Introdução

Bem-vindo à documentação do Sisk!

Finalmente, o que é o Sisk Framework? É uma biblioteca leve de código aberto construída com .NET, projetada para ser minimalista, flexível e abstrata. Ela permite que os desenvolvedores criem serviços de internet rapidamente, com pouca ou nenhuma configuração necessária. O Sisk torna possível que seu aplicativo existente tenha um módulo HTTP gerenciado, completo e descartável.

Os valores do Sisk incluem transparência de código, modularidade, desempenho e escalabilidade, e podem lidar com vários tipos de aplicativos, como Restful, JSON-RPC, Web-sockets e mais.

Seus principais recursos incluem:

Recurso	Descrição
Routing	Um roteador de caminhos que suporta prefixos, métodos personalizados, variáveis de caminho, conversores de valor e mais.
Request Handlers	Também conhecidos como *middlewares*, fornecem uma interface para criar seus próprios manipuladores de solicitações que funcionam com a solicitação antes ou após uma ação.
Compression	Comprima o conteúdo de suas respostas facilmente com o Sisk.
Web sockets	Fornece rotas que aceitam web-sockets completos, para leitura e escrita no cliente.
Server-sent events	Fornece o envio de eventos do servidor para clientes que suportam o protocolo SSE.
Logging	Registro de logs simplificado. Registre erros, acesso, defina logs rotativos por tamanho, múltiplos fluxos de saída para o mesmo log e mais.
Multi-host	Tenha um servidor HTTP para várias portas, e cada porta com seu próprio roteador, e cada roteador com seu próprio aplicativo.
Server handlers	Estenda sua própria implementação do servidor HTTP. Personalize com extensões, melhorias e novos recursos.

Primeiros passos

O Sisk pode ser executado em qualquer ambiente .NET. Neste guia, vamos ensinar como criar um aplicativo Sisk usando .NET. Se você ainda não o instalou, por favor, baixe o SDK aqui.

Neste tutorial, vamos cobrir como criar uma estrutura de projeto, receber uma solicitação, obter um parâmetro de URL e enviar uma resposta. Este guia se concentrará em criar um servidor simples usando C#. Você também pode usar sua linguagem de programação favorita.

> **NOTE**
>
> Você pode estar interessado em um projeto de início rápido. Verifique este repositório para obter mais informações.

Criando um Projeto

Vamos nomear nosso projeto "Meu Aplicativo Sisk". Uma vez que você tenha o .NET configurado, você pode criar seu projeto com o seguinte comando:

```
dotnet new console -n meu-aplicativo-sisk
```

Em seguida, navegue até o diretório do seu projeto e instale o Sisk usando a ferramenta de utilitário .NET:

```
cd meu-aplicativo-sisk
dotnet add package Sisk.HttpServer
```

Você pode encontrar maneiras adicionais de instalar o Sisk em seu projeto aqui.

Agora, vamos criar uma instância do nosso servidor HTTP. Para este exemplo, vamos configurá-lo para ouvir na porta 5000.

Construindo o Servidor HTTP

O Sisk permite que você construa seu aplicativo passo a passo manualmente, pois ele roteia para o objeto HttpServer. No entanto, isso pode não ser muito conveniente para a maioria dos projetos. Portanto, podemos usar o método de construtor, que torna mais fácil colocar nosso aplicativo em execução.

Program.cs C#

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

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

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

É importante entender cada componente vital do Sisk. Mais tarde, neste documento, você aprenderá mais sobre como o Sisk funciona.

Configuração manual (avançada)

Você pode aprender como cada mecanismo do Sisk funciona nesta seção da documentação, que explica o comportamento e as relações entre o HttpServer, Router, ListeningPort e outros componentes.

Instalando

Você pode instalar o Sisk por meio do Nuget, dotnet cli ou outras opções. Você pode configurar facilmente o ambiente do Sisk executando este comando no console do desenvolvedor:

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

Este comando instalará a versão mais recente do Sisk no seu projeto.

Suporte Nativo AOT

.NET Native AOT permite a publicação de aplicativos .NET nativos que são autossuficientes e não requerem o tempo de execução .NET instalado no host de destino. Além disso, o Native AOT fornece benefícios como:

- Aplicativos muito menores
- Inicialização significativamente mais rápida
- Consumo de memória mais baixo

O Sisk Framework, por sua natureza explícita, permite o uso de Native AOT para quase todos os seus recursos sem exigir rework no código-fonte para adaptá-lo ao Native AOT.

Recursos não suportados

No entanto, o Sisk usa reflexão, embora mínima, para alguns recursos. Os recursos mencionados abaixo podem estar parcialmente disponíveis ou completamente indisponíveis durante a execução de código nativo:

- Auto-escaneamento de módulos do roteador: este recurso escaneia os tipos incorporados na Assembly em execução e registra os tipos que são módulos do roteador. Este recurso requer tipos que possam ser excluídos durante a redução da Assembly.

Todos os outros recursos são compatíveis com o AOT no Sisk. É comum encontrar um ou outro método que dá um aviso de AOT, mas o mesmo, se não for mencionado aqui, tem uma sobrecarga que indica a passagem de um tipo, parâmetro ou informação de tipo que ajuda o compilador AOT a compilar o objeto.

Implantando sua Aplicação Sisk

O processo de implantar uma aplicação Sisk consiste em publicar seu projeto em produção. Embora o processo seja relativamente simples, é importante notar detalhes que podem ser letais para a segurança e estabilidade da infraestrutura de implantação.

Idealmente, você deve estar pronto para implantar sua aplicação na nuvem, após realizar todos os testes possíveis para ter sua aplicação pronta.

Publicando sua aplicação

Publicar sua aplicação ou serviço Sisk é gerar binários prontos e otimizados para produção. Neste exemplo, vamos compilar os binários para produção para executar em uma máquina que tenha o .NET Runtime instalado.

Você precisará ter o .NET SDK instalado em sua máquina para compilar sua aplicação, e o .NET Runtime instalado no servidor de destino para executar sua aplicação. Você pode aprender como instalar o .NET Runtime em seu servidor Linux aqui, Windows e Mac OS.

No diretório onde seu projeto está localizado, abra um terminal e use o comando de publicação do .NET:

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

Isso gerará seus binários dentro de `bin/Release/publish/linux-x64`.

> **NOTE**
>
> Se sua aplicação estiver executando usando o pacote Sisk.ServiceProvider, você deve copiar seu `service-config.json` para o servidor de hospedagem junto com todos os binários gerados pelo `dotnet publish`. Você pode deixar o arquivo pré-configurado, com variáveis de ambiente, portas e hosts de escuta e configurações adicionais do servidor.

A próxima etapa é levar esses arquivos para o servidor onde sua aplicação será hospedada.

Depois disso, dê permissões de execução para o arquivo binário. Neste caso, vamos considerar que o nome do nosso projeto é "my-app":

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

Após executar sua aplicação, verifique se ela produz alguma mensagem de erro. Se não produzir, é porque sua aplicação está em execução.

Neste ponto, provavelmente não será possível acessar sua aplicação pela rede externa fora do seu servidor, pois as regras de acesso, como Firewall, não foram configuradas. Vamos considerar isso nas próximas etapas.

Você deve ter o endereço do host virtual onde sua aplicação está escutando. Isso é definido manualmente na aplicação e depende de como você está instanciando seu serviço Sisk.

Se você **não** estiver usando o pacote Sisk.ServiceProvider, você deve encontrar o endereço onde definiu sua instância de HttpServer:

```
HttpServer server = HttpServer.Emit(5000, out HttpServerConfiguration config, out var host,
out var router);
// sisk deve escutar em http://localhost:5000/
```

Associando um ListeningHost manualmente:

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

Ou se você estiver usando o pacote Sisk.ServiceProvider, em seu `service-config.json`:

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

A partir disso, podemos criar um proxy reverso para escutar seu serviço e tornar o tráfego disponível sobre a rede aberta.

Proxyando sua aplicação

Proxyar seu serviço significa não expor diretamente seu serviço Sisk à rede externa. Essa prática é muito comum para implantações de servidor porque:

- Permite associar um certificado SSL à sua aplicação;
- Cria regras de acesso antes de acessar o serviço e evitar sobrecargas;
- Controla a largura de banda e os limites de solicitação;
- Separa os balanceadores de carga para sua aplicação;
- Previne danos de segurança à infraestrutura de falha.

Você pode servir sua aplicação por meio de um proxy reverso como Nginx ou Apache, ou você pode usar um túnel http-over-dns como Cloudflared.

Além disso, lembre-se de resolver corretamente os cabeçalhos de encaminhamento do proxy para obter as informações do cliente, como endereço IP e host, por meio de resolutores de encaminhamento.

A próxima etapa após criar seu túnel, configurar o firewall e ter sua aplicação em execução é criar um serviço para sua aplicação.

> **NOTE**
>
> Usar certificados SSL diretamente no serviço Sisk em sistemas não-Windows não é possível. Isso é um ponto da implementação do HttpListener, que é o módulo central para como a gestão da fila HTTP é feita no Sisk, e essa implementação varia de sistema operacional para sistema operacional. Você pode usar SSL em seu serviço Sisk se associar um certificado ao host virtual com IIS. Para outros sistemas, usar um proxy reverso é altamente recomendado.

Criando um serviço

Criar um serviço fará com que sua aplicação esteja sempre disponível, mesmo após reiniciar a instância do servidor ou uma falha não recuperável.

Neste tutorial simples, vamos usar o conteúdo do tutorial anterior como um exemplo para manter seu serviço sempre ativo.

1. Acesse o diretório onde os arquivos de configuração do serviço estão localizados:

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

2. Crie seu arquivo `my-app.service` e inclua o conteúdo:

my-app.service **INI**

```ini
[Unit]
Description=<descrição sobre sua aplicação>

[Service]
# defina o usuário que lançará o serviço
User=<usuário que lançará o serviço>

# o caminho do ExecStart não é relativo ao WorkingDirectory.
# defina-o como o caminho completo para o arquivo executável
WorkingDirectory=/home/htdocs
ExecStart=/home/htdocs/my-app

# defina o serviço para sempre reiniciar em caso de falha
Restart=always
RestartSec=3

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

3. Reinicie o módulo de gerenciamento de serviços:

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

4. Inicie seu novo serviço criado a partir do nome do arquivo que você definiu e verifique se ele está em execução:

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

5. Agora, se sua aplicação estiver em execução ("Active: active"), habilite seu serviço para continuar em execução após uma reinicialização do sistema:

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

Agora você está pronto para apresentar sua aplicação Sisk a todos.

Trabalhando com SSL

Trabalhar com SSL para desenvolvimento pode ser necessário quando se trabalha em contextos que exigem segurança, como a maioria dos cenários de desenvolvimento web. O Sisk opera em cima do HttpListener, que não suporta HTTPS nativo, apenas HTTP. No entanto, existem soluções que permitem trabalhar com SSL no Sisk. Veja-as abaixo:

Através do IIS no Windows

- Disponível em: Windows
- Esforço: médio

Se você estiver no Windows, pode usar o IIS para habilitar o SSL no seu servidor HTTP. Para que isso funcione, é aconselhável seguir este tutorial antes, se você quiser que sua aplicação esteja ouvindo em um host diferente de "localhost".

Para que isso funcione, você deve instalar o IIS através das recursos do Windows. O IIS está disponível gratuitamente para usuários do Windows e Windows Server. Para configurar o SSL na sua aplicação, tenha o certificado SSL pronto, mesmo que seja autoassinado. Em seguida, você pode ver como configurar o SSL no IIS 7 ou superior.

Através do mitmproxy

- Disponível em: Linux, macOS, Windows
- Esforço: fácil

mitmproxy é uma ferramenta de proxy de interceptação que permite que desenvolvedores e testadores de segurança inspecionem, modifiquem e gravem o tráfego HTTP e HTTPS entre um cliente (como um navegador da web) e um servidor. Você pode usar a utilidade **mitmdump** para iniciar um proxy SSL reverso entre o seu cliente e a aplicação Sisk.

1. Primeiramente, instale o mitmproxy na sua máquina.
2. Inicie a aplicação Sisk. Neste exemplo, usaremos a porta 8000 como a porta HTTP insegura.
3. Inicie o servidor mitmproxy para ouvir a porta segura na porta 8001:

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

E pronto! Você já pode acessar a aplicação através de `https://localhost:8001/`. A aplicação não precisa estar em execução para iniciar o `mitmdump`.

Alternativamente, você pode adicionar uma referência ao helper do mitmproxy no seu projeto. Isso ainda exige que o mitmproxy esteja instalado no seu computador.

Através do pacote Sisk.SslProxy

- Disponível em: Linux, macOS, Windows
- Esforço: fácil

O pacote Sisk.SslProxy é uma maneira simples de habilitar o SSL na aplicação Sisk. No entanto, é um pacote **extremamente experimental**. Pode ser instável trabalhar com esse pacote, mas você pode ser parte do pequeno percentual de pessoas que contribuirão para tornar esse pacote viável e estável. Para começar, você pode instalar o pacote Sisk.SslProxy com:

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

> **NOTE**
>
> Você deve habilitar "Habilitar pacotes de pré-lançamento" no Gerenciador de Pacotes do Visual Studio para instalar o Sisk.SslProxy.

Novamente, é um projeto experimental, então não pense em colocá-lo em produção.

No momento, o Sisk.SslProxy pode lidar com a maioria dos recursos do HTTP/1.1, incluindo HTTP Continue, Chunked-Encoding, WebSockets e SSE. Leia mais sobre o SslProxy aqui.

Configurando reservas de namespace no Windows

Sisk trabalha com a interface de rede HttpListener, que vincula um host virtual ao sistema para ouvir solicitações.

No Windows, essa vinculação é um pouco restritiva, permitindo apenas que o localhost seja vinculado como um host válido. Ao tentar ouvir outro host, um erro de acesso negado é lançado no servidor. Este tutorial explica como conceder autorização para ouvir em qualquer host que você desejar no sistema.

Namespace Setup.bat — BATCH

```batch
@echo off

:: insira o prefixo aqui, sem espaços ou aspas
SET PREFIX=

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

pause
```

Onde em `PREFIX`, é o prefixo ("Host de Escuta→Porta") que o servidor irá ouvir. Ele deve ser formatado com o esquema de URL, host, porta e uma barra no final, exemplo:

Namespace Setup.bat — BATCH

```batch
SET PREFIX=http://meu-aplicativo.exemplo.teste/
```

Para que você possa ser ouvido em sua aplicação por meio de:

Program.cs — C#

```csharp
class Program
{
    static async Task Main(string[] args)
    {
        using var app = HttpServer.CreateBuilder()
```

```
            .UseListeningPort("http://meu-aplicativo.exemplo.teste/")
            .Build();

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

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

Histórico de Versões

Todas as alterações feitas no Sisk são registradas no histórico de versões. Você pode visualizar os históricos de versões para todas as versões do Sisk aqui.

Perguntas Frequentes

Perguntas frequentes sobre Sisk.

O Sisk é de código aberto?

Totalmente. Todo o código-fonte utilizado pelo Sisk é publicado e atualizado frequentemente no GitHub.

Contribuições são aceitas?

Desde que sejam compatíveis com a filosofia do Sisk, todas as contribuições são muito bem-vindas! As contribuições não precisam ser apenas código! Você pode contribuir com documentação, testes, traduções, doações e posts, por exemplo.

O Sisk é financiado?

Não. Nenhuma organização ou projeto atualmente patrocina o Sisk.

Posso usar o Sisk em produção?

Absolutamente. O projeto está em desenvolvimento há mais de três anos e teve testes intensivos em aplicações comerciais que estão em produção desde então. O Sisk é usado em projetos comerciais importantes como

infraestrutura principal.

Um guia sobre como implantar em diferentes sistemas e ambientes foi escrito e está disponível.

O Sisk tem autenticação, monitoramento e serviços de banco de dados?

Não. O Sisk não tem nenhum desses. É um framework para desenvolver aplicações web HTTP, mas é um framework minimalista que entrega o que é necessário para que sua aplicação funcione.

Você pode implementar todos os serviços que desejar usando qualquer biblioteca de terceiros que preferir. O Sisk foi feito para ser agnóstico, flexível e funcionar com qualquer coisa.

Por que devo usar o Sisk em vez de ?

Não sei. Você me diz.

O Sisk foi criado para atender a um cenário genérico para aplicações web HTTP em .NET. Projetos estabelecidos, como o ASP.NET, resolvem vários problemas, mas com diferentes vieses. Diferentemente de frameworks maiores, o Sisk exige que o usuário saiba o que está fazendo e construindo. Noções básicas de desenvolvimento web e do protocolo HTTP são essenciais para trabalhar com o Sisk.

O Sisk é mais próximo do Express do Node.js do que do ASP.NET Core. É uma abstração de alto nível que permite criar aplicações com lógica HTTP que você deseja.

O que preciso aprender para usar o Sisk?

Você precisa dos básicos de:

- Desenvolvimento web (HTTP, Restful, etc.)
- .NET

Isso é tudo. Tendo uma noção desses dois tópicos, você pode dedicar algumas horas para desenvolver uma aplicação avançada com o Sisk.

Posso desenvolver aplicações comerciais com o Sisk?

Absolutamente.

O Sisk foi criado sob a licença MIT, o que significa que você pode usar o Sisk em qualquer projeto comercial, comercial ou não comercial, sem a necessidade de uma licença proprietária.

O que pedimos é que em algum lugar da sua aplicação, você tenha um aviso dos projetos de código aberto utilizados em seu projeto, e que o Sisk esteja lá.

Roteamento

O Router é o primeiro passo na construção do servidor. Ele é responsável por armazenar objetos Route, que são endpoints que mapeiam URLs e seus métodos para ações executadas pelo servidor. Cada ação é responsável por receber uma solicitação e entregar uma resposta ao cliente.

As rotas são pares de expressões de caminho ("padrão de caminho") e o método HTTP que elas podem ouvir. Quando uma solicitação é feita ao servidor, ele tentará encontrar uma rota que corresponda à solicitação recebida, então ele chamará a ação daquela rota e entregará a resposta resultante ao cliente.

Existem várias maneiras de definir rotas no Sisk: elas podem ser estáticas, dinâmicas ou auto-escaneadas, definidas por atributos ou diretamente no objeto Router.

```
Router mainRouter = new Router();

// mapeia o GET / para a ação a seguir
mainRouter.MapGet("/", request => {
    return new HttpResponse("Olá, mundo!");
});
```

Para entender o que uma rota é capaz de fazer, precisamos entender o que uma solicitação é capaz de fazer. Um HttpRequest conterá tudo o que você precisa. O Sisk também inclui alguns recursos extras que aceleram o desenvolvimento geral.

Para cada ação recebida pelo servidor, um delegado do tipo RouteAction será chamado. Este delegado contém um parâmetro que segura um HttpRequest com todas as informações necessárias sobre a solicitação recebida pelo servidor. O objeto resultante deste delegado deve ser um HttpResponse ou um objeto que mapeia para ele por meio de tipos de resposta implícitos.

Correspondência de rotas

Quando uma solicitação é recebida pelo servidor HTTP, o Sisk procura uma rota que satisfaça a expressão do caminho recebido pela solicitação. A expressão é sempre testada entre a rota e o caminho da solicitação, sem considerar a string de consulta.

Este teste não tem prioridade e é exclusivo para uma única rota. Quando nenhuma rota é correspondida com aquela solicitação, a resposta Router.NotFoundErrorHandler é retornada ao cliente. Quando o padrão de caminho é correspondido, mas o método HTTP é incorreto, a resposta Router.MethodNotAllowedErrorHandler é enviada de volta ao cliente.

O Sisk verifica a possibilidade de colisões de rotas para evitar esses problemas. Quando as rotas são definidas, o Sisk procurará por rotas possíveis que possam colidir com a rota sendo definida. Este teste inclui a verificação do caminho e do método que a rota está configurada para aceitar.

Criando rotas usando padrões de caminho

Você pode definir rotas usando vários métodos `SetRoute`.

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

// maneira Map*
mainRouter.MapGet("/form", (request) =>
{
    var formData = request.GetFormData();
    return new HttpResponse(); // 200 ok vazio
});

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

    return new HttpResponse()
    {
        // o StreamContent interno
        // stream é descartado após o envio
        // da resposta.
        Content = new StreamContent(imageStream)
    };
});

// vários parâmetros
mainRouter.MapGet("/hey/<name>/sobrenome/<sobrenome>", (request) =>
{
```

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

A propriedade RouteParameters do HttpResponse contém todas as informações sobre as variáveis de caminho da solicitação recebida.

Cada caminho recebido pelo servidor é normalizado antes que o teste do padrão de caminho seja executado, seguindo essas regras:

- Todos os segmentos vazios são removidos do caminho, por exemplo: `////foo//bar` se torna `/foo/bar`.
- A correspondência de caminho é **sensível a maiúsculas e minúsculas**, a menos que Router.MatchRoutesIgnoreCase seja definido como `true`.

As propriedades Query e RouteParameters do HttpRequest retornam um objeto StringValueCollection, onde cada propriedade indexada retorna um StringValue não nulo, que pode ser usado como uma opção/monad para converter seu valor bruto em um objeto gerenciado.

O exemplo abaixo lê o parâmetro de rota "id" e obtém um `Guid` a partir dele. Se o parâmetro não for um Guid válido, uma exceção é lançada e um erro 500 é retornado ao cliente se o servidor não estiver lidando com Router.CallbackErrorHandler.

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

> [!NOTA] Os caminhos têm sua barra final `/` ignorada em ambos os caminhos da solicitação e da rota, ou seja, se você tentar acessar uma rota definida como `/index/page` você poderá acessá-la usando `/index/page/` também.
>
> Você também pode forçar as URLs a terminar com `/` habilitando a flag ForceTrailingSlash.

Criando rotas usando instâncias de classe

Você também pode definir rotas dinamicamente usando reflexão com o atributo RouteAttribute. Dessa forma, a instância de uma classe na qual seus métodos implementam esse atributo terá suas rotas definidas no roteador de destino.

Para que um método seja definido como uma rota, ele deve ser marcado com um RouteAttribute, como o próprio atributo ou um RouteGetAttribute. O método pode ser estático, de instância, público ou privado. Quando o método `SetObject(type)` ou `SetObject<TType>()` é usado, os métodos de instância são ignorados.

Controller/MyController.cs **C#**

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

    // métodos estáticos também funcionam
    [RouteGet("/hello")]
    static HttpResponse Hello(HttpRequest request)
    {
        HttpResponse res = new HttpResponse();
        res.Content = new StringContent("Olá, mundo!");
        return res;
    }
}
```

A linha abaixo definirá tanto o método `Index` quanto o método `Hello` de `MyController` como rotas, pois ambos são marcados como rotas e uma instância da classe foi fornecida, não seu tipo. Se seu tipo tivesse sido fornecido em vez de uma instância, apenas os métodos estáticos seriam definidos.

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

Desde a versão 0.16 do Sisk, é possível habilitar o AutoScan, que procurará por classes definidas pelo usuário que implementam `RouterModule` e as associará automaticamente ao roteador. Isso não é suportado com compilação AOT.

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

A instrução acima procurará por todos os tipos que implementam `ApiController` , mas não o tipo em si. Os dois parâmetros opcionais indicam como o método procurará por esses tipos. O primeiro argumento implica a

Assembly onde os tipos serão procurados e o segundo indica a forma como os tipos serão definidos.

Rotas de regex

Em vez de usar os métodos de correspondência de caminho HTTP padrão, você pode marcar uma rota para ser interpretada com Regex.

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

Ou com a classe RegexRoute:

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

Você também pode capturar grupos da expressão regular no padrão para o conteúdo de HttpRequest.RouteParameters:

Controller/MyController.cs **C#**

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

Prefixo de rotas

Você pode prefixar todas as rotas em uma classe ou módulo com o atributo RoutePrefix e definir o prefixo como uma string.

Veja o exemplo abaixo usando a arquitetura BREAD (Browse, Read, Edit, Add e Delete):

Controller/Api/UsersController.cs **C#**

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

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

No exemplo acima, o parâmetro HttpResponse é omitido em favor de ser usado por meio do contexto global HttpContext.Current. Leia mais na seção a seguir.

Rotas sem parâmetro de solicitação

As rotas podem ser definidas sem o parâmetro HttpRequest e ainda é possível obter a solicitação e seus componentes no contexto da solicitação. Vamos considerar uma abstração `ControllerBase` que serve como base para todos os controladores de uma API e que fornece a propriedade `Request` para obter a HttpRequest atualmente.

Controller/ControllerBase.cs — C#

```
1   public abstract class ControllerBase
2   {
3       // obtém a solicitação do thread atual
4       public HttpRequest Request { get => HttpContext.Current.Request; }
5
6       // a linha abaixo, quando chamada, obtém o banco de dados da sessão HTTP atual,
7       // ou cria um novo se ele não existir
8       public DbContext Database { get => HttpContext.Current.RequestBag.GetOrAdd<DbContext>
9   (); }
    }
```

E para que todos os seus descendentes possam usar a sintaxe de rota sem o parâmetro de solicitação:

Controller/UsersController.cs — C#

```
1   [RoutePrefix("/api/users")]
2   public class UsersController : ControllerBase
3   {
4       [RoutePost]
5       public async Task<HttpResponse> Create()
6       {
7           // lê os dados JSON do corpo da solicitação atual
8           UserCreationDto? user = JsonSerializer.DeserializeAsync<UserCreationDto>
```

```
    (Request.Body);
            ...
            Database.Users.Add(user);

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

Mais detalhes sobre o contexto atual e injeção de dependência podem ser encontrados no tutorial de injeção de dependência.

Rotas de qualquer método

Você pode definir uma rota para ser correspondida apenas por seu caminho e ignorar o método HTTP. Isso pode ser útil para você fazer a validação do método dentro da callback da rota.

```
// corresponderá ao / em qualquer método HTTP
mainRouter.SetRoute(RouteMethod.Any, "/", callbackFunction);
```

Rotas de qualquer caminho

As rotas de qualquer caminho testam para qualquer caminho recebido pelo servidor HTTP, sujeito ao método da rota sendo testado. Se o método da rota for RouteMethod.Any e a rota usar Route.AnyPath em sua expressão de caminho, essa rota ouvirá todas as solicitações do servidor HTTP e nenhuma outra rota pode ser definida.

```
// a rota a seguir corresponderá a todas as solicitações POST
mainRouter.SetRoute(RouteMethod.Post, Route.AnyPath, callbackFunction);
```

Correspondência de rota ignorando caso

Por padrão, a interpretação de rotas com solicitações é sensível a maiúsculas e minúsculas. Para fazer com que ela ignore o caso, habilite essa opção:

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

Isso também habilitará a opção `RegexOptions.IgnoreCase` para rotas que usam correspondência de regex.

Tratador de callback de não encontrado (404)

Você pode criar um callback personalizado para quando uma solicitação não corresponde a nenhuma rota conhecida.

```
1   mainRouter.NotFoundErrorHandler = () =>
2   {
3       return new HttpResponse(404)
4       {
5           // Desde a v0.14
6           Content = new HtmlContent("<h1>Não encontrado</h1>")
7           // versões anteriores
8           Content = new StringContent("<h1>Não encontrado</h1>", Encoding.UTF8, "text/html")
9       };
10  };
```

Tratador de callback de método não permitido (405)

Você também pode criar um callback personalizado para quando uma solicitação corresponde ao seu caminho, mas não corresponde ao método.

```
1   mainRouter.MethodNotAllowedErrorHandler = (context) =>
2   {
3       return new HttpResponse(405)
4       {
5           Content = new StringContent($"Método não permitido para esta rota.")
```

```
        };
    };
```

Tratador de erro interno

Os callbacks de rota podem lançar erros durante a execução do servidor. Se não forem tratados corretamente, o funcionamento geral do servidor HTTP pode ser interrompido. O roteador tem um callback para quando um callback de rota falha e impede a interrupção do serviço.

Esse método só é alcançável quando ThrowExceptions é definido como `false` .

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

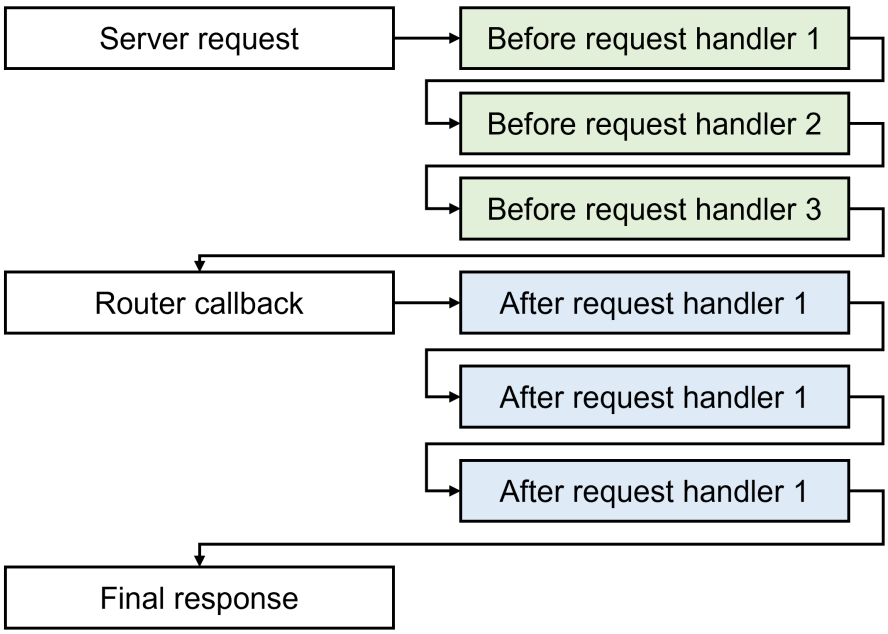
Tratamento de Requisições

Os tratadores de requisições, também conhecidos como "middlewares", são funções que são executadas antes ou após uma requisição ser executada no roteador. Eles podem ser definidos por rota ou por roteador.

Existem dois tipos de tratadores de requisições:

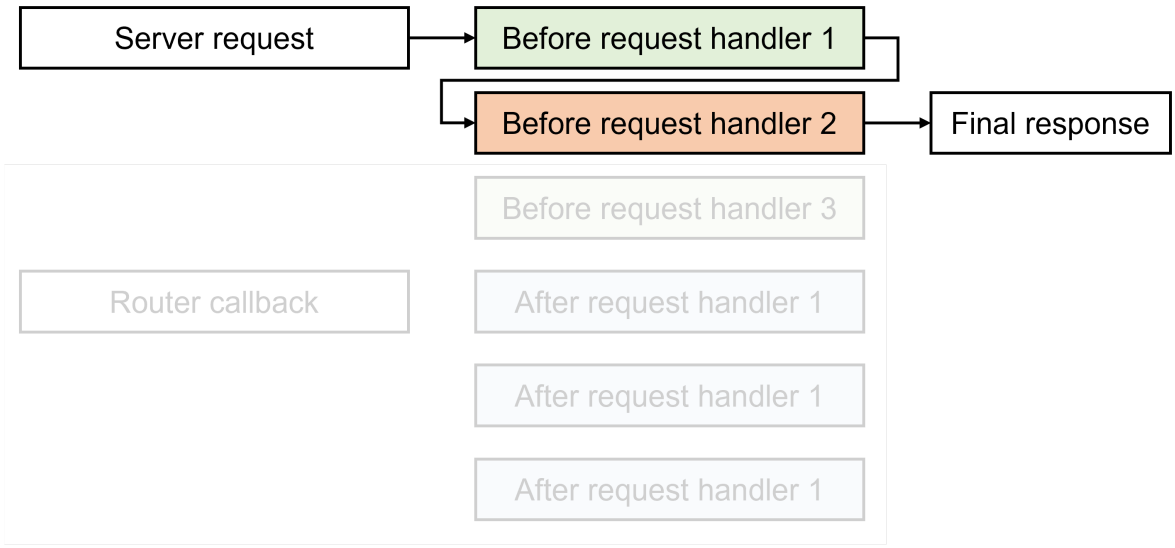
- **BeforeResponse**: define que o tratador de requisição será executado antes de chamar a ação do roteador.
- **AfterResponse**: define que o tratador de requisição será executado após chamar a ação do roteador. Enviar uma resposta HTTP neste contexto substituirá a resposta da ação do roteador.

Ambos os tratadores de requisições podem substituir a resposta da função de callback real do roteador. Além disso, os tratadores de requisições podem ser úteis para validar uma requisição, como autenticação, conteúdo ou qualquer outra informação, como armazenar informações, logs ou outras etapas que podem ser realizadas antes ou após uma resposta.



Dessa forma, um tratador de requisição pode interromper toda a execução e retornar uma resposta antes de finalizar o ciclo, descartando tudo o mais no processo.

Exemplo: suponha que um tratador de requisição de autenticação de usuário não autentique o usuário. Isso impedirá que o ciclo de requisição continue e ficará pendente. Se isso acontecer no tratador de requisição na posição dois, o terceiro e subsequentes não serão avaliados.



Criando um Tratador de Requisição

Para criar um tratador de requisição, podemos criar uma classe que herda a interface IRequestHandler, no seguinte formato:

Middleware/AuthenticateUserRequestHandler.cs C#

```
1   public class AuthenticateUserRequestHandler : IRequestHandler
2   {
3       public RequestHandlerExecutionMode ExecutionMode { get; init; } =
4   RequestHandlerExecutionMode.BeforeResponse;
5
6       public HttpResponse? Execute(HttpRequest request, HttpContext context)
7       {
8           if (request.Headers.Authorization == null)
9           {
10              // Retornar null indica que o ciclo de requisição pode continuar
11              return null;
12          }
13          else
14          {
```

```
            // Retornar um objeto HttpResponse indica que essa resposta substituirá as
respostas adjacentes.
            return new HttpResponse(System.Net.HttpStatusCode.Unauthorized);
        }
    }
}
```

No exemplo acima, indicamos que, se o cabeçalho `Authorization` estiver presente na requisição, deve continuar e a próxima requisição ou a ação do roteador deve ser chamada, dependendo do que vier a seguir. Se um tratador de requisição for executado após a resposta por meio de sua propriedade ExecutionMode e retornar um valor não nulo, ele substituirá a resposta do roteador.

Sempre que um tratador de requisição retorna `null`, isso indica que a requisição deve continuar e o próximo objeto deve ser chamado ou o ciclo deve terminar com a resposta do roteador.

Associando um Tratador de Requisição a uma Rota Única

Você pode definir um ou mais tratadores de requisição para uma rota.

Router.cs — C#

```
mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage, "", new IRequestHandler[]
{
    new AuthenticateUserRequestHandler(),     // antes do tratador de requisição
    new ValidateJsonContentRequestHandler(),  // antes do tratador de requisição
    //                                        -- método IndexPage será executado aqui
    new WriteToLogRequestHandler()            // após o tratador de requisição
});
```

Ou criando um objeto Route:

Router.cs — C#

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


```

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

Associando um Tratador de Requisição a um Roteador

Você pode definir um tratador de requisição global que será executado em todas as rotas de um roteador.

Router.cs C#

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

Associando um Tratador de Requisição a um Atributo

Você pode definir um tratador de requisição em um atributo de método junto com um atributo de rota.

Controller/MyController.cs C#

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

Observe que é necessário passar o tipo de tratador de requisição desejado e não uma instância do objeto. Dessa forma, o tratador de requisição será instanciado pelo analisador do roteador. Você pode passar argumentos no

construtor da classe com a propriedade ConstructorArguments.

Exemplo:

Controller/MyController.cs C#

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

Você também pode criar seu próprio atributo que implementa RequestHandler:

Middleware/Attributes/AuthenticateAttribute.cs C#

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

E usá-lo como:

Controller/MyController.cs C#

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

Ignorando um Tratador de Requisição Global

Depois de definir um tratador de requisição global em uma rota, você pode ignorá-lo em rotas específicas.

Router.cs C#

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

mainRouter.SetRoute(new Route(RouteMethod.Get, "/", "My route", IndexPage, null)
{
    BypassGlobalRequestHandlers = new IRequestHandler[]
    {
        myRequestHandler,                   // ok: a mesma instância do que está nos
tratadores de requisição globais
        new AuthenticateUserRequestHandler() // errado: não ignorará o tratador de
requisição global
    }
});
```

> **NOTE**
>
> Se você estiver ignorando um tratador de requisição, é necessário usar a mesma referência do que foi instanciada anteriormente para ignorar. Criar outra instância do tratador de requisição não ignorará o tratador de requisição global, pois sua referência será alterada. Lembre-se de usar a mesma referência do tratador de requisição usada em ambos GlobalRequestHandlers e BypassGlobalRequestHandlers.

Requests

Requests são estruturas que representam uma mensagem de solicitação HTTP. O objeto HttpRequest contém funções úteis para lidar com mensagens HTTP em toda a sua aplicação.

Uma solicitação HTTP é formada pelo método, caminho, versão, cabeçalhos e corpo.

Neste documento, ensinaremos como obter cada um desses elementos.

Obtendo o método da solicitação

Para obter o método da solicitação recebida, você pode usar a propriedade Method:

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

Esta propriedade retorna o método da solicitação representado por um objeto HttpMethod.

> **NOTE**
>
> Ao contrário dos métodos de rota, esta propriedade não serve o item RouteMethod.Any. Em vez disso, ela retorna o método real da solicitação.

Obtendo componentes da URL da solicitação

Você pode obter vários componentes de uma URL através de certas propriedades de uma solicitação. Para este exemplo, considere a URL:

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

Nome do componente	Descrição	Valor do componente
Path	Obtém o caminho da solicitação.	`/user/login`
FullPath	Obtém o caminho da solicitação e a string de consulta.	`/user/login?email=foo@bar.com`
FullUrl	Obtém a string completa da URL da solicitação.	`http://localhost:5000/user/login?email=foo@bar.com`
Host	Obtém o host da solicitação.	`localhost`
Authority	Obtém o host e a porta da solicitação.	`localhost:5000`
QueryString	Obtém a consulta da solicitação.	`?email=foo@bar.com`
Query	Obtém a consulta da solicitação em uma coleção de valores nomeados.	`{StringValueCollection object}`
IsSecure	Determina se a solicitação está usando SSL (true) ou não (false).	`false`

Você também pode optar por usar a propriedade HttpRequest.Uri, que inclui tudo o que está acima em um único objeto.

Obtendo o corpo da solicitação

Algumas solicitações incluem corpo, como formulários, arquivos ou transações de API. Você pode obter o corpo de uma solicitação a partir da propriedade:

```
1   // obtém o corpo da solicitação como uma string, usando a codificação da solicitação
2   como codificador
3   string body = request.Body;
4
5   // ou obtém-o em um array de bytes
6   byte[] bodyBytes = request.RawBody;
```

```
    // ou, ainda, você pode transmitir.
    Stream requestStream = request.GetRequestStream();
```

Também é possível determinar se há um corpo na solicitação e se ele está carregado com as propriedades HasContents, que determina se a solicitação tem conteúdos, e IsContentAvailable, que indica que o servidor HTTP recebeu totalmente o conteúdo do ponto remoto.

Não é possível ler o conteúdo da solicitação através de `GetRequestStream` mais de uma vez. Se você ler com esse método, os valores em `RawBody` e `Body` também não estarão disponíveis. Não é necessário descartar o fluxo de solicitação no contexto da solicitação, pois ele é descartado no final da sessão HTTP em que foi criado. Além disso, você pode usar a propriedade HttpRequest.RequestEncoding para obter a melhor codificação para decodificar a solicitação manualmente.

O servidor tem limites para leitura do conteúdo da solicitação, que se aplica tanto a HttpRequest.Body quanto a HttpRequest.RawBody. Essas propriedades copiam todo o fluxo de entrada para um buffer local do mesmo tamanho de HttpRequest.ContentLength.

Uma resposta com status 413 Content Too Large é retornada ao cliente se o conteúdo enviado for maior que HttpServerConfiguration.MaximumContentLength definido na configuração do usuário. Além disso, se não houver limite configurado ou se for muito grande, o servidor lançará uma OutOfMemoryException quando o conteúdo enviado pelo cliente exceder Int32.MaxValue (2 GB) e se o conteúdo for tentado acessar através de uma das propriedades mencionadas acima. Você ainda pode lidar com o conteúdo por meio de streaming.

> **NOTE**
>
> Embora o Sisk permita isso, é sempre uma boa ideia seguir a Semântica HTTP para criar sua aplicação e não obter ou servir conteúdo em métodos que não permitem. Leia sobre RFC 9110 "HTTP Semantics".

Obtendo o contexto da solicitação

O HTTP Context é um objeto exclusivo do Sisk que armazena informações do servidor HTTP, rota, roteador e manipulador de solicitação. Você pode usá-lo para se organizar em um ambiente onde esses objetos são difíceis de organizar.

O objeto RequestBag contém informações armazenadas que são passadas de um manipulador de solicitação para outro ponto, e pode ser consumido no destino final. Este objeto também pode ser usado por manipuladores de solicitação que executam após o callback da rota.

> **TIP**
>
> Esta propriedade também é acessível pela propriedade HttpRequest.Bag.

Middleware/AuthenticateUserRequestHandler.cs C#

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

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

O manipulador de solicitação acima definirá `AuthenticatedUser` no request bag, e pode ser consumido mais tarde no callback final:

Controller/MyController.cs C#

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

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

Você também pode usar os métodos auxiliares `Bag.Set()` e `Bag.Get()` para obter ou definir objetos por seus singletons de tipo.

Middleware/Authenticate.cs C#

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

Controller/MyController.cs C#

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

Obtendo dados de formulário

Você pode obter os valores de dados de formulário em uma NameValueCollection com o exemplo abaixo:

Controller/Auth.cs C#

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

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

    if (AttempLogin(username, password))
    {
```

```
            ...
        }
    }
```

Obtendo dados de formulário multipart

A solicitação HTTP do Sisk permite obter conteúdos multipart carregados, como arquivos, campos de formulário ou qualquer conteúdo binário.

Controller/Auth.cs — C#

```
    [RoutePost("/upload-contents")]
    public HttpResponse Index(HttpRequest request)
    {
        // o método seguinte lê todo o input da solicitação em
        // um array de MultipartObjects
        var multipartFormDataObjects = request.GetMultipartFormContent();

        foreach (MultipartObject uploadedObject in multipartFormDataObjects)
        {
            // O nome do arquivo fornecido pelos dados de formulário multipart.
            // Retorna nulo se o objeto não for um arquivo.
            Console.WriteLine("File name       : " + uploadedObject.Filename);

            // O nome do campo do objeto de dados de formulário multipart.
            Console.WriteLine("Field name      : " + uploadedObject.Name);

            // O comprimento do conteúdo do formulário multipart.
            Console.WriteLine("Content length  : " + uploadedObject.ContentLength);

            // Determina o formato da imagem baseado no cabeçalho do arquivo para cada
            // tipo de conteúdo conhecido. Se o conteúdo não for um formato de arquivo comum
    reconhecido, este método abaixo retornará MultipartObjectCommonFormat.Unknown
            Console.WriteLine("Common format   : " + uploadedObject.GetCommonFileFormat());
        }
    }
```

Você pode ler mais sobre os objetos de formulário multipart do Sisk Multipart form objects e seus métodos, propriedades e funcionalidades.

Detectando desconexão do cliente

Desde a versão v1.15 do Sisk, o framework fornece um CancellationToken que é lançado quando a conexão entre o cliente e o servidor é fechada prematuramente antes de receber a resposta. Este token pode ser útil para detectar quando o cliente não deseja mais a resposta e cancelar operações de longa duração.

```
router.MapGet("/connect", async (HttpRequest req) ⇒
{
    // obtém o token de desconexão da solicitação
    var dc = req.DisconnectToken;

    await LongOperationAsync(dc);

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

Este token não é compatível com todos os motores HTTP, e cada um requer uma implementação.

Suporte a eventos enviados pelo servidor

O Sisk suporta Server-sent events, que permite enviar pedaços como um fluxo e manter a conexão entre o servidor e o cliente viva.

Chamar o método HttpRequest.GetEventSource colocará o HttpRequest em seu estado de ouvinte. A partir daí, o contexto desta solicitação HTTP não esperará um HttpResponse, pois irá sobrepor os pacotes enviados por eventos do lado do servidor.

Após enviar todos os pacotes, o callback deve retornar o método Close, que enviará a resposta final ao servidor e indicará que o streaming terminou.

Não é possível prever qual será o comprimento total de todos os pacotes que serão enviados, então não é possível determinar o fim da conexão com o cabeçalho `Content-Length`.

Por padrão, a maioria dos navegadores não suporta enviar cabeçalhos HTTP ou métodos diferentes do método GET. Portanto, tenha cuidado ao usar manipuladores de solicitação com requisições event-source que exigem cabeçalhos específicos na solicitação, pois provavelmente eles não terão.

Além disso, a maioria dos navegadores reinicia streams se o método EventSource.close não for chamado no lado do cliente após receber todos os pacotes, causando processamento adicional infinito no lado do servidor. Para evitar esse tipo de problema, é comum enviar um pacote final indicando que a fonte de eventos terminou de enviar todos os pacotes.

O exemplo abaixo mostra como o navegador pode comunicar ao servidor que suporta eventos do lado do servidor.

sse-example.html **HTML**

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

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

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

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

E enviar progressivamente as mensagens para o cliente:

Controller/MyController.cs **C#**

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

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

```

```
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  }
```

Ao executar este código, esperamos um resultado semelhante a este:

Resolvendo IPs e hosts proxy

O Sisk pode ser usado com proxies, e portanto endereços IP podem ser substituídos pelo endpoint do proxy na transação de um cliente para o proxy.

Você pode definir seus próprios resolvers no Sisk com forwarding resolvers.

Codificação de cabeçalhos

A codificação de cabeçalhos pode ser um problema para algumas implementações. No Windows, cabeçalhos UTF-8 não são suportados, então ASCII é usado. O Sisk possui um conversor de codificação embutido, que pode

ser útil para decodificar cabeçalhos incorretamente codificados.

Esta operação é custosa e desativada por padrão, mas pode ser habilitada sob o sinalizador NormalizeHeadersEncodings.

Respostas

Respostas representam objetos que são respostas HTTP para solicitações HTTP. Elas são enviadas pelo servidor ao cliente como uma indicação da solicitação de um recurso, página, documento, arquivo ou outro objeto.

Uma resposta HTTP é formada por status, cabeçalhos e conteúdo.

Neste documento, ensinaremos como arquitetar respostas HTTP com Sisk.

Definindo um status HTTP

A lista de status HTTP é a mesma desde o HTTP/1.0, e Sisk suporta todos eles.

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

Ou com Sintaxe Fluente:

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

Você pode ver a lista completa de HttpStatusCode disponíveis aqui. Você também pode fornecer seu próprio código de status usando a estrutura HttpStatusInformation.

Corpo e tipo de conteúdo

Sisk suporta objetos de conteúdo .NET nativos para enviar corpos em respostas. Você pode usar a classe StringContent para enviar uma resposta JSON, por exemplo:

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

O servidor sempre tentará calcular o `Content-Length` do que você definiu no conteúdo se você não o definiu explicitamente em um cabeçalho. Se o servidor não puder obter implicitamente o cabeçalho Content-Length do conteúdo da resposta, a resposta será enviada com Chunked-Encoding.

Você também pode transmitir a resposta enviando um StreamContent ou usando o método GetResponseStream.

Cabeçalhos de resposta

Você pode adicionar, editar ou remover cabeçalhos que está enviando na resposta. O exemplo abaixo mostra como enviar uma resposta de redirecionamento para o cliente.

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

Ou com Sintaxe Fluente:

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

Quando você usa o método Add de HttpHeaderCollection, você está adicionando um cabeçalho à solicitação sem alterar os que já foram enviados. O método Set substitui os cabeçalhos com o mesmo nome pelo valor instruído. O indexador de HttpHeaderCollection chama internamente o método Set para substituir os cabeçalhos.

Enviando cookies

Sisk tem métodos que facilitam a definição de cookies no cliente. Cookies definidos por este método já estão codificados em URL e atendem ao padrão RFC-6265.

```
1    HttpResponse res = new HttpResponse();
2    res.SetCookie("nome-do-cookie", "valor-do-cookie");
```

Ou com Sintaxe Fluente:

```
1    new HttpResponse(301)
2     .WithCookie("nome-do-cookie", "valor-do-cookie", expiresAt:
     DateTime.Now.Add(TimeSpan.FromDays(7)));
```

Existem outras versões mais completas do mesmo método.

Respostas chunked

Você pode definir a codificação de transferência como chunked para enviar respostas grandes.

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

Quando usar chunked-encoding, o cabeçalho Content-Length é automaticamente omitido.

Fluxo de resposta

Os fluxos de resposta são uma maneira gerenciada que permite enviar respostas de forma segmentada. É uma operação de nível mais baixo do que usar objetos HttpResponse, pois exigem que você envie os cabeçalhos e o conteúdo manualmente e, em seguida, feche a conexão.

Este exemplo abre um fluxo de leitura somente para o arquivo, copia o fluxo para o fluxo de saída da resposta e não carrega o arquivo inteiro na memória. Isso pode ser útil para servir arquivos médios ou grandes.

```
// obtém o fluxo de saída da resposta
using var fileStream = File.OpenRead("meu-arquivo-grande.zip");
var responseStream = request.GetResponseStream();

// define a codificação de resposta para usar chunked-encoding
// também você não deve enviar o cabeçalho content-length quando usar
// chunked encoding
responseStream.SendChunked = true;
responseStream.SetStatus(200);
responseStream.SetHeader(HttpKnownHeaderNames.ContentType, contentType);

// copia o fluxo do arquivo para o fluxo de saída da resposta
fileStream.CopyTo(responseStream.ResponseStream);

// fecha o fluxo
return responseStream.Close();
```

Compressão GZip, Deflate e Brotli

Você pode enviar respostas com conteúdo comprimido em Sisk com conteúdos HTTP comprimidos. Primeiramente, encapsule seu objeto HttpContent dentro de um dos compressores abaixo para enviar a resposta comprimida ao cliente.

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

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

Você também pode usar esses conteúdos comprimidos com fluxos.

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

 // não aplique "using" aqui. o HttpServer irá descartar seu conteúdo
 // após enviar a resposta.
```

```
    var arquivo = File.OpenRead("/caminho/para/arquivo-grande.zip");

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

Os cabeçalhos Content-Encoding são automaticamente definidos quando usados esses conteúdos.

Compressão automática

É possível comprimir automaticamente respostas HTTP com a propriedade EnableAutomaticResponseCompression. Essa propriedade encapsula automaticamente o conteúdo da resposta do roteador em um conteúdo comprimido que é aceito pela solicitação, desde que a resposta não seja herdada de um CompressedContent.

Apenas um conteúdo comprimido é escolhido para uma solicitação, escolhido de acordo com o cabeçalho Accept-Encoding, que segue a ordem:

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

Se a solicitação especificar que aceita qualquer um desses métodos de compressão, a resposta será automaticamente comprimida.

Tipos de resposta implícitos

Você pode usar outros tipos de retorno além de HttpResponse, mas é necessário configurar o roteador para lidar com cada tipo de objeto.

O conceito é sempre retornar um tipo de referência e transformá-lo em um objeto HttpResponse válido. Rotas que retornam HttpResponse não passam por nenhuma conversão.

Tipos de valor (estruturas) não podem ser usados como tipo de retorno porque não são compatíveis com o RouterCallback, portanto, devem ser encapsulados em um ValueResult para poderem ser usados em manipuladores.

Considere o exemplo a seguir de um módulo de roteador que não usa HttpResponse no tipo de retorno:

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

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

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

 return dUser;
 }

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

 return true;
 }
}
```

Com isso, agora é necessário definir no roteador como ele lidará com cada tipo de objeto. Objetos são sempre o primeiro argumento do manipulador e o tipo de saída deve ser um HttpResponse válido. Além disso, os objetos de saída de uma rota nunca devem ser nulos.

Para tipos ValueResult, não é necessário indicar que o objeto de entrada é um ValueResult e apenas T, pois ValueResult é um objeto refletido de seu componente original.

A associação de tipos não compara o que foi registrado com o tipo do objeto retornado do callback do roteador. Em vez disso, verifica se o tipo do resultado do roteador é atribuível ao tipo registrado.

Registrar um manipulador do tipo Object será um fallback para todos os tipos não validados anteriormente. A ordem de inserção dos manipuladores de valor também importa, portanto, registrar um manipulador de objeto primeiro ignorará todos os outros manipuladores específicos de tipo. Sempre registre manipuladores de valor específicos primeiro para garantir a ordem.

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

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

// registrando um manipulador de valor de objeto deve ser o último
// manipulador de valor que será usado como fallback
r.RegisterValueHandler<object>(fallback ⇒
{
 return new HttpResponse() {
 Status = HttpStatusCode.OK,
 Content = JsonContent.Create(fallback)
 };
});
```

Nota sobre objetos enumeráveis e arrays

Objetos de resposta implícitos que implementam IEnumerable são lidos na memória através do método `ToArray()` antes de serem convertidos através de um manipulador de valor definido. Para que isso ocorra, o objeto `IEnumerable` é convertido em uma matriz de objetos, e o conversor de resposta sempre receberá um `Object[]` em vez do tipo original.

Considere o cenário a seguir:

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

No exemplo acima, o conversor `IEnumerable<string>` **nunca será chamado**, porque o objeto de entrada sempre será um `Object[]` e não é conversível para um `IEnumerable<string>`. No entanto, o conversor abaixo que recebe um `IEnumerable<object>` receberá sua entrada, pois seu valor é compatível.

Se você precisar lidar com o tipo de objeto que será enumerado, você precisará usar reflexão para obter o tipo do elemento da coleção. Todos os objetos enumeráveis (listas, arrays e coleções) são convertidos em uma matriz de objetos pelo conversor de resposta HTTP.

Valores que implementam IAsyncEnumerable são tratados automaticamente pelo servidor se a propriedade ConvertIAsyncEnumerableIntoEnumerable estiver habilitada, semelhante ao que acontece com `IEnumerable`. Uma enumeração assíncrona é convertida em um enumerador bloqueador e, em seguida, convertida em uma matriz de objetos síncronos.

Logging

Você pode configurar o Sisk para escrever logs de acesso e erro automaticamente. É possível definir rotação de logs, extensões e frequência.

A classe LogStream fornece uma maneira assíncrona de escrever logs e mantê-los em uma fila de escrita aguardável.

Neste artigo mostraremos como configurar o logging para sua aplicação.

Logs de acesso baseados em arquivo

Logs para arquivos abrem o arquivo, escrevem o texto da linha e depois fecham o arquivo para cada linha escrita. Esse procedimento foi adotado para manter a responsividade de escrita nos logs.

Program.cs **C#**

```
class Program
{
    static async Task Main(string[] args)
    {
        using var app = HttpServer.CreateBuilder()
            .UseConfiguration(config => {
                config.AccessLogsStream = new LogStream("logs/access.log");
            })
            .Build();

         ...

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

O código acima escreverá todas as requisições recebidas no arquivo `logs/access.log`. Observe que, o arquivo é criado automaticamente se não existir, porém a pasta antes dele não. Não é necessário criar o diretório `logs/` pois a classe LogStream cria-o automaticamente.

Logging baseado em stream

Você pode escrever arquivos de log em instâncias de objetos TextWriter, como `Console.Out`, passando um objeto TextWriter no construtor:

Program.cs — C#

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

Para cada mensagem escrita no log baseado em stream, o método `TextWriter.Flush()` é chamado.

Formatação do log de acesso

Você pode customizar o formato do log de acesso por variáveis predefinidas. Considere a seguinte linha:

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

Ele escreverá uma mensagem como:

```
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]
```

Você pode formatar seu arquivo de log pelo formato descrito na tabela:

Valor	O que representa	Exemplo
%dd	Dia do mês (formatado como dois dígitos)	05
%dmmm	Nome completo do mês	Julho
%dmm	Nome abreviado do mês (três letras)	Jul

Valor	O que representa	Exemplo
%dm	Número do mês (formatado como dois dígitos)	07
%dy	Ano (formatado como quatro dígitos)	2023
%th	Hora em formato 12-horas	03
%tH	Hora em formato 24-horas (HH)	15
%ti	Minutos (formatado como dois dígitos)	30
%ts	Segundos (formatado como dois dígitos)	45
%tm	Milissegundos (formatado como três dígitos)	123
%tz	Deslocamento de fuso horário (horas totais em UTC)	+03:00
%ri	Endereço IP remoto do cliente	192.168.1.100
%rm	Método HTTP (maiúsculas)	GET
%rs	Esquema URI (http/https)	https
%ra	Autoridade URI (domínio)	example.com
%rh	Host da requisição	www.example.com
%rp	Porta da requisição	443
%rz	Caminho da requisição	/path/to/resource
%rq	String de consulta	?key=value&another=123
%sc	Código de status da resposta HTTP	200
%sd	Descrição do status da resposta HTTP	OK
%lin	Tamanho legível do pedido	1.2 KB
%linr	Tamanho bruto do pedido (bytes)	1234
%lou	Tamanho legível da resposta	2.5 KB
%lour	Tamanho bruto da resposta (bytes)	2560
%lms	Tempo decorrido em milissegundos	120
%ls	Status de execução	Executed

Valor	O que representa	Exemplo
%{header-name}	Representa o cabeçalho `header-name` da requisição.	`Mozilla/5.0 (platform; rv:gecko [ ... ]`
%{:res-name}	Representa o cabeçalho `res-name` da resposta.	

Rotação de logs

Você pode configurar o servidor HTTP para rotacionar os arquivos de log para um arquivo .gz comprimido quando atingirem um certo tamanho. O tamanho é verificado periodicamente pelo limite que você definir.

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

O código acima verificará a cada seis horas se o arquivo do LogStream atingiu seu limite de 64 MB. Se sim, o arquivo será comprimido para um arquivo .gz e então o `access.log` será limpo.

Durante esse processo, a escrita no arquivo fica bloqueada até que o arquivo seja comprimido e limpo. Todas as linhas que entrarem para serem escritas nesse período ficarão em uma fila aguardando o fim da compressão.

Esta função funciona apenas com LogStreams baseados em arquivo.

Logging de erros

Quando um servidor não lança erros para o depurador, ele encaminha os erros para escrita de logs quando houver algum. Você pode configurar a escrita de erros com:

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

Esta propriedade escreverá algo no log apenas se o erro não for capturado pelo callback ou pela propriedade Router.CallbackErrorHandler.

O erro escrito pelo servidor sempre grava a data e hora, os cabeçalhos da requisição (não o corpo), a trilha de erro e a trilha da exceção interna, se houver.

Outras instâncias de logging

Sua aplicação pode ter zero ou múltiplos LogStreams, não há limite para quantos canais de log ela pode ter. Portanto, é possível direcionar o log da sua aplicação para um arquivo diferente do AccessLog ou ErrorLog padrão.

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

Extensão do LogStream

Você pode estender a classe `LogStream` para escrever formatos personalizados, compatíveis com o motor de log atual do Sisk. O exemplo abaixo permite escrever mensagens coloridas no Console através da biblioteca Spectre.Console:

CustomLogStream.cs **C#**

```
public class CustomLogStream : LogStream
{
    protected override void WriteLineInternal(string line)
    {
        base.WriteLineInternal($"[{DateTime.Now:g}] {line}");
    }
}
```

Outra forma de escrever logs personalizados automaticamente para cada requisição/resposta é criar um HttpServerHandler. O exemplo abaixo é um pouco mais completo. Ele grava o corpo da requisição e resposta em

JSON no Console. Pode ser útil para depurar requisições em geral. Este exemplo usa ContextBag e HttpServerHandler.

Program.cs C#

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

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

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

JsonMessageHandler.cs C#

```
class JsonMessageHandler : HttpServerHandler
{
    protected override void OnHttpRequestOpen(HttpRequest request)
    {
        if (request.Method == HttpMethod.Get && request.Headers["Content-
Type"]?.Contains("json", StringComparison.InvariantCultureIgnoreCase) == true)
        {
            // Neste ponto, a conexão está aberta e o cliente enviou o
cabeçalho especificando
            // que o conteúdo é JSON. A linha abaixo lê o conteúdo e o deixa armazenado
na requisição.
            //
            // Se o conteúdo não for lido na ação da requisição, o GC provavelmente
```

```
coletará o conteúdo
            // após enviar a resposta ao cliente, então o conteúdo pode não estar
disponível após a resposta ser fechada.
            //
            _ = request.RawBody;

            // adiciona dica no contexto para indicar que esta requisição tem um corpo JSON
            request.Bag.Add("IsJsonRequest", true);
        }
    }

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

        if (result.Request.Bag.ContainsKey("IsJsonRequest"))
        {
            // reformatta o JSON usando a biblioteca CypherPotato.LightJson
            var content = result.Request.Body;
            requestJson = JsonValue.Deserialize(content, new JsonOptions() { WriteIndented
= true }).ToString();
        }

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

            if (content is HttpContent httpContent &&
                // verifica se a resposta é JSON
                httpContent.Headers.ContentType?.MediaType?.Contains("json",
StringComparison.InvariantCultureIgnoreCase) == true)
            {
                string json = await httpContent.ReadAsStringAsync();
                responseJson = JsonValue.Deserialize(json, new JsonOptions() {
WriteIndented = true }).ToString();
            }
        }
        else
        {
            // obtém o status interno de tratamento do servidor
            responseMessage = result.Status.ToString();
        }

        StringBuilder outputMessage = new StringBuilder();
```

```
62
63          if (requestJson != null)
64          {
65              outputMessage.AppendLine("-----");
66              outputMessage.AppendLine($">>> {result.Request.Method} {result.Request.Path}");
67
68              if (requestJson is not null)
69                  outputMessage.AppendLine(requestJson);
70          }
71
72          outputMessage.AppendLine($"<<< {responseMessage}");

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

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

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

Eventos Enviados pelo Servidor

O Sisk suporta o envio de mensagens por meio de Eventos Enviados pelo Servidor fora da caixa. Você pode criar conexões descartáveis e persistentes, obter as conexões durante a execução e usá-las.

Essa funcionalidade tem algumas limitações impostas pelos navegadores, como enviar apenas mensagens de texto e não ser capaz de fechar permanentemente uma conexão. Uma conexão fechada pelo servidor terá um cliente tentando se reconectar periodicamente a cada 5 segundos (3 para alguns navegadores).

Essas conexões são úteis para enviar eventos do servidor para o cliente sem que o cliente solicite as informações todas as vezes.

Criando uma conexão SSE

Uma conexão SSE funciona como uma solicitação HTTP regular, mas em vez de enviar uma resposta e fechar imediatamente a conexão, a conexão é mantida aberta para enviar mensagens.

Chamando o método HttpRequest.GetEventSource(), a solicitação é colocada em um estado de espera enquanto a instância SSE é criada.

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

No código acima, criamos uma conexão SSE e enviamos uma mensagem "Olá, mundo", então fechamos a conexão SSE do lado do servidor.

> **NOTE**
>
> Ao fechar uma conexão do lado do servidor, por padrão o cliente tentará se conectar novamente naquele ponto e a conexão será reiniciada, executando o método novamente, para sempre.
>
> É comum encaminhar uma mensagem de término do servidor sempre que a conexão for fechada do lado do servidor para evitar que o cliente tente se reconectar novamente.

Anexando cabeçalhos

Se você precisar enviar cabeçalhos, você pode usar o método HttpRequestEventSource.AppendHeader antes de enviar quaisquer mensagens.

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

Observe que é necessário enviar os cabeçalhos antes de enviar quaisquer mensagens.

Conexões Wait-For-Fail

As conexões normalmente são encerradas quando o servidor não é mais capaz de enviar mensagens devido a uma possível desconexão do lado do cliente. Com isso, a conexão é automaticamente encerrada e a instância da classe é descartada.

Mesmo com uma reconexão, a instância da classe não funcionará, pois está vinculada à conexão anterior. Em algumas situações, você pode precisar dessa conexão posteriormente e não deseja gerenciá-la por meio do método de retorno de chamada da rota.

Para isso, podemos identificar as conexões SSE com um identificador e obtê-las usando-o posteriormente, até mesmo fora do retorno de chamada da rota. Além disso, marcamos a conexão com WaitForFail para não encerrar a rota e encerrar a conexão automaticamente.

Uma conexão SSE em KeepAlive aguardará um erro de envio (causado por desconexão) para retomar a execução do método. Também é possível definir um tempo limite para isso. Após o tempo, se nenhuma mensagem tiver sido enviada, a conexão será encerrada e a execução será retomada.

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

 sse.WaitForFail(TimeSpan.FromSeconds(15)); // aguarde 15 segundos sem nenhuma mensagem antes de encerrar a conexão

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

O método acima criará a conexão, manipulará-la e aguardará uma desconexão ou erro.

```
HttpRequestEventSource? evs = server.EventSources.GetByIdentifier("minha-conexao-index");
if (evs =/= null)
{
 // a conexão ainda está ativa
 evs.Send("Olá novamente!");
}
```

E o trecho acima tentará procurar a conexão recém-criada e, se existir, enviará uma mensagem para ela.

Todas as conexões ativas do servidor que são identificadas estarão disponíveis na coleção HttpServer.EventSources. Essa coleção armazena apenas conexões ativas e identificadas. Conexões fechadas são removidas da coleção.

> **NOTE**
>
> É importante notar que o keep alive tem um limite estabelecido por componentes que podem estar conectados ao Sisk de forma incontrolável, como um proxy da web, um kernel HTTP ou um driver de rede, e eles fecham conexões ociosas após um determinado período de tempo.
>
> Portanto, é importante manter a conexão aberta enviando pings periódicos ou estendendo o tempo máximo antes que a conexão seja fechada. Leia a próxima seção para entender melhor o envio de pings periódicos.

Configurar política de ping da conexão

A Política de Ping é uma maneira automatizada de enviar mensagens periódicas para o cliente. Essa função permite que o servidor entenda quando o cliente foi desconectado dessa conexão sem ter que manter a conexão aberta indefinidamente.

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

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

No código acima, a cada 5 segundos, uma nova mensagem de ping será enviada para o cliente. Isso manterá a conexão TCP ativa e evitará que ela seja fechada devido à inatividade. Além disso, quando uma mensagem falhar ao ser enviada, a conexão será automaticamente fechada, liberando os recursos usados pela conexão.

Consultando conexões

Você pode pesquisar conexões ativas usando um predicado no identificador da conexão, para poder transmitir, por exemplo.

```
HttpRequestEventSource[] evs = server.EventSources.Find(es ⇒ es.StartsWith("minha-
conexao-"));
foreach (HttpRequestEventSource e in evs)
{
 e.Send("Transmitindo para todas as fontes de eventos que começam com 'minha-conexao-'");
}
```

Você também pode usar o método All para obter todas as conexões SSE ativas.

Web Sockets

Sisk suporta web sockets também, como receber e enviar mensagens para o seu cliente.

Este recurso funciona bem na maioria dos navegadores, mas no Sisk ainda está experimental. Por favor, se encontrar algum bug, reporte no GitHub.

Aceitando mensagens assincronamente

Mensagens WebSocket são recebidas em ordem, enfileiradas até serem processadas por `ReceiveMessageAsync`. Este método não retorna mensagem quando o timeout é atingido, quando a operação é cancelada ou quando o cliente se desconecta.

Só pode ocorrer uma operação de leitura e escrita simultaneamente, portanto, enquanto você espera por uma mensagem com `ReceiveMessageAsync`, não é possível escrever para o cliente conectado.

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

    while (await ws.ReceiveMessageAsync(timeout: TimeSpan.FromSeconds(30)) is {
} receivedMessage)
    {
        string msgText = receivedMessage.GetString();
        Console.WriteLine("Received message: " + msgText);

        await ws.SendAsync("Hello!");
    }

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

Aceitando mensagens sincronamente

O exemplo abaixo contém uma forma de usar um websocket síncrono, sem contexto assíncrono, onde você recebe as mensagens, trata-as e termina usando o socket.

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

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

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

    string name = msg.GetString();

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

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

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

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

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

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

Política de Ping

Semelhante à política de ping em Server Side Events, você também pode configurar uma política de ping para manter a conexão TCP aberta caso haja inatividade nela.

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

Sintaxe de descarte

O servidor HTTP pode ser usado para escutar uma solicitação de retorno de uma ação, como autenticação OAuth, e pode ser descartado após receber essa solicitação. Isso pode ser útil em casos em que você precisa de uma ação em segundo plano, mas não deseja configurar um aplicativo HTTP inteiro para isso.

O exemplo a seguir mostra como criar um servidor HTTP ouvindo na porta 5555 com CreateListener e aguardar o próximo contexto:

```
using (var server = HttpServer.CreateListener(5555))
{
    // aguardar a próxima solicitação HTTP
    var context = await server.WaitNextAsync();
    Console.WriteLine($"Caminho solicitado: {context.Request.Path}");
}
```

A função WaitNext aguarda o próximo contexto de um processamento de solicitação concluído. Uma vez que o resultado dessa operação é obtido, o servidor já tratou completamente a solicitação e enviou a resposta para o cliente.

Instanciando membros por solicitação

É comum dedicar membros e instâncias que duram por toda a vida de uma solicitação, como uma conexão com banco de dados, um usuário autenticado ou um token de sessão. Uma das possibilidades é através do HttpContext.RequestBag, que cria um dicionário que dura por toda a vida de uma solicitação.

Este dicionário pode ser acessado por tratadores de solicitação e definir variáveis durante toda a solicitação. Por exemplo, um tratador de solicitação que autentica um usuário define esse usuário dentro do `HttpContext.RequestBag`, e dentro da lógica da solicitação, esse usuário pode ser recuperado com `HttpContext.RequestBag.Get<User>()`.

Aqui está um exemplo:

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

    public HttpResponse? Execute(HttpRequest request, HttpContext context)
    {
        User authenticatedUser = AuthenticateUser(request);
        context.RequestBag.Set(authenticatedUser);
        return null; // avançar para o próximo tratador de solicitação ou lógica
da solicitação
    }
}

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

Este é um exemplo preliminar dessa operação. A instância de `User` foi criada dentro do tratador de solicitação dedicado à autenticação, e todas as rotas que usam esse tratador de solicitação terão a garantia de que haverá um `User` em sua instância de `HttpContext.RequestBag`.

É possível definir lógica para obter instâncias quando não definidas anteriormente no `RequestBag` através de métodos como GetOrAdd ou GetOrAddAsync.

Desde a versão 1.3, a propriedade estática HttpContext.Current foi introduzida, permitindo o acesso ao `HttpContext` atualmente em execução do contexto da solicitação. Isso permite expor membros do `HttpContext` fora do contexto da solicitação atual e definir instâncias em objetos de rota.

O exemplo abaixo define um controlador que possui membros comumente acessados pelo contexto de uma solicitação.

```
public abstract class Controller : RouterModule
{
    public DbContext Database
    {
        get
        {
            // criar um DbContext ou obter o existente
            return HttpContext.Current.RequestBag.GetOrAdd(() => new DbContext());
        }
    }

    // a linha seguinte lançará uma exceção se a propriedade for acessada quando o User não
    // estiver definido no bag da solicitação
    public User AuthenticatedUser { get => HttpContext.Current.RequestBag.Get<User>(); }

    // Expor a instância HttpRequest também é suportado
    public HttpRequest Request { get => HttpContext.Current.Request; }
}
```

E definir tipos que herdam do controlador:

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

    [RouteGet("<id>")]
    public Post GetPost()
    {
```

```
        int blogId = Request.RouteParameters["id"].GetInteger();

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

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

Para o exemplo acima, você precisará configurar um tratador de valor no seu roteador para que os objetos retornados pelo roteador sejam transformados em um HttpResponse válido.

Observe que os métodos não têm um argumento `HttpRequest request` como presente em outros métodos. Isso ocorre porque, desde a versão 1.3, o roteador suporta dois tipos de delegados para roteamento de respostas: RouteAction, que é o delegado padrão que recebe um argumento `HttpRequest`, e ParameterlessRouteAction. O objeto `HttpRequest` ainda pode ser acessado por ambos os delegados através da propriedade Request do `HttpContext` estático no thread.

No exemplo acima, definimos um objeto descartável, o `DbContext`, e precisamos garantir que todas as instâncias criadas em um `DbContext` sejam descartadas quando a sessão HTTP terminar. Para isso, podemos usar duas maneiras de alcançar isso. Uma é criar um tratador de solicitação que é executado após a ação do roteador, e a outra maneira é através de um tratador de servidor personalizado.

Para o primeiro método, podemos criar o tratador de solicitação inline diretamente no método OnSetup herdado de `RouterModule`:

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

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

        HasRequestHandler(RequestHandler.Create(
            execute: (req, ctx) =>
            {
                // obter um DbContext definido no contexto do tratador de solicitação e
                // descartá-lo
                ctx.RequestBag.GetOrDefault<DbContext>()?.Dispose();
                return null;
            },
            executionMode: RequestHandlerExecutionMode.AfterResponse));
```

```
    }
}
```

O método acima garantirá que o `DbContext` seja descartado quando a sessão HTTP for finalizada. Você pode fazer isso para mais membros que precisam ser descartados no final de uma resposta.

Para o segundo método, você pode criar um tratador de servidor personalizado que descartará o `DbContext` quando a sessão HTTP for finalizada.

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

E usá-lo no seu construtor:

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

Esta é uma maneira de lidar com a limpeza de código e manter as dependências de uma solicitação separadas pelo tipo de módulo que será usado, reduzindo a quantidade de código duplicado dentro de cada ação de um roteador. É uma prática semelhante ao que a injeção de dependência é usada em frameworks como ASP.NET.

Streaming de Conteúdo

O Sisk suporta a leitura e o envio de fluxos de conteúdo para e do cliente. Essa funcionalidade é útil para remover a sobrecarga de memória para serializar e deserializar conteúdo durante a vida útil de uma solicitação.

Fluxo de Conteúdo da Solicitação

Conteúdos pequenos são carregados automaticamente no buffer de memória da conexão HTTP, carregando rapidamente esse conteúdo para HttpRequest.Body e HttpRequest.RawBody. Para conteúdos maiores, o método HttpRequest.GetRequestStream pode ser usado para obter o fluxo de leitura do conteúdo da solicitação.

É importante notar que o método HttpRequest.GetMultipartFormContent lê todo o conteúdo da solicitação na memória, portanto, pode não ser útil para ler conteúdos grandes.

Considere o seguinte exemplo:

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           // solicitação não tem conteúdo
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 HttpResponse () {
22          Content = JsonContent.Create ( new { message = "Arquivo enviado com sucesso." } )
23      };
24  }
```

No exemplo acima, o método `UploadDocument` lê o conteúdo da solicitação e salva o conteúdo em um arquivo. Nenhuma alocação adicional de memória é feita, exceto pelo buffer de leitura usado por `Stream.CopyToAsync`. O exemplo acima remove a pressão de alocação de memória para um arquivo muito grande, o que pode otimizar o desempenho da aplicação.

Uma boa prática é sempre usar um CancellationToken em uma operação que possa ser demorada, como enviar arquivos, pois depende da velocidade da rede entre o cliente e o servidor.

A ajuste com um CancellationToken pode ser feito da seguinte forma:

Controller/UploadDocument.cs **C#**

```
1   // o token de cancelamento abaixo irá lançar uma exceção se o tempo limite de 30 segundos
2   for atingido.
3   CancellationTokenSource copyCancellation = new CancellationTokenSource ( delay:
4   TimeSpan.FromSeconds ( 30 ) );
5
6   try {
7       using (var fs = File.Create ( outputFileName )) {
8           await contentStream.CopyToAsync ( fs, copyCancellation.Token );
9       }
10  }
11  catch (OperationCanceledException) {
12      return new HttpResponse ( HttpStatusInformation.BadRequest ) {
13          Content = JsonContent.Create ( new { Error = "O upload excedeu o tempo máximo de
    upload (30 segundos)." } )
        };
    }
```

Fluxo de Conteúdo da Resposta

Enviar conteúdo de resposta também é possível. Atualmente, existem duas maneiras de fazer isso: através do método HttpRequest.GetResponseStream e usando um conteúdo do tipo StreamContent.

Considere um cenário em que precisamos servir um arquivo de imagem. Para fazer isso, podemos usar o seguinte código:

Controller/ImageController.cs C#

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

    // método de exemplo para obter uma imagem de perfil
    var profilePictureFilename = "profile-picture.jpg";
    byte[] profilePicture = await File.ReadAllBytesAsync ( profilePictureFilename );

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

O método acima faz uma alocação de memória a cada vez que lê o conteúdo da imagem. Se a imagem for grande, isso pode causar um problema de desempenho, e em situações de pico, até mesmo uma sobrecarga de memória e travar o servidor. Nesses casos, o cache pode ser útil, mas não eliminará o problema, pois a memória ainda será reservada para esse arquivo. O cache aliviará a pressão de ter que alocar memória para cada solicitação, mas para arquivos grandes, não será suficiente.

Enviar a imagem por meio de um fluxo pode ser uma solução para o problema. Em vez de ler todo o conteúdo da imagem, um fluxo de leitura é criado no arquivo e copiado para o cliente usando um buffer pequeno.

Enviar através do método GetResponseStream

O método HttpRequest.GetResponseStream cria um objeto que permite enviar pedaços da resposta HTTP à medida que o fluxo de conteúdo é preparado. Esse método é mais manual, exigindo que você defina o status, cabeçalhos e tamanho do conteúdo antes de enviar o conteúdo.

Controller/ImageController.cs C#

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

    var profilePictureFilename = "profile-picture.jpg";

```

```
        // nessa forma de envio, o status e o cabeçalho devem ser definidos
        // antes de enviar o conteúdo
        var requestStreamManager = request.GetResponseStream ();

        requestStreamManager.SetStatus ( System.Net.HttpStatusCode.OK );
        requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentType, "image/jpeg" );
        requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentDisposition, $"inline;
filename={profilePictureFilename}" );

        using (var fs = File.OpenRead ( profilePictureFilename )) {

            // nessa forma de envio, também é necessário definir o tamanho do conteúdo
            // antes de enviá-lo.
            requestStreamManager.SetContentLength ( fs.Length );

            // se você não souber o tamanho do conteúdo, pode usar o codificação em chunk
            // para enviar o conteúdo
            requestStreamManager.SendChunked = true;

            // e então, escrever no fluxo de saída
            await fs.CopyToAsync ( requestStreamManager.ResponseStream );
        }
    }
```

Enviar conteúdo através de um StreamContent

A classe StreamContent permite enviar conteúdo de uma fonte de dados como um fluxo de bytes. Essa forma de envio é mais fácil, removendo os requisitos anteriores, e até mesmo permitindo o uso de codificação de compressão para reduzir o tamanho do conteúdo.

Controller/ImageController.cs **C#**

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

    var profilePictureFilename = "profile-picture.jpg";

    return new HttpResponse () {
        Content = new StreamContent ( File.OpenRead ( profilePictureFilename ) ),
        Headers = new () {
            ContentType = "image/jpeg",
            ContentDisposition = $"inline; filename=\"{profilePictureFilename}\""
        }


```

```
        };
    }
```

> **IMPORTANT**
>
> Nesse tipo de conteúdo, não encapsule o fluxo em um bloco `using`. O conteúdo será automaticamente descartado pelo servidor HTTP quando o fluxo de conteúdo for finalizado, com ou sem erros.

Habilitando CORS (Compartilhamento de Recursos entre Origens) no Sisk

O Sisk possui uma ferramenta que pode ser útil para lidar com Cross-Origin Resource Sharing (CORS) quando expõe seu serviço publicamente. Esse recurso não faz parte do protocolo HTTP, mas é uma funcionalidade específica dos navegadores definidos pelo W3C. Esse mecanismo de segurança impede que uma página web faça requisições para um domínio diferente daquele que forneceu a página. Um provedor de serviço pode permitir que certos domínios acessem seus recursos, ou apenas um.

Mesma Origem

Para que um recurso seja identificado como "mesma origem", uma requisição deve identificar o cabeçalho Origin em sua requisição:

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

E o servidor remoto deve responder com um cabeçalho Access-Control-Allow-Origin com o mesmo valor da origem solicitada:

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

Essa verificação é **explícita**: host, porta e protocolo devem ser iguais ao solicitados. Veja o exemplo:

- Um servidor responde que seu `Access-Control-Allow-Origin` é `https://example.com`:
  - `https://example.net` – o domínio é diferente.
  - `http://example.com` – o esquema é diferente.
  - `http://example.com:5555` – a porta é diferente.
  - `https://www.example.com` – o host é diferente.

Na especificação, apenas a sintaxe é permitida para ambos os cabeçalhos, tanto para requisições quanto respostas. O caminho da URL é ignorado. A porta também é omitida se for uma porta padrão (80 para HTTP e 443 para HTTPS).

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

Habilitando CORS

Nativamente, você tem o objeto CrossOriginResourceSharingHeaders dentro do seu ListeningHost.

Você pode configurar o CORS ao inicializar o servidor:

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

    await app.StartAsync();
}
```

O código acima enviará os seguintes cabeçalhos para **todas as respostas**:

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

Esses cabeçalhos precisam ser enviados para todas as respostas a um cliente web, incluindo erros e redirecionamentos.

Você pode notar que a classe CrossOriginResourceSharingHeaders possui duas propriedades semelhantes: AllowOrigin e AllowOrigins. Observe que uma é plural, enquanto a outra é singular.

- A propriedade **AllowOrigin** é estática: apenas a origem que você especificar será enviada para todas as respostas.
- A propriedade **AllowOrigins** é dinâmica: o servidor verifica se a origem da requisição está contida nesta lista. Se for encontrada, ela será enviada na resposta dessa origem.

Wildcards e cabeçalhos automáticos

Alternativamente, você pode usar um curinga (`*`) na origem da resposta para indicar que qualquer origem pode acessar o recurso. No entanto, esse valor não é permitido para requisições que têm credenciais (cabeçalhos de autorização) e essa operação resultará em um erro.

Você pode contornar esse problema listando explicitamente quais origens serão permitidas através da propriedade AllowOrigins ou também usar a constante AutoAllowOrigin no valor de AllowOrigin. Essa propriedade mágica definirá o cabeçalho `Access-Control-Allow-Origin` para o mesmo valor que o cabeçalho `Origin` da requisição.

Você também pode usar AutoFromRequestMethod e AutoFromRequestHeaders para comportamento semelhante ao `AllowOrigin` , que responde automaticamente com base nos cabeçalhos enviados.

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

        // Responde com base no cabeçalho Origin da requisição
        allowOrigin: CrossOriginResourceSharingHeaders.AutoAllowOrigin,

        // Responde com base no cabeçalho Access-Control-Request-Method ou no método da requisição
        allowMethods: [CrossOriginResourceSharingHeaders.AutoFromRequestMethod],

        // Responde com base no cabeçalho Access-Control-Request-Headers ou nos cabeçalhos enviados
        allowHeaders: [CrossOriginResourceSharingHeaders.AutoFromRequestHeaders]))
```

Outros Métodos de Aplicar CORS

Se você estiver lidando com service providers, pode substituir valores definidos no arquivo de configuração:

```
static async Task Main(string[] args)
{
    using var app = HttpServer.CreateBuilder()
        .UsePortableConfiguration( ... )
        .UseCors(cors ⇒ {
            // Substituirá a origem definida na configuração
            // do arquivo.
            cors.AllowOrigin = "http://example.com";
        })
        .Build();

    await app.StartAsync();
}
```

Desabilitando CORS em Rotas Específicas

A propriedade `UseCors` está disponível para todas as rotas e todos os atributos de rota e pode ser desativada com o exemplo abaixo:

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

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

Substituindo Valores na Resposta

Você pode substituir ou remover valores explicitamente em uma ação de roteador:

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

    public IEnumerable<string> GetWidgets(HttpRequest request) {

        // Remove o cabeçalho Access-Control-Allow-Credentials
        request.Context.OverrideHeaders.AccessControlAllowCredentials = string.Empty;

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

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

Requisições Preflight

Uma requisição preflight é uma requisição do método OPTIONS que o cliente envia antes da requisição real.

O servidor Sisk sempre responderá à requisição com um `200 OK` e os cabeçalhos CORS aplicáveis, e então o cliente pode prosseguir com a requisição real. Essa condição só não é aplicada quando existe uma rota para a requisição com o RouteMethod explicitamente configurado para `Options`.

Desabilitando CORS Globalmente

Não é possível fazer isso. Para não usar CORS, não o configure.

Sisk + JSON-RPC

Sisk possui um módulo experimental de uma API JSON-RPC 2.0, o que possibilita criar aplicações ainda mais simples. Essa extensão implementa extritamente a interface de transporte JSON-RPC 2.0, e oferece transporte via requisições HTTP GET, POST e também web-sockets com Sisk.

Você pode instalar a extensão pelo Nuget com o comando abaixo. Note que, em versões experimentais/betas, deverá ser ativado a opção de buscar por pacotes em pré-lançamento pelo Visual Studio.

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

Interface de transporte

O JSON-RPC é um protocolo de execução remota de procedimentos (RDP) sem estado, assíncrono, e utiliza o JSON para comunicação unilateral dos dados. Uma requisição JSON-RPC é normalmente identificada por um ID, e uma resposta é entregue pelo mesmo ID que foi enviado na requisição. Nem todas as requisições são necessárias de resposta, o que são chamadas de "notificações".

Na especificação do JSON-RPC 2.0 é explicado com detalhes como o transporte funciona. Este transporte é agnóstico de onde será usado. O Sisk implementa esse protocolo através do HTTP, seguindo as conformidades do JSON-RPC over HTTP, que suporta parcialmente requisições GET, mas completamente requisições POST. Também é suportado o uso de web-sockets, o que provê uma comunicação assíncrona de mensagens.

Uma requisição JSON-RPC é parecida com:

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

E uma resposta, quando bem-sucedida, é semelhante:

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

Métodos JSON-RPC

O exemplo a seguir mostra como criar uma API JSON-RPC usando o Sisk. Uma classe de operações matemáticas realiza as operações remotas e entrega a resposta serializada ao cliente.

```
using var app = HttpServer.CreateBuilder(port: 5555)
    .UseJsonRPC((sender, args) =>
    {
        // add all methods tagged with WebMethod to the JSON-RPC handler
        args.Handler.Methods.AddMethodsFromType(new MathOperations());

        // maps the /service route to handle JSON-RPC POST and GET requests
        args.Router.MapPost("/service", args.Handler.Transport.HttpPost);
        args.Router.MapGet("/service", args.Handler.Transport.HttpGet);

        // creates an websocket handler on GET /ws
        args.Router.MapGet("/ws", request =>
        {
            var ws = request.GetWebSocket();
            ws.OnReceive += args.Handler.Transport.WebSocket;

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

await app.StartAsync();

public class MathOperations
{
    [WebMethod]
    public float Sum(float a, float b)
    {
```

```
        return a + b;
    }

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

O exemplo acima irá mapear os métodos `Sum` e `Sqrt` para o handler JSON-RPC, e estes métodos ficarão disponíveis no `GET /service`, `POST /service` e `GET /ws`. O nome dos métodos não são sensíveis a caso.

Os parâmetros dos métodos são deserializados automaticamente para seus tipos específicos. Usar uma requisição com parâmetros nomeados também é suportado. A serialização JSON é feita pela biblioteca LightJson. Quando um tipo não é corretamente deserializado, você poderá criar um conversor JSON específico para aquele tipo e associar ele em seu JsonSerializerOptions posteriormente.

Você também pode obter o objeto `$.params` cru da requisição JSON-RPC diretamente no seu método.

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

Para isso ocorrer, `@params` deve ser o **único** parâmetro no seu método, com exatamente o nome `params` (em C#, o `@` é necessário para escapar o nome deste parâmetro).

A deserialização dos parâmetros ocorre tanto para objetos nomeados ou para arrays posicionais. Por exemplo, o método abaixo pode ser chamado remotamente por ambas requisições:

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

Por um objeto em `params` não é necessário seguir a ordem dos parâmetros:

```
{
    "jsonrpc": "2.0",
    "method": "AddUserToStore",
```

```
    "params": {
        "apiKey": "1234567890",
        "store": {
            "name": "My Store"
        },
        "user": {
            "name": "John Doe",
            "email": "john@example.com"
        }
    },
    "id": 1

}
```

Por array é necessário seguir a ordem dos parâmetros.

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

}
```

Personalizando o serializer

Você pode personalizar o serializador JSON na propriedade JsonRpcHandler.JsonSerializerOptions. Nessa propriedade, você pode ativar o uso de JSON5 para deserialização de mensagens. Por mais que não é conformidade com o JSON-RPC 2.0, JSON5 é uma extensão do JSON que permite uma escrita mais humanizada e legível.

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

        // usa um comparador de nomes sanitizado. esse comparador compara apenas letras
        // e dígitos em um nome, e descarta outros símbolos. ex:
        // foo_bar10 == FooBar10
        e.Handler.JsonSerializerOptions.PropertyNameComparer = new
JsonSanitizedComparer ();

        // habilita JSON5 para o interpretador JSON. mesmo ativando isso, JSON plano ainda
é permitido
        e.Handler.JsonSerializerOptions.SerializationFlags =
LightJson.Serialization.JsonSerializationFlags.Json5;

        // mapeia a rota POST /service para o handler JSON RPC
        e.Router.MapPost ( "/service", e.Handler.Transport.HttpPost );
    } )
    .Build ();

host.Start ();
```

Proxy SSL

> **WARNING**
>
> Este recurso é experimental e não deve ser usado em produção. Consulte este documento se quiser fazer o Sisk funcionar com SSL.

O Proxy SSL do Sisk é um módulo que fornece uma conexão HTTPS para um ListeningHost no Sisk e roteia mensagens HTTPS para um contexto HTTP inseguro. O módulo foi construído para fornecer conexão SSL para um serviço que usa HttpListener para executar, que não suporta SSL.

O proxy é executado dentro do mesmo aplicativo e escuta por mensagens HTTP/1.1, encaminhando-as para o Sisk no mesmo protocolo. Atualmente, esse recurso é altamente experimental e pode ser instável o suficiente para não ser usado em produção.

No momento, o SslProxy suporta quase todos os recursos HTTP/1.1, como keep-alive, codificação chunked, websockets, etc. Para uma conexão aberta ao proxy SSL, uma conexão TCP é criada para o servidor de destino e o proxy é encaminhado para a conexão estabelecida.

O SslProxy pode ser usado com HttpServer.CreateBuilder da seguinte forma:

```
using var app = HttpServer.CreateBuilder(port: 5555)
    .UseRouter(r =>
    {
        r.MapGet("/", request =>
        {
            return new HttpResponse("Hello, world!");
        });
    })
    // adicionar SSL ao projeto
    .UseSsl(
        sslListeningPort: 5567,
        new X509Certificate2(@".\ssl.pfx", password: "12345")
    )
    .Build();

app.Start();
```

Você deve fornecer um certificado SSL válido para o proxy. Para garantir que o certificado seja aceito pelos navegadores, lembre-se de importá-lo no sistema operacional para que ele funcione corretamente.

Autenticação Básica

O pacote Basic Auth adiciona um processador de solicitações capaz de lidar com o esquema de autenticação básica em seu aplicativo Sisk com pouca configuração e esforço. A autenticação HTTP básica é uma forma mínima de entrada de autenticação de solicitações por um ID de usuário e senha, onde a sessão é controlada exclusivamente pelo cliente e não há tokens de autenticação ou acesso.

Leia mais sobre o esquema de autenticação básica na especificação MDN.

Instalando

Para começar, instale o pacote Sisk.BasicAuth em seu projeto:

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

Você pode visualizar mais maneiras de instalá-lo em seu projeto no repositório Nuget.

Criando seu manipulador de autenticação

Você pode controlar o esquema de autenticação para um módulo inteiro ou para rotas individuais. Para isso, primeiro vamos escrever nosso primeiro manipulador de autenticação básica.

No exemplo abaixo, uma conexão é feita com o banco de dados, verifica se o usuário existe e se a senha é válida e, em seguida, armazena o usuário na bolsa de contexto.

```
1   public class UserAuthHandler : BasicAuthenticateRequestHandler
2   {
3       public UserAuthHandler() : base()
4       {
5           Realm = "Para entrar nesta página, informe suas credenciais.";
```

```
    }

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

        // neste caso, estamos usando o email como campo de ID de usuário, então vamos
        // procurar um usuário usando seu email.
        User? user = db.Users.FirstOrDefault(u => u.Email == credentials.UserId);
        if (user == null)
        {
            return base.CreateUnauthorizedResponse("Desculpe! Nenhum usuário foi encontrado
por este email.");
        }

        // valida que a senha das credenciais é válida para este usuário.
        if (!user.ValidatePassword(credentials.Password))
        {
            return base.CreateUnauthorizedResponse("Credenciais inválidas.");
        }

        // adiciona o usuário logado ao contexto HTTP
        // e continua a execução
        context.Bag.Add("loggedUser", user);
        return null;
    }
}
```

Então, basta associar este manipulador de solicitações à nossa rota ou classe.

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

Ou usando a classe RouterModule:

```
public class UsersController : RouterModule
{
    public ClientModule()
    {
        // agora todas as rotas dentro desta classe serão tratadas por
        // UserAuthHandler.
        base.HasRequestHandler(new UserAuthHandler());
    }

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

Observações

A principal responsabilidade da autenticação básica é realizada no lado do cliente. Armazenamento, controle de cache e criptografia são todos gerenciados localmente no cliente. O servidor apenas recebe as credenciais e valida se o acesso é permitido ou não.

Observe que este método não é uma das soluções mais seguras porque coloca uma responsabilidade significativa no cliente, o que pode ser difícil de rastrear e manter a segurança de suas credenciais. Além disso, é essencial que as senhas sejam transmitidas em um contexto de conexão segura (SSL), pois elas não possuem nenhuma criptografia inerente. Uma breve interceptação nos cabeçalhos de uma solicitação pode expor as credenciais de acesso do seu usuário.

Opte por soluções de autenticação mais robustas para aplicativos em produção e evite usar muitos componentes prontos para uso, pois eles podem não se adaptar às necessidades do seu projeto e acabar expondo-o a riscos de segurança.

Service Providers

Service Providers é uma forma de portar sua aplicação Sisk para diferentes ambientes com um arquivo de configuração portátil. Este recurso possibilita alterar funcionamento de portas, parâmetros e demais opções do servidor sem ter que alterar o código do aplicativo para cada ambiente. Este módulo depende da sintaxe de construção do Sisk e pode ser configurada através do método UsePortableConfiguration.

Um provedor de configuração é implementado com IConfigurationProvider, o que provê um leitor de configurações e pode receber qualquer implementação. Por padrão, o Sisk fornece um leitor de configurações em JSON, mas também existe um pacote para arquivos INI. Você também pode criar seu próprio provedor de configurações e registrar com:

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

Como mencionado anteriormente, o provedor padrão é de um arquivo JSON. Por padrão, o nome do arquivo buscado é `service-config.json`, e é buscado na pasta atual (Current Directory) do processo em execução, e não do diretório do executável.

Você pode optar em alterar o nome do arquivo, bem como onde o Sisk deve procurar pelo arquivo de configuração, com:

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

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

O código acima irá procurar o arquivo `config.toml` na pasta atual do processo em execução. Se não encontrado o arquivo, irá então buscar no diretório de onde o executável está localizado. Caso o arquivo não exista, o parâmetro `createIfDontExists` é honrado, criando o arquivo, sem nenhum conteúdo, no último caminho testado (com base em `lookupDirectories`), e um erro é lançado no console, impedindo a inicialização do aplicativo.

> **TIP**
>
> Você pode olhar no código fonte do leitor de configurações de arquivos INI e no de JSON para entender como um IConfigurationProvider é implementado.

Lendo configurações de um arquivo JSON

Por padrão, o Sisk fornece um provedor de configuração que lê configurações em um arquivo JSON. Este arquivo segue uma estrutura fixa e é composto pelos parâmetros:

```
{
    "Server": {
        "DefaultEncoding": "UTF-8",
        "ThrowExceptions": true,
        "IncludeRequestIdHeader": true
    },
    "ListeningHost": {
        "Label": "My sisk application",
        "Ports": [
            "http://localhost:80/",
            "https://localhost:443/",  // Configuration files also supports comments
        ],
        "CrossOriginResourceSharingPolicy": {
            "AllowOrigin": "*",
            "AllowOrigins": [ "*" ],   // new on 0.14
            "AllowMethods": [ "*" ],
            "AllowHeaders": [ "*" ],
            "MaxAge": 3600
        },
        "Parameters": {
            "MySqlConnection": "server=localhost;user=root;"
        }
    }
}
```

Os parâmetros criados a partir de um arquivo de configurações pode ser acessado no construtor do servidor:

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

Cada leitor de configurações fornece uma forma de ler os parâmetros de inicialização do servidor. Algumas propriedades são indicadas a ficarem no ambiente do processo ao invés de serem definidas no arquivo de configuração, como por exemplo, dados sensíveis de API, chaves de API, etc.

Estrutura de arquivo de configuração

O arquivo de configuração JSON é composto pelas propriedades:

Property	Mandatory	Description
Server	Required	Represents the server itself with their settings.
Server.AccessLogsStream	Optional	Default to `console`. Specifies the access log output stream. Can be an filename, `null` or `console`.
Server.ErrorsLogsStream	Optional	Default to `null`. Specifies the error log output stream. Can be an filename, `null` or `console`.
Server.MaximumContentLength	Optional	Default to `0`. Specifies the maximum content length

Property	Mandatory	Description
		in bytes. Zero means infinite.
Server.IncludeRequestIdHeader	Optional	Default to `false`. Specifies if the HTTP server should send the `X-Request-Id` header.
Server.ThrowExceptions	Optional	Default to `true`. Specifies if unhandled exceptions should be thrown. Set to `false` when production and `true` when debugging.
ListeningHost	Required	Represents the server listening host.
ListeningHost.Label	Optional	Represents the application label.
ListeningHost.Ports	Required	Represents an array of strings, matching the ListeningPort syntax.
ListeningHost.CrossOriginResourceSharingPolicy	Optional	Setup the CORS headers for the application.
ListeningHost.CrossOriginResourceSharingPolicy.AllowCredentials	Optional	Defaults to `false`. Specifies the `Allow-Credentials` header.
ListeningHost.CrossOriginResourceSharingPolicy.ExposeHeaders	Optional	Defaults to `null`. This property expects an array of strings. Specifies the `Expose-Headers` header.
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigin	Optional	Defaults to `null`. This property expects an string. Specifies the `Allow-Origin` header.
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigins	Optional	Defaults to `null`. This property expects an array of strings. Specifies multiples `Allow-Origin`

Property	Mandatory	Description
		headers. See AllowOrigins for more information.
ListeningHost.CrossOriginResourceSharingPolicy.AllowMethods	Optional	Defaults to `null`. This property expects an array of strings. Specifies the `Allow-Methods` header.
ListeningHost.CrossOriginResourceSharingPolicy.AllowHeaders	Optional	Defaults to `null`. This property expects an array of strings. Specifies the `Allow-Headers` header.
ListeningHost.CrossOriginResourceSharingPolicy.MaxAge	Optional	Defaults to `null`. This property expects an interger. Specifies the `Max-Age` header in seconds.
ListeningHost.Parameters	Optional	Specifies the properties provided to the application setup method.

Configuração INI

O Sisk tem um método para obter configurações de inicialização além do JSON. Na verdade, qualquer pipeline que implemente IConfigurationReader pode ser usado com PortableConfigurationBuilder.WithConfigurationPipeline, lendo a configuração do servidor de qualquer tipo de arquivo.

O pacote Sisk.IniConfiguration fornece um leitor de arquivos INI baseado em fluxo que não lança exceções para erros de sintaxe comuns e tem uma sintaxe de configuração simples. Esse pacote pode ser usado fora do framework Sisk, oferecendo flexibilidade para projetos que requerem um leitor de documentos INI eficiente.

Instalando

Para instalar o pacote, você pode começar com:

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

Você também pode instalar o pacote core, que não inclui o IConfigurationReader INI, nem a dependência do Sisk, apenas os serializadores INI:

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

Com o pacote principal, você pode usá-lo em seu código como mostrado no exemplo abaixo:

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

O código acima procurará por um arquivo app.ini no diretório atual do processo (CurrentDirectory). O arquivo INI tem a seguinte aparência:

```
1   [Server]
2   # Múltiplos endereços de escuta são suportados
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"
```

Sabor e sintaxe INI

Implementação atual do sabor:

- Nomes de propriedades e seções são **insensíveis a letras maiúsculas e minúsculas**.
- Nomes de propriedades e valores são **recortados**, a menos que os valores sejam citados.
- Valores podem ser citados com aspas simples ou duplas. Aspas podem ter quebras de linha dentro delas.
- Comentários são suportados com `#` e `;`. Além disso, **comentários de tralha são permitidos**.
- Propriedades podem ter múltiplos valores.

Em detalhe, a documentação para o "sabor" do analisador INI usado no Sisk está disponível neste documento.

Usando o seguinte código INI como exemplo:

```
One = 1
Value = este é um valor
Another value = "este valor
    tem uma quebra de linha nele"

; o código abaixo tem algumas cores
[some section]
Color = Red
Color = Blue
Color = Yellow ; não use amarelo
```

Analisá-lo com:

```
// analisa o texto INI da string
IniDocument doc = IniDocument.FromString(iniText);

// obtenha um valor
string? one = doc.Global.GetOne("one");
string? anotherValue = doc.Global.GetOne("another value");

// obtenha múltiplos valores
string[]? colors = doc.GetSection("some section")?.GetMany("color");
```

Parâmetros de configuração

Seção e nome	Permite múltiplos valores	Descrição
`Server.Listen`	Sim	Os endereços/ports de escuta do servidor.
`Server.Encoding`	Não	A codificação padrão do servidor.
`Server.MaximumContentLength`	Não	O tamanho máximo do conteúdo em bytes.
`Server.IncludeRequestIdHeader`	Não	Especifica se o servidor HTTP deve enviar o cabeçalho X-Request-Id.
`Server.ThrowExceptions`	Não	Especifica se as exceções não tratadas devem ser lançadas.
`Server.AccessLogsStream`	Não	Especifica o fluxo de saída do log de acesso.
`Server.ErrorsLogsStream`	Não	Especifica o fluxo de saída do log de erros.
`Cors.AllowMethods`	Não	Especifica o valor do cabeçalho Allow-Methods CORS.
`Cors.AllowHeaders`	Não	Especifica o valor do cabeçalho Allow-Headers CORS.
`Cors.AllowOrigins`	Não	Especifica múltiplos cabeçalhos Allow-Origin, separados por vírgulas. AllowOrigins para mais informações.
`Cors.AllowOrigin`	Não	Especifica um cabeçalho Allow-Origin.
`Cors.ExposeHeaders`	Não	Especifica o valor do cabeçalho Expose-Headers CORS.
`Cors.AllowCredentials`	Não	Especifica o valor do cabeçalho Allow-Credentials CORS.
`Cors.MaxAge`	Não	Especifica o valor do cabeçalho Max-Age CORS.

Configuração manual (avançado)

Nesta seção, vamos criar nosso servidor HTTP sem nenhum padrão predefinido, de uma maneira completamente abstrata. Aqui, você pode construir manualmente como seu servidor HTTP funcionará. Cada `ListeningHost` possui um roteador, e um servidor HTTP pode ter vários `ListeningHosts` , cada um apontando para um host diferente em uma porta diferente.

Primeiro, precisamos entender o conceito de solicitação/resposta. É bastante simples: para cada solicitação, deve haver uma resposta. O Sisk segue esse princípio também. Vamos criar um método que responda com uma mensagem "Olá, Mundo!" em HTML, especificando o código de status e cabeçalhos.

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

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

    return indexResponse;
}
```

O próximo passo é associar esse método a uma rota HTTP.

Roteadores

Roteadores são abstrações de rotas de solicitação e servem como a ponte entre solicitações e respostas para o serviço. Roteadores gerenciam rotas de serviço, funções e erros.

Um roteador pode ter várias rotas, e cada rota pode realizar diferentes operações naquele caminho, como executar uma função, servir uma página ou fornecer um recurso do servidor.

Vamos criar nosso primeiro roteador e associar nosso método `IndexPage` ao caminho index.

```
Router mainRouter = new Router();

// SetRoute associará todas as rotas index às nossas funções.
mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage);
```

Agora nosso roteador pode receber solicitações e enviar respostas. No entanto, `mainRouter` não está vinculado a um host ou servidor, então ele não funcionará sozinho. O próximo passo é criar nosso `ListeningHost`.

Hosts de Escuta e Portas

Um ListeningHost pode hospedar um roteador e várias portas de escuta para o mesmo roteador. Um ListeningPort é um prefixo onde o servidor HTTP irá escutar.

Aqui, podemos criar um `ListeningHost` que aponta para dois endpoints para nosso roteador:

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

Agora nosso servidor HTTP irá escutar os endpoints especificados e redirecionar suas solicitações para nosso roteador.

Configuração do Servidor

A configuração do servidor é responsável por grande parte do comportamento do servidor HTTP em si. Nesta configuração, podemos associar `ListeningHosts` ao nosso servidor.

```
1    HttpServerConfiguration config = new HttpServerConfiguration();
2    config.ListeningHosts.Add(myHost); // Adicione nosso ListeningHost a esta configuração
     do servidor
```

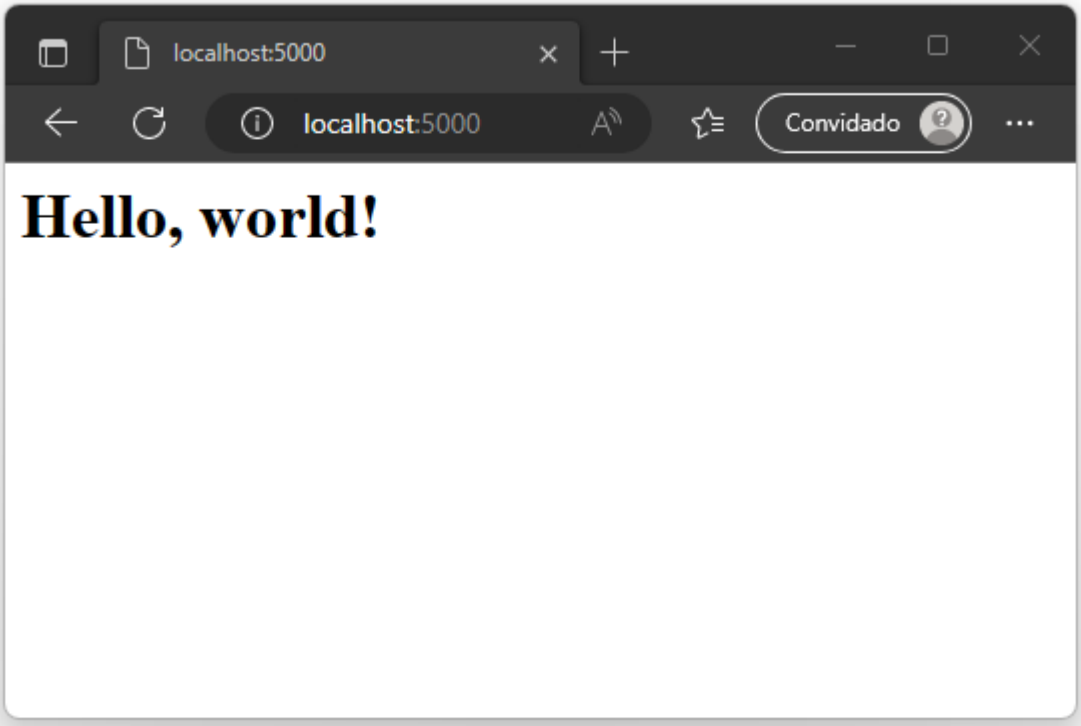
Em seguida, podemos criar nosso servidor HTTP:

```
1    HttpServer server = new HttpServer(config);
2    server.Start();    // Inicia o servidor
3    Console.ReadKey(); // Previne o encerramento da aplicação
```

Agora podemos compilar nosso executável e executar nosso servidor HTTP com o comando:

```
dotnet watch
```

Em tempo de execução, abra seu navegador e navegue até o caminho do servidor, e você deve ver:


localhost:5000
localhost:5000
Convidado
Hello, world!

Ciclo de vida da requisição

Abaixo é explicado o ciclo de vida completo de uma requisição por meio de um exemplo de requisição HTTP.

- **Recebendo a requisição:** cada requisição cria um contexto HTTP entre a requisição em si e a resposta que será entregue ao cliente. Esse contexto vem do ouvinte embutido no Sisk, que pode ser o HttpListener, Kestrel ou Cadente.
  - Validação de requisição externa: a validação de HttpServerConfiguration.RemoteRequestsAction é validada para a requisição.
    - Se a requisição for externa e a propriedade for `Drop`, a conexão é fechada sem uma resposta ao cliente com um `HttpServerExecutionStatus = RemoteRequestDropped`.
  - Configuração do resolvedor de encaminhamento: se um ForwardingResolver estiver configurado, ele chamará o método OnResolveRequestHost no host original da requisição.
  - Combinação de DNS: com o host resolvido e com mais de um ListeningHost configurado, o servidor procurará pelo host correspondente para a requisição.
    - Se nenhum ListeningHost corresponder, uma resposta 400 Bad Request será retornada ao cliente e um status `HttpServerExecutionStatus = DnsUnknownHost` será retornado ao contexto HTTP.
    - Se um ListeningHost corresponder, mas seu Router não estiver inicializado, uma resposta 503 Service Unavailable será retornada ao cliente e um status `HttpServerExecutionStatus = ListeningHostNotReady` será retornado ao contexto HTTP.
  - Associação do roteador: o roteador do ListeningHost correspondente é associado ao servidor HTTP recebido.
    - Se o roteador já estiver associado a outro servidor HTTP, o que não é permitido porque o roteador usa ativamente os recursos de configuração do servidor, uma `InvalidOperationException` será lançada. Isso ocorre apenas durante a inicialização do servidor HTTP, não durante a criação do contexto HTTP.
  - Pré-definição de cabeçalhos:
    - Pré-definir o cabeçalho `X-Request-Id` na resposta se estiver configurado para fazê-lo.
    - Pré-definir o cabeçalho `X-Powered-By` na resposta se estiver configurado para fazê-lo.
  - Validação do tamanho do conteúdo: valida se o conteúdo da requisição é menor que HttpServerConfiguration.MaximumContentLength apenas se for maior que zero.
    - Se a requisição enviar um `Content-Length` maior que o configurado, uma resposta 413 Payload Too Large será retornada ao cliente e um status `HttpServerExecutionStatus = ContentTooLarge` será retornado ao contexto HTTP.
  - O evento `OnHttpRequestOpen` é invocado para todos os manipuladores de servidor HTTP configurados.
- **Roteando a ação:** o servidor invoca o roteador para a requisição recebida.
  - Se o roteador não encontrar uma rota que corresponda à requisição:

        - Se a propriedade Router.NotFoundErrorHandler estiver configurada, a ação será invocada e a resposta da ação será encaminhada ao cliente HTTP.
        - Se a propriedade anterior for nula, uma resposta 404 Not Found padrão será retornada ao cliente.
    - Se o roteador encontrar uma rota correspondente, mas o método da rota não corresponder ao método da requisição:
        - Se a propriedade Router.MethodNotAllowedErrorHandler estiver configurada, a ação será invocada e a resposta da ação será encaminhada ao cliente HTTP.
        - Se a propriedade anterior for nula, uma resposta 405 Method Not Allowed padrão será retornada ao cliente.
    - Se a requisição for do método `OPTIONS`:
        - O roteador retornará uma resposta 200 Ok ao cliente apenas se nenhuma rota corresponder ao método da requisição (o método da rota não é explicitamente RouteMethod.Options).
    - Se a propriedade HttpServerConfiguration.ForceTrailingSlash estiver habilitada, a rota correspondente não for uma expressão regular, o caminho da requisição não terminar com `/` e o método da requisição for `GET`:
        - Uma resposta 307 Temporary Redirect HTTP com o cabeçalho `Location` com o caminho e a consulta para o mesmo local com um `/` no final será retornada ao cliente.
    - O evento `OnContextBagCreated` é invocado para todos os manipuladores de servidor HTTP configurados.
    - Todas as instâncias globais de IRequestHandler com a flag `BeforeResponse` são executadas.
        - Se algum manipulador retornar uma resposta não nula, a resposta do manipulador será encaminhada ao cliente HTTP e o contexto será fechado.
        - Se um erro for lançado nessa etapa e HttpServerConfiguration.ThrowExceptions estiver desabilitado:
            - Se a propriedade Router.CallbackErrorHandler estiver habilitada, ela será invocada e a resposta resultante será retornada ao cliente.
            - Se a propriedade anterior não estiver definida, uma resposta vazia será retornada ao servidor, que encaminhará uma resposta de acordo com o tipo de exceção lançada, que geralmente é 500 Internal Server Error.
    - Todas as instâncias de IRequestHandler definidas na rota e com a flag `BeforeResponse` são executadas.
        - Se algum manipulador retornar uma resposta não nula, a resposta do manipulador será encaminhada ao cliente HTTP e o contexto será fechado.
        - Se um erro for lançado nessa etapa e HttpServerConfiguration.ThrowExceptions estiver desabilitado:
            - Se a propriedade Router.CallbackErrorHandler estiver habilitada, ela será invocada e a resposta resultante será retornada ao cliente.
            - Se a propriedade anterior não estiver definida, uma resposta vazia será retornada ao servidor, que encaminhará uma resposta de acordo com o tipo de exceção lançada, que geralmente é 500 Internal Server Error.

    - A ação do roteador é invocada e transformada em uma resposta HTTP.
        - Se um erro for lançado nessa etapa e HttpServerConfiguration.ThrowExceptions estiver desabilitado:
            - Se a propriedade Router.CallbackErrorHandler estiver habilitada, ela será invocada e a resposta resultante será retornada ao cliente.
            - Se a propriedade anterior não estiver definida, uma resposta vazia será retornada ao servidor, que encaminhará uma resposta de acordo com o tipo de exceção lançada, que geralmente é 500 Internal Server Error.
    - Todas as instâncias globais de IRequestHandler com a flag `AfterResponse` são executadas.
        - Se algum manipulador retornar uma resposta não nula, a resposta do manipulador substituirá a resposta anterior e será imediatamente encaminhada ao cliente HTTP.
        - Se um erro for lançado nessa etapa e HttpServerConfiguration.ThrowExceptions estiver desabilitado:
            - Se a propriedade Router.CallbackErrorHandler estiver habilitada, ela será invocada e a resposta resultante será retornada ao cliente.
            - Se a propriedade anterior não estiver definida, uma resposta vazia será retornada ao servidor, que encaminhará uma resposta de acordo com o tipo de exceção lançada, que geralmente é 500 Internal Server Error.
    - Todas as instâncias de IRequestHandler definidas na rota e com a flag `AfterResponse` são executadas.
        - Se algum manipulador retornar uma resposta não nula, a resposta do manipulador substituirá a resposta anterior e será imediatamente encaminhada ao cliente HTTP.
        - Se um erro for lançado nessa etapa e HttpServerConfiguration.ThrowExceptions estiver desabilitado:
            - Se a propriedade Router.CallbackErrorHandler estiver habilitada, ela será invocada e a resposta resultante será retornada ao cliente.
            - Se a propriedade anterior não estiver definida, uma resposta vazia será retornada ao servidor, que encaminhará uma resposta de acordo com o tipo de exceção lançada, que geralmente é 500 Internal Server Error.
- **Processando a resposta:** com a resposta pronta, o servidor a prepara para envio ao cliente.
    - Os cabeçalhos da Política de Compartilhamento de Recursos de Origem Cruzada (CORS) são definidos na resposta de acordo com o que foi configurado no ListeningHost.CrossOriginResourceSharingPolicy atual.
    - O código de status e os cabeçalhos da resposta são enviados ao cliente.
    - O conteúdo da resposta é enviado ao cliente:
        - Se o conteúdo da resposta for um descendente de ByteArrayContent, os bytes da resposta são copiados diretamente para o fluxo de saída da resposta.
        - Se a condição anterior não for atendida, a resposta é serializada para um fluxo e copiada para o fluxo de saída da resposta.
    - Os fluxos são fechados e o conteúdo da resposta é descartado.

- Se HttpServerConfiguration.DisposeDisposableContextValues estiver habilitado, todos os objetos definidos no contexto da requisição que herdam de IDisposable são descartados.
- O evento `OnHttpRequestClose` é invocado para todos os manipuladores de servidor HTTP configurados.
- Se uma exceção for lançada no servidor, o evento `OnException` é invocado para todos os manipuladores de servidor HTTP configurados.
- Se a rota permitir logging de acesso e HttpServerConfiguration.AccessLogsStream não for nulo, uma linha de log é escrita no fluxo de log de saída.
- Se a rota permitir logging de erros, houver uma exceção e HttpServerConfiguration.ErrorsLogsStream não for nulo, uma linha de log é escrita no fluxo de log de erros de saída.
- Se o servidor estiver aguardando uma requisição por meio de HttpServer.WaitNext, o mutex é liberado e o contexto se torna disponível para o usuário.

Forwarding Resolvers

Um Forwarding Resolver é um auxiliar que ajuda a decodificar informações que identificam o cliente por meio de uma solicitação, proxy, CDN ou balanceadores de carga. Quando seu serviço Sisk é executado por meio de um proxy inverso ou avançado, o endereço IP, o host e o protocolo do cliente podem ser diferentes da solicitação original, pois é um encaminhamento de um serviço para outro. Essa funcionalidade do Sisk permite que você controle e resolva essas informações antes de trabalhar com a solicitação. Esses proxies geralmente fornecem cabeçalhos úteis para identificar seu cliente.

Atualmente, com a classe ForwardingResolver, é possível resolver o endereço IP do cliente, o host e o protocolo HTTP usado. Após a versão 1.0 do Sisk, o servidor não possui mais uma implementação padrão para decodificar esses cabeçalhos por motivos de segurança que variam de serviço para serviço.

Por exemplo, o cabeçalho `X-Forwarded-For` inclui informações sobre os endereços IP que encaminharam a solicitação. Este cabeçalho é usado por proxies para transportar uma cadeia de informações para o serviço final e inclui o IP de todos os proxies usados, incluindo o endereço real do cliente. O problema é: às vezes é desafiador identificar o IP remoto do cliente e não há uma regra específica para identificar este cabeçalho. É altamente recomendável ler a documentação dos cabeçalhos que você está prestes a implementar abaixo:

- Leia sobre o cabeçalho `X-Forwarded-For` aqui.
- Leia sobre o cabeçalho `X-Forwarded-Host` aqui.
- Leia sobre o cabeçalho `X-Forwarded-Proto` aqui.

A classe ForwardingResolver

Esta classe possui três métodos virtuais que permitem a implementação mais adequada para cada serviço. Cada método é responsável por resolver informações da solicitação por meio de um proxy: o endereço IP do cliente, o host da solicitação e o protocolo de segurança usado. Por padrão, o Sisk sempre usará as informações da solicitação original, sem resolver nenhum cabeçalho.

O exemplo abaixo mostra como essa implementação pode ser usada. Este exemplo resolve o endereço IP do cliente através do cabeçalho `X-Forwarded-For` e lança um erro quando mais de um IP foi encaminhado na solicitação.

> [!IMPORTANTE] Não use este exemplo em código de produção. Verifique sempre se a implementação é apropriada para uso. Leia a documentação do cabeçalho antes de implementá-lo.

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

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

        host.Start();
    }

    class Resolver : ForwardingResolver
    {
        public override IPAddress OnResolveClientAddress(HttpRequest request,
IPEndPoint connectingEndpoint)
        {
            string? forwardedFor = request.Headers.XForwardedFor;
            if (forwardedFor is null)
            {
                throw new Exception("O cabeçalho X-Forwarded-For está faltando.");
            }
            string[] ipAddresses = forwardedFor.Split(',');
            if (ipAddresses.Length != 1)
            {
                throw new Exception("Número excessivo de endereços no cabeçalho X-
Forwarded-For.");
            }

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

Handlers de servidor HTTP

Na versão 0.16 do Sisk, introduzimos a classe `HttpServerHandler`, que visa estender o comportamento geral do Sisk e fornecer manipuladores de eventos adicionais ao Sisk, como lidar com solicitações HTTP, roteadores, bolsas de contexto e muito mais.

A classe concentra eventos que ocorrem durante a vida útil do servidor HTTP inteiro e também de uma solicitação. O protocolo HTTP não possui sessões e, portanto, não é possível preservar informações de uma solicitação para outra. O Sisk, por enquanto, fornece uma maneira para você implementar sessões, contextos, conexões de banco de dados e outros provedores úteis para ajudar no seu trabalho.

Consulte esta página para saber onde cada evento é acionado e qual é o seu propósito. Você também pode visualizar o ciclo de vida de uma solicitação HTTP para entender o que acontece com uma solicitação e onde os eventos são disparados. O servidor HTTP permite que você use vários manipuladores ao mesmo tempo. Cada chamada de evento é síncrona, ou seja, bloqueará a thread atual para cada solicitação ou contexto até que todos os manipuladores associados a essa função sejam executados e concluídos.

Ao contrário dos RequestHandlers, eles não podem ser aplicados a alguns grupos de rotas ou rotas específicas. Em vez disso, eles são aplicados a todo o servidor HTTP. Você pode aplicar condições dentro do seu Http Server Handler. Além disso, singletons de cada HttpServerHandler são definidos para cada aplicativo Sisk, portanto, apenas uma instância por `HttpServerHandler` é definida.

Um exemplo prático de uso do HttpServerHandler é descartar automaticamente uma conexão de banco de dados no final da solicitação.

```
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           // verifica se a solicitação definiu um DbContext
10          // em sua bolsa de contexto
11          if (requestBag.IsSet<DbContext>())
12          {
13              var db = requestBag.Get<DbContext>();
14              db.Dispose();
15          }
```

```
    }
}

public static class DatabaseConnectionHandlerExtensions
{
    // permite que o usuário crie um dbcontext a partir de uma solicitação HTTP
    // e armazene-o em sua bolsa de contexto
    public static DbContext GetDbContext(this HttpRequest request)
    {
        var db = new DbContext();
        return request.SetContextBag<DbContext>(db);
    }
}
```

Com o código acima, a extensão `GetDbContext` permite que uma conexão de contexto seja criada diretamente do objeto HttpRequest. Uma conexão não descartada pode causar problemas ao executar com o banco de dados, portanto, é encerrada em `OnHttpRequestClose` .

Você pode registrar um manipulador em um servidor HTTP em seu construtor ou diretamente com HttpServer.RegisterHandler.

```
// Program.cs

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

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

Com isso, a classe `UsersController` pode usar o contexto do banco de dados como:

```
// UserController.cs

[RoutePrefix("/users")]
public class UserController : ApiController
{
    [RouteGet()]
    public async Task<HttpResponse> List(HttpRequest request)
```

```
    {
        var db = request.GetDbContext();
        var users = db.Users.ToArray();

        return JsonOk(users);
    }

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

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

        return JsonOk(user);
    }

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

        ArgumentNullException.ThrowIfNull(user);

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

        return JsonMessage("User added.");
    }
}
```

O código acima usa métodos como `JsonOk` e `JsonMessage` que são integrados ao `ApiController`, que é herdado de um `RouterController`:

```
// ApiController.cs

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

```
            }));
    }

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

Os desenvolvedores podem implementar sessões, contextos e conexões de banco de dados usando esta classe. O código fornecido mostra um exemplo prático com o DatabaseConnectionHandler, automatizando o descarte da conexão do banco de dados no final de cada solicitação.

A integração é simples, com manipuladores registrados durante a configuração do servidor. A classe HttpServerHandler oferece um conjunto poderoso de ferramentas para gerenciar recursos e estender o comportamento do Sisk em aplicações HTTP.

Configuração de vários hosts

O Sisk Framework sempre suportou o uso de mais de um host por servidor, ou seja, um único servidor HTTP pode escutar em múltiplos portas e cada porta possui seu próprio roteador e seu próprio serviço em execução nela.

Isso facilita a separação de responsabilidades e a gestão de serviços em um único servidor HTTP com o Sisk. O exemplo abaixo mostra a criação de dois ListeningHosts, cada um escutando em uma porta diferente, com roteadores e ações diferentes.

Leia criação manual do seu aplicativo para entender os detalhes sobre essa abstração.

```
    static void Main(string[] args)
    {
        // criar dois hosts de escuta, cada um com seu próprio roteador e
        // escutando em sua própria porta
        //
        ListeningHost hostA = new ListeningHost();
        hostA.Ports = [new ListeningPort(12000)];
        hostA.Router = new Router();
        hostA.Router.SetRoute(RouteMethod.Get, "/", request => new
    HttpResponse().WithContent("Olá do host A!"));

        ListeningHost hostB = new ListeningHost();
        hostB.Ports = [new ListeningPort(12001)];
        hostB.Router = new Router();
        hostB.Router.SetRoute(RouteMethod.Get, "/", request => new
    HttpResponse().WithContent("Olá do host B!"));

        // criar uma configuração do servidor e adicionar ambos
        // os hosts de escuta nela
        //
        HttpServerConfiguration configuration = new HttpServerConfiguration();
        configuration.ListeningHosts.Add(hostA);
        configuration.ListeningHosts.Add(hostB);

        // cria um servidor HTTP que usa a configuração especificada
        //
        HttpServer server = new HttpServer(configuration);

        // inicia o servidor
        server.Start();
```

```
31
32      Console.WriteLine("Tente acessar o host A em {0}", server.ListeningPrefixes[0]);
33      Console.WriteLine("Tente acessar o host B em {0}", server.ListeningPrefixes[1]);
34
        Thread.Sleep(-1);
    }
```