Asypi

Asypi Contributors

May 05, 2022

CONTENTS:

CHAPTER

ONE

INTRODUCTION

A simple, asynchronous, and flexible web framework built on `System.Net.HttpListener`. Pronounced "AS-eh-PIE."

1.1 Hello World Example

```
using Asypi;

Server server = new Server(8000, "localhost");

server.Route(
    HttpMethod.Get,
    "/",
    () => { return "Hello World!"; },
    "text/html"
);

server.Run();
```

1.2 Features

- Simple declarative API
- Async support
- Flexible router supporting variable parameters
- Cached static file server supporting custom mount points

CHAPTER

TWO

GETTING STARTED

Asypi is available on nuget. To install the latest version, run `dotnet add package Asypi`.

CHAPTER

THREE

SETUP, ROUTING, AND RESPONDERS

3.1 Setup

Asypi makes setting up a server almost effortless. For example, to set up and run a server listening on `localhost:8000`, one can use the following:

```
using Asypi;

Server server = new Server(8000, "localhost");

server.Run();
```

More constructors for `Server` are available in the API reference.

`Server.Run()` will block the main thread until the server terminates. If you need to do other things while the server runs, consider `Server.RunAsync()`. For example:

```
using Asypi;

Server server = new Server(8000, "localhost");

server.RunAsync();
DoOtherThings();
```

Beware that when the main thread terminates, the server does too.

3.2 Routing

Asypi makes setting up routes a simple and efficient process.

3.2.1 Simple Routes

```
using Asypi;

Server server = new Server(8000, "localhost");

// Respond to GET requests to / with body "Hello World!" and content type flag "text/html
↪"
server.Route(
    HttpMethod.Get,
    "/",
    () => { return "Hello World!"; },
    "text/html"
);

server.Run();
```

3.2.2 Parameterized Routes

Variable parameter names must be surrounded by curly braces.

```
using Asypi;

Server server = new Server(8000, "localhost");

// Respond to GET requests to /{name} with a personalized greeting
server.Route(
    HttpMethod.Get,
    "/{name}",
    (Req req, Res res) => {
        return String.Format(
            "Hello {0}!",
            req.Args[0]
        );
    },
    "text/html"
);

/*

Expected Results:
/Joe -> "Hello Joe!"
/Zoe -> "Hello Zoe!"
/123 -> "Hello 123!"

*/
```

(continues on next page)

(continued from previous page)

```
server.Run();
```

3.3 Responders

In Asypi, each route is assigned a responder that fires whenever the route is accessed. A basic responder has the following signature and returns `void`:

```
(Req req, Res res) => {
    // do something
}
```

For convenience, a number of other responder signatures are supported out of the box:

```
() => {
    // do something
    return bodyText;
}

(Req req, Res res) => {
    // do something
    return bodyText;
}
```

When attaching any of these to a route, a content type must also be provided since the responder cannot reasonably infer one.

CHAPTER

FOUR

SERVING STATIC FILES

Asypi comes with both powerful utilities for routing static files and a versatile `FileServer` for finer control.

4.1 Serving Single Files

```
using Asypi;

Server server = new Server(8000, "localhost");

// Serve content.txt at /
server.RouteStaticFile("/", "content.txt");

server.Run();
```

4.2 Serving Directories

```
using Asypi;

Server server = new Server(8000, "localhost");

// Serve files from ./static at /staticfiles
server.RouteStaticDir("/staticfiles", "./static");

server.Run();
```

4.3 Cache-Control

When using `Asypi.RouteStaticFile()` or `Asypi.RouteStaticDir()`, the `Cache-Control` header is automatically set to the following:

```
Cache-Control: public, max-age=86400
```

4.4 File Compression

With a directory structure such as the following:

```
static
+---foo.txt
+---foo.txt.br
+---foo.txt.gz
```

`Asypi.RouteStaticFile()` and `Asypi.RouteStaticDir()` will first try to satisfy requests to `foo.txt` with `foo.txt.br`, followed by `foo.txt.gz`. Asypi automatically checks the `Accept-Encoding` header and automatically sets the `Content-Encoding` header to ensure compatibility.

4.5 Using `FileServer`

FileServer allows quick, easy, and cached access to files on disk. `Server.RouteStaticFile()` and `Server.RouteStaticDir()` both utilize `FileServer` under the hood.

`FileServer` caches files under an LFU system. The maximum size of the cache and the interval between cache updates can be configured when creating a `Server`. Note that `FileServer` will *not* cache files larger than 50% of the maximum cache size.

```
using Asypi;

Server server = new Server(8000, "localhost");

server.Route(
    HttpMethod.Get,
    "/",
    (Req req, Res res) => {
        byte[] content = FileServer.Get("content.txt");
        string decodedContent = Encoding.UTF8.GetString(content);
        // do something
    }
);

server.Run();
```

CHAPTER

FIVE

MIDDLEWARE

Asypi supports a versatile middleware system, allowing one to view, modify, or cancel request/response pairs before a responder ever sees them.

To understand how middleware in Asypi works, first the "response chain" must be understood. When Asypi receives an HTTP request, a request/response pair is created and then sent as followed: `server -> router -> middleware -> responder`.

Middleware has the power to "break" the response chain, preventing successive middleware or the responder from operating on the request/response pair.

5.1 Writing Middleware

In Asypi, middleware is simply a function with the following signature:

```
(Req req, Res res) => {
    // do something
    return shouldContinue;
}
```

If a middleware function returns false, it will "break the response chain," preventing successive middleware or the responder from operating on the request/response pair. If a middleware function returns true, the request/response pair will be passed down the chain normally.

5.2 Using Middleware

Middleware can be used with a simple call to `Server.Use()`. For example, to register `doStuff()` as middleware for all routes, one would call `Server.Use(".*", doStuff);`.

5.3 Middleware Example

```
// on all routes that start with private/
server.use(@"private\/.*", (Req req, Res res) => {
    // don't load the page if user is not authenticated
    if (isAuthenticated(req)) {
        return true;
    } else {
```

(continues on next page)

(continued from previous page)

```
        return false;
    }
});
```

CHAPTER

SIX

API REFERENCE

6.1 Server

The `Server` class is the heart of Asypi. `Server`s handle receiving requests and sending responses, routing requests, managing routes and responders, and logging diagnostic information.

Due to the way that `Server` interfaces with the static `FileServer`, only one `Server` may be created per run.

6.1.1 Public Fields

`int Port`

The port that the `Server` will bind to when `Server.Run()` or `Server.RunAsync()` is called.

`IEnumerable Hosts`

The hosts that the `Server` will bind to when `Server.Run()` or `Server.RunAsync()` is called.

`LogLevel LogLevel`

The `LogLevel` that the `Server` initialized the logger with.

`int LFUCacheSize`

The LFU cache size, in MiB, that the `Server` initialized `FileServer` with.

---

int FileServerEpochLength

The epoch length, in milliseconds, that the `Server` initialized `FileServer` with.

Responder Responder404

The `Responder` that the `Server` will route requests to when no other valid routes were found.

6.1.2 Constructors

```
public Server(
    int port = 8000,
    string hostname = "localhost",
    LogLevel logLevel = LogLevel.Debug,
    int? LFUCacheSize = null,
    int? fileServerEpochLength = null
)
```

```
public Server(
    int port,
    string[] hostnames,
    LogLevel logLevel = LogLevel.Debug,
    int? LFUCacheSize = null,
    int? fileServerEpochLength = null
)
```

Note that `LFUCacheSize` is measured in MiB, and `fileServerEpochLength` is measured in milliseconds.

If given as null, `LFUCacheSize` and `fileServerEpochLength` will be set to default values under the hood.

6.1.3 Routing Methods

Routes requests to `path` of method `method` to `responder`.

Paths should not contain trailing slashes.

Paths can include variable parameters, the values of which will be forwarded to the responder. Variable parameters must be surrounded by curly braces. For example, `/users/{id}` would match `/users/1`, `/users/joe`, etc., and the responder would receive argument lists `["1"]` and `["joe"]` respectively.

`void Route(HttpMethod method, string path, Responder responder)`

`void Route(HttpMethod method, string path, SimpleTextResponder responder, string contentType, IHeaders headers = null)`

`void Route(HttpMethod method, string path, ComplexTextResponder responder, string contentType, IHeaders headers = null)`

6.1.4 Static File Routing

```
void RouteStaticFile(string path, string filePath, string contentType = null)
```

Routes requests to path to the file at `filePath`. The response will have its content type set to `contentType` if provided. Otherwise, the content type will be guessed.

Note that `DefaultHeaders` contains `X-Content-Type-Options: nosniff`.

```
RouteStaticDir(string mountRoot, string dirRoot, int? maxDepth = null, string match =
".*")
```

Routes requests to paths matching `match` under `mountRoot` to files under `dirRoot`. Content types will be guessed.

If `maxDepth` is not set, will recursively include all subdirectories of `dirRoot`. Otherwise, will only include subdirectories to a depth of `maxDepth`. For example, with `maxDepth` set to 1, will only include files directly under `dirRoot`.

Note that `DefaultHeaders` contains `X-Content-Type-Options: nosniff`.

If finer control is necessary, consider mounting individual files using `Server.RouteStaticFile()`.

6.1.5 404 Responder Setters

```
void Set404Responder(Responder responder)
```

```
void Set404Responder(SimpleTextResponder responder, string contentType)
```

```
void Set404Responder(ComplexTextResponder responder, string contentType)
```

6.1.6 Other Methods

6.1.7 void Use(string matchExpression, Middleware middleware)

Registers the middleware for use on all paths matching `matchExpression`.

```
void Reset()
```

Resets the server. This will remove all previously registered routes.

Exists primarily for testing purposes.

`Task RunAsync()`

Runs the server. For a sync wrapper, consider `Server.Run()`.

`void Run()`

Runs the server. This will block the thread until the server stops running. For async, consider `Server.RunAsync()`.

6.2 Responders

Under the hood, all requests in Asypi are handled by a responder with the following delegate:

```
public delegate void Responder(
    Req request,
    Res response
);
```

However, for convenience, simplified responders are also generally accepted. These are:

```
public delegate string SimpleTextResponder();
public delegate string ComplexTextResponder(Req req, Res res);
```

Under the hood, each of these are wrapped by a `Responder` that sets the body of the `Res` to the string output of the simplified responder, sets content type to a separately provided value, and sets headers.

6.3 HttpMethod

```
public enum HttpMethod {
    Get,
    Head,
    Post,
    Put,
    Delete,
    Patch
}
```

Asypi also contains a few convenience items for working with `HttpMethods`:

`string.ToHttpMethod()`

`HttpMethod.AsString()`

6.4 Req

`Req` is a wrapper over `System.Net.HttpListenerRequest`, exposing relevant fields.

6.4.1 Public Fields

`string BodyText`

Gets the body of the request, as a `string`.

`string BodyBytes`

Gets the body of the request, as a `byte[]`.

6.4.2 `List<string> args`

The values of applicable variable parameters.

For example, if a route was registered with the path `/{name}`, and a user requested `/joe`, the args in the resulting `Req` will contain `["joe"]`.

6.4.3 Other Public Fields

The majority of the fields in `System.Net.HttpListenerRequest` are directly wrapped by `Req`. For more information on these fields, refer to Microsoft's official documentation.

6.5 Res

`Res` is a wrapper over `System.Net.HttpListenerResponse`, exposing relevant fields.

6.5.1 Public Fields

`string BodyText`

Sets the body of the response, as a `string`.

`string BodyBytes`

Sets the body of the response, as a `byte[]`.

6.5.2 Other Public Fields

The majority of the fields in `System.Net.HttpListenerResponse` are directly wrapped by `Res`. For more information on these fields, refer to Microsoft's official documentation.

6.6 IHeaders

```
public interface IHeaders {
    Dictionary<string, string> Values { get; }
}
```

6.7 DefaultHeaders

`DefaultHeaders` is an inheritable class implementing `IHeaders` containing sensible defaults for headers. To be precise, `DefaultHeaders` contains the following:

```
Server: Asypi
X-Content-Type-Options: nosniff
"X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Content-Security-Policy: script-src 'self'; object-src 'none'; require-trusted-types-for
↪'script'
```

Note that `DefaultHeaders` does *NOT* contain `Strict-Transport-Security`, to ensure that Asypi projects work in development. Consider inheriting `DefaultHeaders` and adding this header.

6.8 DefaultServerHeaders

`DefaultServerHeaders` is a static class with a single public member, `Instance`. When no headers are specified for a special responder (e.g. when using `Server.RouteStaticFile()`), `DefaultServerHeaders.Instance` will be loaded by the server.

6.8.1 Public Fields

`static IHeaders Instance`

The internal instance of `DefaultServerHeaders`.

6.9 FileServer

`FileServer` is a static class provided by Asypi. It utilizes an LFU cache, which is set up when `Server` is initialized.

6.9.1 Public Methods

`byte[] Get(string filePath)`

Gets the content of the file at `filePath` as a byte[], and then updates the LFU cache.

6.9.2 `byte[] Read(string filePath)`

Gets the content of the file at `filePath` as a byte[], but does NOT update the LFU cache.