

はじめに

Sisk ドキュメントへようこそ！

最後に、Sisk フレームワークとは何か？Sisk フレームワークは、.NET で構築されたオープンソースの軽量ライブラリで、最小限、柔軟、抽象的であることを目指して設計されています。開発者が、迅速に、必要な設定が少ない、またはまったくないで、インターネットサービスを作成できるようにします。Sisk により、既存のアプリケーションに、管理された HTTP モジュール、完全に破棄可能または完全なものを実現できます。

Sisk の価値観には、コードの透明性、モジュラー性、パフォーマンス、スケーラビリティがあり、Restful、JSON-RPC、Web-sockets などのさまざまなタイプのアプリケーションを処理できます。

主な機能には以下のものがあります：

リソース	説明
ルーティング	プレフィックス、カスタム メソッド、パス変数、値コンバーターなどをサポートするパスルーター。
リクエストハンドラー	ミドルウェアとしても知られており、リクエストの前または後に動作する独自のリクエストハンドラーを構築するためのインターフェイスを提供します。
圧縮	Sisk を使用して、レスポンス コンテンツを簡単に圧縮します。
Web ソケット	完全な Web ソケットを受け入れるルートを提供し、クライアントへの読み取りと書き込みを行います。
サーバー送信イベント	SSE プロトコルをサポートするクライアントにサーバー イベントを送信します。
ログ	ログの簡素化。エラー、ログ、サイズによるローテーション ログ、同じログの複数の出力ストリームなどをログに記録します。
マルチ ホスト	HTTP サーバーを複数のポートで実行し、各ポートに独自のルーターを持ち、各ルーターに独自のアプリケーションを実行します。
サーバーハンドラー	HTTP サーバーの独自の実装を拡張します。拡張機能、改善、ニューフィーチャーでカスタマイズします。

最初のステップ

Sisk は、任意の .NET 環境で実行できます。このガイドでは、.NET を使用して Sisk アプリケーションを作成する方法を説明します。まだインストールしていない場合は、[こちら](#)から SDK をダウンロードしてください。

このチュートリアルでは、プロジェクト構造の作成、リクエストの受信、URL パラメータの取得、レスポンスの送信について説明します。このガイドでは、C# を使用してシンプルなサーバーを構築することに焦点を当てています。好みのプログラミング言語を使用することもできます。

NOTE

クイックスタート プロジェクトに興味がある場合は、[このリポジトリ](#)を参照してください。

プロジェクトの作成

プロジェクト名を "My Sisk Application" とします。.NET を設定したら、次のコマンドを使用してプロジェクトを作成できます。

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

次に、プロジェクトディレクトリに移動し、.NET ユーティリティ ツールを使用して Sisk をインストールします。

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

Sisk をプロジェクトにインストールする方法については、[こちら](#)を参照してください。

ここで、HTTP サーバーのインスタンスを作成します。この例では、ポート 5000 でリッスンするように構成します。

HTTP サーバーの構築

Sisk では、HttpServer オブジェクトにルーティングすることで、アプリケーションを手動で段階的に構築できます。ただし、ほとんどのプロジェクトではこれは便利ではありません。したがって、ビルダー メソッドを使用して、アプリケーションを簡単に起動できます。

Program.cs

C#

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

Sisk の各重要なコンポーネントを理解することが重要です。このドキュメントの後半では、Sisk のしくみについてさらに詳しく説明します。

手動(高度な)設定

Sisk の各メカニズムの動作と、HttpServer、Router、ListeningPort、他のコンポーネント間の関係については、[このセクション](#)のドキュメントを参照してください。

インストール

Sisk を Nuget、dotnet cli、または [他のオプション](#) を介してインストールできます。開発者コンソールで次のコマンドを実行することで、Sisk 環境を簡単に設定できます。

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

このコマンドは、プロジェクトに Sisk の最新バージョンをインストールします。

ネイティブAOTサポート

.NET Native AOTを使用すると、.NETランタイムがターゲットホストにインストールされていない場合でも、自己完結型のネイティブ.NETアプリケーションを公開できます。さらに、ネイティブAOTでは以下のようない利点があります：

- アプリケーションのサイズが大幅に小さくなる
- 初期化が大幅に高速化する
- メモリ消費量が低くなる

Sisk Frameworkは、明示的な性質により、ほとんどの機能でネイティブAOTを使用できます。ソースコードをネイティブAOTに適応させるためのリワークは不要です。

サポートされていない機能

ただし、Siskは一部の機能で反射を使用しています。以下に記載されている機能は、ネイティブコード実行中に部分的に利用可能または完全に利用不可になる可能性があります：

- **モジュールの自動スキャン of the router:** このリソースは、実行中のアセンブリに埋め込まれた型をスキャンし、ルーターモジュールである型を登録します。このリソースでは、アセンブリトリミング中に除外される可能性のある型が必要です。

Siskの他のすべての機能はAOTと互換性があります。AOT警告を出すメソッドが見つかることがあります、ここに記載されていない場合は、型、パラメーター、または型情報を渡すオーバーロードがあり、AOTコンパイラがオブジェクトをコンパイルするのを支援します。

アプリケーションのデプロイ

Sisk アプリケーションのデプロイ プロセスは、プロジェクトを本番環境に公開することです。プロセスは比較的単純ですが、デプロイのインフラストラクチャのセキュリティと安定性に致命的な影響を与える可能性のある詳細に注意する価値があります。

理想的には、アプリケーションをテストして準備し、クラウドにデプロイする準備ができているはずです。

アプリの公開

Sisk アプリケーションまたはサービスを公開するには、生成されたバイナリを本番環境で実行できるように最適化する必要があります。この例では、.NET Runtime がインストールされたマシンで実行するために、バイナリを本番環境用にコンパイルします。

アプリをビルドするには、.NET SDK をインストールし、ターミナル サーバーに .NET Runtime をインストールする必要があります。Linux サーバー、Windows、Mac OS に .NET Runtime をインストールする方法については、[ここ](#)、[ここ](#)、[ここ](#)を参照してください。

プロジェクトが配置されているフォルダーで、ターミナルを開き、.NET 公開コマンドを使用します。

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

これにより、bin/Release/publish/linux-x64 内にバイナリが生成されます。

NOTE

Sisk.ServiceProvider パッケージを使用してアプリを実行している場合、service-config.json ファイルをホスト サーバーにコピーし、dotnet publish で生成されたすべてのバイナリとともに配置する必要があります。ファイルを事前に構成しておくことができます。環境変数、リスニング ポート、ホスト、および追加のサーバー構成が含まれます。

次のステップは、これらのファイルをアプリケーションをホストするサーバーに移動することです。

次に、バイナリ ファイルに実行権限を付与します。この場合、プロジェクト名は "my-app" です。

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

アプリケーションを実行すると、エラー メッセージが表示されない場合は、アプリケーションが実行中であることを意味します。

この時点では、ファイアウォールなどのアクセス ルールが構成されていないため、アプリケーションに外部ネットワークからアクセスすることはできない可能性があります。次のステップでこれを考慮します。

アプリケーションがリスニングしている仮想ホストのアドレスを持っている必要があります。これは、アプリケーション内で手動で設定され、Sisk サービスをインスタンス化する方法によって異なります。

Sisk.ServiceProvider パッケージを使用していない場合、HttpServer インスタンスを定義した場所でこれを見つける必要があります。

```
1 HttpServer server = HttpServer.Emit(5000, out HttpServerConfiguration config, out var host,  
2 out var router);  
// sisk は http://localhost:5000/ でリスニングする必要があります
```

リスニング ホストを手動で関連付ける:

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

または、Sisk.ServiceProvider パッケージを使用している場合、service-config.json 内で:

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

これから、サービスをリスニングし、トラフィックをオープン ネットワークで利用できるようにするために、リバース プロキシを作成できます。

アプリケーションのプロキシ

サービスをプロキシすることは、Sisk サービスを直接外部ネットワークに公開しないことを意味します。この方法は、サーバー デプロイで非常に一般的です。

- アプリケーションに SSL 証明書を関連付けることができます。
- サービスにアクセスする前にアクセス ルールを作成し、過負荷を回避できます。
- バンド幅とリクエストの制限を制御できます。
- アプリケーションの負荷分散を分離できます。
- インフラストラクチャのセキュリティを損なうのを防ぐことができます。

Nginx[↗] または Apache[↗] のようなリバース プロキシを使用してアプリケーションを提供するか、Cloudflare[↗] のような HTTP-over-DNS トンネルを使用することができます。

また、クライアントの情報 (IP アドレスやホストなど) を取得するために、プロキシの転送ヘッダーを正しく解決することを忘れないでください。転送ヘッダー解決を参照してください。

トンネルを作成し、ファイアウォールを構成し、アプリケーションを実行した後、サービスを作成する必要があります。

NOTE

非 Windows システムでは、Sisk サービスで直接 SSL 証明書を使用することはできません。これは、Sisk の HTTP キュー管理の中心となるモジュールである HttpListener の実装によるものであり、オペレーティングシステムによって異なります。IIS で仮想ホストに証明書を関連付けることで、Sisk サービスで SSL を使用できます。ここ[↗]を参照してください。その他のシステムでは、リバース プロキシを使用することを強くお勧めします。

サービスの作成

サービスを作成すると、アプリケーションはサーバー インスタンスの再起動やクラッシュ後も常に利用可能になります。

この簡単なチュートリアルでは、前のチュートリアルからのコンテンツを使用して、サービスを常にアクティブに保つ方法を示します。

1. サービス構成ファイルが配置されているフォルダーにアクセスします。

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

2. my-app.service ファイルを作成し、次の内容を含めます。

my-app.service	INI
----------------	-----

```
1 [Unit]
2 Description=<アプリの説明>
3
4 [Service]
5 # サービスを起動するユーザーを設定
6 User=<サービスを起動するユーザー>
7
8 # ExecStart パスは WorkingDirectory に相対的ではありません。
9 # 実行可能ファイルへの完全パスとして設定します
10 WorkingDirectory=/home/htdocs
11 ExecStart=/home/htdocs/my-app
12
13 # サービスを常に再起動するように設定
14 Restart=always
15 RestartSec=3
16
17 [Install]
18 WantedBy=multi-user.target
```

3. サービス マネージャー モジュールを再起動します。

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

4. ファイル名に基づいて新しく作成したサービスを開始し、実行中であることを確認します。

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

5. アプリが実行中 ("Active: active") である場合、サービスをシステムの再起動後に実行するように有効にします。

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

これで、Sisk アプリケーションを公開する準備ができました。

SSL を使用する

開発の際に SSL を使用する必要がある場合があります。たとえば、ほとんどの Web 開発シナリオではセキュリティが必要です。Sisk は `HttpListener` 上で動作しますが、`HttpListener` ではネイティブの HTTPS はサポートされず、HTTP のみがサポートされます。ただし、Sisk で SSL を使用できるようにするための回避策があります。以下にそれらを示します。

Windows で IIS を使用する

- 使用可能: Windows
- 努力: 中

Windows を使用している場合、IIS を使用して HTTP サーバーで SSL を有効にすることができます。この方法を使用するには、"localhost" 以外のホストでアプリケーションをリッスンさせたい場合は、事前に [このチュートリアル](#) を参照することをお勧めします。

この方法を使用するには、Windows 機能を介して IIS をインストールする必要があります。IIS は、Windows と Windows Server ユーザー向けに無料で提供されています。アプリケーションで SSL を構成するには、セルフサイン証明書であっても SSL 証明書を用意する必要があります。次に、[IIS 7 またはそれ以降で SSL を設定する方法](#) を参照できます。

mitmproxy を使用する

- 使用可能: Linux, macOS, Windows
- 努力: 簡単

mitmproxy は、開発者とセキュリティ テスターがクライアント（たとえば Web ブラウザ）とサーバー之间的 HTTP および HTTPS トラフィックを検査、変更、および記録できるようにするためのインターフェンス プロキシツールです。**mitmdump** ユーティリティを使用して、クライアントと Sisk アプリケーションの間にリバース SSL プロキシを開始できます。

- まず、[mitmproxy](#) をマシンにインストールします。
- Sisk アプリケーションを開始します。この例では、8000 ポートを非安全な HTTP ポートとして使用します。
- mitmproxy サーバーを開始して、8001 ポートでセキュア ポートをリッスンします。

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

これで完了です! すでに <https://localhost:8001/> でアプリケーションを使用できます。 mitmdump を開始するには、アプリケーションが実行中である必要はありません。

代わりに、プロジェクトに [mitmproxy ヘルパー](#) の参照を追加できます。ただし、mitmproxy がコンピューターにインストールされている必要があります。

Sisk.SslProxy パッケージを使用する

- 使用可能: Linux, macOS, Windows
- 努力: 簡単

Sisk.SslProxy パッケージは、Sisk アプリケーションで SSL を有効にするための簡単な方法です。ただし、このパッケージは **非常に実験的** です。このパッケージで作業することは不安定になる可能性がありますが、このパッケージを実用的なものにするために貢献する少数の人々の一人になることができます。開始するには、次のように Sisk.SslProxy パッケージをインストールできます。

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

NOTE

Visual Studio パッケージ マネージャーで "プレビュー パッケージの有効化" を有効にする必要があります。

再び言いますが、このプロジェクトは実験的ですので、生産環境で使用することを考えるべきではありません。

現在、Sisk.SslProxy は、HTTP Continue、チャunk化、WebSockets、SSE を含むほとんどの HTTP/1.1 機能を処理できます。SslProxy についてさらに詳しくは [こちら](#) を参照してください。

Windowsでの名前空間予約の設定

SiskはHttpListenerネットワークインターフェイスを使用しており、仮想ホストをシステムにバインドしてリクエストを待ち受けます。

Windowsでは、このバインドは少し制限があり、localhostのみが有効なホストとしてバインドされます。他のホストをリッスンしようとすると、サーバーでアクセスが拒否されたエラーが発生します。このチュートリアルでは、システムで任意のホストをリッスンするための認証を付与する方法について説明します。

Namespace Setup.bat

BATCH

```
1 @echo off
2
3 :: ここにプレフィックスを入力してください (スペースや引用符なし)
4 SET PREFIX=
5
6 SET DOMAIN=%ComputerName%\%USERNAME%
7 netsh http add urlacl url=%PREFIX% user=%DOMAIN%
8
9 pause
```

ここで、PREFIXは、サーバーがリッスンするプレフィックス("リスニングホスト→ポート")です。URLスキーム、ホスト、ポート、末尾のスラッシュで構成する必要があります。例:

Namespace Setup.bat

BATCH

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

これにより、アプリケーションを介してリッスンできるようになります:

Program.cs

C#

```
1 class Program
2 {
3     static async Task Main(string[] args)
4     {
5         using var app = HttpServer.CreateBuilder()
6             .UseListeningPort("http://my-application.example.test/")
7             .Build();
```

```
8
9     app.Router.MapGet("/", request =>
10    {
11        return new HttpResponse()
12        {
13            Status = 200,
14            Content = new StringContent("Hello, world!")
15        };
16    });
17
18    await app.StartAsync();
19 }
20 }
```

Changelogs

Sisk に行われたすべての変更は、変更ログを通じて記録されます。すべての Sisk バージョンの変更ログは [ここ](#)で確認できます。

Frequently Asked Questions

Sisk に関するよくある質問。

Sisk はオープンソースですか？

完全に。Sisk で使用されるすべてのソースコードは、[GitHub](#) で公開され、頻繁に更新されています。

貢献は受け付けますか？

Sisk の[哲学](#) と互換性がある限り、すべての貢献は大歓迎です！貢献はコードに限りません。ドキュメント、テスト、翻訳、寄付、投稿など、さまざまな形で貢献できます。

Sisk は資金提供されていますか？

いいえ。現在、Sisk はどの組織やプロジェクトからも資金提供を受けておりません。

Sisk を本番環境で使用できますか？

絶対に。プロジェクトは 3 年以上開発されており、商用アプリケーションでのテストが行われてきました。Sisk は、重要な商用プロジェクトの主要インフラストラクチャとして使用されています。

さまざまなシステムや環境での [デプロイ](#) 方法についてのガイドが書かれており、利用可能です。

Sisk には認証、監視、データベースサービスがありますか？

いいえ。Sisk にはこれらのサービスはありません。Sisk は HTTP Web アプリケーションの開発フレームワークですが、まだ最小限のフレームワークであり、アプリケーションが動作するために必要なものだけを提供します。

好みの任意のサードパーティライブラリを使用して、必要なサービスをすべて実装できます。Sisk は、汎用的なシナリオを埋めるために作成され、柔軟性と、他のすべてとの互換性を保っています。

なぜ Sisk を <フレームワーク> よりも使用するべきですか？

わかりません。你が教えてください。

Sisk は、.NET の HTTP Web アプリケーションの汎用的なシナリオを埋めるために作成されました。既存のプロジェクト、たとえば ASP.NET は、さまざまな問題を解決しますが、異なる偏見で解決します。大きいフレームワークとは異なり、Sisk では、ユーザーが何をしているのか、そして何を構築しているのかを知っている必要があります。Web 開発と HTTP プロトコルの基本的な概念は、Sisk を使用するために不可欠です。

Sisk は、Node.js の Express よりも ASP.NET Core に近いです。HTTP ロジックが必要なアプリケーションを作成できる、高レベルの抽象化を提供します。

Sisk を学ぶために何が必要ですか？

以下の基本的な知識が必要です：

- Web 開発 (HTTP、Restful など)
- .NET

これら 2 つのトピックについての基本的な知識があると、Sisk で高度なアプリケーションを開発するのに数時間を使費やすことができます。

Sisk を使用して商用アプリケーションを開発できますか？

絶対に。

Sisk は MIT ライセンスの下で作成されており、Sisk を使用して商用プロジェクトや非商用プロジェクトを作成できます。ただし、プロジェクトで使用されているオープンソースプロジェクトについての通知をどこかに表示し、Sisk も使用していることを示す必要があります。

ルーティング

`Router`は、サーバーを構築するための最初のステップです。ルーティングは、URLとそのメソッドをサーバーが実行するアクションにマッピングするエンドポイントである`Route`オブジェクトを保持する責任があります。各アクションは、リクエストを受信し、クライアントにレスポンスを配信する責任があります。

ルーティングは、`パス式`("パスパターン")とそれがリッスンできるHTTPメソッドのペアです。リクエストがサーバーに送信されると、ルーティングは受信したリクエストに一致するルーティングを検索し、そのルーティングのアクションを呼び出し、結果のレスポンスをクライアントに配信します。

Siskでは、ルーティングを定義する方法は複数あります。静的、動的、または自動スキヤンで定義できます。属性によって定義されることもあり、直接`Router`オブジェクトで定義することもできます。

```
1 Router mainRouter = new Router();
2
3 // GET / ルーティングを次のアクションにマップ
4 mainRouter.MapGet("/", request => {
5     return new HttpResponseMessage("Hello, world!");
6 });


```

ルーティングが何ができるかを理解するには、リクエストが何ができるかを理解する必要があります。

`HttpRequest`には、必要なすべての情報が含まれています。Siskには、開発全体を高速化するいくつかの追加機能も含まれています。

サーバーが受信するすべてのアクションに対して、`RouteAction`タイプのデリゲートが呼び出されます。このデリゲートには、サーバーが受信したリクエストに関するすべての必要な情報を含む`HttpRequest`を保持するパラメーターが含まれています。このデリゲートの結果のオブジェクトは、`HttpResponse`または暗黙的なレスポンスタイプを介してそれにマップされるオブジェクトでなければなりません。

ルーティングのマッチング

HTTPサーバーがリクエストを受信すると、Siskはリクエストを受信したパスの式を満たすルーティングを検索します。式は、常にルーティングとリクエストパスの間でテストされ、クエリ文字列は考慮されません。

このテストには優先順位はありません。单一のルーティングに排他的です。ルーティングがリクエストと一致しない場合、[Router.NotFoundErrorHandler](#)レスポンスがクライアントに返されます。パスパターンが一致するが、HTTPメソッドが一致しない場合、[Router.MethodNotAllowedErrorHandler](#)レスポンスがクライアントに返されます。

Siskは、ルーティングの衝突の可能性をチェックしてこれらの問題を避けます。ルーティングを定義するとき、Siskは定義されているルーティングと衝突する可能性のあるルーティングを検索します。このテストには、ルーティングのパスと受信するメソッドのチェックが含まれます。

パスパターンを使用したルーティングの作成

ルーティングを定義するには、さまざまな SetRoute メソッドを使用できます。

```
1 // SetRoute方式
2 mainRouter.SetRoute(RouteMethod.Get, "/hey/<name>", (request) =>
3 {
4     string name = request.RouteParameters["name"].GetString();
5     return new HttpResponseMessage($"Hello, {name}");
6 });
7
8 // Map*方式
9 mainRouter.MapGet("/form", (request) =>
10 {
11     var formData = request.GetFormData();
12     return new HttpResponseMessage(); // 空の200 OK
13 });
14
15 // Route.*ヘルパーメソッド
16 mainRouter += Route.Get("/image.png", (request) =>
17 {
18     var imageStream = File.OpenRead("image.png");
19
20     return new HttpResponseMessage()
21     {
22         // StreamContent内の
23         // ストリームは、レスポンスを送信した後、破棄されます。
24         Content = new StreamContent(imageStream)
25     };
26 });
27
28 // 複数のパラメーター
29 mainRouter.MapGet("/hey/<name>/surname/<surname>", (request) =>
30 {
31     string name = request.RouteParameters["name"].GetString();
```

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

`RouteParameters`プロパティの`HttpResponse`には、受信したリクエストのパス変数に関するすべての情報が含まれています。

サーバーが受信するすべてのパスは、パスパターンのテストが実行される前に、次のルールに従って正規化されます。

- パスからすべての空のセグメントが削除されます。たとえば、`////foo//bar` は `/foo/bar` になります。
- パスのマッチングは**大文字/小文字を区別します**。ただし、`Router.MatchRoutesIgnoreCase`が `true` に設定されている場合は、区別しません。

`Query`と`RouteParameters`プロパティの`HttpRequest`は、`StringValueCollection`オブジェクトを返します。ここで、各インデックス付きプロパティは、`null`でない`StringValue`を返します。これは、オプション/モナドとして使用して、生の値を管理されたオブジェクトに変換できます。

以下の例では、ルーティングパラメーター「`id`」を読み取り、それから`Guid`を取得します。パラメーターが有効な`Guid`でない場合、例外がスローされ、サーバーが`Router.CallbackErrorHandler`を処理していない場合は、クライアントに500エラーが返されます。

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

ⓘ NOTE

パスの末尾の / は、リクエストパスとルーティングパスの両方で無視されます。つまり、ルーティングが `/index/page` として定義されている場合、`/index/page/` を使用してアクセスすることもできます。

`ForceTrailingSlash`フラグを有効にすると、URLを / で終わらせることもできます。

クラスインスタンスを使用したルーティングの作成

ルーティングを動的に定義するには、`RouteAttribute`属性を使用して、クラスのインスタンスを使用できます。この方法では、ターゲットルーターにルーティングが定義されます。

メソッドがルーティングとして定義されるには、`RouteAttribute`または`RouteGetAttribute`などの属性でマークする必要があります。メソッドは静的、インスタンス、パブリック、またはプライベートにすることができます。

`SetObject(type)` または `SetObject<TType>()` メソッドを使用する場合、インスタンスマソッドは無視されます。

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      // GET / にマッチ
4      [RouteGet]
5      HttpResponseMessage Index(HttpServletRequest request)
6      {
7          HttpResponseMessage res = new HttpResponseMessage();
8          res.Content = new StringContent("Index!");
9          return res;
10     }
11
12     // 静的メソッドも機能します
13     [RouteGet("/hello")]
14     static HttpResponseMessage Hello(HttpServletRequest request)
15     {
16         HttpResponseMessage res = new HttpResponseMessage();
17         res.Content = new StringContent("Hello world!");
18         return res;
19     }
20 }
```

以下の行は、`MyController` の `Index` と `Hello` メソッドの両方をルーティングとして定義します。両方のメソッドがルーティングとしてマークされており、クラスのインスタンスが提供されているためです。クラスの型がインスタンスの代わりに提供された場合、静的メソッドのみが定義されます。

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

Siskバージョン0.16以降、`AutoScan`を有効にすることができます。これにより、`RouterModule`を実装するユーザ一定義のクラスを検索し、ルーターに自動的に関連付けられます。これは、AOTコンパイルではサポートされません。

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

上記の命令は、`ApiController`を実装するすべての型を検索しますが、型自体は検索しません。2つのオプションパラメーターは、メソッドがこれらの型を検索する方法を示します。最初の引数は、型を検索するアセンブリを示し、2番目の引数は、型が定義される方法を示します。

正規表現ルーティング

デフォルトのHTTPパスマッチング方法を使用する代わりに、ルーティングを正規表現で解釈するようにマークできます。

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

または、[RegexRoute](#)クラスを使用することもできます。

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

```

正規表現パターンからグループをキャプチャして、[HttpRequest.RouteParameters](#)の内容に含めることもできます。

Controller/MyController.cs

C#

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

ルーティングのプレフィックス

クラスまたはモジュールのすべてのルーティングにプレフィックスを付けるには、[RoutePrefix](#)属性を使用できます。

以下の例は、BREADアーキテクチャー（Browse、Read、Edit、Add、Delete）を使用しています。

Controller/Api/UsersController.cs

C#

```
1  [RoutePrefix("/api/users")]
2  public class UsersController
3  {
4      // GET /api/users/<id>
5      [RouteGet]
6      public async Task<HttpResponse> Browse()
7      {
8          ...
9      }
10
11     // GET /api/users
12     [RouteGet("/{id}")]
13     public async Task<HttpResponse> Read()
14     {
15         ...
16     }
17
18     // PATCH /api/users/<id>
19     [RoutePatch("/{id}")]
20     public async Task<HttpResponse> Edit()
21     {
22         ...
23     }
24
25     // POST /api/users
26     [RoutePost]
27     public async Task<HttpResponse> Add()
28     {
29         ...
30     }
31
32     // DELETE /api/users/<id>
33     [RouteDelete("/{id}")]
34     public async Task<HttpResponse> Delete()
35     {
36         ...
37     }
38 }
```

上記の例では、`HttpResponse` パラメーターは省略され、代わりにグローバルコンテキスト `HttpContext.Current` を介して使用されます。詳細については、次のセクションを参照してください。

リクエストパラメーターなしのルーティング

ルーティングは、[HttpRequest](#)パラメーターなしで定義でき、依然としてリクエストとそのコンポーネントをリクエストコンテキストで取得できます。すべてのコントローラーの基礎となる ControllerBase 抽象化を考えてみましょう。この抽象化は、Request プロパティを提供して、現在のHttpRequestを取得します。

Controller/BaseController.cs

C#

```
1  public abstract class ControllerBase
2  {
3      // 現在のスレッドからリクエストを取得します。
4      public HttpRequest Request { get => HttpContext.Current.Request; }
5
6      // 次の行は、現在のHTTPセッションからデータベースを取得します。存在しない場合は、新しいものを作成し
7      // ます。
8      public DbContext Database { get => HttpContext.Current.RequestBag.GetOrAdd<DbContext>
9          (); }
10 }
```

そして、すべての派生クラスがリクエストパラメーターなしでルーティング構文を使用できるようにします。

Controller/UsersController.cs

C#

```
1  [RoutePrefix("/api/users")]
2  public class UsersController : ControllerBase
3  {
4      [RoutePost]
5      public async Task<HttpResponse> Create()
6      {
7          // 現在のリクエストからJSONデータを読み取ります。
8          UserCreationDto? user = JsonSerializer.DeserializeAsync<UserCreationDto>
9          (Request.Body);
10         ...
11         Database.Users.Add(user);
12
13         return new HttpResponse(201);
14     }
15 }
```

現在のコンテキストと依存性の注入の詳細については、[依存性の注入](#)チュートリアルを参照してください。

どのメソッドでもマッチするルーティング

ルーティングを定義して、パスのみに基づいてマッチさせ、HTTPメソッドをスキップすることができます。これは、ルーティングのコールバック内でメソッドの検証を行う場合に便利です。

```
1 // どのHTTPメソッドでも / にマッチ
2 mainRouter.SetRoute(RouteMethod.Any, "/", callbackFunction);
```

どのパスでもマッチするルーティング

どのパスでもマッチするルーティングは、ルーティングメソッドをテストするサーバーからのすべてのリクエストにマッチします。ルーティングメソッドが `RouteMethod.Any` で、ルーティングが `Route.AnyPath` をパス式として使用している場合、このルーティングはサーバーからのすべてのリクエストをリッスンし、他のルーティングは定義できません。

```
1 // すべてのPOSTリクエストにマッチ
2 mainRouter.SetRoute(RouteMethod.Post, Route.AnyPath, callbackFunction);
```

大文字/小文字を無視するルーティングのマッチング

デフォルトでは、ルーティングの解釈は大文字/小文字を区別します。無視するには、次のオプションを有効にします。

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

これにより、正規表現マッチングを使用するルーティングの `RegexOptions.IgnoreCase` オプションも有効になります。

見つからない(404) コールバック ハンドラー

ルーティングが見つからない場合のカスタムコールバックを作成できます。

```
1 mainRouter.NotFoundErrorHandler = () =>
2 {
3     return new HttpResponseMessage(404)
4     {
5         // v0.14以降
6         Content = new HtmlContent("<h1>Not found</h1>")
7         // 以前のバージョン
8         Content = new StringContent("<h1>Not found</h1>", Encoding.UTF8, "text/html")
9     };
10};
```

メソッドが許可されていない(405) コールバック ハンドラー

パスが一致するがメソッドが一致しない場合のカスタムコールバックを作成することもできます。

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

内部エラーハンドラー

ルーティングのコールバックは、サーバーの実行中にエラーをスローする可能性があります。適切に処理されない場合、HTTPサーバーの全般的な機能が中断される可能性があります。ルーターには、ルーティングのコールバックが失敗したときに呼び出されるコールバックがあります。

このメソッドは、`ThrowExceptions`が `false` に設定されている場合にのみ到達されます。

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

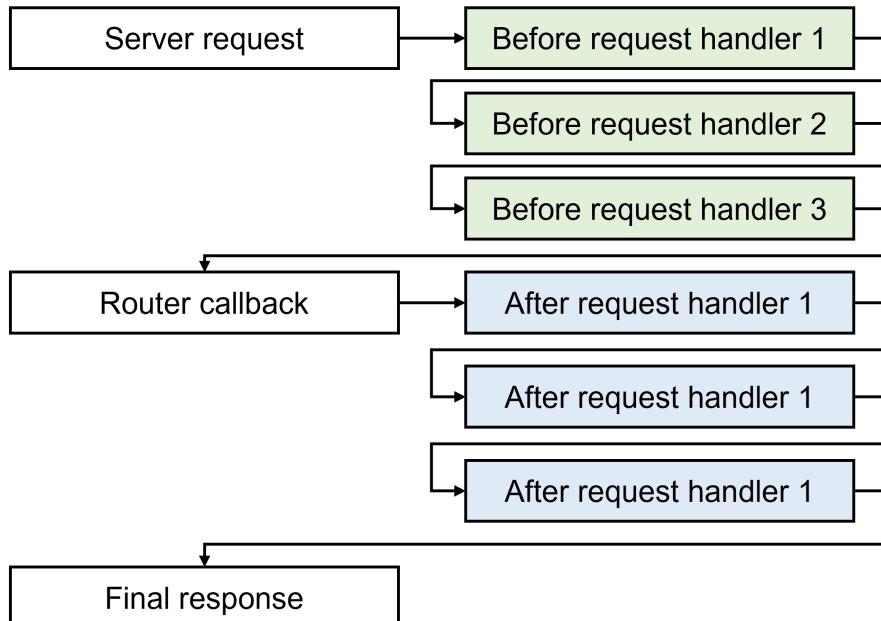
リクエストハンドリング

リクエストハンドラー、別名「ミドルウェア」と呼ばれるものは、リクエストがルーターで実行される前または後に実行される関数です。これらは、ルートごとまたはルーターごとに定義できます。

リクエストハンドラーには2つの種類があります。

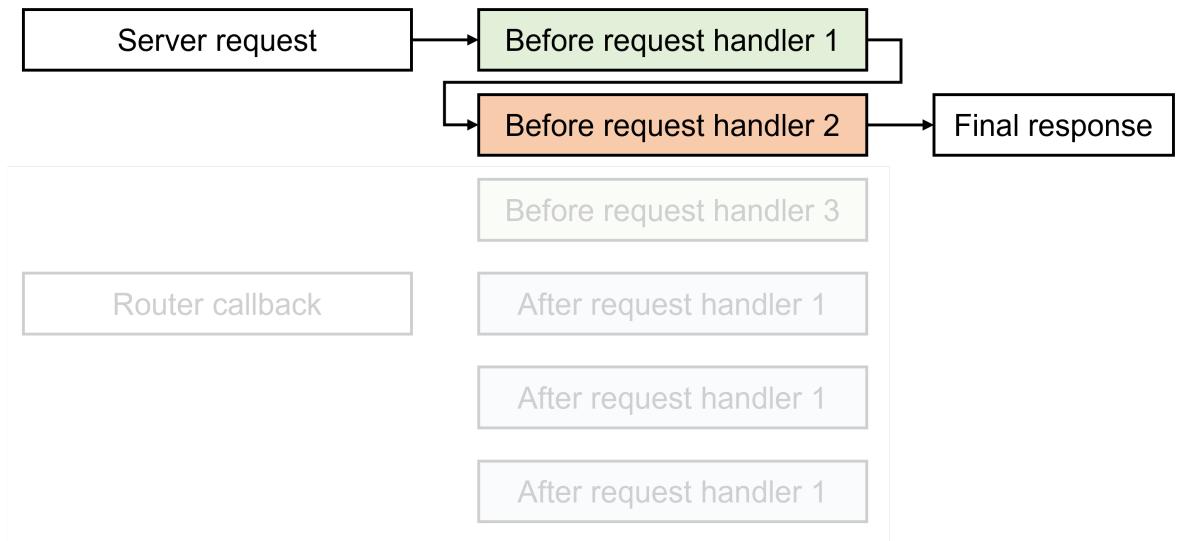
- **BeforeResponse:** リクエストハンドラーがルーターアクションを呼び出す前に実行されることを定義します。
- **AfterResponse:** リクエストハンドラーがルーターアクションを呼び出した後に実行されることを定義します。このコンテキストでHTTPレスポンスを送信すると、ルーターアクションのレスポンスが上書きされます。

両方のリクエストハンドラーは、実際のルーターコールバック関数のレスポンスを上書きできます。さらに、リクエストハンドラーは、認証、コンテンツ、またはその他の情報の検証に役立ちます。また、情報を保存したり、ログを記録したり、レスポンスの前または後に実行できる他のステップもあります。



このように、リクエストハンドラーは実行を中断し、サイクルを終了する前にレスポンスを返すことができます。他のプロセスは破棄されます。

例: ユーザー認証リクエストハンドラーがユーザーを認証しないとします。リクエストライフサイクルは続行されず、ハングします。リクエストハンドラーが2番目の位置にある場合、3番目以降のハンドラーは評価されません。



リクエストハンドラーの作成

リクエストハンドラーを作成するには、[IRequestHandler](#) インターフェイスを継承するクラスを作成できます。

Middleware/AuthenticateUserRequestHandler.cs

C#

```

1  public class AuthenticateUserRequestHandler : IRequestHandler
2  {
3      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
4          RequestHandlerExecutionMode.BeforeResponse;
5
6      public HttpResponse? Execute(HttpContext context)
7      {
8          if (request.Headers.Authorization != null)
9          {
10              // nullを返すと、リクエストサイクルが続行されます
11              return null;
12          }
13          else
14          {
15              // HttpResponseオブジェクトを返すと、隣接するレスポンスが上書きされます
16              return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized);

```

```
17         }
18     }
}
```

上記の例では、Authorizationヘッダーがリクエストに存在する場合、続行し、次のリクエストハンドラーまたはルーターコールバックが呼び出されるようにします。ExecutionModeプロパティによって、レスポンスの前または後に実行されるリクエストハンドラーを指定できます。非null値を返すと、ルーターのレスポンスが上書きされます。

リクエストハンドラーがnullを返すと、リクエストが続行され、次のオブジェクトが呼び出されるか、サイクルがルーターのレスポンスで終了します。

ルートへのリクエストハンドラーの関連付け

ルートに1つまたは複数のリクエストハンドラーを定義できます。

Router.cs

C#

```
1 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage, "", new IRequestHandler[]
2 {
3     new AuthenticateUserRequestHandler(),           // before request handler
4     new ValidateJsonContentRequestHandler(),       // before request handler
5     //                                         -- method IndexPage will be executed here
6     new WriteToLogRequestHandler()                // after request handler
7});
```

または、[Route](#) オブジェクトを作成できます。

Router.cs

C#

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

ルーターへのリクエストハンドラーの関連付け

ルーター全体で実行されるグローバルリクエストハンドラーを定義できます。

Router.cs

C#

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

属性へのリクエストハンドラーの関連付け

ルート属性とともにメソッド属性にリクエストハンドラーを定義できます。

Controller/MyController.cs

C#

```
1 public class MyController
2 {
3     [RouteGet("/")]
4     [RequestHandler<AuthenticateUserRequestHandler>]
5     static HttpResponseMessage Index(HttpContext request)
6     {
7         return new HttpResponseMessage() {
8             Content = new StringContent("Hello world!")
9         };
10    }
11 }
```

注: リクエストハンドラータイプを渡す必要があります。インスタンスを渡すことはできません。そうすると、ルーターがによってリクエストハンドラーがインスタンス化されます。コンストラクタ引数を [ConstructorArguments](#) プロパティで渡すことができます。

例:

Controller/MyController.cs

C#

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

独自の属性を作成してRequestHandlerを実装することもできます。

Middleware/Attributes/AuthenticateAttribute.cs

C#

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

そして、次のように使用します。

Controller/MyController.cs

C#

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

グローバルリクエストハンドラーのバイパス

ルートにグローバルリクエストハンドラーを定義した後、特定のルートでそれを無視できます。

Router.cs

C#

```
1 var myRequestHandler = new AuthenticateUserRequestHandler();
2 mainRouter.GlobalRequestHandlers = new IRequestHandler[]
3 {
4     myRequestHandler
5 };
6
7 mainRouter.SetRoute(new Route(RouteMethod.Get, "/", "My route", IndexPage, null)
8 {
9     BypassGlobalRequestHandlers = new IRequestHandler[]
10    {
11        myRequestHandler, // ok: グローバルリクエストハンドラーと同じインスタンス
12        new AuthenticateUserRequestHandler() // wrong: グローバルリクエストハンドラーはスキップされません
13    }
14 });
});
```

① NOTE

リクエストハンドラーをバイパスする場合、同じ参照を使用してスキップする必要があります。別のリクエストハンドラーインスタンスを作成すると、グローバルリクエストハンドラーはスキップされません。グローバルリクエストハンドラーとバイパスグローバルリクエストハンドラーの両方で同じリクエストハンドラー参照を使用することを確認してください。

Requests

リクエストは HTTP リクエストメッセージを表す構造体です。 [HttpRequest]

(/api/Sisk.Core.Http.HttpRequest) オブジェクトは、アプリケーション全体で HTTP メッセージを処理するための便利な関数を提供します。

HTTP リクエストは、メソッド、パス、バージョン、ヘッダー、ボディで構成されます。

このドキュメントでは、これらの要素を取得する方法を説明します。

リクエストメソッドの取得

受信したリクエストのメソッドを取得するには、 Method プロパティを使用します。

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

このプロパティは、[HttpMethod](#) オブジェクトで表されるリクエストのメソッドを返します。

NOTE

ルートメソッドとは異なり、このプロパティは [RequestMethod.Any] (/api/Sisk.Core.Routing.RouteMethod) アイテムを返しません。代わりに、実際のリクエストメソッドを返します。

リクエスト URL コンポーネントの取得

URL のさまざまなコンポーネントは、リクエストの特定のプロパティを介して取得できます。例として、次の URL を考えます。

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

コンポーネント

ト名	説明	コンポーネント値
Path	リクエストパスを取得します。	/user/login
FullPath	リクエストパスとクエリ文字列を取得します。	/user/login?email=foo@bar.com
FullUrl	完全な URL リクエスト文字列を取得します。	http://localhost:5000/user/login?email=foo@bar.com
Host	リクエストホストを取得します。	localhost
Authority	リクエストホストとポートを取得します。	localhost:5000
QueryString	リクエストクエリを取得します。	?email=foo@bar.com
Query	名前付き値コレクションとしてリクエストクエリを取得します。	{StringValueCollection object}
IsSecure	リクエストが SSL を使用しているかどうかを判定します (true/false)。	false

また、 [HttpRequest.Uri](/api/Sisk.Core.Http.HttpRequest.Uri) プロパティを使用すると、上記のすべてを1つのオブジェクトで取得できます。

リクエストボディの取得

フォーム、ファイル、または API トランザクションなどのボディを含むリクエストがあります。プロパティからリクエストボディを取得できます。

```
1 // リクエストエンコーディングをエンコーダーとして使用して、文字列としてリクエストボディを取得
2 string body = request.Body;
3
4 // またはバイト配列として取得
5 byte[] bodyBytes = request.RawBody;
6
```

```
7 // それ以外の場合はストリームとして取得  
8 Stream requestStream = request.GetRequestStream();
```

リクエストにボディがあるかどうか、およびロードされているかどうかは、`[HasContents]` (`/api/Sisk.Core.Http.HttpRequest.HasContents`) と `[IsContentAvailable]` (`/api/Sisk.Core.Http.HttpRequest.IsContentAvailable`) プロパティで判断できます。`HasContents` はリクエストにコンテンツがあるかを、`IsContentAvailable` は HTTP サーバーがリモートポイントからコンテンツを完全に受信したかを示します。

`GetRequestStream` を使用してリクエストコンテンツを複数回読み取ることはできません。このメソッドで読み取ると、`RawBody` と `Body` の値も利用できなくなります。リクエストストリームは、リクエストのコンテキスト内で破棄する必要はなく、作成された HTTP セッションの終了時に破棄されます。また、

`[HttpRequest.RequestEncoding]` (`/api/Sisk.Core.Http.HttpRequest.RequestEncoding`) プロパティを使用して、リクエストを手動でデコードするための最適なエンコーディングを取得できます。

サーバーはリクエストコンテンツの読み取りに制限を設けており、これは `[HttpRequest.Body]` (`/api/Sisk.Core.Http.HttpRequest.Body`) と `[HttpRequest.RawBody]` (`/api/Sisk.Core.Http.HttpRequest.Body`) の両方に適用されます。これらのプロパティは、入力ストリーム全体を `[HttpRequest.ContentLength]` (`/api/Sisk.Core.Http.HttpRequest.ContentLength`) と同じサイズのローカルバッファにコピーします。

クライアントが送信したコンテンツが `[HttpServerConfiguration.MaximumContentSize]` (`/api/Sisk.Core.Http.HttpServerConfiguration.MaximumContentSize`) を超える場合、ステータス 413 Content Too Large のレスポンスがクライアントに返されます。さらに、設定された制限がない、または制限が大きすぎる場合、クライアントが送信したコンテンツが `[Int32.MaxValue]` (2 GB) を超えると、サーバーは `[OutOfMemoryException]` (<https://learn.microsoft.com/en-us/dotnet/api/system.outofmemoryexception?view=net-8.0>) をスローします。上記のプロパティを介してコンテンツにアクセスしようとするときに発生します。ストリーミングを使用してコンテンツを処理することもできます。

NOTE

Sisk は許可していますが、HTTP セマンティクスに従ってアプリケーションを作成し、許可されていない方法でコンテンツを取得または提供しないようにすることが常に良いアイデアです。RFC 9110 "HTTP Semantics" [を参照してください。](#)

リクエストコンテキストの取得

HTTP コンテキストは、HTTP サーバー、ルート、ルーター、リクエストハンドラー情報を格納する Sisk 専用オブジェクトです。これらのオブジェクトを整理するのが難しい環境で、整理するために使用できます。

[RequestBag] (/api/Sisk.Core.Http.HttpContext.RequestBag) オブジェクトは、リクエストハンドラーから別のポイントへ渡される情報を保持し、最終目的地で消費できます。このオブジェクトは、ルートコールバック後に実行されるリクエストハンドラーでも使用できます。

(i) TIP

このプロパティは [HttpRequest.Bag] (/api/Sisk.Core.Http.HttpRequest.Bag) プロパティからもアクセスできます。

Middleware/AuthenticateUserRequestHandler.cs

C#

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

上記のリクエストハンドラーは `AuthenticatedUser` をリクエストバッグに定義し、後で最終コールバックで消費できます。

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      [RouteGet("/")]
4      [RequestHandler<AuthenticateUserRequestHandler>]
5      static HttpResponse Index(HttpContext request)
6      {
```

```
7     User authUser = request.Context.RequestBag["AuthenticatedUser"];
8
9     return new HttpResponseMessage() {
10         Content = new StringContent($"Hello, {authUser.Name}!")
11     };
12 }
13 }
```

Bag.Set() と Bag.Get() ヘルパーメソッドを使用して、型単位のシングルトンでオブジェクトを取得または設定することもできます。

Middleware/Authenticate.cs

C#

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

Controller/MyController.cs

C#

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

フォームデータの取得

以下の例のように、[\[NameValueCollection\]\(https://learn.microsoft.com/pt-br/dotnet/api/system.collections.specialized.namevaluecollection\)](https://learn.microsoft.com/pt-br/dotnet/api/system.collections.specialized.namevaluecollection) でフォームデータの値を取得できます。

Controller/Auth.cs

C#

```

1 [RoutePost("/auth")]
2 public HttpResponse Index(HttpServletRequest request)
3 {
4     var form = request.GetFormContent();
5
6     string? username = form["username"];
7     string? password = form["password"];
8
9     if (AttemptLogin(username, password))
10    {
11        ...
12    }
13 }

```

マルチパートフォームデータの取得

Sisk の HTTP リクエストは、ファイル、フォームフィールド、または任意のバイナリコンテンツなどのアップロードされたマルチパートコンテンツを取得できます。

Controller/Auth.cs

C#

```

1 [RoutePost("/upload-contents")]
2 public HttpResponse Index(HttpServletRequest request)
3 {
4     // 以下のメソッドは、リクエスト入力全体を
5     // MultipartObjects の配列に読み取ります
6     var multipartFormDataObjects = request.GetMultipartFormContent();
7
8     foreach (MultipartObject uploadedObject in multipartFormDataObjects)
9     {
10         // Multipart フォームデータで提供されたファイル名。
11         // オブジェクトがファイルでない場合は null が返ります。
12         Console.WriteLine("File name : " + uploadedObject.Filename);
13
14         // マルチパートフォームデータオブジェクトのフィールド名。
15         Console.WriteLine("Field name : " + uploadedObject.Name);
16
17         // マルチパートフォームデータのコンテンツ長。
18         Console.WriteLine("Content length : " + uploadedObject.ContentLength);
19

```

```
20      // 既知のコンテンツタイプに基づいて画像フォーマットを判定します。
21      // コンテンツが認識されていない一般的なファイル形式の場合、以下のメソッドは
22      // MultipartObjectCommonFormat.Unknown を返します。
23      Console.WriteLine("Common format : " + uploadedObject.GetCommonFileFormat());
24  }
25 }
```

Sisk の [Multipart form objects](#) とそのメソッド、プロパティ、機能については、さらに詳しく読むことができます。

クライアント切断の検出

Sisk v1.15 以降、フレームワークは、クライアントとサーバー間の接続が応答を受信する前に予期せず閉じられた場合にスローされる `CancellationToken` を提供します。このトークンは、クライアントが応答を望まなくなったときに長時間実行される操作をキャンセルするために役立ちます。

```
1 router.MapGet("/connect", async (HttpRequest req) =>
2 {
3     // リクエストから切断トークンを取得
4     var dc = req.DisconnectToken;
5
6     await LongOperationAsync(dc);
7
8     return new HttpResponseMessage();
9});
```

このトークンはすべての HTTP エンジンと互換性があるわけではなく、各エンジンで実装が必要です。

サーバー送信イベント (SSE) サポート

Sisk は [Server-sent events](#) をサポートし、チャunkをストリームとして送信し、サーバーとクライアント間の接続を維持できます。

`[HttpRequest.GetEventSource] [/api/Sisk.Core.Http.HttpRequest.GetEventSource]` メソッドを呼び出すと、`HttpRequest` がリスナー状態になります。この状態では、HTTP リクエストのコンテキストは `HttpResponse` を期

待せず、サーバー側イベントで送信されるパケットと重複します。

すべてのパケットを送信した後、コールバックは [Close]

(/api/Sisk.Core.Http.HttpRequestEventArgs.Close) メソッドを返す必要があります。これにより、サーバーに最終レスポンスが送信され、ストリーミングが終了したことが示されます。

送信されるすべてのパケットの総長を予測できないため、Content-Length ヘッダーで接続の終了を決定することはできません。

ほとんどのブラウザのデフォルトでは、サーバー側イベントは GET メソッド以外の HTTP ヘッダーやメソッドを送信しません。したがって、イベントソースリクエストで特定のヘッダーが必要なリクエストハンドラーを使用する場合は、ヘッダーがない可能性が高いです。

また、ほとんどのブラウザは、クライアント側で EventSource.close メソッドが呼び出されないと、ストリームを再起動します。これにより、サーバー側で無限に追加処理が発生します。この種の問題を回避するには、イベントソースがすべてのパケットの送信を終了したことを示す最終パケットを送信することが一般的です。

以下の例は、ブラウザがサーバー側イベントをサポートするサーバーと通信する方法を示しています。

sse-example.html

HTML

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

そして、クライアントにメッセージを段階的に送信します。

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

このコードを実行すると、次のような結果が期待されます。



Fruits:

プロキシされた IP とホストの解決

Sisk はプロキシとともに使用でき、したがって IP アドレスはクライアントからプロキシへのトランザクションでプロキシエンドポイントに置き換えられる場合があります。

Sisk で独自のリゾルバを定義するには、[forwarding resolvers](#) を使用します。

ヘッダーエンコーディング

ヘッダーエンコーディングは、いくつかの実装で問題になることがあります。Windows では UTF-8 ヘッダーがサポートされていないため、ASCII が使用されます。Sisk には、誤ってエンコードされたヘッダーをデコードするのに役立つ組み込みエンコーディングコンバータがあります。

この操作はコストが高く、デフォルトでは無効になっていますが、[NormalizeHeadersEncodings](#) フラグで有効になります。

レスポンス

レスポンスは、HTTP リクエストに対する HTTP レスポンスを表すオブジェクトです。これらは、サーバーからクライアントに、リソース、ページ、ドキュメント、ファイルまたはその他のオブジェクトのリクエストの完了を示すために送信されます。

HTTP レスポンスは、ステータス、ヘッダー、およびコンテンツで構成されます。

このドキュメントでは、Sisk を使用して HTTP レスポンスを構築する方法を説明します。

HTTP ステータスの設定

HTTP ステータス一覧は、HTTP/1.0 以降同じであり、Sisk はすべてをサポートしています。

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

または、Fluent Syntax を使用して:

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

使用可能な HttpStatusCode の一覧は、[こちら](#)で確認できます。また、[HttpStatusInformation](#) 構造体を使用して独自のステータス コードを指定することもできます。

ボディとコンテンツタイプ[¶]

Sisk は、レスポンスのボディを送信するために .NET ネイティブ コンテンツ オブジェクトをサポートしています。たとえば、[StringContent](#) クラスを使用して JSON レスポンスを送信できます。

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

サーバーは、ヘッダーで明示的に定義されていない場合、コンテンツから Content-Length を計算しようとします。サーバーがレスポンス コンテンツから Content-Length ヘッダーを暗黙のうちに取得できない場合、レスポンスは Chunked-Encoding で送信されます。

[StreamContent](#) を送信するか、[GetResponseStream](#) メソッドを使用してレスポンスをストリーミングすることもできます。

レスポンスヘッダ

レスポンスで送信するヘッダを追加、編集、または削除できます。以下の例は、クライアントへのリダイレクトレスポンスを送信する方法を示しています。

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

または、Fluent Syntax を使用して:

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

[HttpHeaderCollection](#) の Add メソッドを使用すると、すでに送信されたヘッダを変更せずにヘッダをリクエストに追加します。Set メソッドは、同じ名前のヘッダを指定された値に置き換えます。HttpHeaderCollection のインデクサーは、内部的に Set メソッドを呼び出してヘッダを置き換えます。

クッキーの送信

Sisk には、クライアントでのクッキーの定義を容易にするメソッドがあります。このメソッドで設定されたクッキーは、すでに URL エンコードされ、RFC-6265 標準に適合しています。

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

または、Fluent Syntax を使用して：

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

このメソッドのより完全なバージョンは、[こちら](#) にあります。

チャunkレスポンス

大きなレスポンスを送信するために、転送エンコードをチャunkに設定できます。

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

チャunkエンコードを使用する場合、Content-Length ヘッダーは自動的に省略されます。

レスポンスストリーム

レスポンスストリームは、管理された方法で、セグメント化された方法でレスポンスを送信できるようにします。これは、`HttpResponse` オブジェクトを使用するよりも低レベルの操作であり、ヘッダーとコンテンツを手動で送信し、接続を閉じる必要があります。

この例では、ファイルの読み取り専用ストリームを開き、ストリームをレスポンス出力ストリームにコピーし、ファイルをメモリにロードしません。これは、大きなファイルまたは中程度のファイルをサーブする場合に便利です。

```

1 // レスポンス出力ストリームを取得します
2 using var fileStream = File.OpenRead("my-big-file.zip");
3 var responseStream = request.GetResponseStream();
4
5 // チャンクエンコードを使用するようにレスポンスエンコードを設定します
6 // チャンクエンコードを使用する場合、Content-Length ヘッダーを送信しないでください
7 responseStream.SendChunked = true;
8 responseStream.SetStatus(200);
9 responseStream.SetHeader(HttpKnownHeaderNames.ContentType, contentType);
10
11 // ファイルストリームをレスポンス出力ストリームにコピーします
12 fileStream.CopyTo(responseStream.ResponseStream);
13
14 // ストリームを閉じます
15 return responseStream.Close();

```

GZip、Deflate、およびBrotli圧縮

Sisk で圧縮されたコンテンツを使用してレスポンスを送信できます。まず、以下の圧縮ツールで [HttpContent](#) オブジェクトをカプセル化して、クライアントに圧縮されたレスポンスを送信します。

```

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

```

これらの圧縮コンテンツをストリーミングで使用することもできます。

```

1 router.MapGet("/archive.zip", request => {
2
3     // ここで "using" を適用しないでください。HttpServer はレスポンスを送信した後でコンテンツを破棄します。
4     var archive = File.OpenRead("/path/to/big-file.zip");
5
6     return new HttpResponseMessage () {

```

```
7     Content = new GZipContent(archive)
8 }
9});
```

これらのコンテンツを使用する場合、Content-Encoding ヘッダーは自動的に設定されます。

自動圧縮

[EnableAutomaticResponseCompression](#) プロパティを使用して、HTTP レスポンスを自動的に圧縮することができます。このプロパティは、ルーターからのレスポンス コンテンツを圧縮可能なコンテンツで自動的にカプセル化し、リクエストで受け入れられている場合にのみ圧縮を行います。レスポンスが [CompressedContent](#) から継承されていない場合に限り、圧縮が行われます。

1つのリクエストに対して、1つの圧縮コンテンツのみが選択され、Accept-Encoding ヘッダーに従って、以下の順序で選択されます。

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

リクエストがこれらの圧縮方法のいずれを受け入れることを指定した場合、レスポンスは自動的に圧縮されます。

暗黙のレスポンスタイプ

[HttpResponse](#) 以外の戻り値タイプを使用できますが、ルーターが各タイプのオブジェクトを処理する方法を構成する必要があります。

概念は、常に参照型を返し、それを有効な [HttpResponse](#) オブジェクトに変換することです。[HttpResponse](#) を返すルートは、変換を経ません。

値タイプ (構造体) は、[RouterCallback](#) と互換性がないため、戻り値のタイプとして使用できません。そのため、[ValueResult](#) でラッピングしてハンドラーで使用できるようにする必要があります。

以下の例は、戻り値のタイプとして [HttpResponse](#) を使用しないルーター モジュールの例です。

```

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

```

これにより、ルーターが各タイプのオブジェクトを処理する方法を定義する必要があります。オブジェクトは常にハンドラーの最初の議論であり、出力タイプは有効な `HttpResponse` でなければなりません。また、ルートの出力オブジェクトは、決して `null` になることはできません。

`ValueResult` タイプの場合、入力オブジェクトが `ValueResult` であることを示す必要はなく、`T` のみで十分です。`ValueResult` は、その元のコンポーネントから反映されたオブジェクトであるためです。

タイプの関連付けは、登録されたものとルーター コールバックから返されたオブジェクトのタイプを比較しません。代わりに、ルーター結果のタイプが登録されたタイプに割り当て可能かどうかを確認します。

`Object` ハンドラーの登録は、以前に検証されていないすべてのタイプのフォールバックになります。値ハンドラーの挿入順序も重要であり、`Object` ハンドラーの登録は、他のタイプ固有のハンドラーを無視します。常に特定の値ハンドラーを最初に登録して、順序を確保してください。

```

1 Router r = new Router();
2 r.SetObject(new UsersController());
3
4 r.RegisterValueHandler<ApiResult>(apiResult =>
5 {
6     return new HttpResponseMessage() {
7         Status = apiResult.Success ? HttpStatusCode.OK : HttpStatusCode.BadRequest,
8         Content = apiResult.GetHttpContent(),
9         Headers = apiResult.GetHeaders()
10    };
11 });
12 r.RegisterValueHandler<bool>(bvalue =>
13 {
14     return new HttpResponseMessage() {
15         Status = bvalue ? HttpStatusCode.OK : HttpStatusCode.BadRequest
16    };
17 });
18 r.RegisterValueHandler<IEnumerable<object>>(enumerableValue =>
19 {
20     return new HttpResponseMessage(string.Join("\n", enumerableValue));
21 });
22
23 // オブジェクトの値ハンドラーの登録は最後に実行する必要があります
24 // このハンドラーはフォールバックとして使用されます
25 r.RegisterValueHandler<object>(fallback =>
26 {
27     return new HttpResponseMessage() {
28         Status = HttpStatusCode.OK,
29         Content = JsonContent.Create(fallback)
30    };
31 });

```

列挙オブジェクトと配列に関する注意

[IEnumerable](#) を実装する暗黙のレスポンス オブジェクトは、 `ToArray()` メソッドによってメモリに読み込まれ、定義された値ハンドラーによって変換されます。これが発生するには、 `IEnumerable` オブジェクトがオブジェクトの配列に変換され、レスポンス コンバーターは、元のタイプではなく常に `Object[]` を受け取ります。

以下のシナリオを考えてみましょう。

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

上記の例では、`IEnumerable<string>` コンバーターは呼び出されません。入力オブジェクトは常に `Object[]` であり、`IEnumerable<string>` に変換できません。ただし、以下に示すように、コンバーターが `IEnumerable<object>` を受け取る場合は、入力を受け取ります。

`IAsyncEnumerable` を実装する値は、`ConvertIAsyncEnumerableIntoEnumerable` プロパティが有効になっている場合、サーバーによって自動的に処理されます。非同期列挙は、ブロック列挙に変換され、オブジェクトの同期配列に変換されます。

ログギング

Sisk を設定してアクセスログとエラーログを書き込むように自動で構成できます。ログローテーション、拡張子、頻度を定義することも可能です。

[LogStream](#) クラスは、ログを書き込む非同期方法を提供し、待機可能な書き込みキューに保持します。

この記事では、アプリケーションのロギングを構成する方法を示します。

ファイルベースのアクセスログ

ファイルにログを書き込む際は、ファイルを開き、行テキストを書き込み、書き込みごとにファイルを閉じます。この手順は、ログの書き込み応答性を維持するために採用されました。

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 }
```

上記のコードは、すべての受信リクエストを `logs/access.log` ファイルに書き込みます。ファイルが存在しない場合は自動的に作成されますが、フォルダーは作成されません。`logs/` ディレクトリを作成する必要はありません。`LogStream` クラスが自動的に作成します。

ストリームベースのロギング

TextWriter オブジェクト（例: `Console.Out`）にログファイルを書き込むには、コンストラクタに `TextWriter` オブジェクトを渡します。

Program.cs

C#

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

ストリームベースのログに書き込まれるすべてのメッセージで、`TextWriter.Flush()` メソッドが呼び出されます。

アクセスログのフォーマット

アクセスログのフォーマットは、事前定義された変数でカスタマイズできます。次の行を考えてみてください。

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

これにより、次のようなメッセージが書き込まれます。

```
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]
```

フォーマットは次の表で説明される形式でログファイルを整形できます。

Value	何を表すか	例
%dd	月の日 (2桁)	05
%dmmm	月のフルネーム	July

Value	何を表すか	例
%dmm	月の省略名（3文字）	Jul
%dm	月番号（2桁）	07
%dy	年（4桁）	2023
%th	12時間表記の時間	03
%tH	24時間表記の時間（HH）	15
%ti	分（2桁）	30
%ts	秒（2桁）	45
%tm	ミリ秒（3桁）	123
%tz	タイムゾーンオフセット（UTC 時間）	+03:00
%ri	クライアントのリモート IP アドレス	192.168.1.100
%rm	HTTP メソッド（大文字）	GET
%rs	URI スキーム（http/https）	https
%ra	URI 権限（ドメイン）	example.com
%rh	リクエストのホスト	www.example.com
%rp	リクエストのポート	443
%rz	リクエストのパス	/path/to/resource
%rq	クエリ文字列	?key=value&another=123
%sc	HTTP 応答ステータスコード	200
%sd	HTTP 応答ステータス説明	OK
%lin	リクエストの人間が読めるサイズ	1.2 KB
%linr	リクエストの生サイズ（バイト）	1234
%lou	応答の人間が読めるサイズ	2.5 KB
%lour	応答の生サイズ（バイト）	2560
%lms	ミリ秒単位の経過時間	120

Value	何を表すか	例
%ls	実行ステータス	Executed
%{header-name}	リクエストの header-name ヘッダーを表す。	Mozilla/5.0 (platform; rv:gecko [...]
%{:res-name}	応答の res-name ヘッダーを表す。	

ローテーションログ

HTTP サーバーを構成して、ログファイルが一定サイズに達したら圧縮された .gz ファイルにローテーションすることができます。サイズは、定義した閾値で定期的にチェックされます。

```

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

```

上記のコードは、6 時間ごとに LogStream のファイルが 64MB に達したかどうかを確認します。達した場合、ファイルは .gz ファイルに圧縮され、 access.log がクリアされます。

このプロセス中、ファイルへの書き込みはファイルが圧縮されクリアされるまでロックされます。この期間に書き込まれるすべての行は、圧縮が終了するまでキューに入れられます。

この機能はファイルベースの LogStream でのみ動作します。

エラーロギング

サーバーがデバッガーにエラーをスローしない場合、エラーがあるときにログ書き込みに転送されます。エラー書き込みを構成するには次のようにします。

```

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

```

このプロパティは、エラーがコールバックまたは `Router.CallbackErrorHandler` によってキャプチャされない場合にのみログに書き込みます。

サーバーが書き込むエラーは、常に日付と時刻、リクエストヘッダー（本文は除く）、エラートレース、および内部例外トレース（あれば）を書き込みます。

その他のロギングインスタンス

アプリケーションはゼロまたは複数の `LogStream` を持つことができ、ログチャネルの数に制限はありません。したがって、デフォルトの `AccessLog` または `ErrorLog` 以外のファイルにアプリケーションのログを送ることが可能です。

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

LogStream の拡張

`LogStream` クラスを拡張して、現在の Sisk ログエンジンと互換性のあるカスタムフォーマットを書き込むことができます。以下の例では、`Spectre.Console` ライブラリを使用してコンソールにカラフルなメッセージを書き込みます。

CustomLogStream.cs

C#

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

各リクエスト/レスポンスに対して自動的にカスタムログを書き込むもう一つの方法は、`HttpServerHandler` を作成することです。以下の例は少し完成度が高く、リクエストとレスポンスの本文を JSON でコンソールに書き込みま

す。これは一般的なリクエストのデバッグに役立ちます。この例では ContextBag と HttpServerHandler を使用しています。

Program.cs

C#

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

JsonMessageHandler.cs

C#

```
1  class JsonMessageHandler : HttpServerHandler
2  {
3      protected override void OnHttpRequestOpen(HttpRequest request)
4      {
5          if (request.Method != HttpMethod.Get && request.Headers["Content-
6 Type"]?.Contains("json", StringComparison.InvariantCultureIgnoreCase) == true)
7          {
8              // ここで接続が開かれ、クライアントが JSON コンテンツであることを指定するヘッダーを送信した状
9              態です。
10             // 以下の行はコンテンツを読み取り、リクエストに保持します。
11             //
12             // コンテンツがリクエストアクションで読み取られない場合、GC はレスポンス送信後にコンテンツを收
13             集する可能性があるため、レスポンスが閉じられた後にコンテンツが利用できない場合があります。
```

```

14         //  
15         _ = request.RawBody;  
16  
17         // このリクエストに JSON 本文があることを示すヒントをコンテキストに追加  
18         request.Bag.Add("IsJsonRequest", true);  
19     }  
20 }  
21  
22 protected override async void OnHttpRequestClose(HttpServerExecutionResult result)  
23 {  
24     string? requestJson = null,  
25             responseJson = null,  
26             responseMessage;  
27  
28     if (result.Request.Bag.ContainsKey("IsJsonRequest"))  
29     {  
30         // CypherPotato.LightJson ライブラリを使用して JSON を再フォーマット  
31         var content = result.Request.Body;  
32         requestJson = JsonValue.Deserialize(content, new JsonOptions() { WriteIndented  
= true }).ToString();  
33     }  
34  
35     if (result.Response is {} response)  
36     {  
37         var content = response.Content;  
38         responseMessage = $"{(int)response.Status}  
{HttpStatusInformation.GetStatusCodeDescription(response.Status)}";  
39  
40         if (content is HttpContent httpContent &&  
41             // レスポンスが JSON かどうか確認  
42             httpContent.Headers.ContentType?.MediaType?.Contains("json",  
43             StringComparison.InvariantCultureIgnoreCase) = true)  
44             {  
45                 string json = await httpContent.ReadAsStringAsync();  
46                 responseJson = JsonValue.Deserialize(json, new JsonOptions() {  
47                 WriteIndented = true }).ToString();  
48             }  
49         }  
50     }  
51     else  
52     {  
53         // 内部サーバー処理ステータスを取得  
54         responseMessage = result.Status.ToString();  
55     }  
56  
57     StringBuilder outputMessage = new StringBuilder();  
58     if (requestJson != null)  
59     {  
60

```

```
62     outputMessage.AppendLine("-----");
63     outputMessage.AppendLine($">>> {result.Request.Method} {result.Request.Path}");
64
65     if (requestJson is not null)
66         outputMessage.AppendLine(requestJson);
67 }
68
69     outputMessage.AppendLine($"<<< {responseMessage}");
70
71     if (responseJson is not null)
72         outputMessage.AppendLine(responseJson);
73
74     outputMessage.AppendLine("-----");
75
76     await Console.Out.WriteLineAsync(outputMessage.ToString());
77 }
78 }
```

Server Sent Events

Siskは、Server Sent Eventsをサポートしており、簡単に実装できます。クライアントにメッセージを送信するための使い捨て接続や永続接続を作成し、実行時に接続を取得して使用することができます。

この機能には、ブラウザによって課されるいくつかの制限があります。たとえば、テキストメッセージのみを送信でき、接続を永久に閉じることはできません。サーバー側で接続が閉じられた場合、クライアントは5秒ごとに再接続を試みます（一部のブラウザでは3秒ごと）。

これらの接続は、クライアントが情報を要求するたびにサーバーからクライアントにイベントを送信するために役立ちます。

SSE接続の作成

SSE接続は通常のHTTPリクエストのように動作しますが、レスポンスを送信してすぐに接続を閉じるのではなく、メッセージを送信するために接続が開いたままになります。

[HttpRequest.GetEventSource\(\)](#)メソッドを呼び出すと、リクエストは待機状態になり、SSEインスタンスが作成されます。

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

```

上記のコードでは、SSE接続を作成し、「こんにちは、世界！」というメッセージを送信し、サーバー側からSSE接続を閉じます。

NOTE

サーバー側で接続を閉じると、クライアントはデフォルトで再接続を試み、その接続は再開され、メソッドが再度実行されます。

クライアントが再接続しないように、サーバーが接続を閉じたときに終了メッセージを転送することが一般的です。

ヘッダーの追加

ヘッダーを送信する必要がある場合は、メッセージを送信する前に[HttpRequestEventSource.AppendHeader](#)メソッドを使用できます。

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3     using var sse = req.GetEventSource();
4     sse.AppendHeader("ヘッダーキー", "ヘッダー値");
5
6     sse.Send("こんにちは！");
7
8     return sse.Close();
9   });

```

ヘッダーを送信する前にメッセージを送信しないことが必要です。

待機-失敗接続

接続は通常、クライアント側の切断によりメッセージを送信できなくなると終了します。そのため、接続は自動的に終了し、クラスのインスタンスは破棄されます。

再接続しても、クラスのインスタンスは機能しません。以前の接続にリンクしているからです。場合によっては、この接続を後で必要とし、ルートのコールバックメソッドを介して管理したくことがあります。

このため、識別子でSSE接続を識別し、後でそれを使用して、ルートのコールバック外でも取得できます。さらに、[WaitForFail](#)で接続をマークして、ルートを終了せず、接続を自動的に終了させません。

KeepAliveのSSE接続は、切断による送信エラーが発生するまでメソッドの実行を待機します。タイムアウトを設定することもできます。一定時間内にメッセージが送信されなかった場合、接続は終了し、実行が再開されます。

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3     using var sse = req.GetEventSource("私のインデックス接続");
4
5     sse.WaitForFail(TimeSpan.FromSeconds(15)); // 15秒間メッセージが送信されなかった場合に接続を終了
6
7     return sse.Close();
8   });

```

上記のメソッドは、接続を作成し、処理し、切断またはエラーを待ちます。

```
1  HttpRequestEventSource? evs = server.EventSources.GetByIdentifier("私のインデックス接続");
2  if (evs != null)
3  {
4    // 接続はまだアクティブです
5    evs.Send("もう一度こんにちは！");
6  }

```

上記のコードは、新しく作成された接続を探し、存在する場合にメッセージを送信します。

識別されたアクティブなサーバー接続はすべて `HttpServer.EventSources` コレクションで利用できます。このコレクションには、アクティブで識別された接続のみが保存されます。閉じられた接続はコレクションから削除されます。

NOTE

アイドル接続を一定期間後に閉じるコンポーネント（Webプロキシ、HTTPカーネル、ネットワークドライバーなど）によって、キープアライブには制限があることに注意することが重要です。

したがって、定期的なpingを送信するか、接続が閉じられるまでの最大時間を延長して、接続を開いたままにすることが重要です。次のセクションを読んで、定期的なpingの送信について理解を深めてください。

接続のpingポリシーの設定

Pingポリシーは、クライアントに定期的なメッセージを送信する自動化された方法です。この機能により、サーバーはクライアントが接続から切断したかどうかを理解できます。

```

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

```

上記のコードでは、5秒ごとに新しいpingメッセージがクライアントに送信されます。これにより、TCP接続がアクティブなままになり、非アクティブにより接続が閉じられることがなくなります。また、メッセージの送信に失敗した場合、接続は自動的に閉じられ、接続で使用されるリソースが解放されます。

接続のクエリ

識別子に基づく述語を使用してアクティブな接続を検索することができます。たとえば、ブロードキャストなどです。

```

1 HttpRequestEventSource[] evs = server.EventSources.Find(es => es.StartsWith("私の接続-"));
2 foreach (HttpRequestEventSource e in evs)
3 {
4     e.Send("私の接続-で始まるすべてのイベントソースへのブロードキャスト");
5 }

```

Allメソッドを使用して、すべてのアクティブなSSE接続を取得することもできます。

Web Sockets

Sisk は WebSocket もサポートしており、クライアントへのメッセージの送受信が可能です。

この機能はほとんどのブラウザで正常に動作しますが、Sisk ではまだ実験段階です。バグを発見した場合は GitHub で報告してください。

非同期でメッセージを受信する

WebSocket のメッセージは順序通りに受信され、`ReceiveMessageAsync` で処理されるまでキューに入れられます。このメソッドは、タイムアウトに達したとき、操作がキャンセルされたとき、またはクライアントが切断されたときにメッセージを返しません。

読み取りと書き込みは同時に1つしか実行できないため、`ReceiveMessageAsync` でメッセージを待っている間は、接続されたクライアントに書き込むことはできません。

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

同期でメッセージを受信する

以下の例では、非同期コンテキストを使わずに同期的に WebSocket を使用し、メッセージを受信し、処理し、ソケットの使用を終了する方法を示しています。

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

});
```

Ping ポリシー

Server Side Events の ping ポリシーと同様に、TCP 接続に不活動がある場合に接続を維持するために ping ポリシーを設定できます。

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

Discard syntax

HTTPサーバーは、OAuth認証などのアクションからのコールバック要求を待ち受けるために使用でき、要求を受け取った後には破棄できます。これは、バックグラウンドアクションが必要だが、HTTPアプリケーションを設定たくない場合に便利です。

以下の例は、[CreateListener](#)を使用してポート5555でリスニングHTTPサーバーを作成し、次のコンテキストを待つ方法を示しています。

```
1  using (var server = HttpServer.CreateListener(5555))
2  {
3      // 次のHTTP要求を待つ
4      var context = await server.WaitNextAsync();
5      Console.WriteLine($"要求されたパス: {context.Request.Path}");
6  }
```

[WaitNext](#)関数は、完了した要求処理の次のコンテキストを待ちます。この操作の結果が取得されると、サーバーはすでに要求を完全に処理し、クライアントに応答を送信しています。

依存性注入

リクエストの生存期間中に存在するメンバーとインスタンスを専用にすることは一般的です。たとえば、データベース接続、認証されたユーザー、またはセッショントークンなどです。可能性の1つは、[HttpContext.RequestBag](#)を使用することです。これは、リクエストの生存期間中に存在する辞書を作成します。

この辞書は、[リクエスト ハンドラー](#)によってアクセスされ、リクエスト全体で変数を定義できます。たとえば、ユーザーを認証するリクエストハンドラーは、[HttpContext.RequestBag](#) 内にユーザーを設定し、リクエスト ロジック内では、[HttpContext.RequestBag.Get<User>\(\)](#) でユーザーを取得できます。

以下は例です。

RequestHandlers/AuthenticateUser.cs

C#

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

Controllers/HelloController.cs

C#

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

これは、この操作の初期的な例です。User のインスタンスは、認証用のリクエスト ハンドラー内で作成されました。AuthenticateUser リクエスト ハンドラーを使用するすべてのルートには、HttpContext.RequestBag 内に User が存在することが保証されます。

RequestBag 内に事前に定義されていないインスタンスを取得するロジックを定義するには、[GetOrAdd](#) または [GetOrAddAsync](#) のようなメソッドを使用できます。

バージョン 1.3 以降、静的プロパティ `HttpContext.Current` が導入され、現在実行中のリクエスト コンテキストの `HttpContext` にアクセスできるようになりました。これにより、リクエスト外部で `HttpContext` のメンバーにアクセスし、ルート オブジェクト内でインスタンスを定義できるようになりました。

以下の例では、リクエスト コンテキストで一般的にアクセスされるメンバーを持つコントローラーを定義します。

Controllers/Controller.cs

C#

```
1  public abstract class Controller : RouterModule
2  {
3      public DbContext Database
4      {
5          get
6          {
7              // DbContext を作成または既存のものを取得
8              return HttpContext.Current.RequestBag.GetOrAdd(() => new DbContext());
9          }
10     }
11
12     // 次の行は、プロパティがリクエスト バッグ内に定義されていない場合に例外をスローします
13     public User AuthenticatedUser { get => HttpContext.Current.RequestBag.Get<User>(); }
14
15     // HttpRequest インスタンスの公開もサポートされます
16     public HttpRequest Request { get => HttpContext.Current.Request; }
17 }
```

コントローラーから継承するタイプを定義します。

Controllers/PostsController.cs

C#

```
1  [RoutePrefix("/api/posts")]
2  public class PostsController : Controller
3  {
4      [RouteGet]
5      public IEnumerable<Blog> ListPosts()
6      {
7          return Database.Posts
```

```

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

```

上記の例では、ルーターに [値ハンドラー](#) を構成する必要があります。ルーターによって返されるオブジェクトが有効な [HttpResponse](#) に変換されるようにします。

メソッドに `HttpRequest request` 引数がないことに注意してください。これは、バージョン 1.3 以降、ルーターがルーティング応答の 2 つの種類のデリゲートをサポートしているためです。1 つは、デフォルトのデリゲートである [RouteAction](#) で、`HttpRequest` 引数を受け取ります。もう 1 つは、[ParameterlessRouteAction](#) です。

`HttpRequest` オブジェクトは、[静的な](#) `HttpContext` の [Request](#) プロパティを介して、両方のデリゲートからアクセスできます。

上記の例では、[破棄可能](#)なオブジェクトである `DbContext` を定義しました。HTTP セッションが終了するときに、`DbContext` のすべてのインスタンスが破棄されることを確認する必要があります。これを実現するには、2 つの方法があります。1 つは、ルーターのアクションの後に実行される [リクエストハンドラー](#) を作成することです。もう 1 つは、カスタム [サーバーハンドラー](#) を使用することです。

最初の方法では、`RouterModule` から継承される [OnSetup](#) メソッド内に直接リクエストハンドラーをインラインで作成できます。

Controllers/PostsController.cs

C#

```

1  public abstract class Controller : RouterModule
2  {
3      // ...
4
5      protected override void OnSetup(Router parentRouter)
6      {
7          base.OnSetup(parentRouter);
8
9          HasRequestHandler(RequestHandler.Create(

```

```
10         execute: (req, ctx) =>
11         {
12             // リクエスト ハンドラー コンテキスト内に定義された DbContext を取得し、破棄します
13             ctx.RequestBag.GetOrDefault<DbContext>()?.Dispose();
14             return null;
15         },
16         executionMode: RequestHandlerExecutionMode.AfterResponse));
17     }
18 }
```

① TIP

Sisk バージョン 1.4 以降、プロパティ [HttpServerConfiguration.DisposeDisposableContextValues](#) が導入され、デフォルトで有効になりました。これは、HTTP セッションが閉じられたときに、コンテキスト バッグ内のすべての [IDisposable](#) 値を破棄するかどうかを定義します。

上記の方法では、HTTP 応答が終了したときに [DbContext](#) が破棄されることを保証します。他のメンバーも破棄する必要がある場合は、同様の方法を使用できます。

2 番目の方法では、HTTP セッションが終了したときに [DbContext](#) を破棄するカスタム サーバー ハンドラーを作成できます。

Server/Handlers/ObjectDisposerHandler.cs

C#

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

そして、アプリケーション ビルダーで使用します。

Program.cs

C#

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

これは、コードのクリーンアップを処理し、リクエストの依存関係を使用されるモジュールの種類によって分離する方法です。これは、ASP.NET のようなフレームワークで依存性注入が使用されるのと似た方法です。

コンテンツのストリーミング

Sisk では、クライアントとサーバー間でコンテンツのストリーミングを読み書きすることができます。この機能は、リクエストの生存期間中にコンテンツのシリアル化とデシリアル化のメモリ負荷を削減するために役立ちます。

リクエストコンテンツストリーム

小さなコンテンツは自動的に HTTP 接続バッファメモリに読み込まれ、[HttpRequest.Body](#) と [HttpRequest.RawBody](#) に迅速に読み込まれます。より大きなコンテンツの場合は、[HttpRequest.GetRequestStream](#) メソッドを使用してリクエストコンテンツ読み取りストリームを取得できます。

[HttpRequest.GetMultipartFormContent](#) メソッドは、リクエスト全体のコンテンツをメモリに読み込むため、大きなコンテンツを読み取るには適していないことに注意してください。

以下の例を考えてみましょう：

Controller/UploadDocument.cs

C#

```
1  [RoutePost ( "/api/upload-document/<filename>" )]
2  public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4      var fileName = request.RouteParameters [ "filename" ].GetString ();
5
6      if ( !request.HasContents ) {
7          // リクエストにコンテンツがありません
8          return new HttpResponse ( HttpStatusInformation.BadRequest );
9      }
10
11      var contentStream = request.GetRequestStream ();
12      var outputFileName = Path.Combine (
13          AppDomain.CurrentDomain.BaseDirectory,
14          "uploads",
15          fileName );
16
17      using ( var fs = File.Create ( outputFileName ) ) {
18          await contentStream.CopyToAsync ( fs );
```

```

19     }
20
21     return new HttpResponseMessage() {
22         Content = JsonContent.Create(new { message = "ファイルが正常に送信されました。" } )
23     };
24 }
```

上記の例では、`UploadDocument` メソッドはリクエストコンテンツを読み取り、コンテンツをファイルに保存します。`Stream.CopyToAsync` で使用される読み取りバッファ以外に、追加のメモリ割り当ては行われません。上記の例は、非常に大きなファイルの場合にメモリ割り当ての負担を削減し、アプリケーションのパフォーマンスを最適化するのに役立ちます。

良い実践は、ファイルの送信などの時間のかかる操作では、常に [CancellationToken](#) を使用することです。これは、クライアントとサーバーの間のネットワーク速度に依存するためです。

`CancellationToken` での調整は、以下のように行うことができます:

Controller/UploadDocument.cs

C#

```

1 // 30 秒のタイムアウトに達した場合、以下のキャンセルトークンは例外をスローします。
2 CancellationTokenSource copyCancellation = new CancellationTokenSource( delay:
3 TimeSpan.FromSeconds( 30 ) );
4
5 try {
6     using (var fs = File.Create( outputFileName )) {
7         await contentStream.CopyToAsync( fs, copyCancellation.Token );
8     }
9 }
10 catch (OperationCanceledException) {
11     return new HttpResponseMessage( HttpStatusCode.BadRequest ) {
12         Content = JsonContent.Create( new { Error = "アップロードが最大アップロード時間（30 秒）" +
13 "を超えました。" } );
14     };
15 }
```

レスポンスコンテンツストリーム

レスポンスコンテンツを送信することも可能です。現在、[HttpRequest.GetResponseStream](#) メソッドと [StreamContent](#) タイプのコンテンツを使用するという 2 つの方法があります。

画像ファイルを提供するシナリオを考えてみましょう。以下のコードを使用できます:

Controller/ImageController.cs

C#

```
1 [RouteGet ( "/api/profile-picture" )]
2 public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4     // プロファイル画像を取得するための例メソッド
5     var profilePictureFilename = "profile-picture.jpg";
6     byte[] profilePicture = await File.ReadAllBytesAsync ( profilePictureFilename );
7
8     return new HttpResponse () {
9         Content = new ByteArrayContent ( profilePicture ),
10        Headers = new () {
11            ContentType = "image/jpeg",
12            ContentDisposition = $"inline; filename={profilePictureFilename}"
13        }
14    };
15 }
```

上記のメソッドは、画像コンテンツを読み取るたびにメモリ割り当てを行います。画像が大きい場合、これによりパフォーマンスの問題が発生し、ピーク時にはメモリオーバーロードによりサーバーがクラッシュする可能性があります。このような状況では、キャッシングは役立ちますが、ファイルのためにメモリが依然として予約されるため、問題を完全に解決することはできません。キャッシングは、毎回メモリを割り当てる必要性の圧力を軽減するのに役立ちますが、大きなファイルの場合は十分ではありません。

画像をストリームで送信することが問題の解決策となります。画像コンテンツ全体を読み取るのではなく、ファイル上で読み取りストリームを作成し、クライアントに小さなバッファを使用してコピーします。

GetResponseStream メソッドを使用した送信

`HttpRequest.GetResponseStream` メソッドは、HTTP 応答のコンテンツフローが準備されるにつれて、HTTP 応答のチャunkを送信できるオブジェクトを作成します。このメソッドはより手動で、コンテンツを送信する前にステータス、ヘッダー、コンテンツサイズを定義する必要があります。

Controller/ImageController.cs

C#

```
1 [RouteGet ( "/api/profile-picture" )]
2 public async Task<HttpResponse> UploadDocument ( HttpRequest request ) {
3
4     var profilePictureFilename = "profile-picture.jpg";
5
6     // この形式の送信では、ステータスとヘッダーを事前に定義する必要があります。
```

```

7     var requestStreamManager = request.GetResponseStream ();
8
9     requestStreamManager.SetStatus ( System.Net.HttpStatusCode.OK );
10    requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentType, "image/jpeg" );
11    requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentDisposition, $"inline;
12    filename={profilePictureFilename}" );
13
14    using (var fs = File.OpenRead ( profilePictureFilename )) {
15
16        // この形式の送信では、コンテンツサイズも事前に定義する必要があります。
17        requestStreamManager.SetContentLength ( fs.Length );
18
19        // コンテンツサイズがわからない場合は、チャunk化されたエンコードを使用してコンテンツを送信でき
20        ます。
21        requestStreamManager.SendChunked = true;
22
23        // その後、出力ストリームに書き込みます。
24        await fs.CopyToAsync ( requestStreamManager.ResponseStream );
25    }
26}

```

StreamContent を使用したコンテンツの送信

[StreamContent](#) クラスを使用すると、データソースからバイトストリームとしてコンテンツを送信できます。この形式の送信は、以前の要件を削除し、[圧縮エンコード](#) を使用してコンテンツサイズを削減することもできます。

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 }

```

✖️ IMPORTANT

このタイプのコンテンツでは、ストリームを `using` ブロックで囲むことは避けてください。コンテンツフローが終了すると、HTTP サーバーによってコンテンツが自動的に破棄されます。

SiskでCORS（クロスオリジンリソース共有）を有効にする

Siskには、サービスを公開する際に[クロスオリジンリソース共有 \(CORS\)](#) を扱うのに便利なツールがあります。この機能はHTTPプロトコルの一部ではなく、W3Cによって定義されたWebブラウザ固有の機能です。このセキュリティメカニズムは、WebページがWebページを提供したドメインとは異なるドメインへのリクエストを行うことを防止します。サービスプロバイダーは、特定のドメインにリソースへのアクセスを許可するか、あるいは単一のドメインのみを許可することができます。

同一オリジン

リソースが「同一オリジン」として識別されるには、リクエストがOriginヘッダーをリクエストに含める必要があります。

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

そしてリモートサーバーは、リクエストされたオリジンと同じ値を持つAccess-Control-Allow-Originヘッダーで応答しなければなりません。

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

この検証は明示的です：ホスト、ポート、プロトコルはリクエストされたものと同じでなければなりません。例を確認してください。

- サーバーがAccess-Control-Allow-Originをhttps://example.comと応答する場合：
 - https://example.net - ドメインが異なる。
 - http://example.com - スキームが異なる。
 - http://example.com:5555 - ポートが異なる。

- <https://www.example.com> - ホストが異なる。

仕様では、リクエストとレスポンスの両方のヘッダーに対して構文のみが許可されます。URL パスは無視されます。ポートはデフォルトポート（HTTP は 80、HTTPS は 443）である場合は省略されます。

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

CORS を有効にする

ネイティブに、[ListeningHost](#) 内に [CrossOriginResourceSharingHeaders](#) オブジェクトがあります。

サーバーを初期化するときに CORS を構成できます。

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

上記のコードは [すべてのレスポンス](#) に対して次のヘッダーを送信します。

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

これらのヘッダーは、エラーやリダイレクトを含むすべてのレスポンスに対して送信する必要があります。

[CrossOriginResourceSharingHeaders](#) クラスには、[AllowOrigin](#) と [AllowOrigins](#) の 2 つの似たプロパティがあります。1つは複数形、もう1つは単数形です。

- **AllowOrigin** プロパティは静的です：指定したオリジンのみがすべてのレスポンスで送信されます。
- **AllowOrigins** プロパティは動的です：サーバーはリクエストのオリジンがこのリストに含まれているかを確認します。見つかった場合、そのオリジンのレスポンスに送信されます。

ワイルドカードと自動ヘッダー

代わりに、レスポンスのオリジンにワイルドカード（*）を使用して、任意のオリジンがリソースにアクセスできるように指定できます。ただし、この値は認証情報（認証ヘッダー）を持つリクエストには許可されず、エラーが発生します。

この問題を回避するには、[AllowOrigins](#) プロパティで許可されるオリジンを明示的に列挙するか、[AllowOrigin](#) の値に [AutoAllowOrigin](#) 定数を使用します。このマジックプロパティは、リクエストの `Origin` ヘッダーと同じ値で `Access-Control-Allow-Origin` ヘッダーを定義します。

また、[AllowOrigin](#) と同様の動作を自動で行う [AutoFromRequestMethod](#) と [AutoFromRequestHeaders](#) を使用できます。

```

1  using var host = HttpServer.CreateBuilder()
2      .UseCors(new CrossOriginResourceSharingHeaders(
3
4          // リクエストの Origin ヘッダーに基づいて応答
5          allowOrigin: CrossOriginResourceSharingHeaders.AutoAllowOrigin,
6
7          // Access-Control-Request-Method ヘッダーまたはリクエストメソッドに基づいて応答
8          allowMethods: [CrossOriginResourceSharingHeaders.AutoFromRequestMethod],
9
10         // Access-Control-Request-Headers ヘッダーまたは送信されたヘッダーに基づいて応答
11         allowHeaders: [CrossOriginResourceSharingHeaders.AutoFromRequestHeaders]))
```

CORS を適用する他の方法

[サービスプロバイダー](#) を扱っている場合、設定ファイルで定義された値を上書きできます。

```

1  static async Task Main(string[] args)
2  {
3      using var app = HttpServer.CreateBuilder()
4          .UsePortableConfiguration( ... )
```

```
5     .UseCors(cors => {
6         // 設定ファイルで定義されたオリジンを上書き
7         cors.AllowOrigin = "http://example.com";
8     })
9     .Build();
10
11     await app.StartAsync();
12 }
```

特定のルートで CORS を無効にする

`UseCors` プロパティはルートとすべてのルート属性で利用可能で、次の例のように無効にできます。

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

レスポンス内の値を置き換える

ルーターアクションで明示的に値を置き換えたり削除したりできます。

```
1 [RoutePrefix("api/widgets")]
2 public class WidgetController : Controller {
3
4     public IEnumerable<string> GetWidgets(HttpRequest request) {
5
6         // Access-Control-Allow-Credentials ヘッダーを削除
7         request.Context.OverrideHeaders.AccessControlAllowCredentials = string.Empty;
8     }
9 }
```

```
9      // Access-Control-Allow-Origin を置き換え
10     request.Context.OverrideHeaders.AccessControlAllowOrigin = "https://contorso.com";
11
12     return new[] { "Green widget", "Red widget" };
13 }
14 }
```

プリフライトリクエスト

プリフライトリクエストは、実際のリクエストの前にクライアントが送信する [OPTIONS](#) メソッドリクエストです。

Sisk サーバーは常に `200 OK` と適用される CORS ヘッダーで応答し、クライアントは実際のリクエストを続行できます。この条件は、`Options` に対して明示的に構成された [RouteMethod](#) を持つルートが存在する場合にのみ適用されません。

CORS をグローバルに無効にする

これは不可能です。CORS を使用しない場合は、構成しないでください。

JSON-RPC 拡張

Sisk には、よりシンプルなアプリケーションを作成できる実験的な JSON-RPC 2.0 API 用のモジュールがあります。この拡張機能は、JSON-RPC 2.0 トランスポート インターフェイスを厳密に実装し、HTTP GET、POST リクエスト、以及 Sisk での Web ソケットを介したトランスポートを提供します。

以下のコマンドを使用して、Nuget を介して拡張機能をインストールできます。実験/ベータ バージョンの場合は、Visual Studio でプレリリース パッケージの検索を有効にする必要があります。

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

トランスポート インターフェイス

JSON-RPC は、状態を保持しない非同期リモート手続き呼び出し (RDP) プロトコルで、JSON を使用して一方向のデータ通信を行います。JSON-RPC リクエストは、通常、ID で識別され、レスポンスはリクエストで送信された同じ ID で配信されます。すべてのリクエストがレスポンスを必要とするわけではありません。これらは「通知」と呼ばれます。

[JSON-RPC 2.0仕様](#) では、トランスポートの詳細について説明しています。このトランスポートは、使用される場所に依存しません。Sisk は、このプロトコルを HTTP を介して実装し、[JSON-RPC over HTTP](#) に準拠しています。これは、GET リクエストを部分的にサポートし、POST リクエストを完全にサポートします。Web ソケットもサポートされており、非同期メッセージ通信を提供します。

JSON-RPC リクエストは次のようにになります。

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

そして、成功したレスポンスは次のようになります:

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

JSON-RPC メソッド

以下の例は、Sisk を使用して JSON-RPC API を作成する方法を示しています。数学演算クラスはリモート演算を実行し、シリアル化されたレスポンスをクライアントに配信します。

Program.cs

C#

```
1  using var app = HttpServer.CreateBuilder(port: 5555)
2      .UseJsonRPC((sender, args) =>
3      {
4          // WebMethod 属性が付いたすべてのメソッドを JSON-RPC ハンドラーに追加します
5          args.Handler.Methods.AddMethodsFromType(new MathOperations());
6
7          // /service ルートを JSON-RPC の POST および GET リクエストのハンドラーにマップします
8          args.Router.MapPost("/service", args.Handler.Transport.HttpPost);
9          args.Router.MapGet("/service", args.Handler.Transport.HttpGet);
10
11         // GET /ws に WebSocket ハンドラーを作成します
12         args.Router.MapGet("/ws", request =>
13         {
14             var ws = request.GetWebSocket();
15             ws.OnReceive += args.Handler.Transport.WebSocket;
16
17             ws.WaitForClose(timeout: TimeSpan.FromSeconds(30));
18             return ws.Close();
19         });
20     })
21     .Build();
22
23 await app.StartAsync();
```

MathOperations.cs

C#

```

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

上記の例では、`Sum` と `Sqrt` メソッドを JSON-RPC ハンドラーにマップし、これらのメソッドは GET /service、POST /service、および GET /ws で利用可能になります。メソッド名は大文字と小文字を区別しません。

メソッドのパラメーターは自動的に特定の型にデシリアライズされます。名前付きパラメーターを使用したリクエストもサポートされています。JSON シリアライズは、[LightJson](#) ライブラリによって実行されます。型が正しくデシリアライズされない場合は、その型用に特定の [JSON コンバーター](#) を作成し、後でそれを [JsonSerializerOptions](#) に関連付けることができます。

また、JSON-RPC リクエストから直接 `$.params` の生のオブジェクトをメソッドで取得することもできます。

MathOperations.cs

C#

```

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

これが発生するには、`@params` がメソッドの唯一のパラメーターで、正確に `params` (C# では `@` でエスケープする必要があります) という名前でなければなりません。

パラメーターのデシリアライズは、名前付きオブジェクトまたは位置指定配列の両方で発生します。たとえば、次のメソッドは、両方のリクエストでリモートで呼び出することができます。

```

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

```
4     ...
5 }
```

配列の場合、パラメーターの順序に従う必要があります。

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

シリアルライザーカスタマイズ

`JsonRpcHandler.JsonSerializerOptions` プロパティで JSON シリアルライザーカスタマイズできます。このプロパティでは、メッセージのデシリアライズに [JSON5](#) を使用できるようにすることができます。JSON-RPC 2.0 との準拠ではありませんが、JSON5 は、人間が読み書きしやすい JSON の拡張です。

Program.cs

C#

```
1 using var host = HttpServer.CreateBuilder( 5556 )
2     .UseJsonRPC( ( o, e ) => {
3
4         // 名前比較子を使用して、名前の比較を実行します。
5         // この比較子では、名前の文字と数字のみを比較し、他のシンボルは無視されます。
6         // 例:
7         // foo_bar10 = FooBar10
8         e.Handler.JsonSerializerOptions.PropertyNameComparer = new
9         JsonSanitizedComparer();
10    }
```

```
11      // JSON5 を JSON インタープリターで有効にします。
12      // これを有効にした場合でも、プレーン JSON はまだ許可されます。
13      e.Handler.JsonSerializerOptions.SerializationFlags =
14      LightJson.Serialization.JsonSerializationFlags.Json5;
15
16      // POST /service ルートを JSON-RPC ハンドラーにマップします
17      e.Router.MapPost ( "/service" , e.Handler.Transport.HttpPost );
18  }
19 .Build ();
```

host.Start();

SSL Proxy

① WARNING

この機能は実験的であり、運用環境では使用しないでください。Sisk を SSL で動作させる方法については、[このドキュメント](#)を参照してください。

Sisk SSL Proxy は、Sisk の [ListeningHost](#) に HTTPS 接続を提供し、HTTPS メッセージを非安全な HTTP コンテキストにルーティングするモジュールです。このモジュールは、SSL をサポートしない [HttpListener](#) を使用して実行されるサービスに SSL 接続を提供するために構築されました。

プロキシは同じアプリケーション内で実行され、HTTP/1.1 メッセージをリッスンし、同じプロトコルで Sisk に転送します。現在、この機能は非常に実験的であり、運用環境で使用するには不安定すぎる可能性があります。

現在、SslProxy は、キープアライブ、チャunk化されたエンコード、WebSockets など、ほとんどの HTTP/1.1 機能をサポートしています。SSL プロキシへのオープン接続の場合、ターゲット サーバーに TCP 接続が作成され、プロキシは確立された接続に転送されます。

SslProxy は、次のように [HttpServer.CreateBuilder](#) と共に使用できます。

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

プロキシに有効な SSL 証明書を提供する必要があります。ブラウザによって証明書が受け入れられるようにするには、オペレーティング システムに証明書をインポートして、正しく機能するようにします。

Basic Auth

Basic Auth パッケージは、非常に少ない設定と労力で、Sisk アプリケーションで基本認証スキームを処理できるリクエストハンドラーを追加します。Basic HTTP 認証は、ユーザー ID とパスワードでリクエストを認証する最小限の入力形式であり、セッションはクライアントによって完全に制御され、認証またはアクセストークンはありません。



Basic 認証スキームについては、[MDN の仕様](#) を参照してください。

インストール

開始するには、プロジェクトに Sisk.BasicAuth パッケージをインストールします:

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

プロジェクトにインストールする他の方法については、[NuGet リポジトリ](#) を参照してください。

認証ハンドラーの作成

認証スキームをモジュール全体または個々のルートに対して制御できます。まず、基本認証ハンドラーを作成しましょう。

以下の例では、データベースに接続して、ユーザーが存在し、パスワードが有効であるかどうかを確認し、次にユーザーをコンテキスト バッグに保存します。

```
1  public class UserAuthHandler : BasicAuthenticateRequestHandler
2  {
3      public UserAuthHandler() : base()
4      {
5          Realm = "このページにアクセスするには、資格情報を入力してください。";
```

```

6      }
7
8      public override HttpResponseMessage? OnValidating(BasicAuthenticationCredentials credentials,
9      HttpContext context)
10     {
11         DbContext db = new DbContext();
12
13         // この場合、ユーザー ID フィールドとして電子メールを使用しているため、電子メールでユーザーを検索し
14         // ます。
15         User? user = db.Users.FirstOrDefault(u => u.Email == credentials.UserId);
16         if (user == null)
17         {
18             return base.CreateUnauthorizedResponse("ユーザーが見つかりませんでした。");
19         }
20
21         // 資格情報のパスワードがこのユーザーに対して有効であることを確認します。
22         if (!user.ValidatePassword(credentials.Password))
23         {
24             return base.CreateUnauthorizedResponse("資格情報が無効です。");
25         }
26
27         // ログインしたユーザーを HTTP コンテキストに追加し、実行を続行します。
28         context.Bag.Add("loggedUser", user);
29         return null;
30     }
31 }

```

これで、このリクエスト ハンドラーをルートまたはクラスに関連付けるだけです。

```

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

```

または、**RouterModule** クラスを使用することもできます:

```

1  public class UsersController : RouterModule
2  {
3      public ClientModule()

```

```
4     {
5         // このクラス内のすべてのルートは、UserAuthHandler によって処理されます。
6         base.HasRequestHandler(new UserAuthHandler());
7     }
8
9     [RouteGet("/")]
10    public string Index(HttpContext request)
11    {
12        User loggedUser = (User)request.Context.RequestBag["loggedUser"];
13        return "こんにちは、" + loggedUser.Name + "!";
14    }
15 }
```

備考

基本認証の主な責任はクライアント側で実行されます。ストレージ、キャッシュ制御、暗号化はすべてクライアント側でローカルに処理され、サーバーは資格情報を受け取り、許可されたアクセスかどうかを検証するだけです。

この方法は、クライアントに大きな責任を負わせるため、セキュリティの面で最も安全な方法ではありません。さらに、パスワードは SSL のようなセキュアな接続コンテキストで送信される必要があります。なぜなら、パスワードには固有の暗号化がないからです。リクエストのヘッダーを一時的に傍受するだけで、ユーザーのアクセス資格情報が公開される可能性があります。

本番環境のアプリケーションでは、より堅牢な認証ソリューションを選択し、オフザシェルフのコンポーネントを使用しすぎないようにしてください。そうしないと、プロジェクトがセキュリティリスクにさらされる可能性があります。

サービス プロバイダー

サービス プロバイダーは、Sisk アプリケーションをさまざまな環境に移植するための方法です。ポータブルな構成ファイルを使用して、サーバーのポート、パラメーター、他のオプションを変更できます。各環境ごとにアプリケーション コードを変更する必要はありません。このモジュールは、Sisk の構築構文に依存し、`UsePortableConfiguration` メソッドを使用して構成できます。

構成プロバイダーは、`IConfigurationProvider` を実装して構成リーダーを提供し、任意の実装を受け取ることができます。デフォルトでは、Sisk では JSON 構成リーダーが提供されますが、INI ファイル用のパッケージもあります。独自の構成プロバイダーを作成し、次のように登録することもできます。

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

前述のように、デフォルトのプロバイダーは JSON ファイルです。デフォルトでは、`service-config.json` という名前のファイルが検索され、実行中のプロセスのカレント ディレクトリで検索されますが、実行可能ファイルのディレクトリではありません。

ファイル名を変更したり、Sisk が構成ファイルを検索する場所を指定したりすることもできます。

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

上記のコードは、実行中のプロセスのカレント ディレクトリで `config.toml` ファイルを検索します。如果見つからない場合は、実行可能ファイルのディレクトリで検索します。如果ファイルが存在しない場合、`createIfDontExists`

パラメーターが尊重され、最後にテストされたパス（`lookupDirectories` に基づく）にファイルが作成され、エラーがコンソールに出力され、アプリケーションの初期化が阻止されます。

(i) TIP

INI 構成リーダーと JSON 構成リーダーのソースコードを参照して、`IConfigurationProvider` がどのように実装されるかを理解することができます。

JSON ファイルからの構成の読み取り

デフォルトでは、`Sisk` では JSON ファイルから構成を読み取る構成プロバイダーが提供されます。このファイルは固定構造を持ち、次のパラメーターで構成されます。

```
1  {
2      "Server": {
3          "DefaultEncoding": "UTF-8",
4          "ThrowExceptions": true,
5          "IncludeRequestIdHeader": true
6      },
7      "ListeningHost": {
8          "Label": "My sisk application",
9          "Ports": [
10             "http://localhost:80/",
11             "https://localhost:443/" // 構成ファイルもコメントをサポートします
12         ],
13         "CrossOriginResourceSharingPolicy": {
14             "AllowOrigin": "*",
15             "AllowOrigins": [ "*" ], // 0.14 で新しく追加されました
16             "AllowMethods": [ "*" ],
17             "AllowHeaders": [ "*" ],
18             "MaxAge": 3600
19         },
20         "Parameters": {
21             "MySqlConnection": "server=localhost;user=root;"
22         }
23     }
24 }
```

構成ファイルから作成されたパラメーターは、サーバーのコンストラクターでアクセスできます。

```

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

```

各構成リーダーは、サーバーの初期化パラメーターを読み取る方法を提供します。プロセス環境で定義する必要がある一部のプロパティ（機密性の高い API データ、API キーなど）は、構成ファイルに定義するのではなく、プロセス環境で定義する必要があります。

構成ファイルの構造

JSON 構成ファイルは、次のプロパティで構成されます。

プロパティ	必須	説明
Server	必須	サーバー自身とその設定を表します。
Server.AccessLogsStream	省略可能	デフォルトは <code>console</code> 。アクセスログの出力ストリームを指定します。ファイル名、 <code>null</code> 、または <code>console</code> のいずれかになります。
Server.ErrorsLogsStream	省略可能	デフォルトは <code>null</code> 。エラー ログの出力ストリームを指定します。ファイル名、 <code>null</code> 、または <code>console</code> のいずれかになります。
Server.MaximumContentLength	省略可能	
Server.MaximumContentLength	省略	デフォルトは <code>0</code> 。コンテンツの最大長をバイト単位で指定します。 <code>0</code>

プロパティ	必須 可能	説明
Server.IncludeRequestIdHeader	可能	は無制限を意味します。
Server.ThrowExceptions	省略可能	デフォルトは <code>false</code> 。HTTP サーバーが <code>X-Request-Id</code> ヘッダーを送信するかどうかを指定します。
ListeningHost	必須	サーバーのリスニング ホストを表します。
ListeningHost.Label	省略可能	アプリケーションのラベルを表します。
ListeningHost.Ports	必須	<code>ListeningPort</code> 構文に一致する文字列の配列を表します。
ListeningHost.CrossOriginResourceSharingPolicy	省略可能	アプリケーションの CORS ヘッダーを設定します。
ListeningHost.CrossOriginResourceSharingPolicy.AllowCredentials	省略可能	デフォルトは <code>false</code> 。 <code>Allow-Credentials</code> ヘッダーを指定します。
ListeningHost.CrossOriginResourceSharingPolicy.ExposeHeaders	省略可能	デフォルトは <code>null</code> 。文字列の配列を期待します。 <code>Expose-Headers</code> ヘッダーを指定します。
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigin	省略可能	デフォルトは <code>null</code> 。文字列を期待します。 <code>Allow-Origin</code> ヘッダーを指定します。

プロパティ	必須	説明
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigins	省略可能	デフォルトは <code>null</code> 。文字列の配列を期待します。複数の <code>Allow-Origin</code> ヘッダーを指定します。詳細については、 AllowOrigins を参照してください。
ListeningHost.CrossOriginResourceSharingPolicy.AllowMethods	省略可能	デフォルトは <code>null</code> 。文字列の配列を期待します。 <code>Allow-Methods</code> ヘッダーを指定します。
ListeningHost.CrossOriginResourceSharingPolicy.AllowHeaders	省略可能	デフォルトは <code>null</code> 。文字列の配列を期待します。 <code>Allow-Headers</code> ヘッダーを指定します。
ListeningHost.CrossOriginResourceSharingPolicy.MaxAge	省略可能	デフォルトは <code>null</code> 。整数を期待します。 <code>Max-Age</code> ヘッダーを秒単位で指定します。
ListeningHost.Parameters	省略可能	アプリケーションの設定メソッドに提供されるプロパティを指定します。

INI 構成プロバイダー

Sisk には、JSON 以外の起動構成を取得する方法があります。実際には、 [IConfigurationReader](#) を実装する任意のパイプラインを使用して、[PortableConfigurationBuilder.WithConfigurationPipeline](#) でサーバー構成を任意のファイルタイプから読み取ることができます。

[Sisk.IniConfiguration](#) パッケージでは、共通の構文エラーに対して例外をスローしないストリームベースの INI ファイルリーダーと、シンプルな構成構文が提供されます。このパッケージは、Sisk フレームワークの外部で使用でき、効率的な INI ドキュメントリーダーが必要なプロジェクトに柔軟性を提供します。

インストール

パッケージをインストールするには、次のコマンドから始めることができます。

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

また、INI [IConfigurationReader](#) や Sisk 依存関係を含まないコア パッケージもインストールできます。

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

メイン パッケージを使用すると、次の例のようにコードで使用できます。

```
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                 // IniConfigurationReader 構成リーダーを使用
13                 config.WithConfigurationPipeline<IniConfigurationReader>();
14             });
15
16         Host.Run();
17     }
18 }
```

```

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($"Hello, {name}!");
28     }
29 }
```

上記のコードは、プロセスの現在のディレクトリ (CurrentDirectory) にある app.ini ファイルを探します。INI ファイルの内容は次のようにになります。

```

1 [Server]
2 # 複数のリスニング アドレスがサポートされます
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 フレーバーと構文

現在の実装フレーバー:

- プロパティとセクション名は **大文字小文字を区別しません。**

- プロパティ名と値は **トリミングされます**、ただし値が引用符で囲まれている場合は除きます。
- 値は单一引用符または二重引用符で囲むことができます。引用符内には改行を含めることができます。
- コメントは # と ; でサポートされます。末尾のコメントも許可されます。
- プロパティには複数の値を指定できます。

詳細については、Sisk で使用されている INI パーサーの "フレーバー" のドキュメントが [こちら](#) にあります。

次の INI コードを例として使用します。

```

1  One = 1
2  Value = this is an value
3  Another value = "this value
4      has an line break on it"
5
6  ; 次のコードにはいくつかの色があります
7  [some section]
8  Color = Red
9  Color = Blue
10 Color = Yellow ; 黄色は使用しないでください

```

これを解析するには:

```

1 // 文字列から INI テキストを解析
2 IniDocument doc = IniDocument.FromString(iniText);
3
4 // 1 つの値を取得
5 string? one = doc.Global.GetOne("one");
6 string? anotherValue = doc.Global.GetOne("another value");
7
8 // 複数の値を取得
9 string[]? colors = doc.GetSection("some section")?.GetMany("color");

```

構成パラメーター

セクションと名前	複数の値 を許可	説明
Server.Listen	はい	サーバーのリスニング アドレス/ポート。

セクションと名前	複数の値 を許可	説明
Server.Encoding	いいえ	サーバーの既定のエンコード。
Server.MaximumContentSize	いいえ	サーバーの最大コンテンツ長(バイト単位)。
Server.IncludeRequestIdHeader	いいえ	HTTP サーバーが X-Request-Id ヘッダーを送信するかどうかを指定します。
Server.ThrowExceptions	いいえ	処理されていない例外をスローするかどうかを指定します。
Server.AccessLogsStream	いいえ	アクセス ログの出力ストリームを指定します。
Server.ErrorsLogsStream	いいえ	エラー ログの出力ストリームを指定します。
Cors.AllowMethods	いいえ	CORS Allow-Methods ヘッダー値を指定します。
Cors.AllowHeaders	いいえ	CORS Allow-Headers ヘッダー値を指定します。
Cors.AllowOrigins	いいえ	複数の Allow-Origin ヘッダー、コンマで区切られた値を指定します。 AllowOrigins に関する詳細情報。
Cors.AllowOrigin	いいえ	1つの Allow-Origin ヘッダーを指定します。
Cors.ExposeHeaders	いいえ	CORS Expose-Headers ヘッダー値を指定します。
Cors.AllowCredentials	いいえ	CORS Allow-Credentials ヘッダー値を指定します。
Cors.MaxAge	いいえ	CORS Max-Age ヘッダー値を指定します。

Manual (advanced) setup

このセクションでは、事前に定義された標準なしで、完全に抽象的な方法でHTTPサーバーを作成します。ここでは、HTTPサーバーがどのように機能するかを手動で構築できます。各ListeningHostにはルーターがあり、HTTPサーバーには複数のListeningHostsを持ち、それぞれが異なるホストと異なるポートを指すことができます。

まず、リクエスト/レスポンスの概念を理解する必要があります。非常にシンプルです。各リクエストに対して、レスポンスが必要です。Siskもこの原則に従います。"Hello, World!"メッセージをHTMLで返すメソッドを作成しましょう。ステータスコードとヘッダーを指定します。

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

次のステップは、このメソッドをHTTPルートに関連付けることです。

Routers

ルーターは、リクエストルートの抽象化であり、サービスに対するリクエストとレスポンスの橋渡しとなります。ルーターはサービスルート、関数、エラーを管理します。

ルーターには複数のルートを持ち、それぞれのルートは異なる操作を実行できます。たとえば、関数の実行、ページの提供、サーバーからのリソースの提供などです。

最初のルーターを作成し、`IndexPage` メソッドをインデックスパスに関連付けましょう。

```
1 Router mainRouter = new Router();
2
3 // SetRouteはすべてのインデックスルートをメソッドに関連付けます。
4 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage);
```

今、ルーターはリクエストを受け取り、レスポンスを送信できます。ただし、`mainRouter` はホストまたはサーバーに結び付けられていないため、単独では機能しません。次のステップは、`ListeningHost`を作成することです。

Listening HostsとPorts

`ListeningHost`はルーターをホストし、同じルーターの複数のリスニングポートを持ちます。`ListeningPort`は、HTTPサーバーがリスニングするプレフィックスです。

ここで、ルーターを指す2つのエンドポイントを持つ`ListeningHost`を作成できます。

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

今、HTTPサーバーは指定されたエンドポイントをリスニングし、リクエストをルーターに転送します。

サーバー設定

サーバー設定は、HTTPサーバー自身の動作のほとんどを担当します。この設定では、`ListeningHosts` をサーバーに関連付けることができます。

```
1  HttpServerConfiguration config = new HttpServerConfiguration();
2  config.ListeningHosts.Add(myHost); // ListeningHostをサーバー設定に追加
```

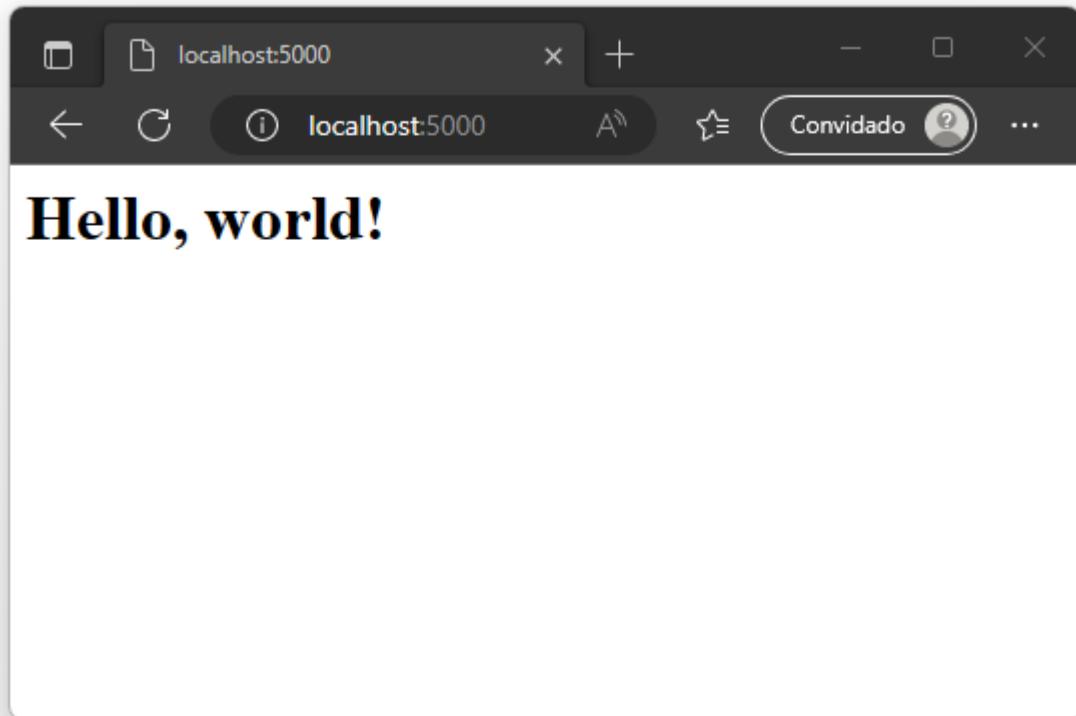
次に、HTTPサーバーを作成できます。

```
1  HttpServer server = new HttpServer(config);
2  server.Start();    // サーバーを起動
3  Console.ReadKey(); // アプリケーションが終了しないようにする
```

今、実行可能ファイルをコンパイルし、次のコマンドでHTTPサーバーを実行できます。

```
dotnet watch
```

実行時に、ブラウザを開き、サーバーパスに移動すると、次のようになります。



リクエストライフサイクル

以下は、HTTP リクエストの例を通じて、リクエストの全ライフサイクルを説明します。

- **リクエストの受信:** 各リクエストは、リクエスト自体とクライアントに配信されるレスポンスの間で HTTP コンテキストを作成します。このコンテキストは、`Sisk` に組み込まれたリスナーから来ます。これは、`HttpListener`、`Kestrel`、または `Cadente` になります。
 - 外部リクエストの検証: `HttpServerConfiguration.RemoteRequestsAction` の検証がリクエストに対して行われます。
 - リクエストが外部で、プロパティが `Drop` の場合、クライアントにレスポンスを返さずに接続が閉じられ、`HttpServerExecutionStatus = RemoteRequestDropped` になります。
 - フォワーディング リゾルバーの構成: フォワーディング リゾルバーが構成されている場合、リクエストの元のホストに対して `OnResolveRequestHost` メソッドが呼び出されます。
 - DNS の一致: ホストが解決され、複数の `ListeningHost` が構成されている場合、サーバーはリクエストに対応するホストを探します。
 - 対応する `ListeningHost` が見つからない場合、クライアントに `400 Bad Request` レスポンスが返され、HTTP コンテキストに `HttpServerExecutionStatus = DnsUnknownHost` ステータスが返されます。
 - `ListeningHost` が見つかるが、その `Router` が初期化されていない場合、クライアントに `503 Service Unavailable` レスポンスが返され、HTTP コンテキストに `HttpServerExecutionStatus = ListeningHostNotReady` ステータスが返されます。
 - ルーターのバインディング: 対応する `ListeningHost` のルーターが受信した HTTP サーバーに結び付けられます。
 - ルーターがすでに別の HTTP サーバーに結び付けられている場合、サーバーの構成リソースをアクティブに使用しているため、`InvalidOperationException` がスローされます。これは、HTTP サーバーの初期化中にのみ発生し、HTTP コンテキストの作成中に発生しません。
 - ヘッダーの事前定義:
 - `X-Request-Id` ヘッダーがレスポンスに事前定義されます (構成されている場合)。
 - `X-Powered-By` ヘッダーがレスポンスに事前定義されます (構成されている場合)。
 - コンテンツ サイズの検証: リクエスト コンテンツが `HttpServerConfiguration.MaximumContentLength` 未満であることを検証します (これが 0 より大きい場合)。
 - リクエストが構成されたものより大きい `Content-Length` を送信した場合、クライアントに `413 Payload Too Large` レスポンスが返され、HTTP コンテキストに `HttpServerExecutionStatus = ContentTooLarge` ステータスが返されます。
 - すべての構成された HTTP サーバーハンドラーに対して `OnHttpRequestOpen` イベントが呼び出されます。
- **アクションのルーティング:** サーバーは受信したリクエストに対してルーターを呼び出します。

- ルーターがリクエストに一致するルートを見つけることができない場合:
 - **Router.NotFoundErrorHandler** プロパティが構成されている場合、アクションが呼び出され、クライアントにアクションのレスポンスが転送されます。
 - 前述のプロパティが null の場合、デフォルトの 404 Not Found レスポンスがクライアントに返されます。
- ルーターがリクエストに一致するルートを見つけるが、ルートのメソッドがリクエストのメソッドと一致しない場合:
 - **Router.MethodNotAllowedErrorHandler** プロパティが構成されている場合、アクションが呼び出され、クライアントにアクションのレスポンスが転送されます。
 - 前述のプロパティが null の場合、デフォルトの 405 Method Not Allowed レスポンスがクライアントに返されます。
- リクエストが OPTIONS メソッドの場合:
 - ルーターは、リクエストメソッドに一致するルートが見つからない場合(ルートのメソッドが明示的に **RouteMethod.Options** ではない場合)に、クライアントに 200 Ok レスポンスを返します。
- **HttpServerConfiguration.ForceTrailingSlash** プロパティが有効で、ルートが正規表現ではなく、リクエストパスが / で終わらず、リクエストメソッドが GET の場合:
 - クライアントに、パスとクエリを同じ場所に / を付けてリダイレクトする 307 Temporary Redirect HTTP レスポンスが返されます。
- すべての構成された HTTP サーバーハンドラーに対して **OnContextBagCreated** イベントが呼び出されます。
- **BeforeResponse** フラグを持つすべてのグローバル **IRequestHandler** インスタンスが実行されます。
 - どのハンドラーも null 以外のレスポンスを返した場合、そのレスポンスはクライアントに転送され、コンテキストは閉じられます。
 - このステップでエラーが発生し、**HttpServerConfiguration.ThrowExceptions** が無効になっている場合:
 - **Router.CallbackErrorHandler** プロパティが有効になっている場合、それが呼び出され、結果のレスポンスがクライアントに返されます。
 - 前述のプロパティが定義されていない場合、空のレスポンスがサーバーに返され、通常は 500 Internal Server Error になります。
- **BeforeResponse** フラグを持つルートで定義されたすべての **IRequestHandler** インスタンスが実行されます。
 - どのハンドラーも null 以外のレスポンスを返した場合、そのレスポンスはクライアントに転送され、コンテキストは閉じられます。
 - このステップでエラーが発生し、**HttpServerConfiguration.ThrowExceptions** が無効になっている場合:
 - **Router.CallbackErrorHandler** プロパティが有効になっている場合、それが呼び出され、結果のレスポンスがクライアントに返されます。
 - 前述のプロパティが定義されていない場合、空のレスポンスがサーバーに返され、通常は 500 Internal Server Error になります。
- ルーターのアクションが呼び出され、HTTP レスポンスに変換されます。

- このステップでエラーが発生し、[HttpServerConfiguration.ThrowExceptions](#) が無効になっている場合:
 - [Router.CallbackErrorHandler](#) プロパティが有効になっている場合、それが呼び出され、結果のレスポンスがクライアントに返されます。
 - 前述のプロパティが定義されていない場合、空のレスポンスがサーバーに返され、通常は 500 Internal Server Error になります。
- AfterResponse フラグを持つすべてのグローバル [IRequestHandler](#) インスタンスが実行されます。
 - どのハンドラーも null 以外のレスポンスを返した場合、そのハンドラーのレスポンスが前のレスポンスに置き換わり、すぐにクライアントに転送されます。
 - このステップでエラーが発生し、[HttpServerConfiguration.ThrowExceptions](#) が無効になっている場合:
 - [Router.CallbackErrorHandler](#) プロパティが有効になっている場合、それが呼び出され、結果のレスポンスがクライアントに返されます。
 - 前述のプロパティが定義されていない場合、空のレスポンスがサーバーに返され、通常は 500 Internal Server Error になります。
- AfterResponse フラグを持つルートで定義されたすべての [IRequestHandler](#) インスタンスが実行されます。
 - どのハンドラーも null 以外のレスポンスを返した場合、そのハンドラーのレスポンスが前のレスポンスに置き換わり、すぐにクライアントに転送されます。
 - このステップでエラーが発生し、[HttpServerConfiguration.ThrowExceptions](#) が無効になっている場合:
 - [Router.CallbackErrorHandler](#) プロパティが有効になっている場合、それが呼び出され、結果のレスポンスがクライアントに返されます。
 - 前述のプロパティが定義されていない場合、空のレスポンスがサーバーに返され、通常は 500 Internal Server Error になります。
- レスポンスの処理: レスポンスが準備できたら、サーバーはそれをクライアントに送信するために準備します。
 - Cross-Origin Resource Sharing Policy (CORS) ヘッダーが、現在の [ListeningHost.CrossOriginResourceSharingPolicy](#) に基づいてレスポンスに定義されます。
 - レスポンスのステータス コードとヘッダーがクライアントに送信されます。
 - レスポンス コンテンツがクライアントに送信されます:
 - レスポンス コンテンツが [ByteArrayContent](#) の派生クラスの場合、レスポンス バイトが直接レスポンス出力ストリームにコピーされます。
 - 前述の条件が満たされない場合、レスポンスはストリームにシリアル化され、レスポンス出力ストリームにコピーされます。
 - ストリームが閉じられ、レスポンス コンテンツが破棄されます。
 - [HttpServerConfiguration.DisposeDisposableContextValues](#) が有効になっている場合、リクエスト コンテキストで定義されたすべての [IDisposable](#) を継承するオブジェクトが破棄されます。
 - すべての構成された HTTP サーバーハンドラーに対して [OnHttpRequestClose](#) イベントが呼び出されます。

- サーバーで例外が発生した場合、すべての構成された HTTP サーバー ハンドラーに対して `OnException` イベントが呼び出されます。
- ルートがアクセス ロギングを許可し、`HttpServerConfiguration.AccessLogsStream` が null でない場合、ログ出力にログ行が書き込まれます。
- ルートがエラー ロギングを許可し、例外が発生し、`HttpServerConfiguration.ErrorsLogsStream` が null でない場合、エラー ログ出力にログ行が書き込まれます。
- サーバーが `HttpServer.WaitNext` を通じてリクエストを待っている場合、ミューテックスが解放され、コンテキストがユーザーに利用可能になります。

フォワーディング リゾルバー

フォワーディング リゾルバーは、クライアントを識別する情報をデコードするヘルパーです。リクエスト、プロキシ、CDN、またはロード バランサーを介して、Sisk サービスがリバース プロキシまたはフォワード プロキシを介して実行される場合、クライアントの IP アドレス、ホスト、およびプロトコルは、元のリクエストとは異なる場合があります。これは、サービス間のフォワーディングであるためです。この Sisk 機能により、リクエストを処理する前にこの情報を解決して制御できます。これらのプロキシは、クライアントを識別するために役立つヘッダーを提供します。

現在、[ForwardingResolver](#) クラスを使用すると、クライアントの IP アドレス、ホスト、および使用される HTTP プロトコルを解決できます。Sisk のバージョン 1.0 以降、サーバーにはこれらのヘッダーをデコードするための標準実装がなくなりました。これは、サービスごとに異なるセキュリティ上の理由があるためです。

たとえば、[X-Forwarded-For](#) ヘッダーには、リクエストをフォワードした IP アドレスに関する情報が含まれます。このヘッダーは、プロキシによって使用され、最終的なサービスに情報のチェーンを運ぶために使用され、クライアントの実際のアドレスを含むすべてのプロキシの IP アドレスが含まれます。問題は、クライアントのリモート IP を識別するのが難しい場合があり、ヘッダーを識別するための特定のルールがないことです。以下のヘッダーを実装する前に、ドキュメントを読むことを強くお勧めします。

- [X-Forwarded-For ヘッダー](#)について読む。
- [X-Forwarded-Host ヘッダー](#)について読む。
- [X-Forwarded-Proto ヘッダー](#)について読む。

ForwardingResolver クラス

このクラスには、各サービスに最も適した実装を可能にする 3 つの仮想メソッドがあります。各メソッドは、プロキシを介したリクエストから情報を解決する責任があります。クライアントの IP アドレス、リクエストのホスト、および使用されるセキュリティ プロトコルです。デフォルトでは、Sisk は常に元のリクエストの情報を使用し、ヘッダーを解決しません。

以下の例は、この実装を使用する方法を示しています。この例では、[X-Forwarded-For](#) ヘッダーを介してクライアントの IP を解決し、リクエストで複数の IP がフォワードされた場合にエラーをスローします。

✖️ IMPORTANT

この例を生産コードで使用しないでください。実装が使用するために適切であることを常に確認してください。
ヘッダーを実装する前にドキュメントを読んでください。

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          using var host = HttpServer.CreateBuilder()
6              .UseForwardingResolver<Resolver>()
7              .UseListeningPort(5555)
8              .Build();
9
10         host.Router.SetRoute(RouteMethod.Any, Route.AnyPath, request =>
11             new HttpResponseMessage("Hello, world!!!"));
12
13         host.Start();
14     }
15
16     class Resolver : ForwardingResolver
17     {
18         public override IPAddress OnResolveClientAddress(HttpRequest request,
19 IPEndPoint connectingEndpoint)
20         {
21             string? forwardedFor = request.Headers.XForwardedFor;
22             if (forwardedFor is null)
23             {
24                 throw new Exception("The X-Forwarded-For header is missing.");
25             }
26             string[] ipAddresses = forwardedFor.Split(',');
27             if (ipAddresses.Length != 1)
28             {
29                 throw new Exception("Too many addresses in the X-Forwarded-For header.");
30             }
31
32             return IPAddress.Parse(ipAddresses[0]);
33         }
34     }
35 }
```

Http サーバー ハンドラー

Sisk バージョン 0.16 では、`HttpServerHandler` クラスが導入され、Sisk の全体的な動作を拡張し、Http リクエスト、ルーター、コンテキスト バッグなどへの追加のイベント ハンドラーを提供します。

このクラスは、HTTP サーバーのライフタイムとリクエストのイベントを集中管理します。Http プロトコルにはセッションがないため、1つのリクエストから別のリクエストへの情報を保持することはできません。Sisk では、セッション、コンテキスト、データベース接続などの有用なプロバイダーを実装する方法を提供します。

各イベントが発生するタイミングと目的については、[このページ](#) を参照してください。また、HTTP リクエストの [ライフサイクル](#) を確認して、リクエストに対して何が起こるかと、イベントがどこで発生するかを理解することもできます。HTTP サーバーでは、同時に複数のハンドラーを使用できます。各イベント呼び出しは同期的であり、関連付けられたすべてのハンドラーが実行され完了するまで、現在のスレッドがロックされます。

`RequestHandlers` と異なり、特定のルート グループまたはルートに適用することはできません。代わりに、全体の HTTP サーバーに適用されます。Http サーバー ハンドラー内で条件を適用することもできます。さらに、各 Sisk アプリケーションに対して、各 `HttpServerHandler` のシングルトンが定義されます。つまり、各 `HttpServerHandler` には1つのインスタンスのみが定義されます。

`HttpServerHandler` を使用する実用的な例は、リクエストの終了時に自動的にデータベース接続を破棄することができます。

```
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         // リクエストがコンテキスト バッグに DbContext を定義しているかどうかを確認します
10        if (requestBag.IsSet<DbContext>())
11        {
12            var db = requestBag.Get<DbContext>();
13            db.Dispose();
14        }
15    }
16 }
17
18 public static class DatabaseConnectionHandlerExtensions
19 {
```

```
20     // ユーザーが HttpRequest から DbContext を作成し、それをリクエスト バッグに保存できるようにします
21     public static DbContext GetDbContext(this HttpRequest request)
22     {
23         var db = new DbContext();
24         return request.SetContextBag<DbContext>(db);
25     }
26 }
```

上記のコードでは、GetDbContext 拡張メソッドにより、HttpRequest オブジェクトから直接接続コンテキストを作成し、それをリクエスト バッグに保存できます。破棄されていない接続はデータベースを実行する際に問題を引き起こす可能性があるため、OnHttpRequestClose で終了されます。

ハンドラーを Http サーバーに登録するには、ビルダーまたは [HttpServer.RegisterHandler](#) を使用できます。

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

これにより、UserController クラスはデータベース コンテキストを使用できます。

```
1 // UserController.cs
2
3 [RoutePrefix("/users")]
4 public class UserController : ApiController
5 {
6     [RouteGet()]
7     public async Task<HttpResponse> List(HttpContext request)
8     {
9         var db = request.GetDbContext();
10        var users = db.Users.ToArray();
11
12        return JsonOk(users);
13    }
14 }
```

```

15     [RouteGet("<id>")]
16     public async Task<HttpResponse> View(HttpRequest request)
17     {
18         var db = request.GetDbContext();
19
20         var userId = request.GetQueryValue<int>("id");
21         var user = db.Users.FirstOrDefault(u => u.Id == userId);
22
23         return JsonOk(user);
24     }
25
26     [RoutePost]
27     public async Task<HttpResponse> Create(HttpRequest request)
28     {
29         var db = request.GetDbContext();
30         var user = JsonSerializer.Deserialize<User>(request.Body);
31
32         ArgumentNullException.ThrowIfNull(user);
33
34         db.Users.Add(user);
35         await db.SaveChangesAsync();
36
37         return JsonMessage("User added.");
38     }
39 }
```

上記のコードでは、JsonOk と JsonMessage などのメソッドを使用しています。これらは ApiController に組み込まれており、RouterController から継承されています。

```

1 // ApiController.cs
2
3 public class ApiController : RouterModule
4 {
5     public HttpResponse JsonOk(object value)
6     {
7         return new HttpResponse(200)
8             .WithContent(JsonContent.Create(value, null, new JsonSerializerOptions()
9                 {
10                     PropertyNameCaseInsensitive = true
11                 }));
12     }
13
14     public HttpResponse JsonMessage(string message, int statusCode = 200)
15     {
16         return new HttpResponse(statusCode)
17             .WithContent(JsonContent.Create(new
```

```
18     {
19         Message = message
20     });
21 }
22 }
```

開発者は、このクラスを使用してセッション、コンテキスト、データベース接続を実装できます。提供されたコードは、`DatabaseConnectionHandler` を使用した実用的な例を示しており、各リクエストの終了時に自動的にデータベース接続を破棄します。

統合は簡単であり、ハンドラーはサーバー設定中に登録されます。`HttpServerHandler` クラスは、HTTP アプリケーションでリソースを管理し、`Sisk` の動作を拡張するための強力なツールセットを提供します。

複数のリスニングホストをサーバーごとに設定

Sisk Frameworkは、常に1つのサーバーに複数のホストを使用することをサポートしています。つまり、単一のHTTPサーバーは複数のポートでリスニングできます。各ポートには独自のルーターとサービスが実行されています。

この方法により、責任を簡単に分離し、Siskを使用した単一のHTTPサーバーでサービスを管理できます。以下の例は、2つのリスニングホストの作成を示しています。各リスニングホストは異なるポートでリスニングし、異なるルーターとアクションを実行しています。

アプリの手動作成を読んで、この抽象化の詳細を理解してください。

```
1  static void Main(string[] args)
2  {
3      // 2つのリスニングホストを作成します。各ホストには独自のルーターとポートがあります
4      //
5      ListeningHost hostA = new ListeningHost();
6      hostA.Ports = [new ListeningPort(12000)];
7      hostA.Router = new Router();
8      hostA.Router.SetRoute(RouteMethod.Get, "/", request => new
9      HttpResponseMessage().WithContent("ホストAからこんにちは！"));
10
11     ListeningHost hostB = new ListeningHost();
12     hostB.Ports = [new ListeningPort(12001)];
13     hostB.Router = new Router();
14     hostB.Router.SetRoute(RouteMethod.Get, "/", request => new
15     HttpResponseMessage().WithContent("ホストBからこんにちは！"));
16
17     // サーバー設定を作成し、両方のリスニングホストを追加します
18     //
19     HttpServerConfiguration configuration = new HttpServerConfiguration();
20     configuration.ListeningHosts.Add(hostA);
21     configuration.ListeningHosts.Add(hostB);
22
23     // 指定された設定を使用するHTTPサーバーを作成します
24     //
25     HttpServer server = new HttpServer(configuration);
26
27     // サーバーを起動します
28     server.Start();
29
30     Console.WriteLine("ホストAに{0}でアクセスしてみてください", server.ListeningPrefixes[0]);
```

```
31     Console.WriteLine("ホストBに{0}でアクセスしてみてください", server.ListeningPrefixes[1]);
32
33     Thread.Sleep(-1);
34 }
```