Routing

Web servers use the request's **path and HTTP method** to look up the correct resource, refers as **"routing"**.

We can define a route by calling a **method named after HTTP verbs**, passing a path and a function to execute when matched.

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .get('/', 'hello')
    .get('/hi', 'hi')
    .listen(3000)
```

We can access the web server by going to http://localhost:3000

By default, web browsers will send a GET method when visiting a page.

TIP

Using an interactive browser above, hover on a blue highlight area to see difference result between each path

Path type

Path in Elysia can be grouped into 3 types:

- **static paths** - static string to locate the resource
- **dynamic paths** - segment can be any value
- **wildcards** - path until a specific point can be anything

You can use all of the path types together to compose a behavior for your web server.

The priorities are as follows:

3. wildcards

If the path is resolved as the static wild dynamic path is presented, Elysia will resolve the static path rather than the dynamic path

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .get('/id/1', 'static path')
    .get('/id/:id', 'dynamic path')
    .get('/id/*', 'wildcard path')
    .listen(3000)
```

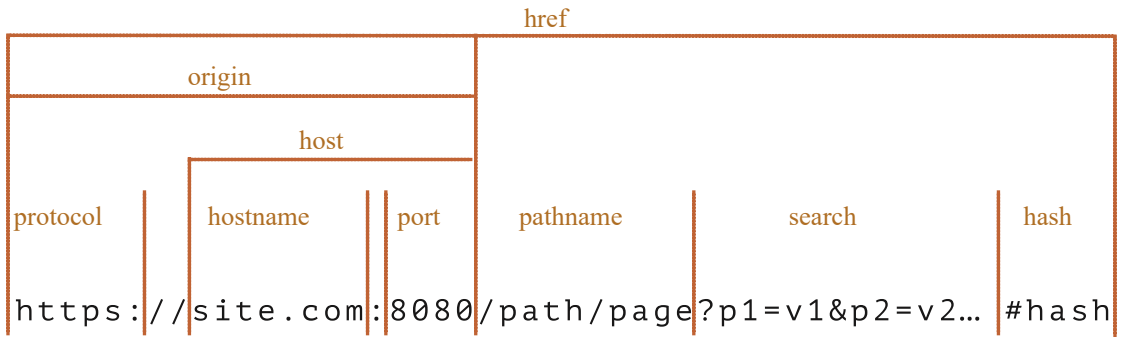
Path	Response
/id/1	static path
/id/2	dynamic path
/id/2/a	wildcard path

Static Path

A path or pathname is an identifier to locate resources of a server.

```bash
http://localhost:/path/page
```

Elysia uses the path and method to look up the correct resource.



A path starts after the origin. Prefix with **/** and ends before search query **(?)**

URL	Path
http://site.com/	/
http://site.com/hello	/hello
http://site.com/hello/world	/hello/world

Outline

http://site.com/hello#title	/hello

TIP

If the path is not specified, the browser and web server will treat the path as '/' as a default value.

Elysia will look up each request for route and response using handler function.

Dynamic path

URLs can be both static and dynamic.

Static paths are hardcoded strings that can be used to locate resources of the server, while dynamic paths match some part and captures the value to

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .get('/id/:id', ({ params:
    .listen(3000)
```

```
params: {
    id: string;
}
```

Here dynamic path is created with `/id/:id` which tells Elysia to match any path up until `/id`. What comes after that is then stored as **params** object.

When requested, the server should return the response as follows:

/id/123	123
/id/anything	anything
/id/anything?name=salt	anything
/id	Not Found
/id/anything/rest	Not Found

Dynamic paths are great to include things like IDs, which then can be used later.

We refer to the named variable path as **path parameter** or **params** for short.

Segment

URL segments are each path that is composed into a full path.

Segments are separated by `/`.

http://elysiajs.com/name/saltyaom

Segment 1 Segment 2

Path parameters in Elysia are represented by prefixing a segment with ':' followed by a name.

/:id

Prefix Path parameter name

Tells Elysia that segment is a dynamic value

Name to be stored into params

Route	Path	Params
/id/:id	/id/1	id=1
/id/:id	/id/hi	id=hi
/id/:name	/id/hi	name=hi

Multiple path parameters

You can have as many path parameters as you like, which will then be stored into a `params` object.

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .get('/id/:id', ({ params: { id } }) => id)
    .get('/id/:id/:name', ({ params: { id, name } }) => id + ' ' + name)
    .listen(3000)
```

```typescript
params: {
    id: string;
    name: string;
}
```

The server will respond as follows:

Path	Response
/id/1	1
/id/123	123
/id/anything	anything
/id/anything?name=salt	anything
/id	Not Found
/id/anything/rest	anything rest

Optional path parameters

We can make a path parameter optional by adding a question mark `?` after the parameter name.

```
params: {
    id?: string | undefined;
}
```

```ts
import { Elysia } from 'elysia'

new Elysia()
    .get('/id/:id?', ({ params: { id } }) => `id ${id}`)
    .listen(3000)
```

localhost /id GET

id: undefined

The server will respond as follows:

/id/1	id 1

Wildcards

Dynamic paths allow capturing certain segments of the URL.

However, when you need a value of the path to be more dynamic and want to capture the rest of the URL segment, d.

```
params: {
    "*": string;
}
```

Wildcards can capture the value afte f amount by using "*".

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .get('/id/*', ({ params }) => params['*'])
    .listen(3000)
```

localhost /id/1 GET

1

In this case the server will respond as follows:

Path	Response
/id/1	1
/id/123	123
/id/anything	anything
/id/anything?name=salt	anything
/id	Not Found
/id/anything/rest	anything/rest

Wildcards are useful for capturing a path until a specific point.

TIP

You can use a wildcard with a path parameter.

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource

There are several HTTP verbs, but the most common ones are:

GET

Requests using GET should only retrieve data.

POST

Submits a payload to the specified resource, often causing state change or side effect.

PUT

Replaces all current representations of the target resource using the request's payload.

PATCH

Applies partial modifications to a resource.

DELETE

To handle each of the different verbs, Elysia has a built-in API for several HTTP verbs by default, similar to `Elysia.get`

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .get('/', 'hello')
    .post('/hi', 'hi')
    .listen(3000)
```

Elysia HTTP methods accepts the following parameters:

- **path**: Pathname

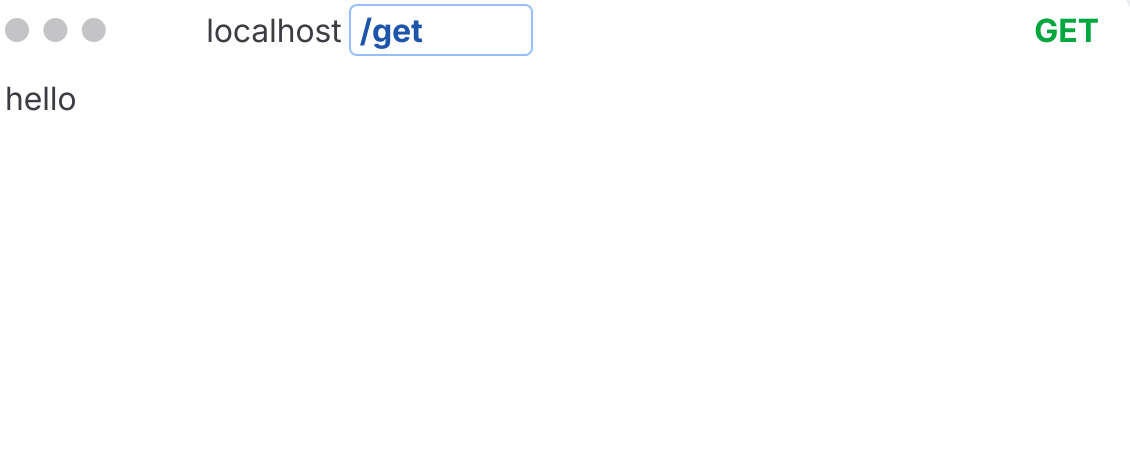
You can read more about the HTTP methods on HTTP Request Methods.

Custom Method

We can accept custom HTTP Methods with `Elysia.route`.

typescript

```
import { Elysia } from 'elysia'

const app = new Elysia()
    .get('/get', 'hello')
    .post('/post', 'hi')
+   .route('M-SEARCH', '/m-search', 'connect')
    .listen(3000)
```



Elysia.route accepts the following:

- **method**: HTTP Verb
- **path**: Pathname
- **function**: Function to response to the client
- **hook**: Additional metadata

When navigating to each method, you should see the results as the following:

Path	Method	Result
/	GET	hello
/	POST	hi
/	M-SEARCH	connect

TIP

Based on RFC 7231, HTTP Verb is case-sensitive.

It's recommended to use the UPPERCASE convention for defining a custom HTTP Verb with Elysia.

Elysia.all

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .all('/', 'hi')
    .listen(3000)
```

localhost / GET

hi

Any HTTP method that matches the path, will be handled as follows:

Path	Method	Result
/	GET	hi
/	POST	hi

Handle

Most developers use REST clients like Postman, Insomnia or Hoppscotch to test their API.

However, Elysia can be programmatically test using `Elysia.handle`.

typescript

```typescript
import { Elysia } from 'elysia'

const app = new Elysia()
    .get('/', 'hello')
    .post('/hi', 'hi')
    .listen(3000)

app.handle(new Request('http://localhost/')).then(console.log)
```

Elysia.handle is a function to process an actual request sent to the server.

TIP

But also useful for simulating or creating unit tests.

404

If no path matches the defined routes, Elysia will pass the request to error life cycle before returning a **"NOT_FOUND"** with an HTTP status of 404.

We can handle a custom 404 error by returning a value from `error` life cycle like this:

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .get('/', 'hi')
    .onError(({ code }) => {
        if (code === 'NOT_FOUND') {
            return 'Route not found :('
        }
    })
    .listen(3000)
```

When navigating to your web server, you should see the result as follows:

Path	Method	Result
/	GET	hi
/	POST	Route not found :(
/hi	GET	Route not found :(

You can learn more about life cycle and error handling in Life Cycle Events and Error Handling.

TIP

HTTP Status is used to indicate the type of response. By default if everything is correct, the server will return a '200 OK' status code (If a route matches and there is no error, Elysia will return 200 as default)

Group

When creating a web server, you would often have multiple routes sharing the same prefix:

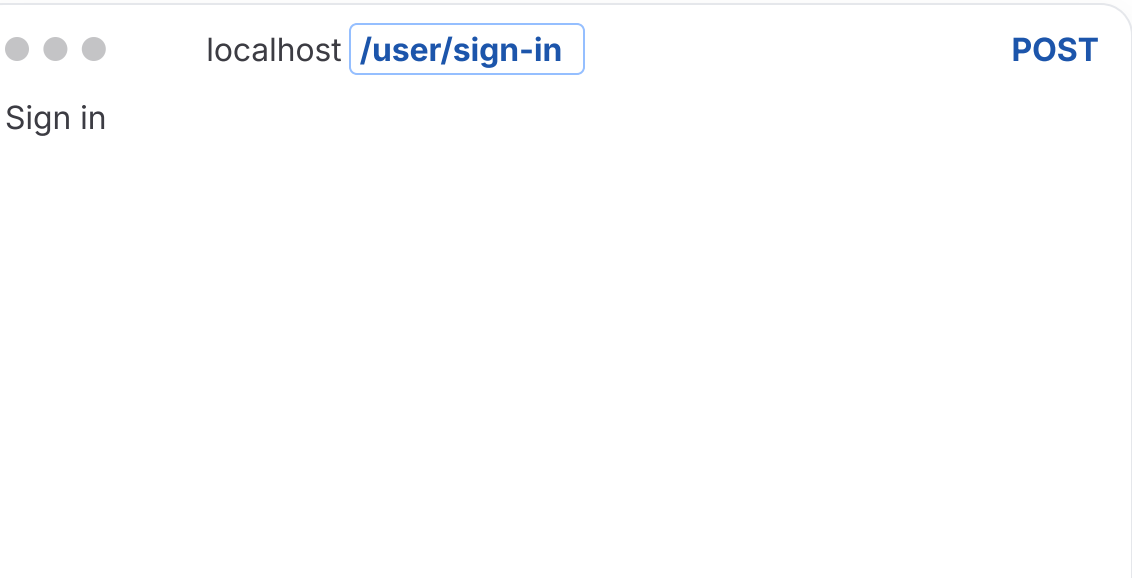
typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .post('/user/sign-in', 'Sign in')
    .post('/user/sign-up', 'Sign up')
    .post('/user/profile', 'Profile')
    .listen(3000)
```

This can be improved with `Elysia.group`, allowing us to apply prefixes to multiple routes at the same time by grouping them together:

typescript

```typescript
import { Elysia } from 'elysia'

new Elysia()
    .group('/user', (app) =>
        app
            .post('/sign-in', 'Sign in')
            .post('/sign-up', 'Sign up')
            .post('/profile', 'Profile')
    )
    .listen(3000)
```



follows:

Path	Result
/user/sign-in	Sign in
/user/sign-up	Sign up
/user/profile	Profile

`.group()` can also accept an optional guard parameter to reduce boilerplate of using groups and guards together:

typescript

```typescript
import { Elysia, t } from 'elysia'

new Elysia()
    .group(
        '/user',
        {
            body: t.Literal('Rikuhachima Aru')
        },
        (app) => app
            .post('/sign-in', 'Sign in')
            .post('/sign-up', 'Sign up')
            .post('/profile', 'Profile')
```

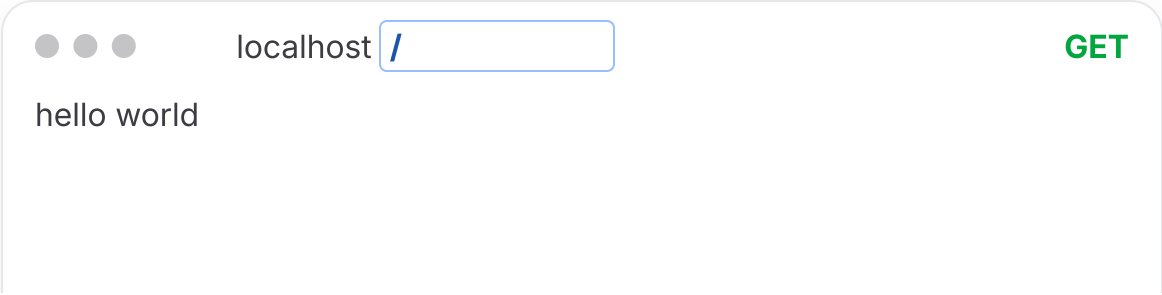
You may find more information about grouped guards in scope.

Prefix

We can separate a group into a separate plugin instance to reduce nesting by providing a **prefix** to the constructor.

typescript

```typescript
import { Elysia } from 'elysia'

const users = new Elysia({ prefix: '/user' })
    .post('/sign-in', 'Sign in')
    .post('/sign-up', 'Sign up')
    .post('/profile', 'Profile')

new Elysia()
    .use(users)
    .get('/', 'hello world')
    .listen(3000)
```



Edit this page on GitHub

Last updated: 3/11/25, 1:13 PM

Previous page
Table of Content

Next page
Handler