Best Practice

Elysia is a pattern-agnostic framework, leaving the decision of which coding patterns to use up to you and your team.

However, there are several concern from trying to adapt an MVC pattern (Model-View-Controller) with Elysia, and found it's hard to decouple and handle types.

This page is a guide to on how to follows Elysia structure best practice combined with MVC pattern but can be adapted to any coding pattern you like.

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.

```
new Elysia()
    .state('build', 1)
+   // Store is strictly typed
    .get('/', ({ store: { build } }) => build)
    .listen(3000)
```

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

❌ 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 '{}'.
```

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

Controller

> 1 Elysia instance = 1 controller

Elysia does a lot to ensure type integrity, if you pass an entire `Context` type to a controller, these might be the problems:

1. Elysia type is complex and heavily depends on plugin and multiple level of chaining.
2. Hard to type, Elysia type could change at anytime, especially with decorators, and store
3. Type casting may lead to a loss of type integrity or an inability to ensure consistency between types and runtime code.
4. This makes it more challenging for Sucrose *(Elysia's "kind of" compiler)* to statically analyze your code

❌ Don't: Create a separate controller

Don't create a separate controller, use Elysia itself as a controller instead:

```typescript
abstract class Controller {
    static root(context: Context) {
        return Service.doStuff(context.stuff)
    }
}

// ❌ Don't
new Elysia()
    .get('/', Controller.hi)
```

By passing an entire `Controller.method` to Elysia is an equivalent of having 2 controllers passing data back and forth. It's against the design of framework and MVC pattern itself.

✅ Do: Use Elysia as a controller

Instead treat an Elysia instance as a controller itself instead.

typescript

```typescript
import { Elysia } from 'elysia'
import { Service } from './service'

new Elysia()
    .get('/', ({ stuff }) => {
```

Testing

You can test your controller using `handle` to directly call a function (and it's lifecycle)

typescript

```typescript
import { Elysia } from 'elysia'
import { Service } from './service'

import { describe, it, should } from 'bun:test'

const app = new Elysia()
    .get('/', ({ stuff }) => {
        Service.doStuff(stuff)

        return 'ok'
    })

describe('Controller', () => {
    it('should work', async () => {
        const response = await app
            .handle(new Request('http://localhost/'))
            .then((x) => x.text())

        expect(response).toBe('ok')
```

You may find more information about testing in Unit Test.

Service

Service is a set of utility/helper functions decoupled as a business logic to use in a module/controller, in our case, an Elysia instance.

Any technical logic that can be decoupled from controller may live inside a **Service**.

There're 2 types of service in Elysia:

1. Non-request dependent service
2. Request dependent service

✅ Do: Non-request dependent service

This kind of service doesn't need to access any property from the request or `Context`, and can be initiated as a static class same as usual MVC service pattern.

typescript

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

```
            return number

        return Service.fibo(number - 1) + Service.fibo(number -
    }
}

new Elysia()
    .get('/fibo', ({ body }) => {
        return Service.fibo(body)
    }, {
        body: t.Numeric()
    })
```

If your service doesn't need to store a property, you may use `abstract class` and `static` instead to avoid allocating class instance.

Request Dependent Service

This kind of service may require some property from the request, and should be **initiated as an Elysia instance.**

❌ Don't: Pass entire `Context` to a service

Context is a highly dynamic type that can be inferred from Elysia instance.

typescript

```typescript
import type { Context } from 'elysia'

class AuthService {
    constructor() {}

    // ❌ Don't do this
    isSignIn({ cookie: { session } }: Context) {
        if (session.value)
            return error(401)
    }
}
```

As Elysia type is complex, and heavily depends on plugin and multiple level of chaining, it can be challenging to manually type as it's highly dynamic.

✅ Do: Use Elysia instance as a service

We recommended to use Elysia instance as a service to ensure type integrity and inference:

typescript

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

// ✅ Do
const AuthService = new Elysia({ name: 'Service.Auth' })
```

```
            user: session.value
        }
    }))
    .macro(({ onBeforeHandle }) => ({
        // This is declaring a service method
        isSignIn(value: boolean) {
            onBeforeHandle(({ Auth, error }) => {
                if (!Auth?.user || !Auth.user) return error(401
            })
        }
    }))

const UserController = new Elysia()
    .use(AuthService)
    .get('/profile', ({ Auth: { user } }) => user, {
        isSignIn: true
    })
```

TIP

Elysia handle plugin deduplication by default so you don't have to worry about performance, as it's going to be Singleton if you specified a **"name"** property.

⚠️ Infers Context from Elysia instance

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

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

class AuthService {
    constructor() {}

    // ✅ Do
    isSignIn({ cookie: { session } }: InferContext<typeof setup
        if (session.value)
            return error(401)
    }
}
```

However we recommend to avoid this if possible, and use Elysia as a service instead.

You may find more about InferContext in Essential: Handler.

Model

and validate it at runtime.

❌ Don't: Declare a class instance as a model

Do not declare a class instance as a model:

typescript

```typescript
// ❌ Don't
class CustomBody {
    username: string
    password: string

    constructor(username: string, password: string) {
        this.username = username
        this.password = password
    }
}

// ❌ Don't
interface ICu
    username
    password
}
```

```typescript
type CustomBody = {
    username: string;
    password: string;
}
```

✅ Do: Use Elysia's validation system

typescript

```typescript
// ✅ Do
import { Elysia, t } from 'elys

const customBody = t.Object({
    username: t.String(),
    password: t.String()
})

// Optional if you want to get the type of the model
// Usually if we didn't use the type, as it's already inferred
type CustomBody = typeof customBody.static



export { customBody }
```

```
body: {
    username: string;
    password: string;
}
```

We can get type of model by using `typeof` with `.static` property from the model.

Then you can use the `CustomBody` type to infer the type of the request body.

typescript

```typescript
// ✅ Do
new Elysia()
    .post('/login', ({ body }) => {
})
```

❌ Don't: Declare type separate from the model

Do not declare a type separate from the model, instead use `typeof` with `.static` property to get the type of the model.

typescript

```typescript
// ❌ Don't
import { Elysia, t } from 'elysia'

const customBody = t.Object({
    username: t.String(),
    password: t.String()
})

type CustomBody = {
    username: string
    password: string
}

// ✅ Do
const customBody = t.Object({
    username: t.String(),
    password: t.String()
```

```typescript
type CustomBody = typeof customBody.static
```

Group

You can group multiple models into a single object to make it more organized.

typescript

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

export const AuthModel = {
    sign: t.Object({
        username: t.String(),
        password: t.String()
    })
}

const models = AuthModel.models
```

```
body: {
    username: string;
    password: string;
}
```

Model Injection

Though this is optional, if you are strictly followir

want to inject like a service into a controller. We recommended using Elysia reference model

Using Elysia's model reference

```
const customBody = t.Object({
    username: t.String(),
    password: t.String()
})

const AuthModel = new Elysia()
    .model({
        'auth.sign': customBody
    })

const models = AuthModel.models

const UserController = new Elysia({ prefix: '/auth' })
    .use(AuthModel)
    .post('/sign-in', async ({ body, cookie: { session } }) ⇒

        return true
    }, {
        body: 'auth.sign'
    })
```

This approach provide several benefits:

1. Allow us to name a model and provide auto-completion.
2. Modify schema for later usage, or perform a remap.

registration.

Reuse a plugin

It's ok to reuse plugins multiple time to provide type inference.

Elysia handle plugin deduplication automatically by default, and the performance is negligible.

To create a unique plugin, you may provide a **name** or optional **seed** to an Elysia instance.

typescript

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

const plugin = new Elysia({ name: 'my-plugin' })
    .decorate("type", "plugin")

const app = new Elysia()
    .use(plugin)
    .use(plugin)
    .use(plugin)
    .use(plugin)
    .listen(3000)
```

Edit this page on GitHub

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

Previous page
Plugin

Next page
Macro