



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

Search

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

Outline

Handler

Handler is a function that responds to the request for each route.

Accepting request information and returning a response to the client.

Alternatively, handler is also known as a **Controller** in other frameworks.

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

typescript

```
new Elysia()  
  // the function `() => 'hello world'` is a handler  
  .get('/', () => 'hello world')  
  .listen(3000)
```

Handler maybe a literal value, and can be inlined.

```
import { Elysia, file } from 'elysia'
```

typescript

```
new Elysia()  
  .get('/', 'Hello Elysia')  
  .get('/video', file('kyuukurarin.mp4'))  
  .listen(3000)
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#)[Plugins](#)[Blog](#)

This allows Elysia to compile the response ahead of time to optimize performance.

TIP

Providing an inline value is not a cache.

Static Resource value, headers and status can be mutate dynamically using lifecycle.

Context

Context contains a request information which unique for each request, and is not shared except for `store` (global mutable state).

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

typescript

```
new Elysia()  
  .get('/', (context) => context.path)  
  // ^ This is a context
```

Context can be only retrieve in a route handler, consists of:

- **path** - Pathname of the request
- **body** - [HTTP message](#), form or file upload.
- **query** - [Query String](#), include additional parameters for search query as JavaScript Object. (Query is extracted from a value after pathname starting from '?' question mark sign)
- **params** - Elysia's path parameters parsed as JavaScript object



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#)[Plugins](#)[Blog](#)

- **request** - [Web Standard Request](#)
- **redirect** - A function to redirect a response
- **store** - A global mutable store for Elysia instance
- **cookie** - A global mutable signal store for interacting with Cookie (including get/set)
- **set** - Property to apply to Response:
 - **status** - [HTTP status](#), defaults to 200 if not set.
 - **headers** - Response headers
 - **redirect** - Response as a path to redirect to
- **error** - A function to return custom status code
- **server** - Bun server instance

Set

set is a mutable property that form a response accessible via `Context.set`.

- **set.status** - Set custom status code
- **set.headers** - Append custom headers
- **set.redirect** - Append redirect

```
import { Elysia } from 'elysia'

new Elysia()
  .get('/', ({ set, error }) => {
    set.headers = { 'X-Teapot': 'true' }

    return error(418, 'I am a teapot')
```

ts



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

status

We can return a custom status code by using either:

- **error** function (recommended)
- **set.status** (legacy)

typescript

```
import { Elysia } from 'elysia'

new Elysia()
  .get('/error', ({ error }) => error(418, 'I am a teapot'))
  .get('/set.status', ({ set }) => {
    set.status = 418
    return 'I am a teapot'
  })
  .listen(3000)
```

set.error

A dedicated **error** function for returning status code with response.

typescript

```
import { Elysia } from 'elysia'

new Elysia()
  .get('/', ({ error }) => error(418, "Kirifuji Nagisa"))
  .listen(3000)
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

It's recommend to use `error` inside main handler as it has better inference:

- allows TypeScript to check if a return value is correctly type to response schema
- autocompletion for type narrowing base on status code
- type narrowing for error handling using End-to-end type safety ([Eden](#))

set.status

Set a default status code if not provided.

It's recommended to use this in a plugin that only needs to return a specific status code while allowing the user to return a custom value. For example, HTTP 201/206 or 403/405, etc.

typescript

```
import { Elysia } from 'elysia'

new Elysia()
  .onBeforeHandle(({ set }) => {
    set.status = 418
```



Getting Started

[At Glance](#)
[Quick Start](#)
[Tutorial](#)
[Key Concept](#)
[Table of Content](#)

Essential

[Route](#)
[Handler](#)
[Life Cycle](#)
[Validation](#)
[Plugin](#)
[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)
[Plugins](#)
[Blog](#)

```
...  
    .listen(3000)
```

Unlike `error` function, `set.status` cannot infer the return value type, therefore it can't check if the return value is correctly type to response schema.

TIP

HTTP Status indicates the type of response. If the route handler is executed successfully without error, Elysia will return the status code 200.

You can also set a status code using the common name of the status code instead of using a number.

```
// @errors 2322  
import { Elysia } from 'elysia'
```

```
new Elysia()  
  .get('/', ({  
    set.statu
```

```
status?: number | "Continue" | "Switching Protocols" | "Processing"  
  | "Early Hints" | "OK" | "Created" | "Accepted" | "Non-Authoritative  
Information" | "No Content" | "Reset Content" | ... 50 more ... |  
undefined
```

```
    return 'Kirifuji Nagisa'  
  })  
  .listen(3000)
```

set.headers

Allowing us to append or delete a response headers represent as Object.



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
new Elysia()
  .get('/', ({ set }) => {
    set.headers['x-powered-by'] = 'Elysia'

    return 'a mimir'
  })
  .listen(3000)
```

WARNING

The names of headers should be lowercase to force case-sensitivity consistency for HTTP headers and auto-completion, eg. use `set-cookie` rather than `Set-Cookie` .

redirect

Redirect a request to another resource.

[typescript](#)

```
import { Elysia } from 'elysia'

new Elysia()
  .get('/', ({ redirect }) => {
    return redirect('https://youtu.be/whpVWVWBW4U?t=8')
  })
  .get('/custom-status', ({ redirect }) => {
    // You can also set custom status to redirect
    return redirect('https://youtu.be/whpVWVWBW4U?t=8', 302)
  })
  .listen(3000)
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

Server

Server instance is accessible via `Context.server` to interact with the server.

Server could be nullable as it could be running in a different environment (test).

If server is running (allocating) using Bun, `server` will be available (not null).

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

typescript

```
new Elysia()  
  .get('/port', ({ server }) => {  
    return server?.port  
  })  
  .listen(3000)
```

Request IP

We can get request IP by using `server.requestIP` method

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

typescript

```
new Elysia()  
  .get('/ip', ({ server, request }) => {  
    return server?.requestIP(request)  
  })
```




Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

Response

Elysia is built on top of Web Standard Request/Response.

To comply with the Web Standard, a value returned from route handler will be mapped into a [Response](#) by Elysia.

Letting you focus on business logic rather than boilerplate code.

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

typescript

```
new Elysia()  
  // Equivalent to "new Response('hi')"  
  .get('/', () => 'hi')  
  .listen(3000)
```

If you prefer an explicit Response class, Elysia also handles that automatically.

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

typescript

```
new Elysia()  
  .get('/', () => new Response('hi'))  
  .listen(3000)
```

TIP



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

Formdata

We may return a `FormData` by using returning `form` utility directly from the handler.

typescript

```
import { Elysia, form, file } from 'elysia'

new Elysia()
  .get('/', () => form({
    name: 'Tea Party',
    images: [file('nagi.web'), file('mika.webp')]
  }))
  .listen(3000)
```

This pattern is useful if even need to return a file or multipart form data.

Return a single file

Or alternatively, you can return a single file by returning `file` directly without `form`.

typescript

```
import { Elysia, file } from 'elysia'

new Elysia()
  .get('/', file('nagi.web'))
  .listen(3000)
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#) [Plugins](#) [Blog](#)

As Elysia is built on top of Web Standard Request, we can programmatically test it using `Elysia.handle`.

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

typescript

```
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

Unlike unit test's mock, **you can expect it to behave like an actual request** sent to the server.
But also useful for simulating or creating unit tests.

Stream

To return a response streaming out of the box by using a generator function with `yield` keyword.

typescript

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



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
yield 1
yield 2
yield 3
})
```

This this example, we may stream a response by using `yield` keyword.

Set headers

Elysia will defers returning response headers until the first chunk is yielded.

This allows us to set headers before the response is streamed.

[typescript](#)

```
import { Elysia } from 'elysia'

const app = new Elysia()
  .get('/ok', function* ({ set }) {
    // This will set headers
    set.headers['x-name'] = 'Elysia'
    yield 1
    yield 2

    // This will do nothing
    set.headers['x-id'] = '1'
    yield 3
  })
```

Once the first chunk is yielded, Elysia will send the headers and the first chunk in the same response.



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#) [Plugins](#) [Blog](#)

Conditional Stream

If the response is returned without yield, Elysia will automatically convert stream to normal response instead.

typescript

```
import { Elysia } from 'elysia'

const app = new Elysia()
  .get('/ok', function* () {
    if (Math.random() > 0.5) return 'ok'

    yield 1
    yield 2
    yield 3
  })
```

This allows us to conditionally stream a response or return a normal response if necessary.

Abort

While streaming a response, it's common that request may be cancelled before the response is fully streamed.

Elysia will automatically stop the generator function when the request is cancelled.

Eden

Eden will interpret a stream response as `AsyncGenerator` allowing us to use `for await` loop to consume the stream.



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
const app = new Elysia()
  .get('/ok', function* () {
    yield 1
    yield 2
    yield 3
  })
```

```
const { data, error } = await treaty(app).ok.get()
if (error) throw error
```

```
for await (const chunk of data)
  console.log(chunk)
```

Extending context

As Elysia only provides essential information, we can customize Context for our specific need for instance:

- extracting user ID as variable
- inject a common pattern repository
- add a database connection

We may extend Elysia's context by using the following APIs to customize the Context:

- [state](#) - a global mutable state
- [decorate](#) - additional property assigned to **Context**
- [derive](#) / [resolve](#) - create a new value from existing property



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

- A property is a global mutable state, and shared across multiple routes using [state](#)
- A property is associated with a request or response using [decorate](#)
- A property is derived from an existing property using [derive](#) / [resolve](#)

Otherwise, we recommend defining a value or function separately than extending the context.

TIP

It's recommended to assign properties related to request and response, or frequently used functions to Context for separation of concerns.

State

State is a global mutable object or state shared across the Elysia app.

Once **state** is called, value will be added to **store** property **once at call time**, and can be used in handler.

```
import { Elysia } from 'elysia'

new Elysia()
  .state('version', 1)
  .get('/a', ({ store: { version } }) => version)
  .get('/b', ({ store }) => store)
```

typescript



Getting Started

At Glance
Quick Start
Tutorial
Key Concept
Table of Content

Essential

Route
Handler
Life Cycle
Validation
Plugin
Best Practice

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#) [Plugins](#) [Blog](#)

When to use

- When you need to share a primitive mutable value across multiple routes
- If you want to use a non-primitive or a **wrapper** value or class that mutate an internal state, use decorate instead.

Key takeaway

- **store** is a representation of a single-source-of-truth global mutable object for the entire Elysia app.
- **state** is a function to assign an initial value to **store**, which could be mutated later.
- Make sure to assign a value before using it in a handler.

typescript

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

```
new Elysia()
```




Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

Property 'counter' does not exist on type '{}'.
|

```
.state('counter', 0)
```

```
// ✅ Because we assigned a counter before, we can now access it
```

```
.get('/', ({ store }) => store.counter)
```



localhost /error

GET

TIP

Beware that we cannot use a state value before assign.

Elysia registers state values into the store automatically without explicit type or additional TypeScript generic needed.

Decorate

decorate assigns an additional property to **Context** directly at call time.



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
class Logger {  
  log(value: string) {  
    console.log(value)  
  }  
}  
  
new Elysia()  
  .decorate('logger', new Logger())  
  // ✅ defined from the previous line  
  .get('/', ({ logger }) => {  
    logger.log('hi')  
  
    return 'hi'  
  })
```

When to use

- A constant or readonly value object to **Context**
- Non primitive value or class that may contain internal mutable state
- Additional functions, singleton, or immutable property to all handlers.

Key takeaway

- Unlike **state**, decorated value **SHOULD NOT** be mutated although it's possible
- Make sure to assign a value before using it in a handler.

Derive



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

new properties from existing properties).

typescript

```
import { Elysia } from 'elysia'

new Elysia()
  .derive(({ headers }) => {
    const auth = headers['authorization']

    return {
      bearer: auth?.startsWith('Bearer ') ? auth.slice(7) : null
    }
  })
  .get('/', ({ bearer }) => bearer)
```



localhost /

GET

12345

Because **derive** is assigned once a new request starts, **derive** can access request properties like **headers**, **query**, **body** where **store**, and **decorate** can't.



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#)[Plugins](#)[Blog](#)

checking

- When you need to access request properties like **headers**, **query**, **body** without validation

Key takeaway

- Unlike **state** and **decorate** instead of assign **at call time**, **derive** is assigned once a new request starts.
- **derive** is called **at transform**, or **before validation** happens, Elysia cannot safely confirm the type of request property resulting in as **unknown**. If you want to assign a new value from typed request properties, you may want to use resolve instead.

Resolve

Same as derive, resolve allow us to assign a new property to context.

Resolve is called at **beforeHandle** lifecycle or **after validation**, allowing us to **derive** request properties safely.

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

```
new Elysia()  
  .guard({  
    headers: t.Object({  
      bearer: t.String({  
        pattern: '^Bearer .+$'  
      })  
    })  
  })
```

typescript



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
    bearer: headers.bearer.slice(7)
  }
})
.get('/', ({ bearer }) => bearer)
```

When to use

- Create a new property from existing properties in **Context** with type integrity (type checked)
- When you need to access request properties like **headers**, **query**, **body** with validation

Key takeaway

- **resolve** is called at **beforeHandle**, or **after validation** happens. Elysia can safely confirm the type of request property resulting in as **typed**.

Error from resolve/derive

As resolve and derive is based on **transform** and **beforeHandle** lifecycle, we can return an error from resolve and derive. If error is returned from **derive**, Elysia will return early exit and return the error as response.

typescript

```
import { Elysia } from 'elysia'

new Elysia()
  .derive(({ headers, error }) => {
    const auth = headers['authorization']

    if(!auth) return error(400)
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
context.decorate('context', { bearer })  
}  
}))  
.get('/', ({ bearer }) => bearer)
```

Pattern

state, **decorate** offers a similar APIs pattern for assigning property to Context as the following:

- key-value
- object
- remap

Where **derive** can be only used with **remap** because it depends on existing value.

key-value

We can use **state**, and **decorate** to assign a value using a key-value pattern.

typescript

```
import { Elysia } from 'elysia'  
  
class Logger {  
  log(value: string) {  
    console.log(value)  
  }  
}
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
.decorate( logger, new Logger() )
```

This pattern is great for readability for setting a single property.

Object

Assigning multiple properties is better contained in an object for a single assignment.

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

typescript

```
new Elysia()  
  .decorate({  
    logger: new Logger(),  
    trace: new Trace(),  
    telemetry: new Telemetry()  
  })
```

The object offers a less repetitive API for setting multiple values.

Remap

Remap is a function reassignment.

Allowing us to create a new value from existing value like renaming or removing a property.

By providing a function, and returning an entirely new object to reassign the value.

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

typescript

```
new Elysia()
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

```
.state(({ version, ...store }) => ({
  ...store,
  elysiaVersion: 1
}))
// ✅ Create from state remap
.get('/elysia-version', ({ store }) => store.elysiaVersion)
// ❌ Excluded from state remap
.get('/version', ({ store }) => store.version)
```

Property 'version' does not exist on type '{ elysiaVersion: number; counte



It's a good idea to use state remap to create a new initial value from the existing value.

However, it's important to note that Elysia doesn't offer reactivity from this approach, as remap only assigns an initial value.

TIP



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

Affix

To provide a smoother experience, some plugins might have a lot of property value which can be overwhelming to remap one-by-one.

The **Affix** function which consists of **prefix** and **suffix**, allowing us to remap all property of an instance.

```
import { Elysia } from 'elysia'

const setup = new Elysia({ name: 'setup' })
  .decorate({
    argon: 'a',
    boron: 'b',
    carbon: 'c'
  })

const app = new Elysia()
  .use(
    setup
    .prefix('decorator', 'setup')
  )
  .get('/', ({ setupCarbon, ...rest }) => setupCarbon)
```

ts



localhost /

GET

c



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

Allowing us to bulk remap a property of the plugin effortlessly, preventing the name collision of the plugin.

By default, **affix** will handle both runtime, type-level code automatically, remapping the property to camelCase as naming convention.

In some condition, we can also remap **all** property of the plugin:

```
import { Elysia } from 'elysia'

const setup = new Elysia({ name: 'setup' })
  .decorate({
    argon: 'a',
    boron: 'b',
    carbon: 'c'
  })

const app = new Elysia()
+ .use(setup.prefix('all', 'setup'))
  .get('/', ({ setupCarbon, ...rest }) => setupCarbon)
```

ts



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[CtrlK](#)[Cheat Sheet](#) [Plugins](#) [Blog](#)

To mutate the state, it's recommended to use **reference** to mutate rather than using an actual value.

When accessing the property from JavaScript, if we define a primitive value from an object property as a new value, the reference is lost, the value is treated as new separate value instead.

For example:

```
const store = {  
  counter: 0  
}
```

```
store.counter++  
console.log(store.counter) //  1
```



typescript

We can use **store.counter** to access and mutate the property.

However, if we define a counter as a new value

```
const store = {  
  counter: 0  
}
```

```
let counter = store.counter
```

```
counter++  
console.log(store.counter) //  0  
console.log(counter) //  1
```

typescript



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

This can apply to `store`, as it's a global mutable object instead.

typescript

```
import { Elysia } from 'elysia'

new Elysia()
  .state('counter', 0)
  // ✅ Using reference, value is shared
  .get('/', ({ store }) => store.counter++)
  // ❌ Creating a new variable on primitive value, the link is lost
  .get('/error', ({ store: { counter } }) => counter)
```



TypeScript

Elysia automatically type context base on various of factors like store, decorators, schema.



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

CtrlK

[Cheat Sheet](#)[Plugins](#)[Blog](#)

- [InferContext](#)
- [InferHandle](#)

InferContext

Infer context is a utility type to help you define a context type based on Elysia instance.

typescript

```
import { Elysia, type InferContext } from 'elysia'

const setup = new Elysia()
  .state('a', 'a')
  .decorate('b', 'b')

type Context = InferContext<typeof setup>

const handler = ({ store }: Context) => store.a
```

InferHandler

Infer handler is a utility type to help you define a handler type based on Elysia instance, path, and schema.

typescript

```
import { Elysia, type InferHandler } from 'elysia'

const setup = new Elysia()
  .state('a', 'a')
  .decorate('b', 'b')
```



Getting Started

[At Glance](#)[Quick Start](#)[Tutorial](#)[Key Concept](#)[Table of Content](#)

Essential

[Route](#)[Handler](#)[Life Cycle](#)[Validation](#)[Plugin](#)[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration

[Cheat Sheet](#)[Plugins](#)[Blog](#)[CtrlK](#)

```
// Elysia instance is based on
typeof setup,
// path
'/path',
// schema
{
  body: string
  response: {
    200: string
  }
}
>

const handler: Handler = ({ body }) => body

const app = new Elysia()
  .get('/', handler)
```

Unlike `InferContext`, `InferHandler` requires a path and schema to define a handler type and can safely ensure type safety of a return type.

[Edit this page on GitHub](#)

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

[Previous page](#)
[Route](#)[Next page](#)
[Life Cycle](#)



CtrlK

[Cheat Sheet](#) [Plugins](#) [Blog](#)

Getting Started

[At Glance](#)
[Quick Start](#)
[Tutorial](#)
[Key Concept](#)
[Table of Content](#)

Essential

[Route](#)
[Handler](#)
[Life Cycle](#)
[Validation](#)
[Plugin](#)
[Best Practice](#)

Patterns

Recipe

Eden

Plugins

Integration