Documentation

Hummingbird Document… / Middleware

Article

Middleware

Processing requests and responses outside of request handlers.

Overview

Middleware can be used to edit requests before they are forwarded to the router, edit the responses returned by the route handlers or even shortcut the router and return their own responses. Middleware is added to the application as follows.

```
let router = Router()
router.add(middleware: MyMiddlware())
```

In the example above the `MyMiddleware` is applied to every request that comes into the server.

Groups

Middleware can also be applied to a specific set of routes using groups. Below is a example of applying an authentication middleware `BasicAuthenticatorMiddleware` to routes that need protected.

```
let router = Router()
router.put("/user", createUser)
router.group()
    .add(middleware: BasicAuthenticatorMiddleware())
    .post("/user", loginUser)
```

The first route that calls `createUser` does not have the `BasicAuthenticator Middleware` applied to it. But the route calling `loginUser` which is inside the group does have the middleware applied.

Middleware result builder

You can add multiple middleware to the router using the middleware stack result builder `MiddlewareFixedTypeBuilder`.

```
let router = Router()
router.add {
    LogRequestsMiddleware()
    MetricsMiddleware()
    TracingMiddleware()
}
```

This gives a slight performance boost over adding them individually.

Writing Middleware

All middleware has to conform to the protocol `RouterMiddleware`. This requires one function `handle(_:context:next)` to be implemented. At some point in this function unless you want to shortcut the router and return your own response you should call `next(request, context)` to continue down the middleware stack and return the result, or a result processed by your middleware.

The following is a simple logging middleware that outputs every URI being sent to the server

```
public struct LogRequestsMiddleware<Context: RequestContext>: RouterMiddlewa
    public func handle(_ request: Request, context: Context, next: (Request,
        // log request URI
        context.logger.log(level: .debug, String(describing:request.uri.path
        // pass request onto next middleware or the router and return respon
        return try await next(request, context)
    }
}
```

See Also

Related Documentation

`protocol RouterMiddleware`

Version of `MiddlewareProtocol` whose Input is `Request` and output is `Response`.

Hummingbird Server

Router

The router directs requests to their handlers based on the contents of their path.

Request Decoding

Decoding of Requests with JSON content and other formats.

Response Encoding

Writing Responses using JSON and other formats.

Request Contexts

Controlling contextual data provided to middleware and route handlers

Error Handling

How to build errors for the server to return.

Logging, Metrics and Tracing

Considered the three pillars of observability, logging, metrics and tracing provide different ways of viewing how your application is working.

Result Builder Router

Building your router using a result builder.

Server protocol

Support for TLS and HTTP2 upgrades

Service Lifecycle

Integration with Swift Service Lifecycle

- Testing

  Using the HummingbirdTesting framework to test your application

- Persistent data

  How to persist data between requests to your server.

- Migrating to Hummingbird v2

  Migration guide for converting Hummingbird v1 applications to Hummingbird v2