Outline

Key Concept

We highly recommend you to read this page before starting to use Elysia.

Although Elysia is a simple library, it has some key concepts that you need to understand to use it effectively.

This page covers most important concepts of Elysia that you should know.

Everything is a component

Every Elysia instance is a component.

A component is a plugin that could plug into other instances.

It could be a router, a store, a service, or anything else.

ts

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

```ts
const router = new Elysia()
    .use(store)
    .get('/increase', ({ store }) => store.visitor++)

const app = new Elysia()
    .use(router)
    .get('/', ({ store }) => store)
    .listen(3000)
```

This forces you to break down your application into small pieces, making it easy for you to add or remove features.

Learn more about this in plugin.

Scope

By default, event/life-cycle in each instance is isolated from each other.

ts

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

const ip = new Elysia()
    .derive(({ server, request }) => ({
        ip: server?.requestIP(request)
```

```
const server = new Elysia()
    .use(ip)
    .get('/ip', ({ ip }) => ip)
Property 'ip' does not exist on type '{ body: unknown; query:
    .listen(3000)
```

In this example, the `ip` property is only shared in its own instance but not in the `server` instance.

To share the lifecycle, in our case, an `ip` property with `server` instance, we need to **explicitly say** that it could be shared.

ts

```
import { Elysia } from 'elysia'

const ip = new Elysia()
    .derive(
+       { as: 'global' },
        ({ server, request }) => ({
            ip: server?.requestIP(request)
        })
    )
    .get('/ip', ({ ip }) => ip)
```

```
.get('/ip', ({ ip }) => ip)
.listen(3000)
```

In this example, `ip` property is shared between `ip` and `server` instance because we define it as `global`.

This forces you to think about the scope of each property, preventing you from accidentally sharing the property between instances.

Learn more about this in scope.

Method Chaining

Elysia code should always use **method chaining**.

As Elysia type system is complex, every methods in Elysia returns a new type reference.

This is important to ensure type integrity and inference.

typescript

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

new Elysia()
    .state('build', 1)
```

```
build: number
```

```
.listen(3000)
```

In the code above, **state** returns a new **ElysiaInstance** type, adding a typed `build` property.

❌ Don't: Use Elysia without method chaining

Without using method chaining, Elysia doesn't save these new types, leading to no type inference.

typescript

```
import { Elysia } from 'elysia'

const app = new Elysia()

app.state('build', 1)

app.get('/', ({ store: { build } }) => build)
  Property 'build' does not exist on type '{}'.

app.listen(3000)
```

We recommend to **always use method chaining** to provide an accurate type inference.

By default, each instance will be re-executed every time it's applied to another instance.

This can cause a duplication of the same method being applied multiple times, whereas some methods, like **lifecycle** or **routes**, should only be called once.

To prevent lifecycle methods from being duplicated, we can add **a unique identifier** to the instance.

ts

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

const ip = new Elysia({ name: 'ip' })
    .derive(
        { as: 'global' },
        ({ server, request }) => ({
            ip: server?.requestIP(request)
        })
    )
    .get('/ip', ({ ip }) => ip)

const router1 = new Elysia()
    .use(ip)
    .get('/ip-1', ({ ip }) => ip)
```

```
    .get('/ip-2', ({ ip }) => ip)

const server = new Elysia()
    .use(router1)
    .use(router2)
```

This will prevent the `ip` property from being called multiple times by applying deduplication using a unique name.

Once `name` is provided, the instance will become a **singleton**, allowing Elysia to apply plugin deduplication.

This allows us to reuse the same instance multiple times without the performance penalty.

This forces you to think about the dependencies of each instance, allowing for easily applied migrations or refactoring.

Learn more about this in plugin deduplication.

Type Inference

```
body: {
    name: string;
}
```

Elysia has a complex type system types from the instance.

```
const app = new Elysia()
    .post('/', ({ body }) => body, {
        body: t.Object({
            name: t.String()
        })
    })
```

If possible, **always use an inline function** to provide an accurate type inference.

If you need to apply a separate function, eg. MVC's controller pattern, it's recommended to destructure properties from inline function to prevent unnecessary type inference.

ts

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

abstract class Controller {
    static greet({ name }: { name: string }) {
        return 'hello ' + name
    }
}
```

```
    body: t.Object({
        name: t.String()
    })
})
```

Learn more about this in Best practice: MVC Controller.

Edit this page on GitHub

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

Previous page
Tutorial

Next page
Table of Content