

Introducción

Bienvenido a la documentación de Sisk.

Finalmente, ¿qué es el Marco de Sisk? Es una biblioteca de código abierto y ligera construida con .NET, diseñada para ser minimalista, flexible y abstracta. Permite a los desarrolladores crear servicios de Internet rápidamente, con poca o ninguna configuración necesaria. Sisk permite que su aplicación existente tenga un módulo HTTP administrado, completo y desecharable o completo.

Los valores de Sisk incluyen la transparencia del código, la modularidad, el rendimiento y la escalabilidad, y pueden manejar varios tipos de aplicaciones, como Restful, JSON-RPC, Web-sockets y más.

Sus características principales incluyen:

Recurso	Descripción
Enrutamiento	Un enrutador de rutas que admite prefijos, métodos personalizados, variables de ruta, convertidores de valores y más.
Controladores de solicitudes	También conocidos como <i>middlewares</i> , proporcionan una interfaz para crear sus propios controladores de solicitudes que funcionan con la solicitud antes o después de una acción.
Compresión	Comprima el contenido de sus respuestas fácilmente con Sisk.
Web sockets	Proporciona rutas que aceptan web sockets completos, para leer y escribir en el cliente.
Eventos enviados por el servidor	Proporciona el envío de eventos del servidor a los clientes que admiten el protocolo SSE.
Registro	Registro simplificado. Registre errores, acceso, defina registros rotativos por tamaño, varias secuencias de salida para el mismo registro y más.
Multi-host	Tener un servidor HTTP para varios puertos, y cada puerto con su propio enrutador, y cada enrutador con su propia aplicación.
Controladores de servidor	Amplíe su propia implementación del servidor HTTP. Personalice con extensiones, mejoras y nuevas características.

Primeros pasos

Sisk puede ejecutarse en cualquier entorno .NET. En esta guía, le enseñaremos cómo crear una aplicación Sisk utilizando .NET. Si aún no lo ha instalado, descargue el SDK desde [aquí](#).

En este tutorial, cubriremos cómo crear una estructura de proyecto, recibir una solicitud, obtener un parámetro de URL y enviar una respuesta. Esta guía se centrará en la creación de un servidor simple utilizando C#. También puede utilizar su lenguaje de programación favorito.

NOTE

Es posible que esté interesado en un proyecto de inicio rápido. Consulte [este repositorio](#) para obtener más información.

Creación de un proyecto

Llame a nuestro proyecto "Mi aplicación Sisk". Una vez que tenga configurado .NET, puede crear su proyecto con el siguiente comando:

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

A continuación, navegue hasta el directorio de su proyecto e instale Sisk utilizando la herramienta de utilidad .NET:

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

Puede encontrar formas adicionales de instalar Sisk en su proyecto [aquí](#).

Ahora, creamos una instancia de nuestro servidor HTTP. En este ejemplo, lo configuraremos para escuchar en el puerto 5000.

Construcción del servidor HTTP

Sisk le permite construir su aplicación paso a paso manualmente, ya que enruta al objeto `HttpServer`. Sin embargo, esto puede no ser muy conveniente para la mayoría de los proyectos. Por lo tanto, podemos utilizar el método del constructor, que facilita la ejecución de nuestra aplicación.

Program.cs

C#

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

Es importante comprender cada componente vital de Sisk. Más adelante en este documento, aprenderá más sobre cómo funciona Sisk.

Configuración manual (avanzada)

Puede aprender cómo funciona cada mecanismo de Sisk en [esta sección](#) de la documentación, que explica el comportamiento y las relaciones entre el `HttpServer`, `Router`, `ListeningPort` y otros componentes.

Instalación

Puedes instalar Sisk a través de Nuget, dotnet cli o [otras opciones](#). Puedes configurar fácilmente tu entorno de Sisk ejecutando este comando en tu consola de desarrollador:

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

Este comando instalará la última versión de Sisk en tu proyecto.

Soporte de AOT Nativo

.NET Native AOT  permite la publicación de aplicaciones .NET nativas que son autosuficientes y no requieren que el tiempo de ejecución de .NET esté instalado en el host de destino. Además, Native AOT proporciona beneficios como:

- Aplicaciones mucho más pequeñas
- Inicialización significativamente más rápida
- Consumo de memoria más bajo

Sisk Framework, por su naturaleza explícita, permite el uso de Native AOT para casi todas sus características sin requerir rework en el código fuente para adaptarlo a Native AOT.

Características no soportadas

Sin embargo, Sisk utiliza la reflexión, aunque mínima, para algunas características. Las características mencionadas a continuación pueden estar parcialmente disponibles o completamente no disponibles durante la ejecución de código nativo:

- **Exploración automática de módulos** del enrutador: este recurso escanea los tipos incrustados en el ensamblado en ejecución y registra los tipos que son **módulos del enrutador**. Este recurso requiere tipos que pueden ser excluidos durante el recorte del ensamblado.

Todas las demás características son compatibles con AOT en Sisk. Es común encontrar uno o otro método que da una advertencia de AOT, pero el mismo, si no se menciona aquí, tiene una sobrecarga que indica el paso de un tipo, parámetro o información de tipo que ayuda al compilador de AOT a compilar el objeto.

Desplegando tu Aplicación Sisk

El proceso de desplegar una aplicación Sisk consiste en publicar tu proyecto en producción. Aunque el proceso es relativamente simple, es importante tener en cuenta detalles que pueden ser letales para la seguridad y la estabilidad de la infraestructura de despliegue.

Idealmente, debes estar listo para desplegar tu aplicación en la nube, después de realizar todas las pruebas posibles para tener tu aplicación lista.

Publicando tu aplicación

Publicar tu aplicación o servicio Sisk es generar binarios listos y optimizados para producción. En este ejemplo, compilaremos los binarios para producción para ejecutarlos en una máquina que tenga el tiempo de ejecución de .NET instalado.

Necesitarás tener el SDK de .NET instalado en tu máquina para compilar tu aplicación, y el tiempo de ejecución de .NET instalado en el servidor objetivo para ejecutar tu aplicación. Puedes aprender a instalar el tiempo de ejecución de .NET en tu servidor Linux [aquí](#), [Windows](#) y [Mac OS](#).

En la carpeta donde se encuentra tu proyecto, abre una terminal y utiliza el comando de publicación de .NET:

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

Esto generará tus binarios dentro de `bin/Release/publish/linux-x64`.

NOTE

Si tu aplicación se ejecuta utilizando el paquete `Sisk.ServiceProvider`, debes copiar tu archivo `service-config.json` en tu servidor de host junto con todos los binarios generados por `dotnet publish`. Puedes dejar el archivo preconfigurado, con variables de entorno, puertos y hosts de escucha, y configuraciones de servidor adicionales.

El siguiente paso es trasladar estos archivos al servidor donde se hospedará tu aplicación.

Después de eso, da permisos de ejecución a tu archivo binario. En este caso, consideremos que el nombre de nuestro proyecto es "my-app":

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

Después de ejecutar tu aplicación, verifica si produce algún mensaje de error. Si no produce ninguno, es porque tu aplicación se está ejecutando.

En este punto, es probable que no sea posible acceder a tu aplicación desde la red externa fuera de tu servidor, ya que no se han configurado las reglas de acceso como el Firewall. Consideraremos esto en los siguientes pasos.

Debes tener la dirección del host virtual donde tu aplicación está escuchando. Esto se establece manualmente en la aplicación y depende de cómo estás instanciando tu servicio Sisk.

Si **no** estás utilizando el paquete Sisk.ServiceProvider, debes encontrarla donde definiste tu instancia de HttpServer:

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

Asociando un ListeningHost manualmente:

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

O si estás utilizando el paquete Sisk.ServiceProvider, en tu archivo `service-config.json`:

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

A partir de esto, podemos crear un proxy inverso para escuchar a tu servicio y hacer que el tráfico esté disponible en la red abierta.

Proxyando tu aplicación

Proxyar tu servicio significa no exponer directamente tu servicio Sisk a una red externa. Esta práctica es muy común para despliegues de servidor porque:

- Permite asociar un certificado SSL en tu aplicación;
- Crea reglas de acceso antes de acceder al servicio y evita sobrecargas;
- Controla el ancho de banda y los límites de solicitudes;
- Separa los equilibradores de carga para tu aplicación;
- Evita daños de seguridad a la infraestructura fallida.

Puedes servir tu aplicación a través de un proxy inverso como [Nginx](#) o [Apache](#), o puedes utilizar un túnel http-over-dns como [Cloudflare](#).

También recuerda resolver correctamente los encabezados de reenvío de tu proxy para obtener la información del cliente, como la dirección IP y el host, a través de [resolutores de reenvío](#).

El siguiente paso después de crear tu túnel, configurar el firewall y tener tu aplicación en ejecución, es crear un servicio para tu aplicación.

NOTE

Utilizar certificados SSL directamente en el servicio Sisk en sistemas no Windows no es posible. Esto es un punto de la implementación de HttpListener, que es el módulo central para la gestión de la cola de HTTP en Sisk, y esta implementación varía de un sistema operativo a otro. Puedes utilizar SSL en tu servicio Sisk si [asocias un certificado con el host virtual con IIS](#). Para otros sistemas, se recomienda altamente utilizar un proxy inverso.

Creando un servicio

Crear un servicio hará que tu aplicación esté siempre disponible, incluso después de reiniciar tu instancia de servidor o un bloqueo no recuperable.

En este tutorial simple, utilizaremos el contenido del tutorial anterior como una demostración para mantener tu servicio siempre activo.

1. Accede a la carpeta donde se encuentran los archivos de configuración del servicio:

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

2. Crea tu archivo `my-app.service` e incluye el contenido:

my-app.service	INI
<pre>1 [Unit] 2 Description=<descripción sobre tu aplicación> 3 4 [Service] 5 # establece el usuario que lanzará el servicio 6 User=<usuario que lanzará el servicio> 7 8 # la ruta de ExecStart no es relativa a WorkingDirectory. 9 # establecela como la ruta completa al archivo ejecutable 10 WorkingDirectory=/home/htdocs 11 ExecStart=/home/htdocs/my-app 12 13 # establece el servicio para que siempre se reinicie en caso de bloqueo 14 Restart=always 15 RestartSec=3 16 17 [Install] 18 WantedBy=multi-user.target</pre>	

3. Reinicia el módulo de administración de servicios:

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

4. Inicia tu servicio recién creado desde el nombre del archivo que estableciste y verifica si se está ejecutando:

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

5. Ahora, si tu aplicación se está ejecutando ("Active: active"), habilita tu servicio para que se mantenga en ejecución después de un reinicio del sistema:

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

Ahora estás listo para presentar tu aplicación Sisk a todos.

Trabajando con SSL

Trabajar con SSL para desarrollo puede ser necesario cuando se trabaja en contextos que requieren seguridad, como la mayoría de los escenarios de desarrollo web. Sisk opera sobre HttpListener, que no admite HTTPS de forma nativa, solo HTTP. Sin embargo, existen soluciones alternativas que te permiten trabajar con SSL en Sisk. Consulta a continuación:

A través de IIS en Windows

- Disponible en: Windows
- Esfuerzo: medio

Si estás en Windows, puedes utilizar IIS para habilitar SSL en tu servidor HTTP. Para que esto funcione, es recomendable que sigas [este tutorial](#) con anticipación si deseas que tu aplicación esté escuchando en un host diferente a "localhost".

Para que esto funcione, debes instalar IIS a través de las características de Windows. IIS está disponible de forma gratuita para usuarios de Windows y Windows Server. Para configurar SSL en tu aplicación, debes tener el certificado SSL listo, incluso si es auto-firmado. A continuación, puedes ver [cómo configurar SSL en IIS 7 o superior](#).

A través de mitmproxy

- Disponible en: Linux, macOS, Windows
- Esfuerzo: fácil

mitmproxy es una herramienta de proxy de interceptación que permite a los desarrolladores y testers de seguridad inspeccionar, modificar y grabar el tráfico HTTP y HTTPS entre un cliente (como un navegador web) y un servidor. Puedes utilizar la utilidad **mitmdump** para iniciar un proxy SSL inverso entre tu cliente y tu aplicación Sisk.

1. Primero, instala [mitmproxy](#) en tu máquina.
2. Inicia tu aplicación Sisk. En este ejemplo, utilizaremos el puerto 8000 como el puerto HTTP inseguro.
3. Inicia el servidor mitmproxy para escuchar el puerto seguro en 8001:

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

Y listo! Ya puedes acceder a tu aplicación a través de `https://localhost:8001/`. Tu aplicación no necesita estar en ejecución para iniciar `mitmdump`.

Alternativamente, puedes agregar una referencia a la [herramienta de ayuda de mitmproxy](#) en tu proyecto. Esto aún requiere que mitmproxy esté instalado en tu computadora.

A través del paquete Sisk.SslProxy

- Disponible en: Linux, macOS, Windows
- Esfuerzo: fácil

El paquete Sisk.SslProxy es una forma sencilla de habilitar SSL en tu aplicación Sisk. Sin embargo, es un paquete **extremadamente experimental**. Puede ser inestable trabajar con este paquete, pero puedes ser parte del pequeño porcentaje de personas que contribuirán a hacer que este paquete sea viable y estable. Para empezar, puedes instalar el paquete Sisk.SslProxy con:

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

NOTE

Debes habilitar "Habilitar paquetes de pre-lanzamiento" en el Administrador de paquetes de Visual Studio para instalar Sisk.SslProxy.

Nuevamente, es un proyecto experimental, así que no pienses en ponerlo en producción.

En este momento, Sisk.SslProxy puede manejar la mayoría de las características de HTTP/1.1, incluyendo HTTP Continue, Chunked-Encoding, WebSockets y SSE. Lee más sobre SslProxy [aquí](#).

Configuración de reservas de namespace en Windows

Sisk funciona con la interfaz de red HttpListener, que enlaza un host virtual al sistema para escuchar solicitudes.

En Windows, esta unión es un poco restrictiva, solo permitiendo que localhost se enlace como un host válido. Al intentar escuchar a otro host, se lanza un error de acceso denegado en el servidor. Este tutorial explica cómo conceder autorización para escuchar en cualquier host que desee en el sistema.

Configuración de namespace.bat

BATCH

```
1 @echo off
2
3 :: insertar prefijo aquí, sin espacios ni comillas
4 SET PREFIX=
5
6 SET DOMAIN=%ComputerName%\%USERNAME%
7 netsh http add urlacl url=%PREFIX% user=%DOMAIN%
8
9 pause
```

Donde en `PREFIX`, es el prefijo ("Host de escucha→Puerto") que su servidor escuchará. Debe estar formateado con el esquema de URL, host, puerto y una barra al final, ejemplo:

Configuración de namespace.bat

BATCH

```
SET PREFIX=http://mi-aplicación.example.test/
```

Para que pueda ser escuchado en su aplicación a través de:

Program.cs

C#

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

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

Registros de cambios

Cada cambio realizado en Sisk se registra a través del registro de cambios. Puedes ver los registros de cambios para todas las versiones de Sisk [aquí](#).

Preguntas Frecuentes

Preguntas frecuentes sobre Sisk.

¿Es Sisk de código abierto?

Totalmente. Todo el código fuente utilizado por Sisk se publica y se actualiza con frecuencia en [GitHub](#).

¿Se aceptan contribuciones?

Siempre y cuando sean compatibles con la [filosofía de Sisk](#), todas las contribuciones son muy bienvenidas. Las contribuciones no tienen que ser solo código. Puedes contribuir con documentación, pruebas, traducciones, donaciones y publicaciones, por ejemplo.

¿Está financiado Sisk?

No. Ninguna organización o proyecto patrocina actualmente a Sisk.

¿Puedo usar Sisk en producción?

Absolutamente. El proyecto lleva en desarrollo más de tres años y ha tenido una intensa prueba en aplicaciones comerciales que han estado en producción desde entonces. Sisk se utiliza en proyectos comerciales importantes

como infraestructura principal.

Una guía sobre cómo [implementar](#) en diferentes sistemas y entornos se ha escrito y está disponible.

¿Tiene Sisk autenticación, monitoreo y servicios de base de datos?

No. Sisk no tiene ninguno de estos. Es un framework para desarrollar aplicaciones web HTTP, pero es un framework minimalista que entrega lo necesario para que tu aplicación funcione.

Puedes implementar todos los servicios que deseas utilizando cualquier biblioteca de terceros que prefieras. Sisk fue diseñado para ser agnóstico, flexible y funcionar con cualquier cosa.

¿Por qué debería usar Sisk en lugar de ?

No lo sé. Tú me lo dices.

Sisk se creó para llenar un escenario genérico para aplicaciones web HTTP en .NET. Proyectos establecidos, como ASP.NET, resuelven varios problemas, pero con diferentes sesgos. A diferencia de los frameworks más grandes, Sisk requiere que el usuario sepa lo que está haciendo y construyendo. Los conceptos básicos de desarrollo web y el protocolo HTTP son esenciales para trabajar con Sisk.

Sisk se parece más a Express de Node.js que a ASP.NET Core. Es una abstracción de alto nivel que te permite crear aplicaciones con lógica HTTP que tú quieras.

¿Qué necesito para aprender Sisk?

Necesitas los conceptos básicos de:

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

Eso es todo. Teniendo una noción de estos dos temas, puedes dedicar unas horas a desarrollar una aplicación avanzada con Sisk.

¿Puedo desarrollar aplicaciones comerciales con Sisk?

Absolutamente.

Sisk se creó bajo la licencia MIT, lo que significa que puedes usar Sisk en cualquier proyecto comercial, comercial o no comercial, sin necesidad de una licencia propietaria.

Lo que pedimos es que en algún lugar de tu aplicación, tengas un aviso de los proyectos de código abierto utilizados en tu proyecto, y que Sisk esté allí.

Enrutamiento

El [Router](#) es el primer paso en la construcción del servidor. Es responsable de contener objetos [Route](#), que son puntos de conexión que asignan URLs y sus métodos a acciones ejecutadas por el servidor. Cada acción es responsable de recibir una solicitud y entregar una respuesta al cliente.

Las rutas son pares de expresiones de ruta ("patrón de ruta") y el método HTTP que pueden escuchar. Cuando se realiza una solicitud al servidor, intentará encontrar una ruta que coincida con la solicitud recibida, luego llamará a la acción de esa ruta y entregará la respuesta resultante al cliente.

Hay varias formas de definir rutas en Sisk: pueden ser estáticas, dinámicas o auto-escaneadas, definidas por atributos o directamente en el objeto Router.

```
1 Router mainRouter = new Router();
2
3 // asigna la ruta GET / a la siguiente acción
4 mainRouter.MapGet("/", request => {
5     return new HttpResponse("Hola, mundo!");
6 });
```

Para entender qué es capaz de hacer una ruta, necesitamos entender qué es capaz de hacer una solicitud. Un [HttpRequest](#) contendrá todo lo que necesite. Sisk también incluye algunas características adicionales que aceleran el desarrollo en general.

Para cada acción recibida por el servidor, se llamará a un delegado de tipo [RouteAction](#). Este delegado contiene un parámetro que contiene un [HttpRequest](#) con toda la información necesaria sobre la solicitud recibida por el servidor. El objeto resultante de este delegado debe ser un [HttpResponse](#) o un objeto que se asigna a él a través de [tipos de respuesta implícitos](#).

Coincidencia de rutas

Cuando se recibe una solicitud por el servidor HTTP, Sisk busca una ruta que satisfaga la expresión de la ruta recibida por la solicitud. La expresión siempre se prueba entre la ruta y la ruta de la solicitud, sin considerar la cadena de consulta.

Esta prueba no tiene prioridad y es exclusiva de una sola ruta. Cuando no se encuentra una ruta que coincida con la solicitud, se devuelve la respuesta [Router.NotFoundErrorHandler](#) al cliente. Cuando se coincide con el patrón de ruta, pero el método HTTP no coincide, se envía la respuesta [Router.MethodNotAllowedErrorHandler](#) al cliente.

Sisk verifica la posibilidad de colisiones de rutas para evitar estos problemas. Cuando se definen rutas, Sisk buscará rutas posibles que puedan colisionar con la ruta que se está definiendo. Esta prueba incluye la comprobación de la ruta y el método que la ruta está configurada para aceptar.

Creación de rutas usando patrones de ruta

Puedes definir rutas usando varios métodos `SetRoute`.

```
1 // forma SetRoute
2 mainRouter.SetRoute(RouteMethod.Get, "/hey/<name>", (request) =>
3 {
4     string name = request.RouteParameters["name"].GetString();
5     return new HttpResponseMessage($"Hola, {name}");
6 });
7
8 // forma Map*
9 mainRouter.MapGet("/form", (request) =>
10 {
11     var formData = request.GetFormData();
12     return new HttpResponseMessage(); // 200 ok vacío
13 });
14
15 // métodos de ayuda Route.*
16 mainRouter += Route.Get("/image.png", (request) =>
17 {
18     var imageStream = File.OpenRead("image.png");
19
20     return new HttpResponseMessage()
21     {
22         // el StreamContent interno
23         // se desecha después de enviar
24         // la respuesta.
25         Content = new StreamContent(imageStream)
26     };
27 });
28
29 // varios parámetros
30 mainRouter.MapGet("/hey/<name>/surname/<surname>", (request) =>
31 {
```

```

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

```

La propiedad [RouteParameters](#) de [HttpResponse](#) contiene toda la información sobre las variables de ruta de la solicitud recibida.

Cada ruta recibida por el servidor se normaliza antes de que se ejecute la prueba del patrón de ruta, siguiendo estas reglas:

- Todos los segmentos vacíos se eliminan de la ruta, por ejemplo: `////foo//bar` se convierte en `/foo/bar`.
- La coincidencia de ruta es **sensible a mayúsculas y minúsculas**, a menos que [Router.MatchRoutesIgnoreCase](#) esté establecido en `true`.

Las propiedades [Query](#) y [RouteParameters](#) de [HttpRequest](#) devuelven un objeto [StringValueCollection](#), donde cada propiedad indexada devuelve un [StringValue](#) no nulo, que se puede usar como una opción/monada para convertir su valor raw en un objeto administrado.

El ejemplo siguiente lee el parámetro de ruta "id" y obtiene un [Guid](#) de él. Si el parámetro no es un Guid válido, se lanza una excepción y se devuelve un error 500 al cliente si el servidor no está manejando [Router.CallbackErrorHandler](#).

```

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

```

[!NOTA] Las rutas ignoran el `/` final en ambas rutas de solicitud y ruta, es decir, si intentas acceder a una ruta definida como `/index/page` podrás acceder usando `/index/page/` también.

También puedes forzar a las URLs a terminar con `/` habilitando la bandera [ForceTrailingSlash](#).

Creación de rutas usando instancias de clase

También puedes definir rutas dinámicamente usando reflexión con el atributo [RouteAttribute](#). De esta manera, la instancia de una clase en la que sus métodos implementan este atributo tendrá sus rutas definidas en el router de destino.

Para que un método se defina como una ruta, debe estar marcado con un [RouteAttribute](#), como el atributo en sí o un [RouteGetAttribute](#). El método puede ser estático, de instancia, público o privado. Cuando se usa el método

`SetObject(type)` o `SetObject<TType>()`, se ignoran los métodos de instancia.

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      // coincide con GET /
4      [RouteGet]
5      HttpResponseMessage Index(HttpRequest request)
6      {
7          HttpResponseMessage res = new HttpResponseMessage();
8          res.Content = new StringContent("Index!");
9          return res;
10     }
11
12     // los métodos estáticos también funcionan
13     [RouteGet("/hello")]
14     static HttpResponseMessage Hello(HttpRequest request)
15     {
16         HttpResponseMessage res = new HttpResponseMessage();
17         res.Content = new StringContent("Hola mundo!");
18         return res;
19     }
20 }
```

La línea siguiente definirá tanto el método `Index` como el método `Hello` de `MyController` como rutas, ya que ambos están marcados como rutas y se ha proporcionado una instancia de la clase, no su tipo. Si se hubiera proporcionado su tipo en lugar de una instancia, solo se habrían definido los métodos estáticos.

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

Desde la versión 0.16 de Sisk, es posible habilitar `AutoScan`, que buscará clases definidas por el usuario que implementen `RouterModule` y las asociará automáticamente con el router. Esto no es compatible con la compilación AOT.

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

La instrucción anterior buscará todos los tipos que implementan `ApiController`, pero no el tipo en sí. Los dos parámetros opcionales indican cómo se buscarán estos tipos. El primer argumento implica el ensamblado donde se buscarán los tipos y el segundo indica la forma en que se definirán los tipos.

Rutas de regex

En lugar de usar los métodos de coincidencia de ruta HTTP predeterminados, puedes marcar una ruta para que se interprete con Regex.

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

O con la clase [RegexRoute](#):

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

También puedes capturar grupos de la expresión regular en el contenido de [HttpRequest.RouteParameters](#):

Controller/MyController.cs

C#

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

Prefijo de rutas

Puedes prefijar todas las rutas en una clase o módulo con el atributo [RoutePrefix](#) y establecer el prefijo como una cadena.

Vea el ejemplo siguiente usando la arquitectura BREAD (Buscar, Leer, Editar, Agregar y Eliminar):

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

En el ejemplo anterior, el parámetro `HttpResponse` se omite a favor de ser utilizado a través del contexto global `HttpContext.Current`. Lea más en la sección que sigue.

Rutas sin parámetro de solicitud

Las rutas se pueden definir sin el parámetro `HttpRequest` y aún es posible obtener la solicitud y sus componentes en el contexto de la solicitud. Consideremos una abstracción `ControllerBase` que sirve como base para todos los controladores de una API y que proporciona la propiedad `Request` para obtener la `HttpRequest` actualmente.

Controller/ControllerBase.cs

C#

```
1  public abstract class ControllerBase
2  {
3      // obtiene la solicitud del subproceso actual
4      public HttpRequest Request { get => HttpContext.Current.Request; }
5
6      // la linea siguiente, cuando se llama, obtiene la base de datos del subproceso
7      // HTTP actual,
8      // o crea una nueva si no existe
9      public DbContext Database { get => HttpContext.Current.RequestBag.GetOrAdd<DbContext>
() ; }
}
```

Y para que todos sus descendientes puedan usar la sintaxis de ruta sin el parámetro de solicitud:

Controller/UsersController.cs

C#

```
1  [RoutePrefix("/api/users")]
2  public class UsersController : ControllerBase
3  {
4      [RoutePost]
5      public async Task<HttpResponse> Create()
6      {
7          // lee los datos JSON de la solicitud actual
8          UserCreationDto? user = JsonSerializer.DeserializeAsync<UserCreationDto>
(Request.Body);
9          ...
10         Database.Users.Add(user);
11
12         return new HttpResponse(201);
13     }
14 }
```

Más detalles sobre el contexto actual y la inyección de dependencias se pueden encontrar en el tutorial de [inyección de dependencias](#).

Rutas de cualquier método

Puedes definir una ruta para que coincida solo con su ruta y omitir el método HTTP. Esto puede ser útil para que valides el método dentro de la acción de la ruta.

```
1 // coincide con / en cualquier método HTTP
2 mainRouter.SetRoute(RouteMethod.Any, "/", callbackFunction);
```

Rutas de cualquier ruta

Las rutas de cualquier ruta prueban cualquier ruta recibida por el servidor HTTP, sujeta al método de ruta que se está probando. Si el método de ruta es RouteMethod.Any y la ruta usa [Route.AnyPath](#) en su expresión de ruta, esta ruta escuchará todas las solicitudes del servidor HTTP, y no se pueden definir otras rutas.

```
1 // la siguiente ruta coincide con todas las solicitudes POST
2 mainRouter.SetRoute(RouteMethod.Post, Route.AnyPath, callbackFunction);
```

Coincidencia de ruta sin distinguir mayúsculas y minúsculas

Por defecto, la interpretación de rutas con solicitudes es sensible a mayúsculas y minúsculas. Para hacer que ignore mayúsculas y minúsculas, habilita esta opción:

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

Esto también habilitará la opción `RegexOptions.IgnoreCase` para rutas que usan coincidencia de regex.

Controlador de errores de no encontrado (404)

Puedes crear un controlador de errores personalizado para cuando una solicitud no coincide con ninguna ruta conocida.

```
1 mainRouter.NotFoundErrorHandler = () =>
2 {
3     return new HttpResponseMessage(404)
4     {
5         // Desde la versión v0.14
6         Content = new HtmlContent("<h1>No encontrado</h1>")
7         // versiones anteriores
8         Content = new StringContent("<h1>No encontrado</h1>", Encoding.UTF8, "text/html")
9     };
10};
```

Controlador de errores de método no permitido (405)

También puedes crear un controlador de errores personalizado para cuando una solicitud coincide con su ruta, pero no coincide con el método.

```
1 mainRouter.MethodNotAllowedErrorHandler = (context) =>
2 {
3     return new HttpResponseMessage(405)
4     {
5         Content = new StringContent($"Método no permitido para esta ruta.")
6     };
7};
```

Controlador de errores internos

Las acciones de ruta pueden lanzar errores durante la ejecución del servidor. Si no se manejan correctamente, el funcionamiento general del servidor HTTP puede interrumpirse. El router tiene un controlador de errores para cuando una acción de ruta falla y evita la interrupción del servicio.

Este método solo es accesible cuando [ThrowExceptions](#) está establecido en `false`.

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

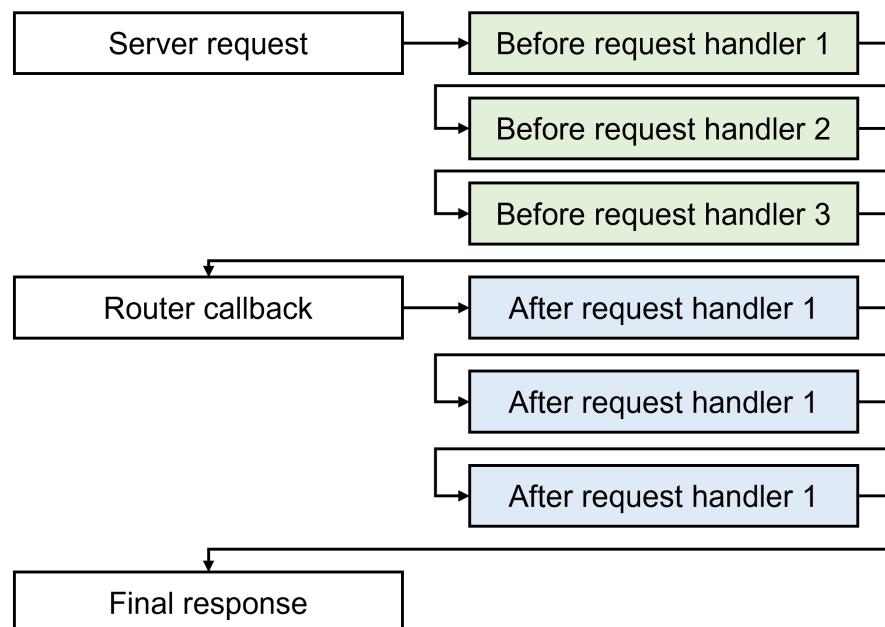
Manejo de solicitudes

Los controladores de solicitudes, también conocidos como "middlewares", son funciones que se ejecutan antes o después de que se ejecute una solicitud en el enrutador. Pueden definirse por ruta o por enrutador.

Existen dos tipos de controladores de solicitudes:

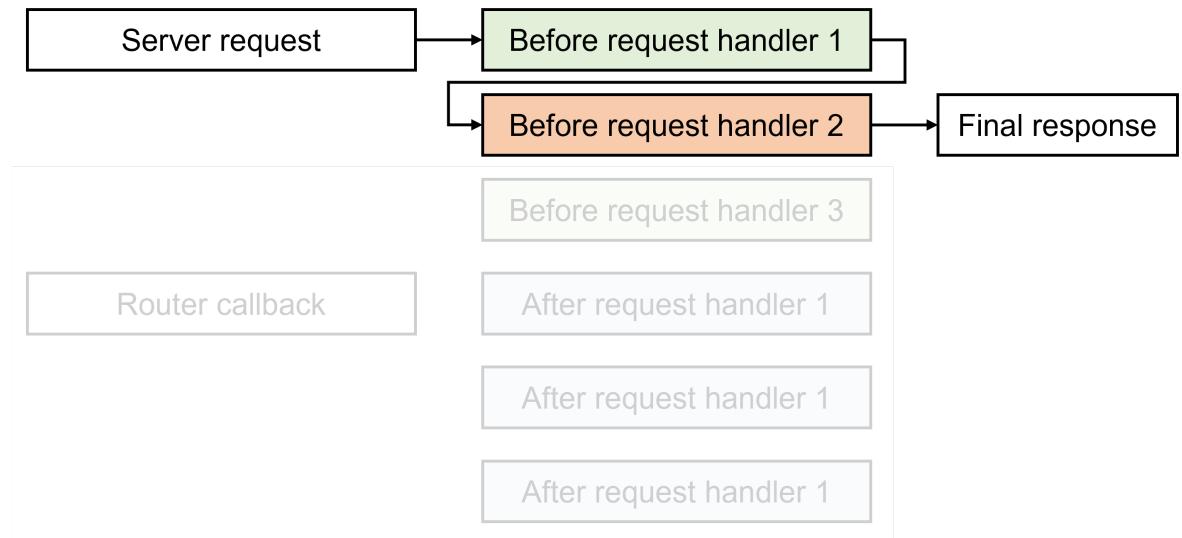
- **BeforeResponse**: define que el controlador de solicitudes se ejecutará antes de llamar a la acción del enrutador.
- **AfterResponse**: define que el controlador de solicitudes se ejecutará después de llamar a la acción del enrutador. Enviar una respuesta HTTP en este contexto sobrescribirá la respuesta de la acción del enrutador.

Ambos controladores de solicitudes pueden anular la respuesta real de la función de devolución de llamada del enrutador. Además, los controladores de solicitudes pueden ser útiles para validar una solicitud, como la autenticación, el contenido o cualquier otra información, como almacenar información, registros o otros pasos que se pueden realizar antes o después de una respuesta.



De esta manera, un controlador de solicitudes puede interrumpir toda esta ejecución y devolver una respuesta antes de terminar el ciclo, descartando todo lo demás en el proceso.

Ejemplo: supongamos que un controlador de solicitudes de autenticación de usuario no autentica al usuario. Evitará que el ciclo de solicitud continúe y se suspenderá. Si esto sucede en el controlador de solicitudes en la posición dos, el tercero y siguientes no se evaluarán.



Creación de un controlador de solicitudes

Para crear un controlador de solicitudes, podemos crear una clase que herede de la interfaz `IRequestHandler`, en este 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(HttpContext context)
7      {
8          if (context.Request.Headers.Authorization != null)
9          {
10              // Devolver null indica que el ciclo de solicitud puede continuar
11          }
12      }
13 }
```

```

11         return null;
12     }
13     else
14     {
15         // Devolver un objeto HttpResponse indica que esta respuesta sobrescribirá las
16     respuestas adyacentes.
17         return new HttpResponseMessage(System.Net.HttpStatusCode.Unauthorized);
18     }
}

```

En el ejemplo anterior, indicamos que si el encabezado `Authorization` está presente en la solicitud, debe continuar y la siguiente solicitud de controlador o la devolución de llamada del enrutador debe ser llamada, lo que sea que venga a continuación. Si es un controlador de solicitudes que se ejecuta después de la respuesta por su propiedad `ExecutionMode` y devuelve un valor no nulo, sobrescribirá la respuesta del enrutador.

Siempre que un controlador de solicitudes devuelva `null`, indica que la solicitud debe continuar y el siguiente objeto debe ser llamado o el ciclo debe terminar con la respuesta del enrutador.

Asociación de un controlador de solicitudes con una sola ruta

Puedes definir uno o más controladores de solicitudes para una ruta.

Router.cs

C#

```

1 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage, "", new IRequestHandler[]
2 {
3     new AuthenticateUserRequestHandler(),           // antes de la solicitud
4     new ValidateJsonContentRequestHandler(),       // antes de la solicitud
5     //                                         -- el método IndexPage se ejecutará aquí
6     new WriteToLogRequestHandler()                 // después de la solicitud
7 });

```

O creando un objeto `Route`:

Router.cs

C#

```

1 Route indexRoute = new Route(RouteMethod.Get, "/", "", IndexPage, null);
2 indexRoute.RequestHandlers = new IRequestHandler[]

```

```
3  {
4      new AuthenticateUserRequestHandler()
5  };
6 mainRouter.SetRoute(indexRoute);
```

Asociación de un controlador de solicitudes con un enrutador

Puedes definir un controlador de solicitudes global que se ejecutará en todas las rutas del enrutador.

Router.cs

C#

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

Asociación de un controlador de solicitudes con un atributo

Puedes definir un controlador de solicitudes en un atributo junto con un atributo de ruta.

Controller/MyController.cs

C#

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

Ten en cuenta que es necesario pasar el tipo de controlador de solicitudes deseado y no una instancia del objeto. De esta manera, el controlador de solicitudes será instanciado por el analizador del enrutador. Puedes pasar argumentos en el constructor de la clase con la propiedad [ConstructorArguments](#).

Ejemplo:

Controller/MyController.cs

C#

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

También puedes crear tu propio atributo que implemente RequestHandler:

Middleware/Attributes/AuthenticateAttribute.cs

C#

```
1 public class AuthenticateAttribute : RequestHandlerAttribute  
2 {  
3     public AuthenticateAttribute() : base(typeof(AuthenticateUserRequestHandler),  
4 ConstructorArguments = new object?[] { "arg1", 123, ... })  
5     {  
6         ;  
7     }  
8 }
```

Y usarlo como:

Controller/MyController.cs

C#

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

Omisión de un controlador de solicitudes global

Después de definir un controlador de solicitudes global en una ruta, puedes ignorar este controlador de solicitudes en rutas específicas.

Router.cs

C#

```
1  var myRequestHandler = new AuthenticateUserRequestHandler();
2  mainRouter.GlobalRequestHandlers = new IRequestHandler[]
3  {
4      myRequestHandler
5  };
6
7  mainRouter.SetRoute(new Route(RouteMethod.Get, "/", "Mi ruta", IndexPage, null)
8  {
9      BypassGlobalRequestHandlers = new IRequestHandler[]
10     {
11         myRequestHandler, // ok: la misma instancia de lo que está en
12         los controladores de solicitudes globales
13         new AuthenticateUserRequestHandler() // incorrecto: no saltará el controlador de
14         solicitudes global
15     }
16 });
17 );
```

NOTE

Si estás omitiendo un controlador de solicitudes, debes usar la misma referencia de lo que instanciaste antes para saltar. Crear otra instancia de controlador de solicitudes no saltará el controlador de solicitudes global, ya que su referencia cambiará. Recuerda usar la misma referencia de controlador de solicitudes utilizada en ambos GlobalRequestHandlers y BypassGlobalRequestHandlers.

Requests

Las solicitudes son estructuras que representan un mensaje de solicitud HTTP. El objeto [HttpRequest](#) contiene funciones útiles para manejar mensajes HTTP en toda tu aplicación.

Una solicitud HTTP se forma por el método, ruta, versión, encabezados y cuerpo.

En este documento, te enseñaremos cómo obtener cada uno de estos elementos.

Obtener el método de la solicitud

Para obtener el método de la solicitud recibida, puedes usar la propiedad `Method`:

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

Esta propiedad devuelve el método de la solicitud representado por un objeto [HttpMethod](#).

NOTE

A diferencia de los métodos de ruta, esta propiedad no sirve el elemento [RouteMethod.Any](#). En su lugar, devuelve el método real de la solicitud.

Obtener componentes de la URL de la solicitud

Puedes obtener varios componentes de una URL mediante ciertas propiedades de una solicitud. Para este ejemplo, consideremos la URL:

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

Nombre del componente	Descripción	Valor del componente
Path	Obtiene la ruta de la solicitud.	/user/login
FullPath	Obtiene la ruta de la solicitud y la cadena de consulta.	/user/login?email=foo@bar.com
FullUrl	Obtiene la cadena completa de la URL de la solicitud.	http://localhost:5000/user/login?email=foo@bar.com
Host	Obtiene el host de la solicitud.	localhost
Authority	Obtiene el host y el puerto de la solicitud.	localhost:5000
QueryString	Obtiene la consulta de la solicitud.	?email=foo@bar.com
Query	Obtiene la consulta de la solicitud en una colección de valores nombrados.	{StringValueCollection object}
IsSecure	Determina si la solicitud está usando SSL (true) o no (false).	false

También puedes optar por usar la propiedad [HttpRequest.Uri](#), que incluye todo lo anterior en un solo objeto.

Obtener el cuerpo de la solicitud

Algunas solicitudes incluyen cuerpo, como formularios, archivos o transacciones API. Puedes obtener el cuerpo de una solicitud desde la propiedad:

```
1 // obtiene el cuerpo de la solicitud como una cadena, usando la codificación de la
2 solicitud como el codificador
3 string body = request.Body;
4
5 // o lo obtiene en un array de bytes
6 byte[ ] bodyBytes = request.RawBody;
7
8
```

```
// o, de lo contrario, puedes transmitirlo.  
Stream requestStream = request.GetRequestStream();
```

También es posible determinar si hay un cuerpo en la solicitud y si está cargado con las propiedades [HasContents](#), que determina si la solicitud tiene contenidos y [IsContentAvailable](#) que indica que el servidor HTTP recibió completamente el contenido del punto remoto.

No es posible leer el contenido de la solicitud a través de `GetRequestStream` más de una vez. Si lees con este método, los valores en `RawBody` y `Body` tampoco estarán disponibles. No es necesario disponer del flujo de solicitud en el contexto de la solicitud, ya que se dispone al final de la sesión HTTP en la que se creó. Además, puedes usar la propiedad [HttpRequest.RequestEncoding](#) para obtener la mejor codificación para decodificar la solicitud manualmente.

El servidor tiene límites para leer el contenido de la solicitud, que se aplica tanto a [HttpRequest.Body](#) como a [HttpRequest.RawBody](#). Estas propiedades copian todo el flujo de entrada a un búfer local del mismo tamaño que [HttpRequest.ContentLength](#).

Se devuelve una respuesta con estado 413 Content Too Large al cliente si el contenido enviado es mayor que [HttpServerConfiguration.MaximumContentSize](#) definido en la configuración del usuario. Además, si no hay límite configurado o si es demasiado grande, el servidor lanzará una [OutOfMemoryException](#) cuando el contenido enviado por el cliente exceda [Int32.MaxValue](#) (2 GB) y si el contenido se intenta acceder a través de una de las propiedades mencionadas anteriormente. Puedes seguir manejando el contenido mediante transmisión.

NOTE

Aunque Sisk lo permite, siempre es una buena idea seguir la Semántica HTTP para crear tu aplicación y no obtener ni servir contenido en métodos que no lo permiten. Lee sobre [RFC 9110 "HTTP Semantics"](#).

Obtener el contexto de la solicitud

El Contexto HTTP es un objeto exclusivo de Sisk que almacena información del servidor HTTP, ruta, enrutador y manejador de solicitud. Puedes usarlo para poder organizarte en un entorno donde estos objetos son difíciles de organizar.

El objeto [RequestBag](#) contiene información almacenada que se pasa de un manejador de solicitud a otro punto, y puede consumirse en el destino final. Este objeto también puede ser usado por manejadores de solicitud que se ejecutan después del callback de ruta.

TIP

Esta propiedad también es accesible por la propiedad `HttpRequest.Bag`.

Middleware/AuthenticateUserRequestHandler.cs

C#

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

El manejador de solicitud anterior definirá `AuthenticatedUser` en el request bag, y puede consumirse más tarde en el callback final:

Controller/MyController.cs

C#

```
1  public class MyController
2  {
3      [RouteGet("/")]
4      [RequestHandler<AuthenticateUserRequestHandler>]
5      static HttpResponse Index(HttpContext request)
6      {
7          User authUser = request.Context.RequestBag["AuthenticatedUser"];
8
9          return new HttpResponseMessage() {
10              Content = new StringContent($"Hello, {authUser.Name}!")
11          };
12      }
13  }
```

También puedes usar los métodos auxiliares `Bag.Set()` y `Bag.Get()` para obtener o establecer objetos por sus tipos singleton.

Middleware/Authenticate.cs

C#

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

Controller/MyController.cs

C#

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

Obtener datos de formulario

Puedes obtener los valores de datos de formulario en una [NameValueCollection](#) con el ejemplo siguiente:

Controller/Auth.cs

C#

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

```
11         ...
12     }
13 }
```

Obtener datos de formulario multipart

La solicitud HTTP de Sisk te permite obtener contenidos multipart cargados, como archivos, campos de formulario o cualquier contenido binario.

Controller/Auth.cs

C#

```
1 [RoutePost("/upload-contents")]
2 public HttpResponse Index(HttpRequest request)
3 {
4     // el siguiente método lee toda la entrada de la solicitud en
5     // un array de MultipartObjects
6     var multipartFormDataObjects = request.GetMultipartFormContent();
7
8     foreach (MultipartObject uploadedObject in multipartFormDataObjects)
9     {
10         // El nombre del archivo proporcionado por los datos de formulario multipart.
11         // Se devuelve null si el objeto no es un archivo.
12         Console.WriteLine("File name      : " + uploadedObject.Filename);
13
14         // El nombre del campo del objeto de datos de formulario multipart.
15         Console.WriteLine("Field name      : " + uploadedObject.Name);
16
17         // La longitud de contenido del formulario multipart.
18         Console.WriteLine("Content length : " + uploadedObject.ContentLength);
19
20         // Determina el formato de imagen basado en el encabezado del archivo para cada
21         // tipo de contenido conocido. Si el contenido no es un formato de archivo
22         común reconocido,
23         // este método devolverá MultipartObjectCommonFormat.Unknown
24         Console.WriteLine("Common format   : " + uploadedObject.GetCommonFileFormat());
25     }
26 }
```

Puedes leer más sobre los [objetos de formulario multipart](#) de Sisk y sus métodos, propiedades y funcionalidades.

Detectar desconexión del cliente

Desde la versión v1.15 de Sisk, el framework proporciona un `CancellationToken` que se lanza cuando la conexión entre el cliente y el servidor se cierra prematuramente antes de recibir la respuesta. Este token puede ser útil para detectar cuando el cliente ya no desea la respuesta y cancelar operaciones de larga duración.

```
1  router.MapGet("/connect", async (HttpRequest req) =>
2  {
3      // obtiene el token de desconexión de la solicitud
4      var dc = req.DisconnectToken;
5
6      await LongOperationAsync(dc);
7
8      return new HttpResponseMessage();
9  });

```

Este token no es compatible con todos los motores HTTP, y cada uno requiere una implementación.

Soporte de eventos enviados por el servidor

Sisk soporta [Server-sent events](#), que permite enviar fragmentos como un flujo y mantener la conexión entre el servidor y el cliente viva.

Llamar al método `HttpRequest.GetEventSource` pondrá la `HttpRequest` en su estado de escucha. Desde esto, el contexto de esta solicitud HTTP no esperará una `HttpResponse` ya que superpondrá los paquetes enviados por eventos del lado del servidor.

Después de enviar todos los paquetes, el callback debe devolver el método `Close`, que enviará la respuesta final al servidor e indicará que el streaming ha terminado.

No es posible predecir cuál será la longitud total de todos los paquetes que se enviarán, por lo que no es posible determinar el final de la conexión con la cabecera `Content-Length`.

Por defecto en la mayoría de los navegadores, los eventos del lado del servidor no soportan enviar encabezados HTTP o métodos distintos al método GET. Por lo tanto, ten cuidado al usar manejadores de solicitud con solicitudes event-source que requieran encabezados específicos en la solicitud, ya que probablemente no los tendrán.

Además, la mayoría de los navegadores reinician los flujos si el método `EventSource.close` no se llama en el lado del cliente después de recibir todos los paquetes, causando procesamiento adicional infinito en el lado del servidor. Para evitar este tipo de problema, es común enviar un paquete final indicando que la fuente de eventos ha terminado de enviar todos los paquetes.

El ejemplo siguiente muestra cómo el navegador puede comunicar al servidor que soporta eventos del lado del servidor.

sse-example.html HTML

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

Y enviar progresivamente los mensajes al cliente:

Controller/MyController.cs C#

```
1 public class MyController
2 {
3     [RouteGet("/event-source")]
4     public async Task<HttpResponse> ServerEventsResponse(HttpContext request)
5     {
6         var sse = await request.GetEventSourceAsync();
7
8         string[] fruits = new[] { "Apple", "Banana", "Watermelon", "Tomato" };
9     }
}
```

```
10     foreach (string fruit in fruits)
11     {
12         await serverEvents.SendAsync(fruit);
13         await Task.Delay(1500);
14     }
15
16     return serverEvents.Close();
17 }
18 }
```

Al ejecutar este código, esperamos un resultado similar a este:



Fruits:

Resolver IPs y hosts proxied

Sisk puede usarse con proxies, y por lo tanto las direcciones IP pueden ser reemplazadas por el punto final del proxy en la transacción de un cliente al proxy.

Puedes definir tus propios resolutores en Sisk con [forwarding resolvers](#).

Codificación de encabezados

La codificación de encabezados puede ser un problema para algunas implementaciones. En Windows, los encabezados UTF-8 no son compatibles, por lo que se usa ASCII. Sisk tiene un convertidor de codificación

incorporado, que puede ser útil para decodificar encabezados codificados incorrectamente.

Esta operación es costosa y está deshabilitada por defecto, pero puede habilitarse bajo la bandera [NormalizeHeadersEncodings](#).

Respuestas

Las respuestas representan objetos que son respuestas HTTP a solicitudes HTTP. Son enviadas por el servidor al cliente como una indicación de la solicitud de un recurso, página, documento, archivo u otro objeto.

Una respuesta HTTP se compone de estado, encabezados y contenido.

En este documento, te enseñaremos a arquitecturar respuestas HTTP con Sisk.

Establecer un estado HTTP

La lista de estados HTTP es la misma desde HTTP/1.0, y Sisk los admite todos.

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

O con sintaxis fluida:

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

Puedes ver la lista completa de HttpStatusCode disponibles [aquí](#). También puedes proporcionar tu propio código de estado utilizando la estructura [HttpStatusInformation](#).

Cuerpo y tipo de contenido

Sisk admite objetos de contenido .NET nativos para enviar el cuerpo en las respuestas. Puedes utilizar la clase [StringContent](#) para enviar una respuesta JSON, por ejemplo:

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

El servidor siempre intentará calcular el Content-Length a partir de lo que has definido en el contenido si no lo has definido explícitamente en un encabezado. Si el servidor no puede obtener implícitamente el encabezado Content-Length del contenido de la respuesta, la respuesta se enviará con Chunked-Encoding.

También puedes transmitir la respuesta enviando un [StreamContent](#) o utilizando el método [GetResponseStream](#).

Encabezados de respuesta

Puedes agregar, editar o eliminar encabezados que estás enviando en la respuesta. El ejemplo siguiente muestra cómo enviar una respuesta de redirección al cliente.

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

O con sintaxis fluida:

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

Cuando utilices el método [Add](#) de [HttpHeaderCollection](#), estás agregando un encabezado a la solicitud sin alterar los que ya se han enviado. El método [Set](#) reemplaza los encabezados con el mismo nombre con el valor indicado. El indexador de [HttpHeaderCollection](#) llama internamente al método [Set](#) para reemplazar los encabezados.

Enviar cookies

Sisk tiene métodos que facilitan la definición de cookies en el cliente. Las cookies establecidas por este método ya están codificadas en URL y se ajustan al estándar RFC-6265.

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

O con sintaxis fluida:

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

Hay otras [versiones más completas](#) del mismo método.

Respuestas fragmentadas

Puedes establecer la codificación de transferencia en fragmentada para enviar respuestas grandes.

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

Cuando utilices codificación fragmentada, el encabezado Content-Length se omite automáticamente.

Flujo de respuesta

Los flujos de respuesta son una forma administrada que te permiten enviar respuestas de manera segmentada. Es una operación de nivel inferior que utilizar objetos `HttpResponse`, ya que requieren que envíes los encabezados y el contenido manualmente, y luego cierres la conexión.

Este ejemplo abre un flujo de lectura solo para el archivo, copia el flujo en el flujo de salida de la respuesta y no carga todo el archivo en la memoria. Esto puede ser útil para servir archivos medianos o grandes.

```

1 // obtiene el flujo de salida de la respuesta
2 using var fileStream = File.OpenRead("my-big-file.zip");
3 var responseStream = request.GetResponseStream();
4
5 // establece la codificación de la respuesta para utilizar codificación fragmentada
6 // también no deberías enviar el encabezado content-length cuando utilices
7 // codificación fragmentada
8 responseStream.SendChunked = true;
9 responseStream.StatusCode = 200;
10 responseStream.setHeader(HttpKnownHeaderNames.ContentType, contentType);
11
12 // copia el flujo del archivo en el flujo de salida de la respuesta
13 fileStream.CopyTo(responseStream.ResponseStream);
14
15 // cierra el flujo
16 return responseStream.Close();

```

Compresión GZip, Deflate y Brotli

Puedes enviar respuestas con contenido comprimido en Sisk con contenidos HTTP comprimidos. Primero, encapsula tu objeto [HttpContent](#) dentro de uno de los compresores siguientes para enviar la respuesta comprimida al cliente.

```

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

```

También puedes utilizar estos contenidos comprimidos con flujos.

```

1 router.MapGet("/archive.zip", request => {
2
3     // no apliques "using" aquí. el HttpServer descartará tu contenido
4     // después de enviar la respuesta.

```

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

Los encabezados Content-Encoding se establecen automáticamente cuando se utilizan estos contenidos.

Compresión automática

Es posible comprimir automáticamente las respuestas HTTP con la propiedad [EnableAutomaticResponseCompression](#). Esta propiedad encapsula automáticamente el contenido de la respuesta del enrutador en un contenido compressible que es aceptado por la solicitud, siempre y cuando la respuesta no sea heredada de un [CompressedContent](#).

Solo se elige un contenido compressible para una solicitud, elegido según el encabezado Accept-Encoding, que sigue el orden:

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

Si la solicitud especifica que acepta cualquiera de estos métodos de compresión, la respuesta se comprimirá automáticamente.

Tipos de respuesta implícitos

Puedes utilizar otros tipos de retorno además de HttpResponseMessage, pero es necesario configurar el enrutador para que sepa cómo manejará cada tipo de objeto.

El concepto es siempre devolver un tipo de referencia y convertirlo en un objeto HttpResponseMessage válido. Las rutas que devuelven HttpResponseMessage no se someten a conversión.

Los tipos de valor (estructuras) no se pueden utilizar como tipo de retorno porque no son compatibles con el [RouterCallback](#), por lo que deben estar envueltos en un [ValueResult](#) para poder ser utilizados en controladores.

Considera el siguiente ejemplo de un módulo de enrutador que no utiliza [HttpResponse](#) en el tipo de retorno:

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

Con esto, ahora es necesario definir en el enrutador cómo manejará cada tipo de objeto. Los objetos siempre son el primer argumento del controlador y el tipo de salida debe ser un [HttpResponse](#) válido. Además, los objetos de salida de una ruta nunca deben ser nulos.

Para tipos [ValueResult](#) no es necesario indicar que el objeto de entrada es un [ValueResult](#) y solo T, ya que [ValueResult](#) es un objeto reflejado de su componente original.

La asociación de tipos no compara lo que se registró con el tipo del objeto devuelto del controlador de enrutador. En su lugar, verifica si el tipo del resultado del enrutador es assignable al tipo registrado.

Registrar un controlador de tipo Object actuará como una reserva para todos los tipos no validados previamente. El orden de inserción de los controladores de valor también importa, por lo que registrar un controlador de Object ignorará todos los controladores específicos de tipo. Siempre registre controladores de valor específicos primero para asegurarse del orden.

```
1 Router r = new Router();
2 r.SetObject(new UsersController());
3
4 r.RegisterValueHandler<ApiResult>(apiResult =>
5 {
6     return new HttpResponseMessage() {
7         Status = apiResult.Success ? HttpStatusCode.OK : HttpStatusCode.BadRequest,
8         Content = apiResult.GetHttpContent(),
9         Headers = apiResult.GetHeaders()
10    };
11 });
12 r.RegisterValueHandler<bool>(bvalue =>
13 {
14     return new HttpResponseMessage() {
15         Status = bvalue ? HttpStatusCode.OK : HttpStatusCode.BadRequest
16     };
17 });
18 r.RegisterValueHandler<IEnumerable<object>>(enumerableValue =>
19 {
20     return new HttpResponseMessage(string.Join("\n", enumerableValue));
21 });
22
23 // registrar un controlador de valor de objeto debe ser el último
24 // controlador de valor que se utilizará como una reserva
25 r.RegisterValueHandler<object>(fallback =>
26 {
27     return new HttpResponseMessage() {
28         Status = HttpStatusCode.OK,
29         Content = JsonContent.Create(fallback)
30     };
31 });
```

Nota sobre objetos enumerables y matrices

Los objetos de respuesta implícitos que implementan [IEnumerable](#) se leen en la memoria a través del método `ToArray()` antes de ser convertidos a través de un controlador de valor definido. Para que esto ocurra, el objeto `IEnumerable` se convierte en una matriz de objetos, y el convertidor de respuesta siempre recibirá un `Object[]` en lugar del tipo original.

Considera el siguiente escenario:

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

En el ejemplo anterior, el convertidor `IEnumerable<string>` **nunca se llamará**, porque el objeto de entrada siempre será un `Object[]` y no es convertible a un `IEnumerable<string>`. Sin embargo, el convertidor siguiente que recibe un `IEnumerable<object>` recibirá su entrada, ya que su valor es compatible.

Si necesitas manejar realmente el tipo de objeto que se enumerará, necesitarás utilizar reflexión para obtener el tipo del elemento de la colección. Todos los objetos enumerables (listas, matrices y colecciones) se convierten en una matriz de objetos por el convertidor de respuesta HTTP.

Los valores que implementan [IAsyncEnumerable](#) se manejan automáticamente por el servidor si la propiedad `ConvertIAsyncEnumerableIntoEnumerable` está habilitada, de manera similar a lo que sucede con `IEnumerable`. Una enumeración asíncrona se convierte en un enumerador bloqueante y luego se convierte en una matriz de objetos sincrónico.

Registro

Puedes configurar Sisk para escribir automáticamente registros de acceso y errores. Es posible definir la rotación de logs, extensiones y frecuencia.

La clase [LogStream](#) proporciona una forma asíncrona de escribir logs y mantenerlos en una cola de escritura esperable.

En este artículo mostraremos cómo configurar el registro para tu aplicación.

Registros de acceso basados en archivos

Los logs a archivos abren el archivo, escriben el texto de la línea y luego cierran el archivo por cada línea escrita. Este procedimiento se adoptó para mantener la respuesta de escritura en los logs.

Program.cs

C#

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

El código anterior escribirá todas las solicitudes entrantes en el archivo `logs/access.log`. Ten en cuenta que, el archivo se crea automáticamente si no existe, sin embargo la carpeta antes de él no. No es necesario crear el directorio `logs/` ya que la clase `LogStream` lo crea automáticamente.

Registro basado en flujo

Puedes escribir archivos de registro en instancias de objetos `TextWriter`, como `Console.Out`, pasando un objeto `TextWriter` en el constructor:

Program.cs

C#

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

Para cada mensaje escrito en el registro basado en flujo, se llama al método `TextWriter.Flush()`.

Formateo del registro de acceso

Puedes personalizar el formato del registro de acceso mediante variables predefinidas. Considera la siguiente línea:

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

Escribirá un mensaje 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]
```

Puedes formatear tu archivo de registro con el formato descrito en la tabla:

Valor	Qué representa	Ejemplo
%dd	Día del mes (formateado como dos dígitos)	05
%dmmm	Nombre completo del mes	July

Valor	Qué representa	Ejemplo
%dmm	Nombre abreviado del mes (tres letras)	Jul
%dm	Número del mes (formateado como dos dígitos)	07
%dy	Año (formateado como cuatro dígitos)	2023
%th	Hora en formato de 12 horas	03
%tH	Hora en formato de 24 horas (HH)	15
%ti	Minutos (formateados como dos dígitos)	30
%ts	Segundos (formateados como dos dígitos)	45
%tm	Milisegundos (formateados como tres dígitos)	123
%tz	Desplazamiento de zona horaria (horas totales en UTC)	+03:00
%ri	Dirección IP remota del cliente	192.168.1.100
%rm	Método HTTP (mayúsculas)	GET
%rs	Esquema URI (http/https)	https
%ra	Autoridad URI (dominio)	example.com
%rh	Host de la solicitud	www.example.com
%rp	Puerto de la solicitud	443
%rz	Ruta de la solicitud	/path/to/resource
%rq	Cadena de consulta	?key=value&another=123
%sc	Código de estado de respuesta HTTP	200
%sd	Descripción del estado de respuesta HTTP	OK
%lin	Tamaño legible del pedido	1.2 KB
%linr	Tamaño bruto del pedido (bytes)	1234
%lou	Tamaño legible de la respuesta	2.5 KB
%lour	Tamaño bruto de la respuesta (bytes)	2560
%lms	Tiempo transcurrido en milisegundos	120

Valor	Qué representa	Ejemplo
%ls	Estado de ejecución	Executed
%{header-name}	Representa el encabezado <code>header-name</code> de la solicitud.	Mozilla/5.0 (platform; rv:gecko [...]
%{:res-name}	Representa el encabezado <code>res-name</code> de la respuesta.	

Rotación de logs

Puedes configurar el servidor HTTP para rotar los archivos de registro a un archivo comprimido .gz cuando alcancen un cierto tamaño. El tamaño se verifica periódicamente según el límite que definas.

```

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

```

El código anterior verificará cada seis horas si el archivo de LogStream ha alcanzado su límite de 64 MB. Si es así, el archivo se comprime a un archivo .gz y luego se limpia `access.log`.

Durante este proceso, la escritura en el archivo está bloqueada hasta que el archivo se comprime y limpia. Todas las líneas que se ingresen para escribir en este período estarán en una cola esperando el final de la compresión.

Esta función solo funciona con LogStreams basados en archivos.

Registro de errores

Cuando un servidor no lanza errores al depurador, reenvía los errores a la escritura de logs cuando hay alguno.

Puedes configurar la escritura de errores con:

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

Esta propiedad solo escribirá algo en el registro si el error no es capturado por la devolución de llamada o la propiedad [Router.CallbackErrorHandler](#).

El error escrito por el servidor siempre escribe la fecha y hora, los encabezados de la solicitud (no el cuerpo), la traza del error y la traza de la excepción interna, si hay alguna.

Otras instancias de registro

Tu aplicación puede tener cero o múltiples LogStreams, no hay límite en cuántos canales de registro puede tener. Por lo tanto, es posible dirigir el registro de tu aplicación a un archivo distinto del AccessLog o ErrorLog predeterminado.

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

Extender LogStream

Puedes extender la clase `LogStream` para escribir formatos personalizados, compatibles con el motor de registro actual de Sisk. El ejemplo siguiente permite escribir mensajes coloridos en la Consola mediante la biblioteca Spectre.Console:

CustomLogStream.cs

C#

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

```
    }  
}
```

Otra forma de escribir automáticamente logs personalizados para cada solicitud/respuesta es crear un [HttpServerHandler](#). El ejemplo siguiente es un poco más completo. Escribe el cuerpo de la solicitud y respuesta en JSON a la Consola. Puede ser útil para depurar solicitudes en general. Este ejemplo hace uso de ContextBag y HttpServerHandler.

Program.cs

C#

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

JsonMessageHandler.cs

C#

```
1  class JsonMessageHandler : HttpServerHandler  
2  {  
3      protected override void OnHttpRequestOpen(HttpRequest request)  
4      {  
5          if (request.Method != HttpMethod.Get && request.Headers["Content-  
6              Type"]?.Contains("json", StringComparison.InvariantCultureIgnoreCase) == true)  
7          {
```

```

8             // En este punto, la conexión está abierta y el cliente ha enviado el
9             encabezado especificando
10            // que el contenido es JSON. La línea siguiente lee el contenido y lo deja
11            almacenado en la solicitud.
12            //
13            // Si el contenido no se lee en la acción de solicitud, es probable que el GC
14            lo recoja
15            // después de enviar la respuesta al cliente, por lo que el contenido puede no
16            estar disponible después de cerrar la respuesta.
17            //
18            _ = request.RawBody;
19
20            // agrega una pista en el contexto para indicar que esta solicitud tiene un
21            cuerpo JSON
22            request.Bag.Add("IsJsonRequest", true);
23        }
24    }
25
26    protected override async void OnHttpRequestClose(HttpServerExecutionResult result)
27    {
28        string? requestJson = null,
29                    responseJson = null,
30                    responseMessage;
31
32        if (result.Request.Bag.ContainsKey("IsJsonRequest"))
33        {
34            // reformatea el JSON usando la biblioteca CypherPotato.LightJson
35            var content = result.Request.Body;
36            requestJson = JsonValue.Deserialize(content, new JsonOptions() { WriteIndented
37 = true }).ToString();
38        }
39
40        if (result.Response is {} response)
41        {
42            var content = response.Content;
43            responseMessage = $"{(int)response.Status}
{HttpStatusInformation.GetStatusCodeDescription(response.Status)};
44
45
46            if (content is HttpContent httpContent &&
47                // verifica si la respuesta es JSON
48                httpContent.Headers.ContentType?.MediaType?.Contains("json",
49                StringComparison.InvariantCultureIgnoreCase) = true)
50            {
51                string json = await httpContent.ReadAsStringAsync();
52                responseJson = JsonValue.Deserialize(json, new JsonOptions() {
53                WriteIndented = true }).ToString();
54            }
55        }

```

```
56     else
57     {
58         // obtiene el estado interno de manejo del servidor
59         responseMessage = result.Status.ToString();
60     }
61
62     StringBuilder outputMessage = new StringBuilder();
63
64     if (requestJson != null)
65     {
66         outputMessage.AppendLine("-----");
67         outputMessage.AppendLine($">>> {result.Request.Method} {result.Request.Path}");
68
69         if (requestJson is not null)
70             outputMessage.AppendLine(requestJson);
71     }
72
73     outputMessage.AppendLine($"<<< {responseMessage}");
74
75     if (responseJson is not null)
76         outputMessage.AppendLine(responseJson);
77
78     outputMessage.AppendLine("-----");
79
80     await Console.Out.WriteLineAsync(outputMessage.ToString());
81 }
82 }
```

Eventos Enviados por Servidor

Sisk admite el envío de mensajes a través de Eventos Enviados por Servidor de forma predeterminada. Puedes crear conexiones desechables y persistentes, obtener las conexiones durante el tiempo de ejecución y utilizarlas.

Esta característica tiene algunas limitaciones impuestas por los navegadores, como el envío de solo mensajes de texto y no poder cerrar permanentemente una conexión. Una conexión cerrada en el lado del servidor hará que un cliente intente reconectar periódicamente cada 5 segundos (3 para algunos navegadores).

Estas conexiones son útiles para enviar eventos desde el servidor al cliente sin que el cliente solicite la información cada vez.

Creando una conexión SSE

Una conexión SSE funciona como una solicitud HTTP regular, pero en lugar de enviar una respuesta y cerrar inmediatamente la conexión, la conexión se mantiene abierta para enviar mensajes.

Al llamar al método [HttpRequest.GetEventSource\(\)](#), la solicitud se pone en un estado de espera mientras se crea la instancia de SSE.

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

En el código anterior, creamos una conexión SSE y enviamos un mensaje "Hola, mundo", luego cerramos la conexión SSE desde el lado del servidor.

NOTE

Al cerrar una conexión en el lado del servidor, por defecto el cliente intentará conectarse de nuevo en ese extremo y la conexión se reiniciará, ejecutando el método de nuevo, por siempre.

Es común reenviar un mensaje de terminación desde el servidor cada vez que la conexión se cierra desde el servidor para evitar que el cliente intente reconectar de nuevo.

Agregando encabezados

Si necesitas enviar encabezados, puedes utilizar el método [HttpRequestEventArgs.AppendHeader](#) antes de enviar cualquier mensaje.

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

```

Ten en cuenta que es necesario enviar los encabezados antes de enviar cualquier mensaje.

Conexiones Wait-For-Fail

Las conexiones normalmente se terminan cuando el servidor ya no puede enviar mensajes debido a una posible desconexión en el lado del cliente. De esta manera, la conexión se termina automáticamente y la instancia de la clase se descarta.

Incluso con una reconexión, la instancia de la clase no funcionará, ya que está vinculada a la conexión anterior. En algunas situaciones, es posible que necesites esta conexión más adelante y no quieras administrarla a través del método de devolución de llamada de la ruta.

Para esto, podemos identificar las conexiones SSE con un identificador y obtenerlas utilizando más tarde, incluso fuera de la devolución de llamada de la ruta. Además, marcamos la conexión con [WaitForFail](#) para no terminar la ruta y terminar la conexión automáticamente.

Una conexión SSE en KeepAlive esperará un error de envío (causado por la desconexión) para reanudar la ejecución del método. También es posible establecer un tiempo de espera para esto. Después del tiempo, si no se ha enviado ningún mensaje, la conexión se termina y la ejecución se reanuda.

```
1   r += new Route(RouteMethod.Get, "/", (req) =>
2   {
3     using var sse = req.GetEventSource("mi-indice-conexion");
4
5     sse.WaitForFail(TimeSpan.FromSeconds(15)); // esperar 15 segundos sin ningún mensaje antes
6     de terminar la conexión
7
8     return sse.Close();
9   });
10 }
```

El método anterior creará la conexión, la manejará y esperará una desconexión o error.

```
1   HttpRequestEventSource? evs = server.EventSources.GetByIdentifier("mi-indice-conexion");
2   if (evs != null)
3   {
4     // la conexión todavía está viva
5     evs.Send("Hola de nuevo!");
6   }
7 }
```

Y el fragmento de código anterior intentará buscar la conexión recién creada y, si existe, enviará un mensaje a ella.

Todas las conexiones activas del servidor que estén identificadas estarán disponibles en la colección [HttpServer.EventSources](#). Esta colección solo almacena conexiones activas e identificadas. Las conexiones cerradas se eliminan de la colección.

NOTE

Es importante tener en cuenta que Keep Alive tiene un límite establecido por componentes que pueden estar conectados a Sisk de una manera incontrolable, como un proxy web, un kernel HTTP o un controlador de red, y cierran las conexiones inactivas después de un período determinado de tiempo.

Por lo tanto, es importante mantener la conexión abierta enviando pings periódicos o extendiendo el tiempo máximo antes de que se cierre la conexión. Lee la siguiente sección para comprender mejor el envío de pings periódicos.

Configurar política de pings de conexión

La política de pings es una forma automatizada de enviar mensajes periódicos a tu cliente. Esta función permite al servidor comprender cuándo el cliente se ha desconectado de esa conexión sin tener que mantener la conexión abierta indefinidamente.

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

En el código anterior, cada 5 segundos, se enviará un nuevo mensaje de ping al cliente. Esto mantendrá viva la conexión TCP y evitará que se cierre debido a la inactividad. Además, cuando un mensaje falla al enviarse, la conexión se cierra automáticamente, liberando los recursos utilizados por la conexión.

Consultar conexiones

Puedes buscar conexiones activas utilizando un predicado en el identificador de conexión, para poder transmitir, por ejemplo.

```
1 HttpRequestEventSource[] evs = server.EventSources.Find(es => es.StartsWith("mi-
2 conexión-"));
3 foreach (HttpRequestEventSource e in evs)
4 {
5     e.Send("Transmisión a todas las fuentes de eventos que comienzan con 'mi-conexion-'");
6 }
```

También puedes utilizar el método [All](#) para obtener todas las conexiones SSE activas.

Web Sockets

Sisk soporta websockets también, como recibir y enviar mensajes a su cliente.

Esta característica funciona bien en la mayoría de los navegadores, pero en Sisk todavía es experimental. Por favor, si encuentras algún error, repórtalo en [github](#).

Aceptar mensajes de forma asíncrona

Los mensajes WebSocket se reciben en orden, se encolan hasta ser procesados por `ReceiveMessageAsync`. Este método no devuelve ningún mensaje cuando se alcanza el tiempo de espera, cuando la operación es cancelada o cuando el cliente se desconecta.

Solo puede ocurrir una operación de lectura y escritura simultáneamente, por lo tanto, mientras esperas un mensaje con `ReceiveMessageAsync`, no es posible escribir al cliente conectado.

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

Aceptar mensajes de forma síncrona

El ejemplo siguiente contiene una forma de usar un websocket síncrono, sin un contexto asíncrono, donde recibes los mensajes, los manejas y terminas de usar el socket.

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

});
```

Política de Ping

Similar a cómo funciona la política de ping en Server Side Events, también puedes configurar una política de ping para mantener la conexión TCP abierta si hay inactividad en ella.

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

Sintaxis de descarte

El servidor HTTP se puede utilizar para escuchar una solicitud de devolución de llamada desde una acción, como la autenticación OAuth, y se puede descartar después de recibir esa solicitud. Esto puede ser útil en casos donde necesite una acción en segundo plano pero no desee configurar una aplicación HTTP completa para ello.

El siguiente ejemplo muestra cómo crear un servidor HTTP de escucha en el puerto 5555 con [CreateListener](#) y esperar el siguiente contexto:

```
1  using (var server = HttpServer.CreateListener\(5555\))
2  {
3      // espera la próxima solicitud HTTP
4      var context = await server.WaitNextAsync\(\);
5      Console.WriteLine\(\$"Ruta solicitada: {context.Request.Path}"\);
6  }
```

La función [WaitNext](#) espera el próximo contexto de un procesamiento de solicitud completado. Una vez que se obtiene el resultado de esta operación, el servidor ya ha procesado completamente la solicitud y ha enviado la respuesta al cliente.

Inyección de dependencias

Es común dedicar miembros e instancias que duran toda la vida de una solicitud, como una conexión a una base de datos, un usuario autenticado o un token de sesión. Una de las posibilidades es a través de [HttpContext.RequestBag](#), que crea un diccionario que dura toda la vida de una solicitud.

Este diccionario se puede acceder desde [manejadores de solicitudes](#) y definir variables a lo largo de esa solicitud. Por ejemplo, un manejador de solicitudes que autentica a un usuario establece este usuario dentro de `HttpContext.RequestBag`, y dentro de la lógica de la solicitud, este usuario se puede recuperar con `HttpContext.RequestBag.Get<User>()`.

Aquí hay un ejemplo:

RequestHandlers/AuthenticateUser.cs

C#

```
1  public class AuthenticateUser : IRequestHandler
2  {
3      public RequestHandlerExecutionMode ExecutionMode { get; init; } =
4      RequestHandlerExecutionMode.BeforeResponse;
5
6      public HttpResponse? Execute(HttpRequest request, HttpContext context)
7      {
8          User authenticatedUser = AuthenticateUser(request);
9          context.RequestBag.Set(authenticatedUser);
10         return null; // avanzar a la siguiente solicitud de manejo o lógica de solicitud
11     }
12 }
```

Controllers/HelloController.cs

C#

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

Este es un ejemplo preliminar de esta operación. La instancia de `User` se creó dentro del manejador de solicitudes dedicado a la autenticación, y todas las rutas que utilizan este manejador de solicitudes tendrán la garantía de que habrá un `User` en su instancia de `HttpContext.RequestBag`.

Es posible definir lógica para obtener instancias cuando no se han definido previamente en `RequestBag` a través de métodos como `GetOrAdd` o `GetOrAddAsync`.

Desde la versión 1.3, se introdujo la propiedad estática `HttpContext.Current`, que permite acceder al `HttpContext` actualmente en ejecución del contexto de la solicitud. Esto permite exponer miembros del `HttpContext` fuera de la solicitud actual y definir instancias en objetos de rutas.

El ejemplo siguiente define un controlador que tiene miembros comúnmente accedidos por el contexto de una solicitud.

Controllers/Controller.cs

C#

```
1  public abstract class Controller : RouterModule
2  {
3      public DbContext Database
4      {
5          get
6          {
7              // crear un DbContext o obtener el existente
8              return HttpContext.Current.RequestBag.GetOrAdd(() => new DbContext());
9          }
10     }
11
12     // la siguiente linea lanzará una excepción si la propiedad se accede cuando el
13     Usuario no
14     // está definido en la bolsa de solicitudes
15     public User AuthenticatedUser { get => HttpContext.Current.RequestBag.Get<User>(); }
16
17     // También se admite la exposición de la instancia de HttpRequest
18     public HttpRequest Request { get => HttpContext.Current.Request; }
19 }
```

Y define tipos que heredan del controlador:

Controllers/PostsController.cs

C#

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

```

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

```

Para el ejemplo anterior, necesitarás configurar un [manejador de valores](#) en tu enrutador para que los objetos devueltos por el enrutador se transformen en un [HttpResponse](#) válido.

Tenga en cuenta que los métodos no tienen un argumento `HttpRequest request` como está presente en otros métodos. Esto se debe a que, desde la versión 1.3, el enrutador admite dos tipos de delegados para respuestas de enrutamiento: [RouteAction](#), que es el delegado predeterminado que recibe un argumento `HttpRequest`, y [ParameterlessRouteAction](#). El objeto `HttpRequest` aún se puede acceder desde ambos delegados a través de la propiedad `Request` del `HttpContext` estático en el subproceso.

En el ejemplo anterior, definimos un objeto desecharable, el `DbContext`, y necesitamos asegurarnos de que todas las instancias creadas en un `DbContext` se desechen cuando la sesión HTTP finalice. Para ello, podemos utilizar dos formas de lograrlo. Una es crear un [manejador de solicitudes](#) que se ejecute después de la acción del enrutador, y la otra forma es a través de un [manejador de servidor personalizado](#).

Para el primer método, podemos crear el manejador de solicitudes directamente en el método `OnSetup` heredado de `RouterModule`:

Controllers/PostsController.cs

C#

```

1  public abstract class Controller : RouterModule
2  {
3      ...
4
5      protected override void OnSetup(Router parentRouter)
6      {

```

```

7         base.OnSetup(parentRouter);
8
9         HasRequestHandler(RequestHandler.Create(
10             execute: (req, ctx) =>
11             {
12                 // obtener un DbContext definido en el contexto del manejador de
13                 solicitudes y
14                 // desecharlo
15                 ctx.RequestBag.GetOrDefault<DbContext>()?.Dispose();
16                 return null;
17             },
18             executionMode: RequestHandlerExecutionMode.AfterResponse));
19     }
}

```

TIP

Desde Sisk versión 1.4, la propiedad `HttpServerConfiguration.DisposeDisposableContextValues` se introdujo y se habilitó de forma predeterminada, lo que define si el servidor HTTP debe desechar todos los valores `IDisposable` en la bolsa de contexto cuando se cierra una sesión HTTP.

El método anterior garantizará que el `DbContext` se deseche cuando la sesión HTTP se finalice. Puedes hacer esto para más miembros que necesitan desecharse al final de una respuesta.

Para el segundo método, puedes crear un [manejador de servidor personalizado](#) que desechará el `DbContext` cuando la sesión HTTP se finalice.

Server/Handlers/ObjectDisposerHandler.cs

C#

```

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

```

Y usarlo en tu constructor de aplicaciones:

Program.cs

C#

```

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

```

Esta es una forma de manejar la limpieza de código y mantener las dependencias de una solicitud separadas por el tipo de módulo que se utilizará, reduciendo la cantidad de código duplicado dentro de cada acción de un enrutador. Es una práctica similar a la que se utiliza la inyección de dependencias en frameworks como ASP.NET.

Transmisión de contenido

El Sisk admite la lectura y el envío de flujos de contenido desde y hacia el cliente. Esta característica es útil para eliminar la sobrecarga de memoria para serializar y deserializar contenido durante la vida útil de una solicitud.

Flujo de contenido de la solicitud

Los contenidos pequeños se cargan automáticamente en la memoria del búfer de la conexión HTTP, cargando rápidamente este contenido en [HttpRequest.Body](#) y [HttpRequest.RawBody](#). Para contenidos más grandes, se puede utilizar el método [HttpRequest.GetRequestStream](#) para obtener el flujo de lectura del contenido de la solicitud.

Es importante destacar que el método [HttpRequest.GetMultipartFormContent](#) lee todo el contenido de la solicitud en la memoria, por lo que puede no ser útil para leer contenidos grandes.

Consideremos el siguiente ejemplo:

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          // la solicitud no tiene contenido
8          return new HttpResponse ( HttpStatusInformation.BadRequest );
9      }
10
11      var contentStream = request.GetRequestStream ();
12      var outputFileName = Path.Combine (
13          AppDomain.CurrentDomain.BaseDirectory,
14          "uploads",
15          fileName );
16
17      using ( var fs = File.Create ( outputFileName ) ) {
18          await contentStream.CopyToAsync ( fs );
```

```

19     }
20
21     return new HttpResponseMessage() {
22         Content = JsonContent.Create(new { message = "Archivo enviado con éxito." } )
23     };
24 }
```

En el ejemplo anterior, el método `UploadDocument` lee el contenido de la solicitud y lo guarda en un archivo. No se realiza ninguna asignación de memoria adicional, excepto por el búfer de lectura utilizado por `Stream.CopyToAsync`. El ejemplo anterior elimina la presión de asignación de memoria para un archivo muy grande, lo que puede optimizar el rendimiento de la aplicación.

Es una buena práctica utilizar siempre un [CancellationToken](#) en una operación que pueda ser larga, como enviar archivos, ya que depende de la velocidad de la red entre el cliente y el servidor.

El ajuste con un `CancellationToken` se puede realizar de la siguiente manera:

Controller/UploadDocument.cs

C#

```

1 // el token de cancelación a continuación lanzará una excepción si se alcanza el tiempo de
2 espera de 30 segundos.
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 HttpResponseMessage( HttpStatusCode.BadRequest ) {
13         Content = JsonContent.Create( new { Error = "La carga superó el tiempo de carga
máximo (30 segundos)." } )
14     };
15 }
```

Flujo de contenido de la respuesta

Enviar contenido de respuesta también es posible. Actualmente, hay dos formas de hacerlo: a través del método `HttpRequest.GetResponseStream` y utilizando un contenido de tipo `StreamContent`.

Consideremos un escenario en el que necesitamos servir un archivo de imagen. Para hacer esto, podemos utilizar el siguiente código:

Controller/ImageController.cs

C#

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

El método anterior realiza una asignación de memoria cada vez que se lee el contenido de la imagen. Si la imagen es grande, esto puede causar un problema de rendimiento, y en situaciones de pico, incluso una sobrecarga de memoria y caída del servidor. En estas situaciones, la caché puede ser útil, pero no eliminará el problema, ya que la memoria seguirá reservada para ese archivo. La caché aliviará la presión de tener que asignar memoria para cada solicitud, pero para archivos grandes, no será suficiente.

Enviar la imagen a través de un flujo puede ser una solución al problema. En lugar de leer todo el contenido de la imagen, se crea un flujo de lectura en el archivo y se copia al cliente utilizando un búfer pequeño.

Enviar a través del método `GetResponseStream`

El método `HttpRequest.GetResponseStream` crea un objeto que permite enviar fragmentos de la respuesta HTTP a medida que se prepara el flujo de contenido. Este método es más manual, requiriendo que se defina el estado, los encabezados y el tamaño del contenido antes de enviar el contenido.

Controller/ImageController.cs

C#

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

```

3
4     var profilePictureFilename = "profile-picture.jpg";
5
6     // en esta forma de envío, el estado y el encabezado deben definirse
7     // antes de enviar el contenido
8     var requestStreamManager = request.GetResponseStream ();
9
10    requestStreamManager.SetStatus ( System.Net.HttpStatusCode.OK );
11    requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentType, "image/jpeg" );
12    requestStreamManager.SetHeader ( HttpKnownHeaderNames.ContentDisposition, $"inline;
13    filename={profilePictureFilename}" );
14
15    using (var fs = File.OpenRead ( profilePictureFilename )) {
16
17        // en esta forma de envío, también es necesario definir el tamaño del contenido
18        // antes de enviarlo.
19        requestStreamManager.SetContentLength ( fs.Length );
20
21        // si no se conoce el tamaño del contenido, se puede utilizar codificación
22        por fragmentos
23        // para enviar el contenido
24        requestStreamManager.SendChunked = true;
25
26        // y luego, escribir en el flujo de salida
27        await fs.CopyToAsync ( requestStreamManager.ResponseStream );
28    }
29}

```

Enviar contenido a través de un StreamContent

La clase [StreamContent](#) permite enviar contenido desde una fuente de datos como un flujo de bytes. Esta forma de envío es más fácil, eliminando los requisitos anteriores, y incluso permitiendo el uso de [codificación de compresión](#) para reducir el tamaño del contenido.

Controller/ImageController.cs

C#

```

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

```

```
10         ContentDisposition = $"inline; filename=\"{profilePictureFilename}\""
11     }
12 }
13 }
```

✖️ IMPORTANT

En este tipo de contenido, no encapsule el flujo en un bloque `using`. El contenido se descartará automáticamente por el servidor HTTP cuando se finalice el flujo de contenido, con o sin errores.

Habilitando CORS (Compartición de Recursos entre Orígenes) en Sisk

Sisk tiene una herramienta que puede ser útil para manejar la Compartición de Recursos entre Orígenes (CORS) cuando expones tu servicio públicamente. Esta característica no forma parte del protocolo HTTP sino que es una característica específica de los navegadores web definida por el W3C. Este mecanismo de seguridad evita que una página web haga solicitudes a un dominio diferente al que proporcionó la página web. Un proveedor de servicio puede permitir que ciertos dominios accedan a sus recursos, o solo uno.

Misma Origen

Para que un recurso sea identificado como “misma origen”, una solicitud debe identificar la cabecera `Origin` en su solicitud:

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

Y el servidor remoto debe responder con una cabecera `Access-Control-Allow-Origin` con el mismo valor que el origen solicitado:

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

Esta verificación es **explicita**: el host, puerto y protocolo deben ser los mismos que los solicitados. Verifica el ejemplo:

- Un servidor responde que su `Access-Control-Allow-Origin` es `https://example.com` :
 - `https://example.net` - el dominio es diferente.
 - `http://example.com` - el esquema es diferente.
 - `http://example.com:5555` - el puerto es diferente.

- `https://www.example.com` - el host es diferente.

En la especificación, solo se permite la sintaxis para ambas cabeceras, tanto para solicitudes como respuestas. La ruta URL se ignora. El puerto también se omite si es un puerto predeterminado (80 para HTTP y 443 para HTTPS).

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

Habilitando CORS

Nativamente, tienes el objeto `CrossOriginResourceSharingHeaders` dentro de tu `ListeningHost`.

Puedes configurar CORS al inicializar el servidor:

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

El código anterior enviará las siguientes cabeceras para **todas las respuestas**:

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

Estas cabeceras deben enviarse para todas las respuestas a un cliente web, incluidos errores y redirecciones.

Puedes notar que la clase `CrossOriginResourceSharingHeaders` tiene dos propiedades similares: `AllowOrigin` y `AllowOrigins`. Ten en cuenta que una es plural, mientras que la otra es singular.

- La propiedad `AllowOrigin` es estática: solo el origen que especifiques se enviará para todas las respuestas.
- La propiedad `AllowOrigins` es dinámica: el servidor verifica si el origen de la solicitud está contenido en esta lista. Si se encuentra, se envía para la respuesta de ese origen.

Caracteres comodín y cabeceras automáticas

Alternativamente, puedes usar un comodín (*) en el origen de la respuesta para indicar que cualquier origen está permitido acceder al recurso. Sin embargo, este valor no está permitido para solicitudes que tengan credenciales (cabeceras de autorización) y esta operación [resultará en un error](#).

Puedes solucionar este problema enumerando explícitamente los orígenes que serán permitidos a través de la propiedad `AllowOrigins` o también usar la constante `AutoAllowOrigin` en el valor de `AllowOrigin`. Esta propiedad mágica definirá la cabecera `Access-Control-Allow-Origin` con el mismo valor que la cabecera `Origin` de la solicitud.

También puedes usar `AutoFromRequestMethod` y `AutoFromRequestHeaders` para un comportamiento similar a `AllowOrigin`, que responde automáticamente según las cabeceras enviadas.

```
1  using var host = HttpServer.CreateBuilder()
2      .UseCors(new CrossOriginResourceSharingHeaders(
3
4          // Responde según la cabecera Origin de la solicitud
5          allowOrigin: CrossOriginResourceSharingHeaders.AutoAllowOrigin,
6
7          // Responde según la cabecera Access-Control-Request-Method o el método de
8          la solicitud
9          allowMethods: [CrossOriginResourceSharingHeaders.AutoFromRequestMethod],
10
11         // Responde según la cabecera Access-Control-Request-Headers o las
12         cabeceras enviadas
13         allowHeaders: [CrossOriginResourceSharingHeaders.AutoFromRequestHeaders]))
```

Otras Formas de Aplicar CORS

Si estás trabajando con `service providers`, puedes sobreescibir valores definidos en el archivo de configuración:

```
1 static async Task Main(string[] args)
2 {
3     using var app = HttpServer.CreateBuilder()
4         .UsePortableConfiguration( ... )
5         .UseCors(cors => {
6             // Sobrescribirá el origen definido en la configuración
7             // archivo.
8             cors.AllowOrigin = "http://example.com";
9         })
10        .Build();
11
12     await app.StartAsync();
13 }
```

Deshabilitando CORS en Rutas Específicas

La propiedad `UseCors` está disponible tanto para rutas como para todos los atributos de ruta y puede desactivarse con el siguiente ejemplo:

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

Reemplazando Valores en la Respuesta

Puedes reemplazar o eliminar valores explícitamente en una acción de enrutador:

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

Solicitudes Preflight

Una solicitud preflight es una solicitud de método [OPTIONS](#) que el cliente envía antes de la solicitud real.

El servidor Sisk siempre responderá a la solicitud con un `200 OK` y las cabeceras CORS aplicables, y luego el cliente puede proceder con la solicitud real. Esta condición solo no se aplica cuando existe una ruta para la solicitud con el [RouteMethod](#) configurado explícitamente para `Options`.

Deshabilitando CORS Globalmente

No es posible hacer esto. Para no usar CORS, no lo configures.

Extensión JSON-RPC

Sisk tiene un módulo experimental para una API [JSON-RPC 2.0](#), que te permite crear aplicaciones aún más simples. Esta extensión implementa estrictamente la interfaz de transporte JSON-RPC 2.0 y ofrece transporte a través de HTTP GET, solicitudes POST y también web-sockets con Sisk.

Puedes instalar la extensión a través de Nuget con el comando siguiente. Ten en cuenta que, en versiones experimentales/beta, debes habilitar la opción para buscar paquetes prelanzamiento en Visual Studio.

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

Interfaz de transporte

JSON-RPC es un protocolo de ejecución remota de procedimientos (RDP) sin estado y asíncrono que utiliza JSON para la comunicación de datos unidireccional. Una solicitud JSON-RPC se identifica típicamente por un ID, y una respuesta se entrega con el mismo ID que se envió en la solicitud. No todas las solicitudes requieren una respuesta, que se llaman "notificaciones".

La [especificación JSON-RPC 2.0](#) explica en detalle cómo funciona el transporte. Este transporte es agnóstico de dónde se utilizará. Sisk implementa este protocolo a través de HTTP, siguiendo las conformidades de [JSON-RPC sobre HTTP](#), que admite parcialmente las solicitudes GET, pero admite completamente las solicitudes POST. También se admiten los web-sockets, que proporcionan una comunicación de mensajes asíncrona.

Una solicitud JSON-RPC se parece a:

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

Y una respuesta exitosa se parece a:

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

Métodos JSON-RPC

El siguiente ejemplo muestra cómo crear una API JSON-RPC utilizando Sisk. Una clase de operaciones matemáticas realiza las operaciones remotas y entrega la respuesta serializada al cliente.

Program.cs

C#

```
1  using var app = HttpServer.CreateBuilder(port: 5555)
2      .UseJsonRPC((sender, args) =>
3      {
4          // agregar todos los métodos con la etiqueta WebMethod al controlador JSON-RPC
5          args.Handler.Methods.AddMethodsFromType(new MathOperations());
6
7          // asigna la ruta /service para manejar solicitudes JSON-RPC POST y GET
8          args.Router.MapPost("/service", args.Handler.Transport.HttpPost);
9          args.Router.MapGet("/service", args.Handler.Transport.HttpGet);
10
11         // crea un controlador de web-sockets en GET /ws
12         args.Router.MapGet("/ws", request =>
13         {
14             var ws = request.GetWebSocket();
15             ws.OnReceive += args.Handler.Transport.WebSocket;
16
17             ws.WaitForClose(timeout: TimeSpan.FromSeconds(30));
18             return ws.Close();
19         });
20     })
21     .Build();
22
23 await app.StartAsync();
```

MathOperations.cs

C#

```

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

El ejemplo anterior asignará los métodos `Sum` y `Sqrt` al controlador JSON-RPC, y estos métodos estarán disponibles en `GET /service`, `POST /service` y `GET /ws`. Los nombres de los métodos son insensibles a mayúsculas y minúsculas.

Los parámetros de los métodos se deserializan automáticamente en sus tipos específicos. También se admite el uso de parámetros con nombre en las solicitudes. La serialización JSON se realiza mediante la biblioteca [LightJson](#). Cuando un tipo no se deserializa correctamente, puedes crear un convertidor JSON personalizado para ese tipo y asociarlo con tus opciones de serializador JSON más adelante.

También puedes obtener el objeto `$.params` crudo de la solicitud JSON-RPC directamente en tu método.

MathOperations.cs

C#

```

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

Para que esto ocurra, `@params` debe ser el **único** parámetro en tu método, con exactamente el nombre `params` (en C#, el `@` es necesario para escapar este nombre de parámetro).

La deserialización de parámetros ocurre tanto para objetos con nombre como para matrices posicionales. Por ejemplo, el siguiente método se puede llamar de forma remota mediante ambas solicitudes:

```

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

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

Para una matriz, el orden de los parámetros debe seguirse.

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

Personalización del serializador

Puedes personalizar el serializador JSON en la propiedad `JsonRpcHandler.JsonSerializerOptions`. En esta propiedad, puedes habilitar el uso de [JSON5](#) para deserializar mensajes. Aunque no es una conformidad con JSON-RPC 2.0, JSON5 es una extensión de JSON que permite una escritura más legible y humana.

Program.cs

C#

```
1 using var host = HttpServer.CreateBuilder( 5556 )
2     .UseJsonRPC( ( o, e ) => {
3
4         // utiliza un comparador de nombres sanitizado. este comparador compara solo letras
5         // y dígitos en un nombre, y descarta otros símbolos. por ejemplo:
6         // foo_bar10 = FooBar10
7         e.Handler.JsonSerializerOptions.PropertyNameComparer = new
8         JsonSanitizedComparer();
9
10        // habilita JSON5 para el intérprete JSON. incluso activando esto, el JSON plano
```

```
11     todavia se admite
12         e.Handler.JsonSerializerOptions.SerializationFlags =
13     LightJson.Serialization.JsonSerializationFlags.Json5;
14
15         // asigna la ruta POST /service al controlador JSON-RPC
16         e.Router.MapPost ( "/service", e.Handler.Transport.HttpPost );
17     } )
18
19     .Build ();
20
21
22 host.Start ();
```

Proxy SSL

⚠️ WARNING

Esta característica es experimental y no debe usarse en producción. Por favor, consulte [este documento](#) si desea hacer que Sisk funcione con SSL.

El Proxy SSL de Sisk es un módulo que proporciona una conexión HTTPS para un `ListeningHost` en Sisk y enruta los mensajes HTTPS a un contexto HTTP inseguro. El módulo se creó para proporcionar una conexión SSL para un servicio que utiliza [HttpListener](#) para ejecutarse, que no admite SSL.

El proxy se ejecuta dentro de la misma aplicación y escucha los mensajes HTTP/1.1, reenviándolos en el mismo protocolo a Sisk. Actualmente, esta característica es muy experimental y puede ser lo suficientemente inestable como para no usarse en producción.

En la actualidad, el `SslProxy` admite casi todas las características de HTTP/1.1, como keep-alive, codificación chunked, websockets, etc. Para una conexión abierta al proxy SSL, se crea una conexión TCP al servidor de destino y el proxy se reenvía a la conexión establecida.

El `SslProxy` se puede utilizar con `HttpServer.CreateBuilder` de la siguiente manera:

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

Debes proporcionar un certificado SSL válido para el proxy. Para asegurarte de que el certificado sea aceptado por los navegadores, recuerda importarlo en el sistema operativo para que funcione correctamente.

Autenticación Básica

El paquete de Autenticación Básica agrega un controlador de solicitudes capaz de manejar el esquema de autenticación básica en su aplicación Sisk con muy poca configuración y esfuerzo. La autenticación HTTP básica es una forma minimalista de autenticar solicitudes mediante un identificador de usuario y una contraseña, donde la sesión es controlada exclusivamente por el cliente y no hay tokens de autenticación o acceso.



Lea más sobre el esquema de autenticación básica en la [especificación de MDN](#).

Instalación

Para empezar, instale el paquete Sisk.BasicAuth en su proyecto:

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

Puede ver más formas de instalarlo en su proyecto en el [repositorio de Nuget](#).

Creación de su controlador de autenticación

Puede controlar el esquema de autenticación para un módulo completo o para rutas individuales. Para ello, primero escribamos nuestro primer controlador de autenticación básica.

En el ejemplo a continuación, se establece una conexión con la base de datos, se verifica si el usuario existe y si la contraseña es válida, y después de eso, se almacena el usuario en la bolsa de contexto.

```
1  public class UserAuthHandler : BasicAuthenticateRequestHandler
2  {
3      public UserAuthHandler() : base()
4      {
5          Realm = "Para entrar en esta página, por favor, informe sus credenciales.";
```

```

6      }
7
8      public override HttpResponseMessage? OnValidating(BasicAuthenticationCredentials credentials,
9      HttpContext context)
10     {
11         DbContext db = new DbContext();
12
13         // en este caso, estamos utilizando el correo electrónico como el campo de
14         identificador de usuario, así que
15         // vamos a buscar un usuario utilizando su correo electrónico.
16         User? user = db.Users.FirstOrDefault(u => u.Email == credentials.UserId);
17         if (user == null)
18         {
19             return base.CreateUnauthorizedResponse("Lo sentimos, no se encontró ningún
20             usuario con este correo electrónico.");
21         }
22
23         // valida que la contraseña de las credenciales sea válida para este usuario.
24         if (!user.ValidatePassword(credentials.Password))
25         {
26             return base.CreateUnauthorizedResponse("Credenciales inválidas.");
27         }
28
29         // agrega el usuario conectado a la bolsa de contexto
30         // y continúa la ejecución
31         context.Bag.Add("loggedUser", user);
32         return null;
33     }
34 }

```

Así que solo asocie este controlador de solicitudes con nuestra ruta o clase.

```

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

```

O utilizando la clase [RouterModule](#):

```

1  public class UsersController : RouterModule
2  {
3      public ClientModule()
4      {
5          // ahora todas las rutas dentro de esta clase serán manejadas por
6          // UserAuthHandler.
7          base.HasRequestHandler(new UserAuthHandler());
8      }
9
10     [RouteGet("/")]
11     public string Index(HttpRequest request)
12     {
13         User loggedUser = (User)request.Context.RequestBag["loggedUser"];
14         return "Hola, " + loggedUser.Name + "!";
15     }
16 }
```

Observaciones

La responsabilidad principal de la autenticación básica se lleva a cabo en el lado del cliente. El almacenamiento, el control de caché, y el cifrado se manejan localmente en el cliente. El servidor solo recibe las credenciales y valida si se permite o no el acceso.

Tenga en cuenta que este método no es uno de los más seguros porque coloca una gran responsabilidad en el cliente, lo que puede ser difícil de rastrear y mantener la seguridad de sus credenciales. Además, es fundamental que las contraseñas se transmitan en un contexto de conexión segura (SSL), ya que no tienen cifrado inherente. Una breve intercepción en los encabezados de una solicitud puede exponer las credenciales de acceso de su usuario.

Opte por soluciones de autenticación más robustas para aplicaciones en producción y evite utilizar demasiados componentes prefabricados, ya que pueden no adaptarse a las necesidades de su proyecto y terminar exponiéndolo a riesgos de seguridad.

Proveedores de Servicios

Los Proveedores de Servicios son una forma de portar su aplicación Sisk a diferentes entornos con un archivo de configuración portátil. Esta característica permite cambiar el puerto del servidor, parámetros y otras opciones sin tener que modificar el código de la aplicación para cada entorno. Este módulo depende de la sintaxis de construcción de Sisk y se puede configurar a través del método `UsePortableConfiguration`.

Un proveedor de configuración se implementa con `IConfigurationProvider`, que proporciona un lector de configuración y puede recibir cualquier implementación. Por defecto, Sisk proporciona un lector de configuración JSON, pero también hay un paquete para archivos INI. También puede crear su propio proveedor de configuración y registrararlo con:

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

Como se mencionó anteriormente, el proveedor predeterminado es un archivo JSON. Por defecto, el nombre del archivo que se busca es `service-config.json`, y se busca en el directorio actual del proceso en ejecución, no en el directorio del ejecutable.

Puede elegir cambiar el nombre del archivo, así como dónde Sisk debe buscar el archivo de configuración, con:

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

El código anterior buscará el archivo config.toml en el directorio actual del proceso en ejecución. Si no se encuentra, luego buscará en el directorio donde se encuentra el ejecutable. Si el archivo no existe, el parámetro `createIfDontExists` se honra, creando el archivo, sin contenido, en la última ruta de acceso probada (basada en `lookupDirectories`), y se lanza un error en la consola, evitando que la aplicación se inicialice.

TIP

Puede ver el código fuente del lector de configuración INI y el lector de configuración JSON para entender cómo se implementa un `IConfigurationProvider`.

Lectura de configuraciones desde un archivo JSON

Por defecto, Sisk proporciona un proveedor de configuración que lee configuraciones desde un archivo JSON. Este archivo sigue una estructura fija y está compuesto por los siguientes parámetros:

```
1  {
2      "Server": {
3          "DefaultEncoding": "UTF-8",
4          "ThrowExceptions": true,
5          "IncludeRequestIdHeader": true
6      },
7      "ListeningHost": {
8          "Label": "Mi aplicación Sisk",
9          "Ports": [
10              "http://localhost:80/",
11              "https://localhost:443/", // Los archivos de configuración también
12      admitten comentarios
13      ],
14      "CrossOriginResourceSharingPolicy": {
15          "AllowOrigin": "*",
16          "AllowOrigins": [ "*" ], // Nuevo en 0.14
17          "AllowMethods": [ "*" ],
18          "AllowHeaders": [ "*" ],
19          "MaxAge": 3600
20      },
21      "Parameters": {
22          "MySqlConnection": "server=localhost;user=root;"
23      }
24  }
```

Los parámetros creados a partir de un archivo de configuración se pueden acceder en el constructor del servidor:

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

Cada lector de configuración proporciona una forma de leer los parámetros de inicialización del servidor. Algunas propiedades se indican que deben estar en el entorno del proceso en lugar de estar definidas en el archivo de configuración, como datos de API sensibles, claves de API, etc.

Estructura del archivo de configuración

El archivo de configuración JSON está compuesto por las siguientes propiedades:

Propiedad	Obligatorio	Descripción
Server	Requerido	Representa el servidor en sí con sus configuraciones.
Server.AccessLogsStream	Opcional	Por defecto, es <code>console</code> . Especifica la secuencia de salida de los registros de acceso. Puede ser un nombre de archivo, <code>null</code> o <code>console</code> .
Server.ErrorsLogsStream	Opcional	Por defecto, es <code>null</code> . Especifica la secuencia de salida de los registros de errores. Puede ser un nombre de archivo, <code>null</code> o <code>console</code> .

Propiedad	Obligatorio	Descripción
Server.MaximumContentLength	Opcional	
Server.MaximumContentLength	Opcional	Por defecto, es <code>0</code> . Especifica la longitud máxima de contenido en bytes. Cero significa infinito.
Server.IncludeRequestIdHeader	Opcional	Por defecto, es <code>false</code> . Especifica si el servidor HTTP debe enviar el encabezado <code>X-Request-Id</code> .
Server.ThrowExceptions	Opcional	Por defecto, es <code>true</code> . Especifica si las excepciones no controladas deben lanzarse. Establezca en <code>false</code> cuando esté en producción y <code>true</code> cuando esté depurando.
ListeningHost	Requerido	Representa el host de escucha del servidor.
ListeningHost.Label	Opcional	Representa la etiqueta de la aplicación.
ListeningHost.Ports	Requerido	Representa una matriz de cadenas, que coincide con la sintaxis de ListeningPort .
ListeningHost.CrossOriginResourceSharingPolicy	Opcional	Configura los encabezados CORS para la aplicación.
ListeningHost.CrossOriginResourceSharingPolicy.AllowCredentials	Opcional	Por defecto, es <code>false</code> . Especifica el encabezado <code>Allow-Credentials</code> .
ListeningHost.CrossOriginResourceSharingPolicy.ExposeHeaders	Opcional	Por defecto, es <code>null</code> . Esta propiedad espera una matriz de cadenas.

Propiedad	Obligatorio	Descripción
ListeningHost.CrossOriginResourceSharingPolicy.ExposeHeaders		Especifica el encabezado Expose-Headers .
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigin	Opcional	Por defecto, es null . Esta propiedad espera una cadena. Especifica el encabezado Allow-Origin .
ListeningHost.CrossOriginResourceSharingPolicy.AllowOrigins	Opcional	Por defecto, es null . Esta propiedad espera una matriz de cadenas. Especifica múltiples encabezados Allow-Origin . Consulte AllowOrigins para obtener más información.
ListeningHost.CrossOriginResourceSharingPolicy.AllowMethods	Opcional	Por defecto, es null . Esta propiedad espera una matriz de cadenas. Especifica el encabezado Allow-Methods .
ListeningHost.CrossOriginResourceSharingPolicy.AllowHeaders	Opcional	Por defecto, es null . Esta propiedad espera una matriz de cadenas. Especifica el encabezado Allow-Headers .
ListeningHost.CrossOriginResourceSharingPolicy.MaxAge	Opcional	Por defecto, es null . Esta propiedad espera un entero. Especifica el encabezado Max-Age en segundos.
ListeningHost.Parameters	Opcional	Especifica las propiedades proporcionadas al método de configuración de la aplicación.

Proveedor de configuración INI

Sisk tiene un método para obtener configuraciones de inicio diferentes a JSON. De hecho, cualquier canal que implemente [IConfigurationReader](#) se puede utilizar con [PortableConfigurationBuilder.WithConfigurationPipeline](#), leyendo la configuración del servidor desde cualquier tipo de archivo.

El paquete [Sisk.IniConfiguration](#) proporciona un lector de archivos INI basado en flujo que no lanza excepciones por errores de sintaxis comunes y tiene una sintaxis de configuración simple. Este paquete se puede utilizar fuera del marco de Sisk, ofreciendo flexibilidad para proyectos que requieren un lector de documentos INI eficiente.

Instalación

Para instalar el paquete, puede comenzar con:

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

También puede instalar el paquete principal, que no incluye el lector de configuración INI [IConfigurationReader](#), ni la dependencia de Sisk, solo los serializadores INI:

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

Con el paquete principal, puede utilizarlo en su código como se muestra en el ejemplo a continuación:

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

El código anterior buscará un archivo app.ini en el directorio actual del proceso (CurrentDirectory). El archivo INI se ve así:

```

1 [Server]
2 # Se admiten varias direcciones de escucha
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 y sintaxis INI

Implementación actual del sabor:

- Los nombres de propiedades y secciones son **insensibles a mayúsculas y minúsculas**.
- Los nombres de propiedades y valores son **recortados**, a menos que los valores estén entre comillas.
- Los valores pueden estar entre comillas simples o dobles. Las comillas pueden tener saltos de línea dentro de ellas.
- Se admiten comentarios con # y ; . También se admiten **comentarios al final**.
- Las propiedades pueden tener varios valores.

En detalle, la documentación para el "sabor" del analizador INI utilizado en Sisk está [disponible en este documento](#).

Utilizando el siguiente código INI como ejemplo:

```
1  One = 1
2  Value = this is an value
3  Another value = "this value
4      has an line break on it"
5
6  ; el código a continuación tiene algunos colores
7  [some section]
8  Color = Red
9  Color = Blue
10 Color = Yellow ; no use yellow
```

Analícelo con:

```
1  // analice el texto INI desde la cadena
2  IniDocument doc = IniDocument.FromString(iniText);
3
4  // obtenga un valor
5  string? one = doc.Global.GetOne("one");
6  string? anotherValue = doc.Global.GetOne("another value");
7
8  // obtenga varios valores
9  string[]? colors = doc.GetSection("some section")?.GetMany("color");
```

Parámetros de configuración

Sección y nombre	Admite varios valores	Descripción
Server.Listen	Sí	Las direcciones/puertos de escucha del servidor.
Server.Encoding	No	La codificación predeterminada del servidor.
Server.MaximumContentSize	No	El tamaño máximo de contenido en bytes del servidor.
Server.IncludeRequestIdHeader	No	Especifica si el servidor HTTP debe enviar el encabezado X-Request-Id.
Server.ThrowExceptions	No	Especifica si las excepciones no controladas deben lanzarse.
Server.AccessLogsStream	No	Especifica la secuencia de salida de registros de acceso.
Server.ErrorsLogsStream	No	Especifica la secuencia de salida de registros de errores.
Cors.AllowMethods	No	Especifica el valor del encabezado CORS Allow-Methods.
Cors.AllowHeaders	No	Especifica el valor del encabezado CORS Allow-Headers.
Cors.AllowOrigins	No	Especifica varios encabezados Allow-Origin, separados por comas. AllowOrigins para más información.
Cors.AllowOrigin	No	Especifica un encabezado Allow-Origin.
Cors.ExposeHeaders	No	Especifica el valor del encabezado CORS Expose-Headers.
Cors.AllowCredentials	No	Especifica el valor del encabezado CORS Allow-Credentials.
Cors.MaxAge	No	Especifica el valor del encabezado CORS Max-Age.

Configuración manual (avanzada)

En esta sección, crearemos nuestro servidor HTTP sin ningún estándar predefinido, de una manera completamente abstracta. Aquí, puedes construir manualmente cómo funcionará tu servidor HTTP. Cada ListeningHost tiene un enrutador, y un servidor HTTP puede tener varios ListeningHosts, cada uno apuntando a un host diferente en un puerto diferente.

Primero, necesitamos entender el concepto de solicitud/respuesta. Es bastante simple: para cada solicitud, debe haber una respuesta. Sisk sigue este principio también. Creamos un método que responda con un mensaje "Hola, mundo!" en HTML, especificando el código de estado y los encabezados.

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

El siguiente paso es asociar este método con una ruta HTTP.

Enrutadores

Los enrutadores son abstracciones de rutas de solicitud y sirven como el puente entre solicitudes y respuestas para el servicio. Los enrutadores gestionan las rutas del servicio, las funciones y los errores.

Un enrutador puede tener varias rutas, y cada ruta puede realizar diferentes operaciones en esa ruta, como ejecutar una función, servir una página o proporcionar un recurso desde el servidor.

Creemos nuestro primer enrutador y asociemos nuestro método `IndexPage` con la ruta de índice.

```
1 Router mainRouter = new Router();
2
3 // SetRoute asociará todas las rutas de indice con nuestro método.
4 mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage);
```

Ahora nuestro enrutador puede recibir solicitudes y enviar respuestas. Sin embargo, `mainRouter` no está vinculado a un host o un servidor, por lo que no funcionará por sí solo. El siguiente paso es crear nuestro `ListeningHost`.

Hosts y puertos de escucha

Un `ListeningHost` puede hospedar un enrutador y varios puertos de escucha para el mismo enrutador. Un `ListeningPort` es un prefijo donde el servidor HTTP escuchará.

Aquí, podemos crear un `ListeningHost` que apunte a dos puntos finales para nuestro enrutador:

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

Ahora nuestro servidor HTTP escuchará en los puntos finales especificados y redirigirá sus solicitudes a nuestro enrutador.

Configuración del servidor

La configuración del servidor es responsable de la mayoría del comportamiento del servidor HTTP en sí. En esta configuración, podemos asociar `ListeningHosts` con nuestro servidor.

```
1  HttpServerConfiguration config = new HttpServerConfiguration();  
2  config.ListeningHosts.Add(myHost); // Agregue nuestro ListeningHost a esta configuración  
   del servidor
```

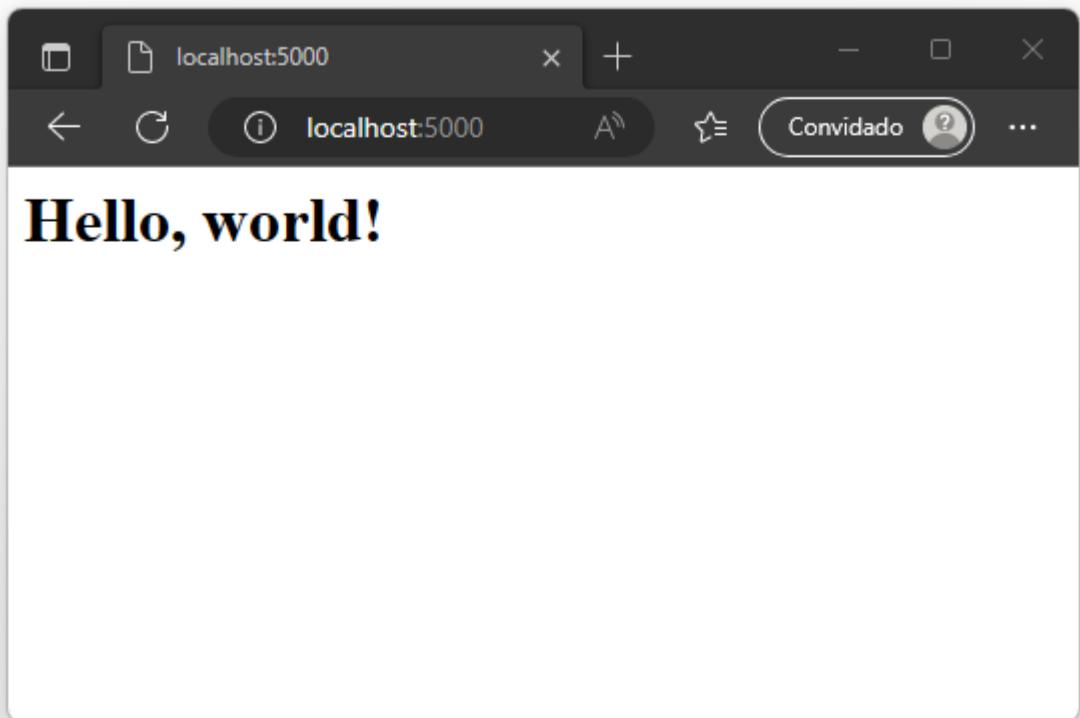
A continuación, podemos crear nuestro servidor HTTP:

```
1  HttpServer server = new HttpServer(config);  
2  server.Start(); // Inicia el servidor  
3  Console.ReadKey(); // Evita que la aplicación se cierre
```

Ahora podemos compilar nuestro ejecutable y ejecutar nuestro servidor HTTP con el comando:

```
dotnet watch
```

En tiempo de ejecución, abra su navegador y navegue hasta la ruta del servidor, y debería ver:



Ciclo de vida de la solicitud

A continuación se explica todo el ciclo de vida de una solicitud a través de un ejemplo de una solicitud HTTP.

- **Recepción de la solicitud:** cada solicitud crea un contexto HTTP entre la solicitud en sí y la respuesta que se entregará al cliente. Este contexto proviene del listener integrado en Sisk, que puede ser el [HttpListener](#), [Kestrel](#), o [Cadente](#).
 - Validación de solicitud externa: se valida la [HttpServerConfiguration.RemoteRequestsAction](#) para la solicitud.
 - Si la solicitud es externa y la propiedad es `Drop`, la conexión se cierra sin una respuesta al cliente con un `HttpServerExecutionStatus = RemoteRequestDropped`.
 - Configuración del resolutor de reenvío: si un [ForwardingResolver](#) está configurado, se llama al método [OnResolveRequestHost](#) en el host original de la solicitud.
 - Coincidencia de DNS: con el host resuelto y con más de un [ListeningHost](#) configurado, el servidor busca el host correspondiente para la solicitud.
 - Si no hay un ListeningHost que coincida, se devuelve una respuesta 400 Bad Request al cliente y un estado `HttpServerExecutionStatus = DnsUnknownHost` se devuelve al contexto HTTP.
 - Si un ListeningHost coincide, pero su [Router](#) no está inicializado, se devuelve una respuesta 503 Service Unavailable al cliente y un estado `HttpServerExecutionStatus = ListeningHostNotReady` se devuelve al contexto HTTP.
 - Enlace del router: el router del ListeningHost correspondiente se asocia con el servidor HTTP recibido.
 - Si el router ya está asociado con otro servidor HTTP, lo que no está permitido porque el router utiliza activamente los recursos de configuración del servidor, se lanza una excepción [InvalidOperationException](#). Esto solo ocurre durante la inicialización del servidor HTTP, no durante la creación del contexto HTTP.
 - Predefinición de encabezados:
 - Predefine el encabezado `X-Request-Id` en la respuesta si está configurado para hacerlo.
 - Predefine el encabezado `X-Powered-By` en la respuesta si está configurado para hacerlo.
 - Validación del tamaño del contenido: valida si el contenido de la solicitud es menor que [HttpServerConfiguration.MaximumContentLength](#) solo si es mayor que cero.
 - Si la solicitud envía un `Content-Length` mayor que el configurado, se devuelve una respuesta 413 Payload Too Large al cliente y un estado `HttpServerExecutionStatus = ContentTooLarge` se devuelve al contexto HTTP.
 - El evento `OnHttpRequestOpen` se invoca para todos los controladores de servidor HTTP configurados.
- **Enrutamiento de la acción:** el servidor invoca el router para la solicitud recibida.
 - Si el router no encuentra una ruta que coincide con la solicitud:

- Si la propiedad [Router.NotFoundErrorHandler](#) está configurada, se invoca la acción y la respuesta de la acción se reenvía al cliente HTTP.
- Si la propiedad anterior es nula, se devuelve una respuesta 404 Not Found por defecto al cliente.
- Si el router encuentra una ruta coincidente, pero el método de la ruta no coincide con el método de la solicitud:
 - Si la propiedad [Router.MethodNotAllowedErrorHandler](#) está configurada, se invoca la acción y la respuesta de la acción se reenvía al cliente HTTP.
 - Si la propiedad anterior es nula, se devuelve una respuesta 405 Method Not Allowed por defecto al cliente.
- Si la solicitud es del método `OPTIONS` :
 - El router devuelve una respuesta 200 Ok al cliente solo si no hay una ruta que coincida con el método de la solicitud (el método de la ruta no es explícitamente [RouteMethod.Options](#)).
- Si la propiedad [HttpServerConfiguration.ForceTrailingSlash](#) está habilitada, la ruta coincidente no es una expresión regular, la ruta de la solicitud no termina con `/` y el método de la solicitud es `GET` :
 - Se devuelve una respuesta HTTP 307 Temporary Redirect con el encabezado `Location` con la ruta y la consulta a la misma ubicación con un `/` al final al cliente.
- El evento `OnContextBagCreated` se invoca para todos los controladores de servidor HTTP configurados.
- Se ejecutan todas las instancias globales de [IRequestHandler](#) con la bandera `BeforeResponse` .
 - Si algún controlador devuelve una respuesta no nula, esa respuesta se reenvía al cliente HTTP y el contexto se cierra.
 - Si se lanza un error en este paso y [HttpServerConfiguration.ThrowExceptions](#) está deshabilitado:
 - Si la propiedad [Router.CallbackErrorHandler](#) está habilitada, se invoca y la respuesta resultante se devuelve al cliente.
 - Si la propiedad anterior no está definida, se devuelve una respuesta vacía al servidor, que reenvía una respuesta según el tipo de excepción lanzada, que generalmente es 500 Internal Server Error.
- Se ejecutan todas las instancias de [IRequestHandler](#) definidas en la ruta y con la bandera `BeforeResponse` .
 - Si algún controlador devuelve una respuesta no nula, esa respuesta se reenvía al cliente HTTP y el contexto se cierra.
 - Si se lanza un error en este paso y [HttpServerConfiguration.ThrowExceptions](#) está deshabilitado:
 - Si la propiedad [Router.CallbackErrorHandler](#) está habilitada, se invoca y la respuesta resultante se devuelve al cliente.
 - Si la propiedad anterior no está definida, se devuelve una respuesta vacía al servidor, que reenvía una respuesta según el tipo de excepción lanzada, que generalmente es 500 Internal Server Error.
- Se invoca la acción del router y se convierte en una respuesta HTTP.
 - Si se lanza un error en este paso y [HttpServerConfiguration.ThrowExceptions](#) está deshabilitado:

- Si la propiedad [Router.CallbackErrorHandler](#) está habilitada, se invoca y la respuesta resultante se devuelve al cliente.
 - Si la propiedad anterior no está definida, se devuelve una respuesta vacía al servidor, que reenvía una respuesta según el tipo de excepción lanzada, que generalmente es 500 Internal Server Error.
- Se ejecutan todas las instancias globales de [IRequestHandler](#) con la bandera `AfterResponse`.
 - Si algún controlador devuelve una respuesta no nula, la respuesta del controlador reemplaza la respuesta anterior y se reenvía inmediatamente al cliente HTTP.
 - Si se lanza un error en este paso y [HttpServerConfiguration.ThrowExceptions](#) está deshabilitado:
 - Si la propiedad [Router.CallbackErrorHandler](#) está habilitada, se invoca y la respuesta resultante se devuelve al cliente.
 - Si la propiedad anterior no está definida, se devuelve una respuesta vacía al servidor, que reenvía una respuesta según el tipo de excepción lanzada, que generalmente es 500 Internal Server Error.
- Se ejecutan todas las instancias de [IRequestHandler](#) definidas en la ruta y con la bandera `AfterResponse`.
 - Si algún controlador devuelve una respuesta no nula, la respuesta del controlador reemplaza la respuesta anterior y se reenvía inmediatamente al cliente HTTP.
 - Si se lanza un error en este paso y [HttpServerConfiguration.ThrowExceptions](#) está deshabilitado:
 - Si la propiedad [Router.CallbackErrorHandler](#) está habilitada, se invoca y la respuesta resultante se devuelve al cliente.
 - Si la propiedad anterior no está definida, se devuelve una respuesta vacía al servidor, que reenvía una respuesta según el tipo de excepción lanzada, que generalmente es 500 Internal Server Error.
- **Procesamiento de la respuesta:** con la respuesta lista, el servidor la prepara para enviarla al cliente.
 - Se definen los encabezados de la política de recursos compartidos de origen cruzado (CORS) en la respuesta según lo configurado en el [ListeningHost.CrossOriginResourceSharingPolicy](#) actual.
 - Se envían el código de estado y los encabezados de la respuesta al cliente.
 - Se envía el contenido de la respuesta al cliente:
 - Si el contenido de la respuesta es un descendiente de [ByteArrayContent](#), los bytes de la respuesta se copian directamente en el flujo de salida de la respuesta.
 - Si la condición anterior no se cumple, la respuesta se serializa en un flujo y se copia en el flujo de salida de la respuesta.
 - Se cierran los flujos y se descarta el contenido de la respuesta.
 - Si [HttpServerConfiguration.DisposeDisposableContextValues](#) está habilitado, se descartan todos los objetos definidos en el contexto de la solicitud que heredan de [IDisposable](#).
 - El evento `OnHttpRequestClose` se invoca para todos los controladores de servidor HTTP configurados.
 - Si se lanzó una excepción en el servidor, el evento `OnException` se invoca para todos los controladores de servidor HTTP configurados.

- Si la ruta permite el registro de acceso y `HttpServerConfiguration.AccessLogsStream` no es nulo, se escribe una línea de registro en el flujo de registro.
- Si la ruta permite el registro de errores, hay una excepción y `HttpServerConfiguration.ErrorsLogsStream` no es nulo, se escribe una línea de registro en el flujo de registro de errores.
- Si el servidor está esperando una solicitud a través de `HttpServer.WaitNext`, se libera el mutex y el contexto se vuelve disponible para el usuario.

Resolutores de Reenvío

Un Resolutor de Reenvío es un ayudante que ayuda a decodificar la información que identifica al cliente a través de una solicitud, proxy, CDN o balanceadores de carga. Cuando su servicio Sisk se ejecuta a través de un proxy inverso o directo, la dirección IP del cliente, el host y el protocolo pueden ser diferentes de la solicitud original, ya que es un reenvío de un servicio a otro. Esta funcionalidad de Sisk le permite controlar y resolver esta información antes de trabajar con la solicitud. Estos proxies suelen proporcionar encabezados útiles para identificar a su cliente.

Actualmente, con la clase [ForwardingResolver](#), es posible resolver la dirección IP del cliente, el host y el protocolo HTTP utilizado. Después de la versión 1.0 de Sisk, el servidor ya no tiene una implementación estándar para decodificar estos encabezados por razones de seguridad que varían de servicio a servicio.

Por ejemplo, el encabezado `X-Forwarded-For` incluye información sobre las direcciones IP que reenviaron la solicitud. Este encabezado es utilizado por los proxies para transportar una cadena de información al servicio final e incluye la dirección IP de todos los proxies utilizados, incluyendo la dirección real del cliente. El problema es: a veces es difícil identificar la dirección IP remota del cliente y no hay una regla específica para identificar este encabezado. Se recomienda encarecidamente leer la documentación de los encabezados que se van a implementar a continuación:

- Lea sobre el encabezado `X-Forwarded-For` [aquí](#).
- Lea sobre el encabezado `X-Forwarded-Host` [aquí](#).
- Lea sobre el encabezado `X-Forwarded-Proto` [aquí](#).

La clase ForwardingResolver

Esta clase tiene tres métodos virtuales que permiten la implementación más adecuada para cada servicio. Cada método es responsable de resolver la información de la solicitud a través de un proxy: la dirección IP del cliente, el host de la solicitud y el protocolo de seguridad utilizado. Por defecto, Sisk siempre utilizará la información de la solicitud original, sin resolver ningún encabezado.

El ejemplo a continuación muestra cómo se puede utilizar esta implementación. Este ejemplo resuelve la dirección IP del cliente a través del encabezado `X-Forwarded-For` y lanza un error cuando se reenvían más de una dirección IP en la solicitud.

IMPORTANT

No utilice este ejemplo en código de producción. Siempre verifique si la implementación es adecuada para su uso. Lea la documentación del encabezado antes de implementarlo.

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          using var host = HttpServer.CreateBuilder()
6              .UseForwardingResolver<Resolver>()
7              .UseListeningPort(5555)
8              .Build();
9
10         host.Router.SetRoute(RouteMethod.Any, Route.AnyPath, request =>
11             new HttpResponseMessage("Hello, world!!!"));
12
13         host.Start();
14     }
15
16     class Resolver : ForwardingResolver
17     {
18         public override IPAddress OnResolveClientAddress(HttpContext request,
19 IPEndPoint connectingEndpoint)
20         {
21             string? forwardedFor = request.Headers.XForwardedFor;
22             if (forwardedFor is null)
23             {
24                 throw new Exception("El encabezado X-Forwarded-For está ausente.");
25             }
26             string[] ipAddresses = forwardedFor.Split(',');
27             if (ipAddresses.Length != 1)
28             {
29                 throw new Exception("Demasiadas direcciones en el encabezado X-
30 Forwarded-For.");
31             }
32
33             return IPAddress.Parse(ipAddresses[0]);
34         }
35     }
36 }
```

Controladores de servidor HTTP

En la versión 0.16 de Sisk, se ha introducido la clase `HttpServerHandler`, que tiene como objetivo ampliar el comportamiento general de Sisk y proporcionar controladores de eventos adicionales a Sisk, como el manejo de solicitudes HTTP, enrutadores, bolsas de contexto y más.

La clase concentra los eventos que ocurren durante la vida útil de todo el servidor HTTP y también de una solicitud. El protocolo HTTP no tiene sesiones, y por lo tanto no es posible conservar información de una solicitud a otra. Sisk proporciona por ahora una forma de implementar sesiones, contextos, conexiones de base de datos y otros proveedores útiles para ayudar en su trabajo.

Consulte [esta página](#) para leer dónde se desencadena cada evento y cuál es su propósito. También puede ver el [ciclo de vida de una solicitud HTTP](#) para entender qué sucede con una solicitud y dónde se disparan los eventos. El servidor HTTP permite utilizar varios controladores al mismo tiempo. Cada llamada a un evento es síncrona, es decir, bloqueará el subproceso actual para cada solicitud o contexto hasta que todos los controladores asociados con esa función se ejecuten y completen.

A diferencia de los controladores de solicitudes, no se pueden aplicar a grupos de rutas o rutas específicas. En su lugar, se aplican a todo el servidor HTTP. Puede aplicar condiciones dentro de su controlador de servidor HTTP. Además, se definen singletones de cada `HttpServerHandler` para cada aplicación Sisk, por lo que solo se define una instancia por `HttpServerHandler`.

Un ejemplo práctico de uso de `HttpServerHandler` es para desechar automáticamente una conexión de base de datos al final de la solicitud.

```
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         // comprueba si la solicitud ha definido un DbContext
10        // en su bolsa de contexto
11        if (requestBag.IsSet<DbContext>())
12        {
13            var db = requestBag.Get<DbContext>();
14            db.Dispose();
15        }
16    }
17}
```

```

16     }
17 }
18
19 public static class DatabaseConnectionHandlerExtensions
20 {
21     // permite al usuario crear un contexto de base de datos a partir de una solicitud HTTP
22     // y almacenarlo en su bolsa de contexto
23     public static DbContext GetDbContext(this HttpRequest request)
24     {
25         var db = new DbContext();
26         return request.SetContextBag<DbContext>(db);
27     }
28 }

```

Con el código anterior, la extensión `GetDbContext` permite crear un contexto de conexión directamente desde el objeto `HttpRequest`. Una conexión no desechar puede causar problemas al ejecutar con la base de datos, por lo que se termina en `OnHttpRequestClose`.

Puede registrar un controlador en un servidor HTTP en su constructor o directamente con `HttpServer.RegisterHandler`.

```

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

```

Con esto, la clase `UserController` puede utilizar el contexto de base de datos como:

```

1 // UserController.cs
2
3 [RoutePrefix("/users")]
4 public class UserController : ApiController
5 {
6     [RouteGet()]
7     public async Task<HttpResponse> List(HttpContext request)

```

```

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

```

El código anterior utiliza métodos como `JsonOk` y `JsonMessage` que están integrados en `ApiController`, que hereda de `RouterController`:

```

1 // ApiController.cs
2
3 public class ApiController : RouterModule
4 {
5     public HttpResponse JsonOk(object value)
6     {
7         return new HttpResponse(200)
8             .WithContent(JsonContent.Create(value, null, new JsonSerializerOptions()
9                 {
10                     PropertyNameCaseInsensitive = true

```

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

Los desarrolladores pueden implementar sesiones, contextos y conexiones de base de datos utilizando esta clase. El código proporcionado muestra un ejemplo práctico con el `DatabaseConnectionHandler`, automatizando el descarte de la conexión de base de datos al final de cada solicitud.

La integración es sencilla, con controladores registrados durante la configuración del servidor. La clase `HttpServerHandler` ofrece un conjunto de herramientas poderosas para administrar recursos y ampliar el comportamiento de Sisk en aplicaciones HTTP.

Varios hosts de escucha por servidor

El Framework Sisk siempre ha soportado el uso de más de un host por servidor, es decir, un solo servidor HTTP puede escuchar en varios puertos y cada puerto tiene su propio enrutador y su propio servicio ejecutándose en él.

De esta manera, es fácil separar responsabilidades y gestionar servicios en un solo servidor HTTP con Sisk. El ejemplo a continuación muestra la creación de dos ListeningHosts, cada uno escuchando en un puerto diferente, con diferentes enrutadores y acciones.

Lea [creación manual de su aplicación](#) para entender los detalles sobre esta abstracción.

```
1  static void Main(string[] args)
2  {
3      // crea dos hosts de escucha, cada uno con su propio enrutador y
4      // escuchando en su propio puerto
5      //
6      ListeningHost hostA = new ListeningHost();
7      hostA.Ports = [new ListeningPort(12000)];
8      hostA.Router = new Router();
9      hostA.Router.SetRoute(RouteMethod.Get, "/", request => new
10     HttpResponseMessage().WithContent("Hola desde el host A!"));
11
12     ListeningHost hostB = new ListeningHost();
13     hostB.Ports = [new ListeningPort(12001)];
14     hostB.Router = new Router();
15     hostB.Router.SetRoute(RouteMethod.Get, "/", request => new
16     HttpResponseMessage().WithContent("Hola desde el host B!"));
17
18     // crea una configuración de servidor y agrega ambos
19     // hosts de escucha en ella
20     //
21     HttpServerConfiguration configuration = new HttpServerConfiguration();
22     configuration.ListeningHosts.Add(hostA);
23     configuration.ListeningHosts.Add(hostB);
24
25     // crea un servidor HTTP que utiliza la configuración
26     // especificada
27     //
28     HttpServer server = new HttpServer(configuration);
29
30     // inicia el servidor
```

```
31     server.Start();
32
33     Console.WriteLine("Intenta llegar al host A en {0}", server.ListeningPrefixes[0]);
34     Console.WriteLine("Intenta llegar al host B en {0}", server.ListeningPrefixes[1]);
35
36     Thread.Sleep(-1);
37 }
```