Server ermöglicht es Ihnen, mehrere Handler gleichzeitig zu verwenden. Jeder Ereignisanruf ist synchron, d. h. er blockiert den aktuellen Thread für jede Anfrage oder jeden Kontext, bis alle zugehörigen Handler ausgeführt und abgeschlossen sind.

Im Gegensatz zu RequestHandlern können sie nicht auf bestimmte Routengruppen oder spezifische Routen angewendet werden. Stattdessen werden sie auf den gesamten HTTP-Server angewendet. Sie können Bedingungen innerhalb Ihres Http-Server-Handlers anwenden. Darüber hinaus werden Singleton-Instanzen jedes HttpServerHandlers für jede Sisk-Anwendung definiert, so dass nur eine Instanz pro `HttpServerHandler` definiert ist.

Ein praktisches Beispiel für die Verwendung von HttpServerHandler ist die automatische Entsorgung einer Datenbankverbindung am Ende der Anfrage.

```
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           // prüft, ob die Anfrage einen DbContext definiert hat
10          // in ihrem Kontextbeutel
11          if (requestBag.IsSet<DbContext>())
12          {
13              var db = requestBag.Get<DbContext>();
14              db.Dispose();
```

```
        }
    }
}

public static class DatabaseConnectionHandlerExtensions
{
    // ermöglicht es dem Benutzer, einen DbContext aus einer Http-Anfrage zu erstellen
    // und ihn in seinem Kontextbeutel zu speichern
    public static DbContext GetDbContext(this HttpRequest request)
    {
        var db = new DbContext();
        return request.SetContextBag<DbContext>(db);
    }
}
```

Mit dem obigen Code ermöglicht die `GetDbContext` -Erweiterung die Erstellung eines Verbindungskontexts direkt aus dem HttpRequest-Objekt. Eine nicht entsorgte Verbindung kann Probleme beim Ausführen mit der Datenbank verursachen, daher wird sie in `OnHttpRequestClose` beendet.

Sie können einen Handler auf einem Http-Server in Ihrem Builder oder direkt mit HttpServer.RegisterHandler registrieren.

```
// Program.cs

class Program
{
    static void Main(string[] args)
    {
        using var app = HttpServer.CreateBuilder()
            .UseHandler<DatabaseConnectionHandler>()
            .Build();

        app.Router.SetObject(new UserController());
        app.Start();
    }
}
```

Mit diesem Code kann die `UsersController` -Klasse den Datenbankkontext wie folgt verwenden:

```
// UserController.cs

[RoutePrefix("/users")]
public class UserController : ApiController
{
    [RouteGet()]
```

```
    public async Task<HttpResponse> List(HttpRequest request)
    {
        var db = request.GetDbContext();
        var users = db.Users.ToArray();

        return JsonOk(users);
    }

    [RouteGet("<id>")]
    public async Task<HttpResponse> View(HttpRequest request)
    {
        var db = request.GetDbContext();

        var userId = request.GetQueryValue<int>("id");
        var user = db.Users.FirstOrDefault(u => u.Id == userId);

        return JsonOk(user);
    }

    [RoutePost]
    public async Task<HttpResponse> Create(HttpRequest request)
    {
        var db = request.GetDbContext();
        var user = JsonSerializer.Deserialize<User>(request.Body);

        ArgumentNullException.ThrowIfNull(user);

        db.Users.Add(user);
        await db.SaveChangesAsync();

        return JsonMessage("Benutzer hinzugefügt.");
    }
}
```

Der obige Code verwendet Methoden wie `JsonOk` und `JsonMessage`, die in `ApiController` integriert sind, die von `RouterController` abgeleitet ist:

```
// ApiController.cs

public class ApiController : RouterModule
{
    public HttpResponse JsonOk(object value)
    {
        return new HttpResponse(200)
            .WithContent(JsonContent.Create(value, null, new JsonSerializerOptions()
            {
```

```
                PropertyNameCaseInsensitive = true
            }));
    }

    public HttpResponse JsonMessage(string message, int statusCode = 200)
    {
        return new HttpResponse(statusCode)
            .WithContent(JsonContent.Create(new
            {
                Message = message
            }));
    }
}
```

Entwickler können Sitzungen, Kontexte und Datenbankverbindungen mithilfe dieser Klasse implementieren. Der bereitgestellte Code zeigt ein praktisches Beispiel mit dem DatabaseConnectionHandler, der die automatische Entsorgung einer Datenbankverbindung am Ende jeder Anfrage ermöglicht.

Die Integration ist einfach, mit Handlern, die während der Servereinrichtung registriert werden. Die HttpServerHandler-Klasse bietet ein leistungsfähiges Werkzeugset für die Verwaltung von Ressourcen und die Erweiterung des Sisk-Verhaltens in HTTP-Anwendungen.

Mehrere Lauscher-Hosts pro Server

Das Sisk Framework unterstützt seit jeher die Verwendung von mehr als einem Host pro Server, d.h. ein einzelner HTTP-Server kann auf mehreren Ports hören und jeder Port hat seinen eigenen Router und seinen eigenen Dienst, der darauf läuft.

Auf diese Weise ist es einfach, Verantwortlichkeiten zu trennen und Dienste auf einem einzelnen HTTP-Server mit Sisk zu verwalten. Das folgende Beispiel zeigt die Erstellung von zwei ListeningHosts, von denen jeder auf einem anderen Port hört, mit unterschiedlichen Routern und Aktionen.

Lesen Sie manuell Ihre App erstellen, um die Details über diese Abstraktion zu verstehen.

```
1   static void Main(string[] args)
2   {
3       // Erstellen von zwei Listening-Hosts, von denen jeder seinen eigenen Router und
4       // Port hat
5       //
6       ListeningHost hostA = new ListeningHost();
7       hostA.Ports = [new ListeningPort(12000)];
8       hostA.Router = new Router();
9       hostA.Router.SetRoute(RouteMethod.Get, "/", request => new
10  HttpResponse().WithContent("Hallo vom Host A!"));
11
12      ListeningHost hostB = new ListeningHost();
13      hostB.Ports = [new ListeningPort(12001)];
14      hostB.Router = new Router();
15      hostB.Router.SetRoute(RouteMethod.Get, "/", request => new
16  HttpResponse().WithContent("Hallo vom Host B!"));
17
18      // Erstellen einer Server-Konfiguration und Hinzufügen beider
19      // Listening-Hosts
20      //
21      HttpServerConfiguration configuration = new HttpServerConfiguration();
22      configuration.ListeningHosts.Add(hostA);
23      configuration.ListeningHosts.Add(hostB);
24
25      // Erstellen eines HTTP-Servers, der die angegebene
26      // Konfiguration verwendet
27      //
28      HttpServer server = new HttpServer(configuration);
29
30      // Starten des Servers
```

```
31      server.Start();
32
33      Console.WriteLine("Versuchen Sie, Host A unter {0} zu
34  erreichen", server.ListeningPrefixes[0]);
35      Console.WriteLine("Versuchen Sie, Host B unter {0} zu
    erreichen", server.ListeningPrefixes[1]);

        Thread.Sleep(-1);
    }
```