Getting started

Welcome to the Sisk documentation!

Finally, what is the Sisk Framework? It is an open-source lightweight library built with .NET, designed to be minimalist, flexible, and abstract. It allows developers to create internet services quickly, with little or no necessary configuration. Sisk makes it possible for your existing application to have a managed HTTP module, complete and disposable or complete.

Sisk's values include code transparency, modularity, performance, and scalability, and can handle various types of applications, such as Restful, JSON-RPC, Web-sockets, and more.

It's main features includes:

Resource	Description
Routing	A path router that supports prefixes, custom methods, path variables, value converters and more.
Request Handlers	Also known as *middlewares*, provides an interface to build your own request-handlers that work with the request before or after an action.
Compression	Compress your response contents easily with Sisk.
Web sockets	Provides routes that accept complete web-sockets, for reading and writing to the client.
Server-sent events	Provides the sending of server events to clients that support the SSE protocol.
Logging	Simplified logging. Log errors, access, define rotating logs by size, multiple output streams for the same log, and more.
Multi-host	Have an HTTP server for multiple ports, and each port with its own router, and each router with its own application.
Server handlers	Extend your own implementation of the HTTP server. Customize with extensions, improvements, and new features.

First steps

Sisk can run in any .NET environment. In this guide, we will teach you how to create a Sisk application using .NET. If you haven't installed it yet, please download the SDK from here.

In this tutorial, we will cover how to create a project structure, receive a request, obtain a URL parameter, and send a response. This guide will focus on building a simple server using C#. You can also use your favorite programming language.

> **NOTE**
>
> You may be interested in a quickstart project. Check this repository for more information.

Creating a Project

Let's name our project "My Sisk Application." Once you have .NET set up, you can create your project with the following command:

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

Next, navigate to your project directory and install Sisk using the .NET utility tool:

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

You can find additional ways to install Sisk in your project here.

Now, let's create an instance of our HTTP server. For this example, we will configure it to listen on port 5000.

Building the HTTP Server

Sisk allows you to build your application step by step manually, as it routes to the HttpServer object. However, this may not be very convenient for most projects. Therefore, we can use the builder method, which makes it easier to get our app up and running.

Program.cs C#

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

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

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

It's important to understand each vital component of Sisk. Later in this document, you will learn more about how Sisk works.

Manual (advanced) setup

You can learn how each Sisk mechanism works in this section of the documentation, which explains the behavior and relationships between the HttpServer, Router, ListeningPort, and other components.

Installing

You can install Sisk through Nuget, dotnet cli or another options. You can easily setup your Sisk environment by running this command in your developer console:

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

This command will install the latest version of Sisk in your project.

Native AOT Support

.NET Native AOT allows the publication of native .NET applications that are self-sufficient and do not require the .NET runtime installed on the target host. Additionally, Native AOT provides benefits such as:

- Much smaller applications
- Significantly faster initialization
- Lower memory consumption

Sisk Framework, by its explicit nature, allows the use of Native AOT for almost all it's features without requiring rework on the source code to adapt it to Native AOT.

Not supported features

However, Sisk does use reflection, albeit minimal, for some features. The features mentioned below may be partially available or entirely unavailable during native code execution:

- Auto-scanning of modules of the router: this resource scans the types embedded in the executing Assembly and registers the types that are router modules. This resource requires types that can be excluded during assembly trimming.

All other features are compatible with AOT in Sisk. It is common to find one or another method that gives an AOT warning, but the same, if not mentioned here, has an overload that indicates the passing of a type, parameter, or type information that assists the AOT compiler in compiling the object.

Deploying your Sisk Application

The process of deploying a Sisk application consists of publishing your project into production. Although the process is relatively simple, it is worth noting details that can be lethal to the security and stability of the deployment's infrastructure.

Ideally, you should be ready to deploy your application to the cloud, after carrying out all possible tests to have your application ready.

Publishing your app

Publishing your Sisk application or service is generating binaries ready and optimized for production. In this example, we will compile the binaries for production to run on a machine that has the .NET Runtime installed on the machine.

You will need .NET SDK installed in your machine in order to build your app, and .NET Runtime installed on the target server to run your app. You can learn how to install .NET Runtime in your Linux server here, Windows and Mac OS.

In the folder where your project is located, open a terminal and use the .NET publish command:

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

This will generate your binaries inside `bin/Release/publish/linux-x64`.

> **NOTE**
>
> If your app is running using Sisk.ServiceProvider package, you should copy your `service-config.json` into your host server along all binaries generated by `dotnet publish`. You can leave the file preconfigured, with environment variables, listening ports and hosts, and additional server configurations.

The next step is to take these files to the server where your application will be hosted.

After that, give execution permissions to your binary file. In this case, let's consider that our project name is "my-app":

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

After running your application, check to see if it produces any error messages. If it didn't produce, it's because your application is running.

At this point, it will probably not be possible to access your application by external net ouside your server, as access rules such as Firewall have not been configured. We will consider this in the next steps.

You should have the address of the virtual host where your application is listening to. This is set manually in the application, and depends on how you are instantiating your Sisk service.

If you're **not** using the Sisk.ServiceProvider package, you should find it where you defined your HttpServer instance:

```
HttpServer server = HttpServer.Emit(5000, out HttpServerConfiguration config, out var host,
out var router);
// sisk should listen on http://localhost:5000/
```

Associating an ListeningHost manually:

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

Or if you're using the Sisk.ServiceProvider package, in your service-config.json:

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

From this, we can create a reverse proxy to listen to your service and make the traffic available over the open network.

Proxying your application

Proxying your service means not directly exposing your Sisk service to an external network. This practice is very common for server deployments because:

- Allows you to associate an SSL certificate in your application;
- Create access rules before accessing the service and avoid overloads;
- Control bandwidth and request limits;
- Separate load-balancers for your application;
- Prevent security damage to failing infrastructure.

You can serve your application through a reverse proxy like Nginx or Apache, or you can use an http-over-dns tunnel like Cloudflared.

Also, remember to correctly resolve your proxy's forwarding headers to obtain your client's information, such as IP address and host, through forwarding resolvers.

The next step after creating your tunnel, firewall configuration and having your application running, is to create a service for your application.

> **NOTE**
>
> Using SSL certificates directly in the Sisk service on non-Windows systems is not possible. This is a point of the implementation of HttpListener, which is the central module for how HTTP queue management is done in Sisk, and this implementation varies from operating system to operating system. You can use SSL in your Sisk service if you associate a certificate with the virtual host with IIS. For other systems, using a reverse proxy is highly recommended.

Creating an service

Creating a service will make your application always available, even after restarting your server instance or a non-recoverable crash.

In this simple tutorial, we will use the content from the previous tutorial as a showcase to keep your service always active.

1. Access the folder where the service configuration files are located:

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

2. Create your `my-app.service` file and include the contents:

my-app.service — INI

```ini
[Unit]
Description=<description about your app>

[Service]
# set the user which will launch the service on
User=<user which will launch the service>

# the ExecStart path is not relative to WorkingDirectory.
# set it as the full path to the executeable file
WorkingDirectory=/home/htdocs
ExecStart=/home/htdocs/my-app

# set the service to always restart on crash
Restart=always
RestartSec=3

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

3. Restart your service manager module:

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

4. Start your new created service from the name of the file you set and check if they are running:

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

5. Now if your app is running ("Active: active"), enable your service to keep run after an system reboot:

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

Now you're ready to go and present your Sisk application to everyone.

Working with SSL

Working with SSL for development may be necessary when working in contexts that require security, such as most web development scenarios. Sisk operates on top of HttpListener, which does not support native HTTPS, only HTTP. However, there are workarounds that allow you to work with SSL in Sisk. See them below:

Through IIS on Windows

- Available on: Windows
- Effort: medium

If you are on Windows, you can use IIS to enable SSL on your HTTP server. For this to work, it is advisable that you follow this tutorial beforehand if you want your application to be listening on a host other than "localhost."

For this to work, you must install IIS through Windows features. IIS is available for free to Windows and Windows Server users. To configure SSL in your application, have the SSL certificate ready, even if it is self-signed. Next, you can see how to set up SSL on IIS 7 or higher.

Through mitmproxy

- Available on: Linux, macOS, Windows
- Effort: easy

mitmproxy is an interception proxy tool that allows developers and security testers to inspect, modify, and record HTTP and HTTPS traffic between a client (such as a web browser) and a server. You can use the **mitmdump** utility to start an reverse SSL proxy between your client and your Sisk application.

1. Firstly, install the mitmprxy in your machine.
2. Start your Sisk application. For this example, we'll use the 8000 port as the insecure HTTP port.
3. Start the mitmproxy server to listen the secure port at 8001:

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

And you're ready to go! You can already your application through `https://localhost:8001/`. Your application does not need to be running for you to start `mitmdump`.

Alternatively, you can add a reference to the mitmproxy helper in your project. This still requires that mitmproxy is installed on your computer.

Through Sisk.SslProxy package

- Available on: Linux, macOS, Windows
- Effort: easy

The Sisk.SslProxy package is a simple way to enable SSL on your Sisk application. However, it is an **extremely experimental** package. It may be unstable to work with this package, but you can be part of the small percentage of people who will contribute to making this package viable and stable. To get started, you can install the Sisk.SslProxy package with:

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

> **NOTE**
>
> You must enable "Enable pre-release packages" in the Visual Studio Package Manger to install Sisk.SslProxy.

Again, it is an experimental project, so don't even think about putting it into production.

At the moment, Sisk.SslProxy can handle most HTTP/1.1 features, including HTTP Continue, Chunked-Encoding, WebSockets, and SSE. Read more about SslProxy here.

Configuring namespace reservations on Windows

Sisk works with the HttpListener network interface, which binds a virtual host to the system to listen for requests.

On Windows, this binding is a bit restrictive, only allowing localhost to be bound as a valid host. When attempting to listen to another host, an access denied error is thrown on the server. This tutorial explains how to grant authorization to listen on any host you want on the system.

Namespace Setup.bat — BATCH

```batch
@echo off

:: insert prefix here, without spaces or quotes
SET PREFIX=

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

pause
```

Where in `PREFIX`, is the prefix ("Listening Host→Port") that your server will listen to. It must be formatted with the URL scheme, host, port and a slash at the end, example:

Namespace Setup.bat — BATCH

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

So that you can be listened in your application through:

Program.cs — C#

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

```
            .UseListeningPort("http://my-application.example.test/")
            .Build();

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

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

Changelogs

Every change made to Sisk is recorded through the changelog. You can view the changelogs for all Sisk versions here.

Frequently Asked Questions

Frequently asked questions about Sisk.

Is Sisk open-source?

Totally. All source code used by Sisk is published and frequently updated on GitHub.

Are contributions accepted?

As long as they are compatible with the Sisk philosophy, all contributions are very welcome! Contributions don't have to be just code! You can contribute with documentation, tests, translations, donations, and posts, for example.

Is Sisk funded?

No. No organization or project currently sponsors Sisk.

Can I use Sisk in production?

Absolutely. The project has been in development for more than three years and has had intense testing in commercial applications that have been in production since then. Sisk is used in important commercial projects

as main infrastructure.

A guide on how to deploy in different systems and environments has been written and is available.

Does Sisk have authentication, monitoring, and database services?

No. Sisk does not have any of these. It's a framework for developing HTTP web applications, but it's still a minimal framework that delivers what's needed for your application to work.

You can implement all the services you want using any third-party library you prefer. Sisk was made to be agnostic, flexible, and work with anything.

Why should I use Sisk instead of ?

I don't know. You tell me.

Sisk was created to fill a generic scenario for HTTP web applications in .NET. Established projects, such as ASP.NET, solve various problems, but with different biases. Unlike larger frameworks, Sisk requires the user to know what they're doing and building. Basic notions of web development and the HTTP protocol are essential for working with Sisk.

Sisk is closer to the Express of Node.js than ASP.NET Core. It's a high-level abstraction that allows you to create applications with HTTP logic that you want.

What do I need to learn Sisk?

You need the basics of:

- Web development (HTTP, Restful, etc.)
- .NET

That's it. Having a notion of what these two topics are, you can dedicate a few hours to developing an advanced application with Sisk.

Can I develop commercial applications with Sisk?

Absolutely.

Sisk was created under the MIT license, which means you can use Sisk in any commercial project, commercially or non-commercially, without the need for a proprietary license.

What we ask is that somewhere in your application, you have a notice of the open-source projects used in your project, and that Sisk is there.

Routing

The Router is the first step in building the server. It is responsible for housing Route objects, which are endpoints that map URLs and their methods to actions executed by the server. Each action is responsible for receiving a request and delivering a response to the client.

The routes are pairs of path expressions ("path pattern") and the HTTP method that they can listen to. When a request is made to the server, it will attempt to find a route that matches the received request, then it will call the action of that route and deliver the resulting response to the client.

There are multiple ways to define routes in Sisk: they can be static, dynamic or auto-scanned, defined by attributes, or directly in the Router object.

```
Router mainRouter = new Router();

// maps the GET / route into the following action
mainRouter.MapGet("/", request => {
    return new HttpResponse("Hello, world!");
});
```

To understand what a route is capable of doing, we need to understand what a request is capable of doing. An HttpRequest will contain everything you need. Sisk also includes some extra features that speed up the overral development.

For every action received by the server, a delegate of type RouteAction will be called. This delegate contains an parameter holding an HttpRequest with all the necessary information about the request received by the server. The resulting object from this delegate must be an HttpResponse or an object that maps to it through implicit response types.

Matching routes

When a request is received by the HTTP server, Sisk searches for a route that satisfies the expression of the path received by the request. The expression is always tested between the route and the request path, without considering the query string.

This test does not have priority and is exclusive to a single route. When no route is matched with that request, the Router.NotFoundErrorHandler response is returned to the client. When the path pattern is matched, but the HTTP method is mismatched, the Router.MethodNotAllowedErrorHandler response is sent back to the client.

Sisk checks for the possibility of route collisions to avoid these problems. When defining routes, Sisk will look for possible routes that might collide with the route being defined. This test includes checking the path and the method that the route is set to accept.

Creating routes using path patterns

You can define routes using various `SetRoute` methods.

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

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

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

    return new HttpResponse()
    {
        // the StreamContent inner
        // stream is disposed after sending
        // the response.
        Content = new StreamContent(imageStream)
    };
});

// multiple parameters
mainRouter.MapGet("/hey/<name>/surname/<surname>", (request) =>
{
    string name = request.RouteParameters["name"].GetString();
```

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

The RouteParameters property of HttpResponse contains all the information about the path variables of the received request.

Every path received by the server is normalized before the path pattern test is executed, following these rules:

- All empty segments are removed from the path, eg: `////foo//bar` becomes `/foo/bar`.
- Path matching is **case-sensitive**, unless Router.MatchRoutesIgnoreCase is set to `true`.

The Query and RouteParameters properties of HttpRequest return a StringValueCollection object, where each indexed property returns a non-null StringValue, which can be used as an option/monad to convert its raw value into a managed object.

The example below reads the route parameter "id" and obtains a `Guid` from it. If the parameter is not a valid Guid, an exception is thrown, and a 500 error is returned to the client if the server is not handling Router.CallbackErrorHandler.

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

> **NOTE**
>
> Paths have their trailing `/` ignored in both request and route path, that is, if you try to access a route defined as `/index/page` you'll be able to access using `/index/page/` too.
>
> You can also force URLs to terminate with `/` enabling the ForceTrailingSlash flag.

Creating routes using class instances

You can also define routes dynamically using reflection with the attribute RouteAttribute. This way, the instance of a class in which its methods implement this attribute will have their routes defined in the target router.

For a method to be defined as a route, it must be marked with a RouteAttribute, such as the attribute itself or a RouteGetAttribute. The method can be static, instance, public, or private. When the method `SetObject(type)` or `SetObject<TType>()` is used, instance methods are ignored.

Controller/MyController.cs C#

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

    // static methods works too
    [RouteGet("/hello")]
    static HttpResponse Hello(HttpRequest request)
    {
        HttpResponse res = new HttpResponse();
        res.Content = new StringContent("Hello world!");
        return res;
    }
}
```

The line below will define both the `Index` and `Hello` methods of `MyController` as routes, as both are marked as routes, and an instance of the class has been provided, not its type. If its type had been provided instead of an instance, only the static methods would be defined.

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

Since Sisk version 0.16, it is possible to enable AutoScan, which will search for user-defined classes that implement `RouterModule` and will automatically associate it with the router. This is not supported with AOT compilation.

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

The above instruction will search for all types which implements `ApiController` but not the type itself. The two optional parameters indicate how the method will search for these types. The first argument implies the Assembly where the types will be searched and the second indicates the way in which the types will be defined.

Regex routes

Instead of using the default HTTP path matching methods, you can mark a route to be interpreted with Regex.

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

Or with RegexRoute class:

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

You can also capture groups from the regex pattern into the HttpRequest.RouteParameters contents:

Controller/MyController.cs C#

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

Prefixing routes

You can prefix all routes in a class or module with the RoutePrefix attribute and set the prefix as a string.

See the example below using the BREAD architecture (Browse, Read, Edit, Add and Delete):

Controller/Api/UsersController.cs C#

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

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

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

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

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

In the above example, the HttpResponse parameter is omitted in favor of being used through the global context HttpContext.Current. Read more in the section that follows.

Routes without request parameter

Routes can be defined without the HttpRequest parameter and still be possible to obtain the request and its components in the request context. Let's consider an abstraction `ControllerBase` that serves as a foundation for all controllers of an API, and that abstraction provides the `Request` property to obtain the HttpRequest currently.

Controller/ControllerBase.cs C#

```
public abstract class ControllerBase
{
    // gets the request from the current thread
    public HttpRequest Request { get => HttpContext.Current.Request; }

    // the line below, when called, gets the database from the current HTTP session,
    // or creates a new one if it doesn't exist
    public DbContext Database { get => HttpContext.Current.RequestBag.GetOrAdd<DbContext>
(); }
}
```

And for all it's descendants to be able to use the route syntax without the request parameter:

Controller/UsersController.cs C#

```
[RoutePrefix("/api/users")]
public class UsersController : ControllerBase
{
    [RoutePost]
    public async Task<HttpResponse> Create()
    {
        // reads the JSON data from the current request
        UserCreationDto? user = JsonSerializer.DeserializeAsync<UserCreationDto>
(Request.Body);
         ...
        Database.Users.Add(user);

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

More details about the current context and dependency injection can be found in the dependency injection tutorial.

Any method routes

You can define a route to be matched only by its path and skip the HTTP method. This can be useful for you to do method validation inside the route callback.

```
1    // will match / on any HTTP method
2    mainRouter.SetRoute(RouteMethod.Any, "/", callbackFunction);
```

Any path routes

Any path routes test for any path received by the HTTP server, subject to the route method being tested. If the route method is RouteMethod.Any and the route uses Route.AnyPath in its path expression, this route will listen to all requests from the HTTP server, and no other routes can be defined.

```
1    // the following route will match all POST requests
2    mainRouter.SetRoute(RouteMethod.Post, Route.AnyPath, callbackFunction);
```

Ignore case route matching

By default, the interpretation of routes with requests are case-sensitive. To make it ignore case, enable this option:

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

This will also enable the option `RegexOptions.IgnoreCase` for routes where it's regex-matching.

Not Found (404) callback handler

You can create a custom callback for when a request doesn't match any known routes.

```
mainRouter.NotFoundErrorHandler = () ⇒
{
    return new HttpResponse(404)
    {
        // Since v0.14
        Content = new HtmlContent("<h1>Not found</h1>")
        // older versions
        Content = new StringContent("<h1>Not found</h1>", Encoding.UTF8, "text/html")
    };
};
```

Method not allowed (405) callback handler

You can also create a custom callback for when a request matches it's path, but doens't match the method.

```
mainRouter.MethodNotAllowedErrorHandler = (context) ⇒
{
    return new HttpResponse(405)
    {
        Content = new StringContent($"Method not allowed for this route.")
    };
};
```

Internal error handler

Route callbacks can throw errors during server execution. If not handled correctly, the overall functioning of the HTTP server can be terminated. The router has a callback for when a route callback fails and prevents service interruption.

This method is only reacheable when ThrowExceptions is set to false.

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

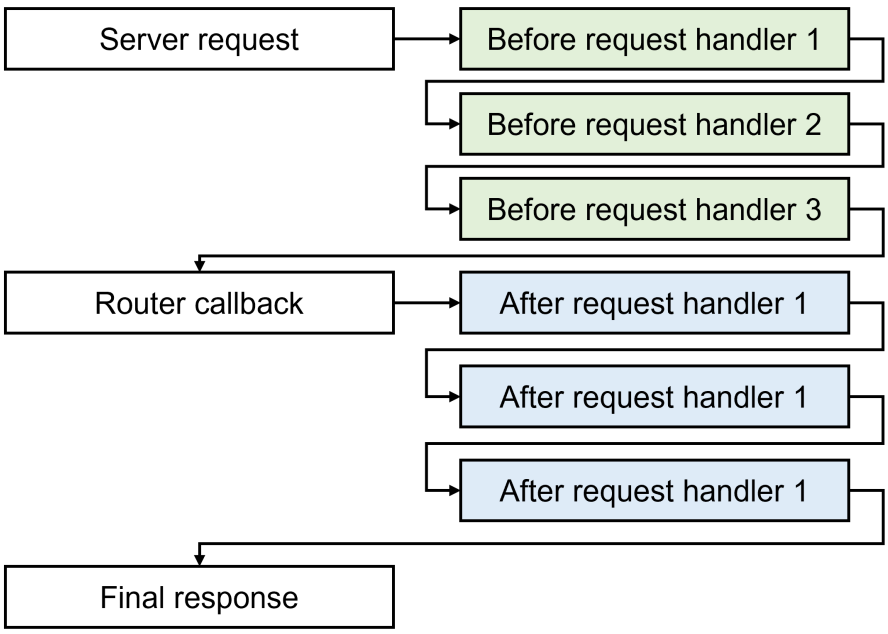
Request handling

Request handlers, also known as "middlewares", are functions that run before or after a request is executed on the router. They can be defined per route or per router.

There are two types of request handlers:

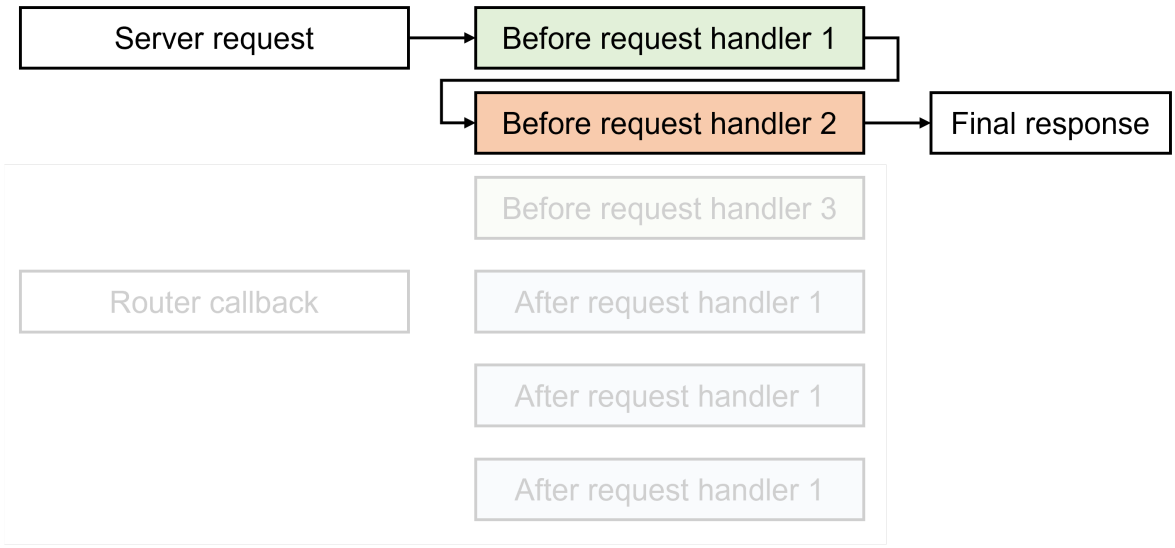
- **BeforeResponse**: defines that the request handler will be executed before calling the router action.
- **AfterResponse**: defines that the request handler will be executed after calling the router action. Sending an HTTP response in this context will overwrite the router's action response.

Both requests handlers can override the actual router callback function response. By the way, request handlers can be useful for validating a request, such as authentication, content, or any other information, such as storing information, logs, or other steps that can be performed before or after a response.



This way, a request handler can interrupt all this execution and return a response before finishing the cycle, discarding everything else in the process.

Example: let's assume that a user authentication request handler does not authenticate him. It will prevent the request lifecycle from being continued and will hang. If this happens in the request handler at position two, the third and onwards will not be evaluated.



Creating an request handler

To create a request handler, we can create a class that inherits the IRequestHandler interface, in this format:

Middleware/AuthenticateUserRequestHandler.cs C#

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

    public HttpResponse? Execute(HttpRequest request, HttpContext context)
    {
        if (request.Headers.Authorization == null)
        {
            // Returning null indicates that the request cycle can be continued
            return null;
        }
        else
        {
            // Returning an HttpResponse object indicates that this response will overwrite
adjacent responses.
```

```
            return new HttpResponse(System.Net.HttpStatusCode.Unauthorized);
        }
    }
}
```

In the above example, we indicated that if the `Authorization` header is present in the request, it should continue and the next request handler or the router callback should be called, whichever comes next. If it's a request handler is executed after the response by their property ExecutionMode and return an non-null value, it will overwrite the router's response.

Whenever a request handler returns `null`, it indicates that the request must continue and the next object must be called or the cycle must end with the router's response.

Associating a request handler with a single route

You can define one or more request handlers for a route.

Router.cs C#

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

Or creating an Route object:

Router.cs C#

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

Associating a request handler with a router

You can define a global request handler that will runned on all routes on a router.

Router.cs C#

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

Associating a request handler with an attribute

You can define a request handler on a method attribute along with a route attribute.

Controller/MyController.cs C#

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

Note that it is necessary to pass the desired request handler type and not an object instance. That way, the request handler will be instantiated by the router parser. You can pass arguments in the class constructor with the ConstructorArguments property.

Example:

Controller/MyController.cs C#

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

You can also create your own attribute that implements RequestHandler:

Middleware/Attributes/AuthenticateAttribute.cs C#

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

And use it as:

Controller/MyController.cs C#

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

Bypassing an global request handler

After defining a global request handler on a route, you can ignore this request handler on specific routes.

Router.cs C#

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

mainRouter.SetRoute(new Route(RouteMethod.Get, "/", "My route", IndexPage, null)
{
    BypassGlobalRequestHandlers = new IRequestHandler[]
    {
        myRequestHandler,                   // ok: the same instance of what is in the
global request handlers
        new AuthenticateUserRequestHandler() // wrong: will not skip the global
request handler
    }
});
```

> **NOTE**
>
> If you're bypassing a request handler you must use the same reference of what you instanced before to skip. Creating another request handler instance will not skip the global request handler since it's reference will change. Remember to use the same request handler reference used in both GlobalRequestHandlers and BypassGlobalRequestHandlers.

Requests

Requests are structures that represent an HTTP request message. The HttpRequest object contains useful functions for handling HTTP messages throughout your application.

An HTTP request is formed by the method, path, version, headers and body.

In this document, we will teach you how to obtain each of these elements.

Getting the request method

To obtain the method of the received request, you can use the Method property:

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

This property returns the request's method represented by an HttpMethod object.

> **NOTE**
>
> Unlike route methods, this property does not serves the RouteMethod.Any item. Instead, it returns the real request method.

Getting request url components

You can get various component from a URL through certain properties of a request. For this example, let's consider the URL:

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

Component name	Description	Component value
Path	Gets the request path.	`/user/login`
FullPath	Gets the request path and the query string.	`/user/login?email=foo@bar.com`
FullUrl	Gets the entire URL request string.	`http://localhost:5000/user/login?email=foo@bar.com`
Host	Gets the request host.	`localhost`
Authority	Gets the request host and port.	`localhost:5000`
QueryString	Gets the request query.	`?email=foo@bar.com`
Query	Gets the request query in a named value collection.	`{StringValueCollection object}`
IsSecure	Determines if the request is using SSL (true) or not (false).	`false`

You can also opt by using the HttpRequest.Uri property, which includes everything above in one object.

Getting the request body

Some requests include body such as forms, files, or API transactions. You can get the body of a request from the property:

```
1   // gets the request body as an string, using the request encoding as the encoder
2   string body = request.Body;
3
4   // or gets it in an byte array
5   byte[] bodyBytes = request.RawBody;
6
7   // or else, you can stream it.
8   Stream requestStream = request.GetRequestStream();
```

It is also possible to determine if there is a body in the request and if it is loaded with the properties HasContents, which determines if the request has contents and IsContentAvailable which indicates that the HTTP server fully received the content from the remote point.

It is not possible to read the request content through `GetRequestStream` more than once. If you read with this method, the values in `RawBody` and `Body` will also not be available. It's not necessary to dispose the request stream in the context of the request, as it is disposed at the end of the HTTP session in which it is created. Also, you can use HttpRequest.RequestEncoding property to get the best encoding to decode the request manually.

The server has limits for reading the request content, which applies to both HttpRequest.Body and HttpRequest.RawBody. These properties copies the entire input stream to an local buffer of the same size of HttpRequest.ContentLength.

A response with status 413 Content Too Large is returned to the client if the content sent is larger than HttpServerConfiguration.MaximumContentLength defined in the user configuration. Additionally, if there is no configured limit or if it is too large, the server will throw an OutOfMemoryException when the content sent by the client exceeds Int32.MaxValue (2 GB) and if the content is attempted to be accessed through one of the properties mentioned above. You can still deal with the content through streaming.

> **NOTE**
>
> Although Sisk allows it, it is always a good idea to follow HTTP Semantics to create your application and not obtain or serve content in methods that do not allow it. Read about RFC 9110 "HTTP Semantics".

Getting the request context

The HTTP Context is an exclusive Sisk object that stores HTTP server, route, router and request handler information. You can use it to be able to organize yourself in an environment where these objects are difficult to organize.

The RequestBag object contains stored information that is passed from an request handler to another point, and can be consumed at the final destination. This object can also be used by request handlers that run after the route callback.

> **TIP**
>
> This property is also acessible by HttpRequest.Bag property.

Middleware/AuthenticateUserRequestHandler.cs C#

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

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

The above request handler will define `AuthenticatedUser` in the request bag, and can be consumed later in the final callback:

Controller/MyController.cs C#

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

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

You can also use the `Bag.Set()` and `Bag.Get()` helper methods to get or set objects by their type singletons.

Middleware/Authenticate.cs C#

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

Controller/MyController.cs C#

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

Getting form data

You can get the values of a form data in an NameValueCollection with the example below:

Controller/Auth.cs C#

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

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

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

Getting multipart form data

Sisk's HTTP request lets you get uploaded multipart contents, such as a files, form fields, or any binary content.

Controller/Auth.cs C#

```cs
[RoutePost("/upload-contents")]
public HttpResponse Index(HttpRequest request)
{
    // the following method reads the entire request input into an
    // array of MultipartObjects
    var multipartFormDataObjects = request.GetMultipartFormContent();

    foreach (MultipartObject uploadedObject in multipartFormDataObjects)
    {
        // The name of the file provided by Multipart form data.
        // Null is returned if the object is not a file.
        Console.WriteLine("File name       : " + uploadedObject.Filename);

        // The multipart form data object field name.
        Console.WriteLine("Field name      : " + uploadedObject.Name);

        // The multipart form data content length.
        Console.WriteLine("Content length  : " + uploadedObject.ContentLength);

        // Determine the image format based in the file header for each
        // known content type. If the content ins't an recognized common file
        // format, this method below will return MultipartObjectCommonFormat.Unknown
        Console.WriteLine("Common format   : " + uploadedObject.GetCommonFileFormat());
    }
}
```

You can read more about Sisk Multipart form objects and it's methods, properties and functionalities.

Detecting client disconnection

Since version v1.15 of Sisk, the framework provides a CancellationToken that is thrown when the connection between the client and the server is prematurely closed before receiving the response. This token can be useful

for detecting when the client no longer wants the response and canceling long-running operations.

```
router.MapGet("/connect", async (HttpRequest req) =>
{
    // gets the disconnection token from the request
    var dc = req.DisconnectToken;

    await LongOperationAsync(dc);

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

This token is not compatible with all HTTP engines, and each requires an implementation.

Server-sent events support

Sisk supports Server-sent events, which allows sending chunks as an stream and keeping the connection between the server and the client alive.

Calling the HttpRequest.GetEventSource method will put the HttpRequest in it's listener state. From this, the context of this HTTP request will not expect an HttpResponse as it will overlap the packets sent by server side events.

After sending all packets, the callback must return the Close method, which will send the final response to the server and indicate that the streaming has ended.

It's not possible to predict what the total length of all packets that will be sent, so it is not possible to determine the end of the connection with `Content-Length` header.

By most browsers defaults, server-side events does not support sending HTTP headers or methods other than the GET method. Therefore, be careful when using request handlers with event-source requests that require specific headers in the request, as it probably they ins't going to have them.

Also, most browsers restart streams if the EventSource.close method ins't called on the client side after receiving all the packets, causing infinite additional processing on the server side. To avoid this kind of problem, it's common to send an final packet indicating that the event source has finished sending all packets.

The example below shows how the browser can communicate with the server that supports Server-side events.

sse-example.html HTML

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

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

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

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

And progressively send the messages to the client:

Controller/MyController.cs C#

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

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

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

        return serverEvents.Close();
```

```
17        }
18    }
```

When running this code, we expect a result similar to this:

Resolving proxied IPs and hosts

Sisk can be used with proxies, and therefore IP addresses can be replaced by the proxy endpoint in the transaction from a client to the proxy.

You can define your own resolvers in Sisk with forwarding resolvers.

Headers encoding

Header encoding can be a problem for some implementations. On Windows, UTF-8 headers are not supported, so ASCII is used. Sisk has a built-in encoding converter, which can be useful for decoding incorrectly encoded headers.

This operation is costly and disabled by default, but can be enabled under the NormalizeHeadersEncodings flag.

Responses

Responses represent objects that are HTTP responses to HTTP requests. They are sent by the server to the client as an indication of the request for a resource, page, document, file or other object.

An HTTP response is formed up of status, headers and content.

In this document, we will teach you how to architect HTTP responses with Sisk.

Setting an HTTP status

The HTTP status list is the same since HTTP/1.0, and Sisk supports all of them.

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

Or with Fluent Syntax:

```
new HttpResponse()
    .WithStatus(200) // or
    .WithStatus(HttpStatusCode.Ok) // or
    .WithStatus(HttpStatusInformation.Ok);
```

You can see the full list of available HttpStatusCode here. You can also provide your own status code by using the HttpStatusInformation structure.

Body and content-type

Sisk supports native .NET content objects to send body in responses. You can use the StringContent class to send a JSON response for example:

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

The server will always attempt to calculate the `Content-Length` from what you have defined in the content if you haven't explicitly defined it in a header. If the server cannot implicitly obtain the Content-Length header from the response content, the response will be sent with Chunked-Encoding.

You can also stream the response by sending a StreamContent or using the method GetResponseStream.

Response headers

You can add, edit or remove headers you're sending in the response. The example below shows how to send an redirect response to the client.

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

Or with Fluent Syntax:

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

When you use the Add method of HttpHeaderCollection, you are adding a header to the request without altering the ones already sent. The Set method replaces the headers with the same name with the instructed value. The indexer of HttpHeaderCollection internally calls the Set method to replace the headers.

Sending cookies

Sisk has methods that facilitate the definition of cookies in the client. Cookies set by this method are already URL encoded and fit the RFC-6265 standard.

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

Or with Fluent Syntax:

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

There are other more complete versions of the same method.

Chunked responses

You can set the transfer encoding to chunked to send large responses.

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

When using chunked-encoding, the Content-Length header is automatically omitted.

Response stream

Response streams are an managed way that allow you to send responses in a segmented way. It's a lower level operation than using HttpResponse objects, as they require you to send the headers and content manually, and then close the connection.

This example opens an read-only stream for the file, copies the stream to the response output stream and doens't loads the entire file in the memory. This can be useful to serving medium or big files.

```
// gets the response output stream
using var fileStream = File.OpenRead("my-big-file.zip");
var responseStream = request.GetResponseStream();

```

```
    // sets the response encoding to use chunked-encoding
    // also you shouldn't send content-length header when using
    // chunked encoding
    responseStream.SendChunked = true;
    responseStream.SetStatus(200);
    responseStream.SetHeader(HttpKnownHeaderNames.ContentType, contentType);

    // copies the file stream to the response output stream
    fileStream.CopyTo(responseStream.ResponseStream);

    // closes the stream
    return responseStream.Close();
```

GZip, Deflate and Brotli compression

You can send responses with compressed content in Sisk with compressing HTTP contents. Firstly, encapsulate your HttpContent object within one of the compressors below to send the compressed response to the client.

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

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

You can also use these compressed contents with streams.

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

        // do not apply "using" here. the HttpServer will discard your content
        // after sending the response.
        var archive = File.OpenRead("/path/to/big-file.zip");

        return new HttpResponse () {
            Content = new GZipContent(archive)


```

```
        }
    });
```

The Content-Encoding headers are automatically set when using these contents.

Automatic compression

It is possible to automatically compress HTTP responses with the EnableAutomaticResponseCompression property. This property automatically encapsulates the response content from the router in a compressible content that is accepted by the request, provided the response is not inherited from a CompressedContent.

Only one compressible content is chosen for a request, chosen according to the Accept-Encoding header, which follows the order:

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

If the request specifies that it accepts any of these compression methods, the response will be automatically compressed.

Implicit response types

You can use other return types besides HttpResponse, but it is necessary to configure the router how it will handle each type of object.

The concept is to always return a reference type and turn it into a valid HttpResponse object. Routes that return HttpResponse do not undergo any conversion.

Value types (structures) cannot be used as a return type because they are not compatible with the RouterCallback, so they must be wrapped in a ValueResult to be able to be used in handlers.

Consider the following example from a router module not using HttpResponse in the return type:

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

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

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

        return dUser;
    }

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

        return true;
    }
}
```

With that, now it is necessary to define in the router how it will deal with each type of object. Objects are always the first argument of the handler and the output type must be a valid HttpResponse. Also, the output objects of a route should never be null.

For ValueResult types it is not necessary to indicate that the input object is a ValueResult and only T, since ValueResult is an object reflected from its original component.

The association of types does not compare what was registered with the type of the object returned from the router callback. Instead, it checks whether the type of the router result is assignable to the registered type.

Registering a handler of type Object will fallback to all previously unvalidated types. The inserting order of the value handlers also matters, so registering an Object handler will ignore all other type-specific handlers. Always register specific value handlers first to ensure order.

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

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

// registering an value handler of object must be the last
// value handler which will be used as an fallback
r.RegisterValueHandler<object>(fallback =>
{
    return new HttpResponse() {
        Status = HttpStatusCode.OK,
        Content = JsonContent.Create(fallback)
    };
});
```

Note on enumerable objects and arrays

Implicit response objects that implement IEnumerable are read into memory through the `ToArray()` method before being converted through a defined value handler. For this to occur, the `IEnumerable` object is converted to an array of objects, and the response converter will always receive an `Object[]` instead of the original type.

Consider the following scenario:

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

In the above example, the `IEnumerable<string>` converter **will never be called**, because the input object will always be an `Object[]` and it is not convertible to an `IEnumerable<string>`. However, the converter below that receives an `IEnumerable<object>` will receive its input, since its value is compatible.

If you need to actually handle the type of the object that will be enumerated, you will need to use reflection to get the type of the collection element. All enumerable objects (lists, arrays, and collections) are converted to an array of objects by the HTTP response converter.

Values that implements IAsyncEnumerable are handled automatically by the server if the ConvertIAsyncEnumerableIntoEnumerable property is enabled, similar to what happens with `IEnumerable`. An asynchronous enumeration is converted to a blocking enumerator, and then converted to a synchronous array of objects.

Logging

You can configure Sisk to write access and error logs automatically. It is possible to define log rotation, extensions and frequency.

The LogStream class provides an asynchronous way of writing logs and keeping them in an awaitable write queue.

In this article we will show you how to configure logging for your application.

File based access logs

Logs to files open the file, write the line text, and then close the file for every line written. This procedure was adopted to maintain write responsiveness in the logs.

Program.cs C#

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

         ...

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

The above code will write all incoming requests to the `logs/access.log` file. Note that, the file is created automatically if it does not exist, however the folder before it does not. It's not necessary to create the `logs/` directory as the LogStream class automatically creates it.

Stream based logging

You can write log files to TextWriter objects instances, such as `Console.Out`, by passing an TextWriter object in the constructor:

Program.cs C#

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

For every message written in the stream-based log, the `TextWriter.Flush()` method is called.

Access log formatting

You can customize the access log format by predefined variables. Consider the following line:

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

It will write an message like:

```
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]
```

You can format your log file by the format described by the table:

Value	What it represents	Example
%dd	Day of the month (formatted as two digits)	05
%dmmm	Full name of the month	July
%dmm	Abbreviated name of the month (three letters)	Jul

Value	What it represents	Example
%dm	Month number (formatted as two digits)	07
%dy	Year (formatted as four digits)	2023
%th	Hour in 12-hour format	03
%tH	Hour in 24-hour format (HH)	15
%ti	Minutes (formatted as two digits)	30
%ts	Seconds (formatted as two digits)	45
%tm	Milliseconds (formatted as three digits)	123
%tz	Time zone offset (total hours in UTC)	+03:00
%ri	Client's remote IP address	192.168.1.100
%rm	HTTP method (uppercase)	GET
%rs	URI scheme (http/https)	https
%ra	URI authority (domain)	example.com
%rh	Host of the request	www.example.com
%rp	Port of the request	443
%rz	Path of the request	/path/to/resource
%rq	Query string	?key=value&another=123
%sc	HTTP response status code	200
%sd	HTTP response status description	OK
%lin	Human-readable size of the request	1.2 KB
%linr	Raw size of the request (bytes)	1234
%lou	Human-readable size of the response	2.5 KB
%lour	Raw size of the response (bytes)	2560
%lms	Elapsed time in milliseconds	120
%ls	Execution status	Executed

Value	What it represents	Example
%{header-name}	Represents the `header-name` header of the request.	`Mozilla/5.0 (platform; rv:gecko [...]`
%{:res-name}	Represents the `res-name` header of the response.	

Rotating logs

You can configure the HTTP server to rotate the log files to a compressed .gz file when they reach a certain size. The size is checked periodically by the limiar you define.

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

The above code will check every six hours if the LogStream's file has reached it's 64MB limit. If so, the file is compressed to an .gz file and it then `access.log` is cleaned.

During this process, writing to the file is locked until the file is compressed and cleaned. All lines that enter to be written in this period will be in a queue waiting for the end of compression.

This function only works with file-based LogStreams.

Error logging

When a server is not throwing errors to the debugger, it forwards the errors to log writing when there are any. You can configure error writing with:

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

This property will only write something to the log if the error is not captured by the callback or the Router.CallbackErrorHandler property.

The error written by the server always writes the date and time, the request headers (not the body), the error trace, and the inner exception trace, if theres any.

Other logging instances

Your application can have zero or multiple LogStreams, there is no limit on how many log channels it can have. Therefore, it is possible to direct your application's log to a file other than the default AccessLog or ErrorLog.

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

Extending LogStream

You can extend the `LogStream` class to write custom formats, compatible with the current Sisk log engine. The example below allows to write colorful messages into the Console through Spectre.Console library:

CustomLogStream.cs — C#

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

Another way to automatically write custom logs for each request/response is to create an HttpServerHandler. The example below is a little more complete. It writes the body of the request and response in JSON to the Console. It can be useful for debugging requests in general. This example makes use of ContextBag and HttpServerHandler.

Program.cs C#

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

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

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

JsonMessageHandler.cs C#

```
class JsonMessageHandler : HttpServerHandler
{
    protected override void OnHttpRequestOpen(HttpRequest request)
    {
        if (request.Method == HttpMethod.Get && request.Headers["Content-
Type"]?.Contains("json", StringComparison.InvariantCultureIgnoreCase) == true)
        {
            // At this point, the connection is open and the client has sent the
header specifying
            // that the content is JSON.The line below reads the content and leaves it
stored in the request.
            //
            // If the content is not read in the request action, the GC is likely to
collect the content
            // after sending the response to the client, so the content may not be
available after the response is closed.
```

```
            //
            _ = request.RawBody;

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

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

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

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

            if (content is HttpContent httpContent &&
                // check if the response is JSON
                httpContent.Headers.ContentType?.MediaType?.Contains("json",
StringComparison.InvariantCultureIgnoreCase) == true)
            {
                string json = await httpContent.ReadAsStringAsync();
                responseJson = JsonValue.Deserialize(json, new JsonOptions() {
WriteIndented = true }).ToString();
            }
        }
        else
        {
            // gets the internal server handling status
            responseMessage = result.Status.ToString();
        }

        StringBuilder outputMessage = new StringBuilder();

        if (requestJson != null)
        {
```

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

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

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

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

Server Sent Events

Sisk supports sending messages through Server Sent Events out of the box. You can create disposable and persistent connections, get the connections during runtime and use them.

This feature has some limitations imposed by browsers, such as sending only texts messages and not being able to permanently close a connection. A server-side closed connection will have a client periodically trying to reconnect every 5 seconds (3 for some browsers).

These connections are useful for sending events from the server to the client without having the client request the information every time.

Creating an SSE connection

A SSE connection works like a regular HTTP request, but instead of sending a response and immediately closing the connection, the connection is kept open to send messages.

Calling the HttpRequest.GetEventSource() method, the request is put in a waiting state while the SSE instance is created.

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

    sse.Send("Hello, world!");

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

In the above code, we create an SSE connection and send a "Hello, world" message, then we close the SSE connection from the server side.

> **NOTE**
>
> When closing a server-side connection, by default the client will try to connect again at that end and the connection will be restarted, executing the method again, forever.
>
> It's common to forward a termination message from the server whenever the connection is closed from the server to prevent the client from trying to reconnect again.

Appending headers

If you need to send headers, you can use the HttpRequestEventSource.AppendHeader method before sending any messages.

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

Note that it is necessary to send the headers before sending any messages.

Wait-For-Fail connections

Connections are normally terminated when the server is no longer able to send messages due to an possible client-side disconnection. With that, the connection is automatically terminated and the instance of the class is discarded.

Even with a reconnection, the instance of the class will not work, as it is linked to the previous connection. In some situations, you may need this connection later and you don't want to manage it via the callback method of the route.

For this, we can identify the SSE connections with an identifier and get them using it later, even outside the callback of the route. In addition, we mark the connection with WaitForFail so as not to terminate the route and terminate the connection automatically.

An SSE connection in KeepAlive will wait for a send error (caused by disconnection) to resume method execution. It is also possible to set a Timeout for this. After the time, if no message has been sent, the connection is terminated and execution resumes.

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

    sse.WaitForFail(TimeSpan.FromSeconds(15)); // wait for 15 seconds without any message
before terminating the connection

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

The above method will create the connection, handle it and wait for a disconnection or error.

```
HttpRequestEventSource? evs = server.EventSources.GetByIdentifier("my-index-connection");
if (evs =/= null)
{
    // the connection is still alive
    evs.Send("Hello again!");
}
```

And the snippet above will try to look for the newly created connection, and if it exists, it will send a message to it.

All active server connections that are identified will be available in the collection HttpServer.EventSources. This collection only stores active and identified connections. Closed connections are removed from the collection.

> **NOTE**
>
> It is important to note that keep alive has a limit established by components that may be connected to Sisk in an uncontrollable way, such as an web proxy, an HTTP kernel or a network driver, and they close idle connections after a certain period of time.
>
> Therefore, it is important to keep the connection open by sending periodic pings or extending the maximum time before the connection is closed. Read the next section to better understand sending periodic pings.

Setup connections ping policy

Ping Policy is an automated way of sending periodic messages to your client. This function allows the server to understand when the client has disconnected from that connection without having to keep the connection open indefinitely.

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

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

In the code above, every 5 seconds, a new ping message will be sent to the client. This will keep the TCP connection alive and prevent it from being closed due to inactivity. Also, when a message fails to be sent, the connection is automatically closed, freeing up the resources used by the connection.

Querying connections

You can search for active connections using a predicate on the connection identifier, to be able to broadcast, for example.

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

You can also use the All method to get all active SSE connections.

Web Sockets

Sisk supports web sockets as well, such as receiving and sending messages to their client.

This feature works fine in most browsers, but in Sisk it is still experimental. Please, if you find any bugs, report it on github.

Accepting messages asynchronously

WebSocket messages are received in order, queued until processed by `ReceiveMessageAsync`. This method returns no message when the timeout is reached, when the operation is canceled, or when the client is disconnected.

Only one read and write operation can occur simultaneously, therefore, while you are waiting for a message with `ReceiveMessageAsync`, it is not possible to write to the connected client.

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

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

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

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

Accepting messages synchronously

The example below contains a way for you to use a synchronous websocket, without an asynchronous context, where you receive the messages, deal with them, and finish using the socket.

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

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

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

    string name = msg.GetString();

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

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

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

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

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

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

Ping Policy

Similar to how ping policy in Server Side Events works, you can also configure a ping policy to keep the TCP connection open if there is inactivity in it.

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

Discard syntax

The HTTP server can be used to listen for a callback request from an action, such as OAuth authentication, and can be discarded after receiving that request. This can be useful in cases where you need a background action but do not want to set up an entire HTTP application for it.

The following example show us how to create an listening HTTP server at port 5555 with CreateListener and wait the next context:

```
using (var server = HttpServer.CreateListener(5555))
{
    // wait for the next http request
    var context = await server.WaitNextAsync();
    Console.WriteLine($"Requested path: {context.Request.Path}");
}
```

The WaitNext function waits for the next context of a completed request processing. Once the result of this operation is obtained, the server has already fully handled the request and sent the response to the client.

Dependency injection

It is common to dedicate members and instances that last for the lifetime of a request, such as a database connection, an authenticated user, or a session token. One of the possibilities is through the HttpContext.RequestBag, which creates a dictionary that lasts for the entire lifetime of a request.

This dictionary can be accessed by request handlers and define variables throughout that request. For example, a request handler that authenticates a user sets this user within the `HttpContext.RequestBag`, and within the request logic, this user can be retrieved with `HttpContext.RequestBag.Get<User>()`.

Here's an example:

RequestHandlers/AuthenticateUser.cs — C#

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

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

Controllers/HelloController.cs — C#

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

This is a preliminary example of this operation. The instance of `User` was created within the request handler dedicated to authentication, and all routes that use this request handler will have the guarantee that there will be a `User` in their instance of `HttpContext.RequestBag`.

It is possible to define logic to obtain instances when not previously defined in the `RequestBag` through methods like GetOrAdd or GetOrAddAsync.

Since version 1.3, the static property HttpContext.Current was introduced, allowing access to the currently executing `HttpContext` of the request context. This enables exposing members of the `HttpContext` outside the current request and defining instances in route objects.

The example below defines a controller that has members commonly accessed by the context of a request.

Controllers/Controller.cs C#

```
public abstract class Controller : RouterModule
{
    public DbContext Database
    {
        get
        {
            // create an DbContext or get the existing one
            return HttpContext.Current.RequestBag.GetOrAdd(() => new DbContext());
        }
    }

    // the following line will throw if the property is accessed when the User is not
    // defined in the request bag
    public User AuthenticatedUser { get => HttpContext.Current.RequestBag.Get<User>(); }

    // Exposing the HttpRequest instance is supported too
    public HttpRequest Request { get => HttpContext.Current.Request; }
}
```

And define types that inherit from the controller:

Controllers/PostsController.cs C#

```
[RoutePrefix("/api/posts")]
public class PostsController : Controller
{
    [RouteGet]
    public IEnumerable<Blog> ListPosts()
    {
```

```
            return Database.Posts
                .Where(post => post.AuthorId == AuthenticatedUser.Id)
                .ToList();
        }

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

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

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

For the example above, you will need to configure a value handler in your router so that the objects returned by the router are transformed into a valid HttpResponse.

Note that the methods do not have an `HttpRequest request` argument as present in other methods. This is because, since version 1.3, the router supports two types of delegates for routing responses: RouteAction, which is the default delegate that receives an `HttpRequest` argument, and ParameterlessRouteAction. The `HttpRequest` object can still be accessed by both delegates through the Request property of the static `HttpContext` on the thread.

In the example above, we defined a disposable object, the `DbContext`, and we need to ensure that all instances created in a `DbContext` are disposed of when the HTTP session ends. For this, we can use two ways to achieve this. One is to create a request handler that is executed after the router's action, and the other way is through a custom server handler.

For the first method, we can create the request handler inline directly in the OnSetup method inherited from `RouterModule`:

Controllers/PostsController.cs **C#**

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

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

```

```
        HasRequestHandler(RequestHandler.Create(
            execute: (req, ctx) =>
            {
                // get one DbContext defined in the request handler context and
                // dispose it
                ctx.RequestBag.GetOrDefault<DbContext>()?.Dispose();
                return null;
            },
            executionMode: RequestHandlerExecutionMode.AfterResponse));
    }
}
```

> **TIP**
>
> Since Sisk version 1.4, the property HttpServerConfiguration.DisposeDisposableContextValues is introduced and enabled by default, which defines whether the HTTP server should dispose all `IDisposable` values in the context bag when an HTTP session is closed.

The method above will ensure that the `DbContext` is disposed of when the HTTP session is finalized. You can do this for more members that need to be disposed of at the end of a response.

For the second method, you can create a custom server handler that will dispose of the `DbContext` when the HTTP session is finalized.

Server/Handlers/ObjectDisposerHandler.cs — C#

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

And use it in your app builder:

Program.cs — C#

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

This is a way to handle code cleanup and keep the dependencies of a request separated by the type of module that will be used, reducing the amount of duplicated code within each action of a router. It is a practice similar to what dependency injection is used for in frameworks like ASP.NET.

Streaming Content

The Sisk supports reading and sending streams of content to and from the client. This feature is useful for removing memory overhead for serializing and deserializing content during the lifetime of a request.

Request content stream

Small contents are automatically loaded into the HTTP connection buffer memory, quickly loading this content to HttpRequest.Body and HttpRequest.RawBody. For larger contents, the HttpRequest.GetRequestStream method can be used to obtain the request content read stream.

It is worth noting that the HttpRequest.GetMultipartFormContent method reads the entire request content into memory, so it may not be useful for reading large contents.

Consider the following example:

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           // request does not have content
8           return new HttpResponse ( HttpStatusInformation.BadRequest );
9       }
10
11      var contentStream = request.GetRequestStream ();
12      var outputFileName = Path.Combine (
13          AppDomain.CurrentDomain.BaseDirectory,
14          "uploads",
15          fileName );
16
17      using (var fs = File.Create ( outputFileName )) {
18          await contentStream.CopyToAsync ( fs );
19      }
```

```
20
21      return new HttpResponse () {
22          Content = JsonContent.Create ( new { message = "File sent successfully." } )
23      };
24  }
```

In the example above, the `UploadDocument` method reads the request content and saves the content to a file. No additional memory allocation is made except for the read buffer used by `Stream.CopyToAsync`. The example above removes the pressure of memory allocation for a very large file, which can optimize application performance.

A good practice is to always use a CancellationToken in an operation that can be time-consuming, such as sending files, as it depends on the network speed between the client and the server.

The adjustment with a CancellationToken can be made in the following way:

Controller/UploadDocument.cs **C#**

```
1   // the cancellation token below will throw an exception if the 30-second timeout
2   is reached.
3   CancellationTokenSource copyCancellation = new CancellationTokenSource ( delay:
4   TimeSpan.FromSeconds ( 30 ) );
5
6   try {
7       using (var fs = File.Create ( outputFileName )) {
8           await contentStream.CopyToAsync ( fs, copyCancellation.Token );
9       }
10  }
11  catch (OperationCanceledException) {
12      return new HttpResponse ( HttpStatusInformation.BadRequest ) {
13          Content = JsonContent.Create ( new { Error = "The upload exceeded the maximum
    upload time (30 seconds)." } )
        };
    }
```

Response content stream

Sending response content is also possible. Currently, there are two ways to do this: through the HttpRequest.GetResponseStream method and using a content of type StreamContent.

Consider a scenario where we need to serve an image file. To do this, we can use the following code:

Controller/ImageController.cs C#

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

    // example method to obtain a profile picture
    var profilePictureFilename = "profile-picture.jpg";
    byte[] profilePicture = await File.ReadAllBytesAsync ( profilePictureFilename );

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

The method above makes a memory allocation every time it reads the image content. If the image is large, this can cause a performance problem, and in peak situations, even a memory overload and crash the server. In these situations, caching can be useful, but it will not eliminate the problem, since memory will still be reserved for that file. Caching will alleviate the pressure of having to allocate memory for every request, but for large files, it will not be enough.

Sending the image through a stream can be a solution to the problem. Instead of reading the entire image content, a read stream is created on the file and copied to the client using a tiny buffer.

Sending through the GetResponseStream method

The HttpRequest.GetResponseStream method creates an object that allows sending chunks of the HTTP response as the content flow is prepared. This method is more manual, requiring you to define the status, headers, and content size before sending the content.

Controller/ImageController.cs C#

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

    var profilePictureFilename = "profile-picture.jpg";

    // in this form of sending, the status and header must be defined
```

```
        // before the content is sent
        var requestStreamManager = request.GetResponseStream ();

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

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

            // in this form of sending, it is also necessary to define the content size
            // before sending it.
            requestStreamManager.SetContentLength ( fs.Length );

            // if you don't know the content size, you can use chunked-encoding
            // to send the content
            requestStreamManager.SendChunked = true;

            // and then, write to the output stream
            await fs.CopyToAsync ( requestStreamManager.ResponseStream );
        }
    }
```

Sending content through a StreamContent

The StreamContent class allows sending content from a data source as a byte stream. This form of sending is easier, removing the previous requirements, and even allowing the use of compression encoding to reduce the content size.

Controller/ImageController.cs — C#

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

    var profilePictureFilename = "profile-picture.jpg";

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

> **IMPORTANT**
>
> In this type of content, do not encapsulate the stream in a `using` block. The content will be automatically discarded by the HTTP server when the content flow is finalized, with or without errors.

Enabling CORS (Cross-Origin Resource Sharing) in Sisk

Sisk has a tool that can be useful for handling Cross-Origin Resource Sharing (CORS) when exposing your service publicly. This feature is not part of the HTTP protocol but a specific feature of web browsers defined by the W3C. This security mechanism prevents a web page from making requests to a different domain than the one that provided the web page. A service provider can allow certain domains to access its resources, or just one.

Same Origin

For a resource to be identified as "same origin", a request must identify the Origin header in its request:

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

And the remote server must respond with an Access-Control-Allow-Origin header with the same value as the requested origin:

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

This verification is **explicit**: the host, port, and protocol must be the same as requested. Check the example:

- A server responds that its `Access-Control-Allow-Origin` is `https://example.com`:
  - `https://example.net` - the domain is different.
  - `http://example.com` - the scheme is different.
  - `http://example.com:5555` - the port is different.
  - `https://www.example.com` - the host is different.

In the specification, only the syntax is allowed for both headers, whether for requests and responses. The URL path is ignored. The port is also omitted if it is a default port (80 for HTTP and 443 for HTTPS).

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

Enabling CORS

Natively, you have the CrossOriginResourceSharingHeaders object within your ListeningHost.

You can configure CORS when initializing the server:

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

    await app.StartAsync();
}
```

The code above will send the following headers for **all responses**:

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

These headers need to be sent for all responses to a web client, including errors and redirects.

You may notice that the CrossOriginResourceSharingHeaders class has two similar properties: AllowOrigin and AllowOrigins. Note that one is plural, while the other is singular.

- The **AllowOrigin** property is static: only the origin you specify will be sent for all responses.

- The **AllowOrigins** property is dynamic: the server checks if the request's origin is contained in this list. If it is found, it is sent for the response of that origin.

Wildcards and automatic headers

Alternatively, you can use a wildcard (`*`) in the response's origin to specify that any origin is allowed to access the resource. However, this value is not allowed for requests that have credentials (authorization headers) and this operation will result in an error.

You can work around this problem by explicitly listing which origins will be allowed through the AllowOrigins property or also use the AutoAllowOrigin constant in the value of AllowOrigin. This magic property will define the `Access-Control-Allow-Origin` header for the same value as the `Origin` header of the request.

You can also use AutoFromRequestMethod and AutoFromRequestHeaders for behavior similar to `AllowOrigin`, which automatically responds based on the headers sent.

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

        // Responds based on the request's Origin header
        allowOrigin: CrossOriginResourceSharingHeaders.AutoAllowOrigin,

        // Responds based on the Access-Control-Request-Method header or the request method
        allowMethods: [CrossOriginResourceSharingHeaders.AutoFromRequestMethod],

        // Responds based on the Access-Control-Request-Headers header or the sent headers
        allowHeaders: [CrossOriginResourceSharingHeaders.AutoFromRequestHeaders]))
```

Other Ways to Apply CORS

If you are dealing with service providers, you can override values defined in the configuration file:

```
static async Task Main(string[] args)
{
    using var app = HttpServer.CreateBuilder()
        .UsePortableConfiguration( ... )
        .UseCors(cors => {
            // Will override the origin defined in the configuration
```

```
            // file.
            cors.AllowOrigin = "http://example.com";
        })
        .Build();

    await app.StartAsync();
}
```

Disabling CORS on Specific Routes

The `UseCors` property is available for both routes and all route attributes and can be disabled with the following example:

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

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

Replacing Values in the Response

You can replace or remove values explicitly in a router action:

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

    public IEnumerable<string> GetWidgets(HttpRequest request) {

        // Removes the Access-Control-Allow-Credentials header
        request.Context.OverrideHeaders.AccessControlAllowCredentials = string.Empty;

```

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

Preflight Requests

A preflight request is an OPTIONS method request that the client sends before the actual request.

The Sisk server will always respond to the request with a `200 OK` and the applicable CORS headers, and then the client can proceed with the actual request. This condition is only not applied when a route exists for the request with the RouteMethod explicitly configured for `Options`.

Disabling CORS Globally

It is not possible to do this. To not use CORS, do not configure it.

JSON-RPC Extension

Sisk has an experimental module for a JSON-RPC 2.0 API, which allows you to create even simpler applications. This extension strictly implements the JSON-RPC 2.0 transport interface and offers transport via HTTP GET, POST requests, and also web-sockets with Sisk.

You can install the extension via Nuget with the command below. Note that, in experimental/beta versions, you should enable the option to search for pre-release packages in Visual Studio.

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

Transport Interface

JSON-RPC is a stateless, asynchronous remote procedure execution (RDP) protocol that uses JSON for unilateral data communication. A JSON-RPC request is typically identified by an ID, and a response is delivered by the same ID that was sent in the request. Not all requests require a response, which are called "notifications".

The JSON-RPC 2.0 specification explains in detail how the transport works. This transport is agnostic of where it will be used. Sisk implements this protocol through HTTP, following the conformities of JSON-RPC over HTTP, which partially supports GET requests, but completely supports POST requests. Web-sockets are also supported, providing asynchronous message communication.

A JSON-RPC request looks similar to:

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

And a successful response looks similar to:

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

JSON-RPC Methods

The following example shows how to create a JSON-RPC API using Sisk. A mathematical operations class performs the remote operations and delivers the serialized response to the client.

Program.cs C#

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

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

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

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

await app.StartAsync();
```

MathOperations.cs C#

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

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

The above example will map the `Sum` and `Sqrt` methods to the JSON-RPC handler, and these methods will be available at `GET /service`, `POST /service` and `GET /ws`. Method names are case-insensitive.

Method parameters are automatically deserialized to their specific types. Using a request with named parameters is also supported. JSON serialization is done by the LightJson library. When a type is not correctly deserialized, you can create a specific JSON converter for that type and associate it with your JsonSerializerOptions later.

You can also get the `$.params` raw object from the JSON-RPC request directly in your method.

MathOperations.cs **C#**

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

For this to occur, `@params` must be the **only** parameter in your method, with exactly the name `params` (in C#, the `@` is necessary to escape this parameter name).

Parameter deserialization occurs for both named objects or positional arrays. For example, the following method can be called remotely by both requests:

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

```
        ...
    }
```

For an array, the order of the parameters must be followed.

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

}
```

Customizing the serializer

You can customize the JSON serializer in the JsonRpcHandler.JsonSerializerOptions property. In this property, you can enable the use of JSON5 for deserializing messages. Although not a conformity with JSON-RPC 2.0, JSON5 is an extension of JSON that allows for more human-readable and legible writing.

Program.cs C#

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

        // uses a sanitized name comparer. this comparer compares only letters
        // and digits in a name, and discards other symbols. ex:
        // foo_bar10 == FooBar10
        e.Handler.JsonSerializerOptions.PropertyNameComparer = new
JsonSanitizedComparer ();

        // enables JSON5 for the JSON interpreter. even activating this, plain JSON is
```

```
still allowed
        e.Handler.JsonSerializerOptions.SerializationFlags =
LightJson.Serialization.JsonSerializationFlags.Json5;

        // maps the POST /service route to the JSON RPC handler
        e.Router.MapPost ( "/service", e.Handler.Transport.HttpPost );
    } )
    .Build ();

host.Start ();
```

SSL Proxy

> **WARNING**
>
> This feature is experimental and should not be used in production. Please refer to this document if you want to make Sisk work with SSL.

The Sisk SSL Proxy is a module that provides an HTTPS connection for a ListeningHost in Sisk and routes HTTPS messages to an insecure HTTP context. The module was built to provide SSL connection for a service that uses HttpListener to run, which does not support SSL.

The proxy runs within the same application and listens for HTTP/1.1 messages, forwarding them in the same protocol to Sisk. Currently, this feature is highly experimental and may be unstable enough to not be used in production.

At present, the SslProxy supports almost all HTTP/1.1 features, such as keep-alive, chunked encoding, websockets, etc. For an open connection to the SSL proxy, a TCP connection is created to the target server, and the proxy is forwarded to the established connection.

The SslProxy can be used with HttpServer.CreateBuilder as follows:

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

app.Start();
```

You must provide a valid SSL certificate for the proxy. To ensure that the certificate is accepted by browsers, remember to import it into the operating system so that it functions correctly.

Basic Auth

The Basic Auth package adds a request handler capable of handling basic authentication scheme in your Sisk application with very little configuration and effort. Basic HTTP authentication is a minimal input form of authenticating requests by an user id and password, where the session is controlled exclusively by the client and there are no authentication or access tokens.

Read more about the Basic authentication scheme in the MDN specification.

Installing

To get started, install the Sisk.BasicAuth package in your project:

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

You can view more ways to install it in your project in the Nuget repository.

Creating your auth handler

You can control the authentication scheme for an entire module or for individual routes. For that, let's first write our first basic authentication handler.

In the example below, a connection is made to the database, it checks if the user exists and if the password is valid, and after that, stores the user in the context bag.

```
1   public class UserAuthHandler : BasicAuthenticateRequestHandler
2   {
3       public UserAuthHandler() : base()
4       {
5           Realm = "To enter this page, please, inform your credentials.";
```

```
    }

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

        // in this case, we're using the email as the user id field, so we're
        // going to search for an user using their email.
        User? user = db.Users.FirstOrDefault(u => u.Email == credentials.UserId);
        if (user == null)
        {
            return base.CreateUnauthorizedResponse("Sorry! No user was found by
this email.");
        }

        // validates that the credentials password is valid for this user.
        if (!user.ValidatePassword(credentials.Password))
        {
            return base.CreateUnauthorizedResponse("Invalid credentials.");
        }

        // adds the logged user to the http context
        // and continues the execution
        context.Bag.Add("loggedUser", user);
        return null;
    }
}
```

So, just associate this request handler with our route or class.

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

Or using RouterModule class:

```
public class UsersController : RouterModule
{
    public ClientModule()
    {
        // now all routes inside this class will be handled by
        // UserAuthHandler.
        base.HasRequestHandler(new UserAuthHandler());
    }

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

Remarks

The primary responsibility of basic authentication is carried out on the client-side. Storage, cache control, and encryption are all handled locally on the client. The server only receives the credentials and validates whether access is allowed or not.

Note that this method is not one of the most secure because it places a significant responsibility on the client, which can be difficult to trace and maintain the security of its credentials. Additionally, it is crucial for passwords to be transmitted in a secure connection context (SSL), as they do not have any inherent encryption. A brief interception in the headers of a request can expose the access credentials of your user.

Opt for more robust authentication solutions for applications in production and avoid using too many off-the-shelf components, as they may not adapt to the needs of your project and end up exposing it to security risks.

Service Providers

Service Providers is a way to port your Sisk application to different environments with a portable configuration file. This feature allows you to change the server's port, parameters, and other options without having to modify the application code for each environment. This module depends on the Sisk construction syntax and can be configured through the UsePortableConfiguration method.

A configuration provider is implemented with IConfigurationProvider, which provides a configuration reader and can receive any implementation. By default, Sisk provides a JSON configuration reader, but there is also a package for INI files. You can also create your own configuration provider and register it with:

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

As mentioned earlier, the default provider is a JSON file. By default, the file name searched for is service-config.json, and it is searched in the current directory of the running process, not the executable directory.

You can choose to change the file name, as well as where Sisk should look for the configuration file, with:

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

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

The code above will look for the config.toml file in the current directory of the running process. If not found, it will then search in the directory where the executable is located. If the file does not exist, the createIfDontExists

parameter is honored, creating the file, without any content, in the last tested path (based on lookupDirectories), and an error is thrown in the console, preventing the application from initializing.

> **TIP**
>
> You can look at the source code of the INI configuration reader and the JSON configuration reader to understand how an IConfigurationProvider is implemented.

Reading configurations from a JSON file

By default, Sisk provides a configuration provider that reads configurations from a JSON file. This file follows a fixed structure and is composed of the following parameters:

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

The parameters created from a configuration file can be accessed in the server constructor:

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

Each configuration reader provides a way to read the server initialization parameters. Some properties are indicated to be in the process environment instead of being defined in the configuration file, such as sensitive API data, API keys, etc.

Configuration file structure

The JSON configuration file is composed of the following properties:

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

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

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

INI configuration provider

Sisk has a method for obtaining startup configurations other than JSON. In fact, any pipeline that implements IConfigurationReader can be used with PortableConfigurationBuilder.WithConfigurationPipeline, reading the server configuration from any file type.

The Sisk.IniConfiguration package provides a stream-based INI file reader that does not throw exceptions for common syntax errors and has a simple configuration syntax. This package can be used outside the Sisk framework, offering flexibility for projects that require an efficient INI document reader.

Installing

To install the package, you can start with:

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

You can also install the core package, which doens't includes the INI IConfigurationReader, neither the Sisk dependency, just the INI serializers:

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

With the main package, you can use it in your code as shown in the example below:

```
1   class Program
2   {
3       static HttpServerHostContext Host = null!;
4
5       static void Main(string[] args)
6       {
7           Host = HttpServer.CreateBuilder()
8               .UsePortableConfiguration(config =>
9               {
10                  config.WithConfigFile("app.ini", createIfDontExists: true);
11
12                  // uses the IniConfigurationReader configuration reader
```

```
                config.WithConfigurationPipeline<IniConfigurationReader>();
            })
            .UseRouter(r =>
            {
                r.MapGet("/", SayHello);
            })
            .Build();

        Host.Start();
    }

    static HttpResponse SayHello(HttpRequest request)
    {
        string? name = Host.Parameters["name"] ?? "world";
        return new HttpResponse($"Hello, {name}!");
    }
}
```

The code above will look for an app.ini file in the process's current directory (CurrentDirectory). The INI file looks like this:

```
[Server]
# Multiple listen addresses are supported
Listen = http://localhost:5552/
Listen = http://localhost:5553/
ThrowExceptions = false
AccessLogsStream = console

[Cors]
AllowMethods = GET, POST
AllowHeaders = Content-Type, Authorization
AllowOrigin = *

[Parameters]
Name = "Kanye West"
```

INI flavor and syntax

Current implementation flavor:

- Properties and section names are **case-insensitive**.

- Properties names and values are **trimmed**, unless values are quoted.
- Values can be quoted with single or double quotes. Quotes can have line-breaks inside them.
- Comments are supported with `#` and `;`. Also, **trailing comments are allowed**.
- Properties can have multiple values.

In detail, the documentation for the "flavor" of the INI parser used in Sisk is available in this document.

Using the following ini code as example:

```
One = 1
Value = this is an value
Another value = "this value
    has an line break on it"

; the code below has some colors
[some section]
Color = Red
Color = Blue
Color = Yellow ; do not use yellow
```

Parse it with:

```
// parse the ini text from the string
IniDocument doc = IniDocument.FromString(iniText);

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

// get multiple values
string[]? colors = doc.GetSection("some section")?.GetMany("color");
```

Configuration parameters

Section and name	Allow multiple values	Description
`Server.Listen`	Yes	The server listening addresses/ports.

Section and name	Allow multiple values	Description
`Server.Encoding`	No	The server default encoding.
`Server.MaximumContentLength`	No	The server max content-length size in bytes.
`Server.IncludeRequestIdHeader`	No	Specifies if the HTTP server should send the X-Request-Id header.
`Server.ThrowExceptions`	No	Specifies if unhandled exceptions should be thrown.
`Server.AccessLogsStream`	No	Specifies the access log output stream.
`Server.ErrorsLogsStream`	No	Specifies the error log output stream.
`Cors.AllowMethods`	No	Specifies the CORS Allow-Methods header value.
`Cors.AllowHeaders`	No	Specifies the CORS Allow-Headers header value.
`Cors.AllowOrigins`	No	Specifies multiples Allow-Origin headers, separated by commas. AllowOrigins for more information.
`Cors.AllowOrigin`	No	Specifies one Allow-Origin header.
`Cors.ExposeHeaders`	No	Specifies the CORS Expose-Headers header value.
`Cors.AllowCredentials`	No	Specifies the CORS Allow-Credentials header value.
`Cors.MaxAge`	No	Specifies the CORS Max-Age header value.

Manual (advanced) setup

In this section, we will create our HTTP server without any predefined standards, in a completely abstract way. Here, you can manually build how your HTTP server will function. Each ListeningHost has a router, and an HTTP server can have multiple ListeningHosts, each pointing to a different host on a different port.

First, we need to understand the request/response concept. It is quite simple: for every request, there must be a response. Sisk follows this principle as well. Let's create a method that responds with a "Hello, World!" message in HTML, specifying the status code and headers.

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

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

    return indexResponse;
}
```

The next step is to associate this method with an HTTP route.

Routers

Routers are abstractions of request routes and serve as the bridge between requests and responses for the service. Routers manage service routes, functions, and errors.

A router can have several routes, and each route can perform different operations on that path, such as executing a function, serving a page, or providing a resource from the server.

Let's create our first router and associate our `IndexPage` method with the index path.

```
Router mainRouter = new Router();

// SetRoute will associate all index routes with our method.
mainRouter.SetRoute(RouteMethod.Get, "/", IndexPage);
```

Now our router can receive requests and send responses. However, `mainRouter` is not tied to a host or a server, so it will not work on its own. The next step is to create our ListeningHost.

Listening Hosts and Ports

A ListeningHost can host a router and multiple listening ports for the same router. A ListeningPort is a prefix where the HTTP server will listen.

Here, we can create a `ListeningHost` that points to two endpoints for our router:

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

Now our HTTP server will listen to the specified endpoints and redirect its requests to our router.

Server Configuration

Server configuration is responsible for most of the behavior of the HTTP server itself. In this configuration, we can associate `ListeningHosts` with our server.

```
1    HttpServerConfiguration config = new HttpServerConfiguration();
2    config.ListeningHosts.Add(myHost); // Add our ListeningHost to this server configuration
```

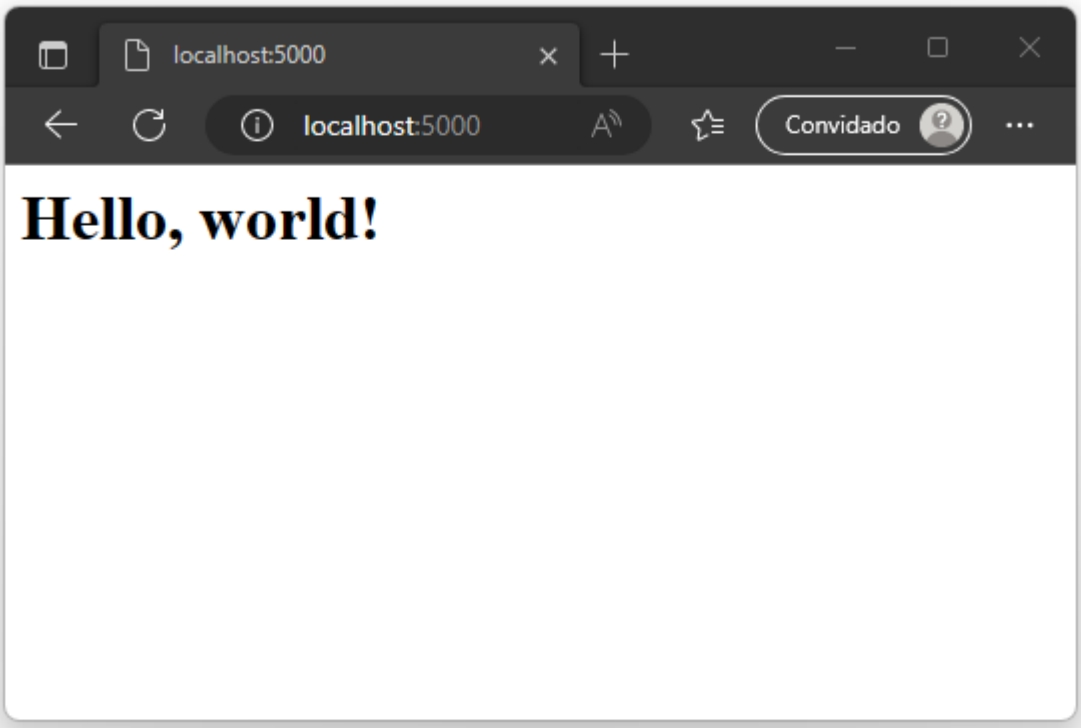
Next, we can create our HTTP server:

```
1    HttpServer server = new HttpServer(config);
2    server.Start();    // Starts the server
3    Console.ReadKey(); // Prevents the application from exiting
```

Now we can compile our executable and run our HTTP server with the command:

```
dotnet watch
```

At runtime, open your browser and navigate to the server path, and you should see:



Request lifecycle

Below is explained the entire life cycle of a request through an example of an HTTP request.

- **Receiving the request:** each request creates an HTTP context between the request itself and the response that will be delivered to the client. This context comes from the built-in listener in Sisk, which can be the HttpListener, Kestrel, or Cadente.
  - External request validation: the validation of HttpServerConfiguration.RemoteRequestsAction is validated for the request.
    - If the request is external and the property is `Drop`, the connection is closed without a response to the client with an `HttpServerExecutionStatus = RemoteRequestDropped`.
  - Forwarding Resolver configuration: if a ForwardingResolver is configured, it will call the OnResolveRequestHost method on the original host of the request.
  - DNS matching: with the host resolved and with more than one ListeningHost configured, the server will look for the corresponding host for the request.
    - If no ListeningHost matches, a 400 Bad Request response is returned to the client and an `HttpServerExecutionStatus = DnsUnknownHost` status is returned to the HTTP context.
    - If a ListeningHost matches, but its Router is not yet initialized, a 503 Service Unavailable response is returned to the client and an `HttpServerExecutionStatus = ListeningHostNotReady` status is returned to the HTTP context.
  - Router binding: the router of the corresponding ListeningHost is associated with the received HTTP server.
    - If the router is already associated with another HTTP server, which is not allowed because the router actively uses the server's configuration resources, an `InvalidOperationException` is thrown. This only occurs during the initialization of the HTTP server, not during the creation of the HTTP context.
  - Pre-definition of headers:
    - Predefines the `X-Request-Id` header in the response if it is configured to do so.
    - Predefines the `X-Powered-By` header in the response if it is configured to do so.
  - Content size validation: validates if the request content is less than HttpServerConfiguration.MaximumContentLength only if it is greater than zero.
    - If the request sends a `Content-Length` greater than the configured one, a 413 Payload Too Large response is returned to the client and an `HttpServerExecutionStatus = ContentTooLarge` status is returned to the HTTP context.
  - The `OnHttpRequestOpen` event is invoked for all configured HTTP server handlers.
- **Routing the action:** the server invokes the router for the received request.
  - If the router does not find a route that matches the request:

        - If the Router.NotFoundErrorHandler property is configured, the action is invoked, and the response of the action is forwarded to the HTTP client.
        - If the previous property is null, a default 404 Not Found response is returned to the client.
    - If the router finds a matching route, but the route's method does not match the request's method:
        - If the Router.MethodNotAllowedErrorHandler property is configured, the action is invoked and the response of the action is forwarded to the HTTP client.
        - If the previous property is null, a default 405 Method Not Allowed response is returned to the client.
    - If the request is of the `OPTIONS` method:
        - The router returns a 200 Ok response to the client only if no route matches the request method (the route's method is not explicitly RouteMethod.Options).
    - If the HttpServerConfiguration.ForceTrailingSlash property is enabled, the matched route is not a regex, the request path does not end with `/`, and the request method is `GET`:
        - A 307 Temporary Redirect HTTP response with the `Location` header with the path and query to the same location with a `/` at the end is returned to the client.
    - The `OnContextBagCreated` event is invoked for all configured HTTP server handlers.
    - All global IRequestHandler instances with the `BeforeResponse` flag are executed.
        - If any handler returns a non-null response, that response is forwarded to the HTTP client and the context is closed.
        - If an error is thrown in this step and HttpServerConfiguration.ThrowExceptions is disabled:
            - If the Router.CallbackErrorHandler property is enabled, it is invoked and the resulting response is returned to the client.
            - If the previous property is not defined, an empty response is returned to the server, which forwards a response according to the type of exception thrown, which is usually 500 Internal Server Error.
    - All IRequestHandler instances defined in the route and with the `BeforeResponse` flag are executed.
        - If any handler returns a non-null response, that response is forwarded to the HTTP client and the context is closed.
        - If an error is thrown in this step and HttpServerConfiguration.ThrowExceptions is disabled:
            - If the Router.CallbackErrorHandler property is enabled, it is invoked and the resulting response is returned to the client.
            - If the previous property is not defined, an empty response is returned to the server, which forwards a response according to the type of exception thrown, which is usually 500 Internal Server Error.
    - The router's action is invoked and transformed into an HTTP response.
        - If an error is thrown in this step and HttpServerConfiguration.ThrowExceptions is disabled:
            - If the Router.CallbackErrorHandler property is enabled, it is invoked and the resulting response is returned to the client.
            - If the previous property is not defined, an empty response is returned to the server, which forwards a response according to the type of exception thrown, which is usually 500 Internal

Server Error.

- All global IRequestHandler instances with the `AfterResponse` flag are executed.
  - If any handler returns a non-null response, the handler's response replaces the previous response and is immediately forwarded to the HTTP client.
  - If an error is thrown in this step and HttpServerConfiguration.ThrowExceptions is disabled:
    - If the Router.CallbackErrorHandler property is enabled, it is invoked and the resulting response is returned to the client.
    - If the previous property is not defined, an empty response is returned to the server, which forwards a response according to the type of exception thrown, which is usually 500 Internal Server Error.
- All IRequestHandler instances defined in the route and with the `AfterResponse` flag are executed.
  - If any handler returns a non-null response, the handler's response replaces the previous response and is immediately forwarded to the HTTP client.
  - If an error is thrown in this step and HttpServerConfiguration.ThrowExceptions is disabled:
    - If the Router.CallbackErrorHandler property is enabled, it is invoked and the resulting response is returned to the client.
    - If the previous property is not defined, an empty response is returned to the server, which forwards a response according to the type of exception thrown, which is usually 500 Internal Server Error.

- **Processing the response:** with the response ready, the server prepares it for sending to the client.
  - The Cross-Origin Resource Sharing Policy (CORS) headers are defined in the response according to what was configured in the current ListeningHost.CrossOriginResourceSharingPolicy.
  - The status code and headers of the response are sent to the client.
  - The response content is sent to the client:
    - If the response content is a descendant of ByteArrayContent, the response bytes are directly copied to the response output stream.
    - If the previous condition is not met, the response is serialized to a stream and copied to the response output stream.
  - The streams are closed and the response content is discarded.
  - If HttpServerConfiguration.DisposeDisposableContextValues is enabled, all objects defined in the request context that inherit from IDisposable are discarded.
  - The `OnHttpRequestClose` event is invoked for all configured HTTP server handlers.
  - If an exception was thrown on the server, the `OnException` event is invoked for all configured HTTP server handlers.
  - If the route allows access-logging and HttpServerConfiguration.AccessLogsStream is not null, a log line is written to the log output.
  - If the route allows error-logging, there is an exception, and HttpServerConfiguration.ErrorsLogsStream is not null, a log line is written to the error log output.
  - If the server is waiting for a request through HttpServer.WaitNext, the mutex is released and the context becomes available to the user.

Forwarding Resolvers

A Forwarding Resolver is a helper that helps decode information that identifies the client through a request, proxy, CDN or load-balancers. When your Sisk service runs through a reverse or forward proxy, the client's IP address, host and protocol may be different from the original request as it is a forwarding from one service to another. This Sisk functionality allows you to control and resolve this information before working with the request. These proxies usually provide useful headers to identify their client.

Currently, with the ForwardingResolver class, it is possible to resolve the client IP address, host, and HTTP protocol used. After version 1.0 of Sisk, the server no longer has a standard implementation to decode these headers for security reasons that vary from service to service.

For example, the `X-Forwarded-For` header includes information about the IP addresses that forwarded the request. This header is used by proxies to carry a chain of information to the final service and includes the IP of all proxies used, including the client's real address. The problem is: sometimes it is challenging to identify the client's remote IP and there is no specific rule to identify this header. It is highly recommended to read the documentation for the headers you are about to implement below:

- Read about the `X-Forwarded-For` header here.
- Read about the `X-Forwarded-Host` header here.
- Read about the `X-Forwarded-Proto` header here.

The ForwardingResolver class

This class has three virtual methods that allow the most appropriate implementation for each service. Each method is responsible for resolving information from the request through a proxy: the client's IP address, the host of the request and the security protocol used. By default, Sisk will always use the information from the original request, without resolving any headers.

The example below shows how this implementation can be used. This example resolves the client's IP through the `X-Forwarded-For` header and throws an error when more than one IP was forwarded in the request.

> **IMPORTANT**
>
> Do not use this example in production code. Always check if the implementation is appropriate for use. Read the header documentation before implementing it.

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

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

        host.Start();
    }

    class Resolver : ForwardingResolver
    {
        public override IPAddress OnResolveClientAddress(HttpRequest request,
IPEndPoint connectingEndpoint)
        {
            string? forwardedFor = request.Headers.XForwardedFor;
            if (forwardedFor is null)
            {
                throw new Exception("The X-Forwarded-For header is missing.");
            }
            string[] ipAddresses = forwardedFor.Split(',');
            if (ipAddresses.Length != 1)
            {
                throw new Exception("Too many addresses in the X-Forwarded-For header.");
            }

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

Http server handlers

In Sisk version 0.16, we've introduced the `HttpServerHandler` class, which aims to extend the overral Sisk behavior and provide additional event handlers to Sisk, such as handling Http requests, routers, context bags and more.

The class concentrates events that occur during the lifetime of the entire HTTP server and also of a request. The Http protocol does not have sessions, and therefore it is not possible to preserve information from one request to another. Sisk for now provides a way for you to implement sessions, contexts, database connections and other useful providers to help your work.

Please refer to this page to read where each event is triggered and what its purpose is. You can also view the lifecycle of an HTTP request to understand what happens with a request and where events are fired. The HTTP server allows you to use multiple handlers at the same time. Each event call is synchronous, that is, it will blocked the current thread for each request or context until all handlers associated with that function are executed and completed.

Unlike RequestHandlers, they cannot be applied to some route groups or specific routes. Instead, they are applied to the entire HTTP server. You can apply conditions within your Http Server Handler. Furthermore, singletons of each HttpServerHandler are defined for every Sisk application, so only one instance per `HttpServerHandler` is defined.

A practical example of using HttpServerHandler is to automatically dispose a database connection at the end of the request.

```
// DatabaseConnectionHandler.cs

public class DatabaseConnectionHandler : HttpServerHandler
{
    public override void OnHttpRequestClose(HttpServerExecutionResult result)
    {
        var requestBag = result.Request.Context.RequestBag;

        // checks if the request has defined an DbContext
        // in it's context bag
        if (requestBag.IsSet<DbContext>())
        {
            var db = requestBag.Get<DbContext>();
            db.Dispose();
        }
```

```
    }
}

public static class DatabaseConnectionHandlerExtensions
{
    // allows the user to create an dbcontext from an http request
    // and store it in its request bag
    public static DbContext GetDbContext(this HttpRequest request)
    {
        var db = new DbContext();
        return request.SetContextBag<DbContext>(db);
    }
}
```

With the code above, the `GetDbContext` extension allows a connection context to be created directly from the HttpRequest object. An undisposed connection can cause problems when running with the database, so it is terminated in `OnHttpRequestClose`.

You can register a handler on an Http server in your builder or directly with HttpServer.RegisterHandler.

```
// Program.cs

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

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

With this, the `UsersController` class can make use of the database context as:

```
// UserController.cs

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

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

        return JsonOk(users);
    }

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

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

        return JsonOk(user);
    }

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

        ArgumentNullException.ThrowIfNull(user);

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

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

The code above uses methods like `JsonOk` and `JsonMessage` that are built into `ApiController`, which is inherited from a `RouterController`:

```
// ApiController.cs

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

```
    }

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

Developers can implement sessions, contexts, and database connections using this class. The provided code showcases a practical example with the DatabaseConnectionHandler, automating database connection disposal at the end of each request.

Integration is straightforward, with handlers registered during server setup. The HttpServerHandler class offers a powerful toolset for managing resources and extending Sisk behavior in HTTP applications.

Multiple listening hosts per server

The Sisk Framework has always supported the use of more than one host per server, that is, a single HTTP server can listen on multiple ports and each port has its own router and its own service running on it.

This way, it is easy to separate responsibilities and manage services on a single HTTP server with Sisk. The example below shows the creation of two ListeningHosts, each listening to a different port, with different routers and actions.

Read manually creating your app to understand the details about this abstraction.

```
static void Main(string[] args)
{
    // create two listening hosts, which each one has it's own router and
    // listens to it's own port
    //
    ListeningHost hostA = new ListeningHost();
    hostA.Ports = [new ListeningPort(12000)];
    hostA.Router = new Router();
    hostA.Router.SetRoute(RouteMethod.Get, "/", request => new
HttpResponse().WithContent("Hello from the host A!"));

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

    // create an server configuration and adds both
    // listening hosts on it
    //
    HttpServerConfiguration configuration = new HttpServerConfiguration();
    configuration.ListeningHosts.Add(hostA);
    configuration.ListeningHosts.Add(hostB);

    // creates an http server which uses the specified
    // configuration
    //
    HttpServer server = new HttpServer(configuration);

    // starts the server
    server.Start();
```

```
32
33      Console.WriteLine("Try to reach host A in {0}", server.ListeningPrefixes[0]);
34      Console.WriteLine("Try to reach host B in {0}", server.ListeningPrefixes[1]);
35
        Thread.Sleep(-1);
    }
```