Documentation

Hummingbird Document… / Request Contexts

Article

Request Contexts

Controlling contextual data provided to middleware and route handlers

Overview

All request handlers and middleware handlers have two function parameters: the request and a context. The context provides contextual data for processing your request. The context parameter is a generic value which must conform to the protocol `RequestContext`. This requires a minimal set of values needed by Hummingbird to process your request. This includes a `Logger`, request decoder, response encoder and the resolved endpoint path.

When you create your `Router` you provide the request context type you want to use. If you don't provide a context it will default to using `BasicRequestContext` the default implementation of a request context provided by Hummingbird.

```
let router = Router(context: MyRequestContext.self)
```

Creating a context type

As mentioned above your context type must conform to `RequestContext`. This requires an `init(source:)` and a single member variable `coreContext`.

```
struct MyRequestContext: RequestContext {
    var coreContext: CoreRequestContextStorage
```

```
    init(source: Source) {
        self.coreContext = .init(source: source)
    }
}
```

The `CoreRequestContextStorage` holds the base set of information needed by the Hummingbird `Router` to process a `Request`.

The `init` takes one parameter of type `Source`. `Source` is an associatedtype for the `RequestContext` protocol and provides setup data for the `RequestContext`. By default this is set to `ApplicationRequestContextSource` which provides access to the `Channel` that created the request.

If you are using HummingbirdLambda your RequestContext will need to conform to `LambdaRequestContext` and in that case the `Source` is a `LambdaRequestContextSource` which provide access to the `Event` that triggered the lambda and the `LambdaContext` from swift-aws-lambda-runtime.

Encoding/Decoding

By default request decoding and response encoding uses `JSONDecoder` and `JSONEncoder` respectively. You can override this by setting the `requestDecoder` and `responseEncoder` member variables in your `RequestContext`. Below we are setting the `requestDecoder` and `responseEncoder` to a decode/encode JSON with a `date DecodingStratrgy` of seconds since 1970. The default in Hummingbird is ISO8601.

```
struct MyRequestContext: RequestContext {
    /// Set request decoder to be JSONDecoder with alternate dataDecodingStr
    var requestDecoder: MyDecoder {
        var decoder = JSONDecoder()
        decoder.dateEncodingStrategy = .secondsSince1970
        return decoder
    }
    /// Set response encoder to be JSONEncode with alternate dataDecodingStr
    var responseEncoder: MyEncoder {
        var encoder = JSONEncoder()
        encoder.dateEncodingStrategy = .secondsSince1970
        return encoder
    }
```

```
}
```

You can find out more about request decoding and response encoding in Request Decoding and Response Encoding.

Passing data forward

The other reason for using a custom context is to pass data you have extracted in a middleware to subsequent middleware or the route handler.

```
/// Example request context with an additional field
struct MyRequestContext: RequestContext {
    var coreContext: CoreRequestContextStorage
    var additionalData: String?

    init(source: Source) {
        self.coreContext = .init(source: source)
        self.additionalData = nil
    }
}

/// Middleware that sets the additional field in
struct MyMiddleware: MiddlewareProtocol {
    func handle(
        _ request: Request,
        context: MyRequestContext,
        next: (Request, MyRequestContext) async throws -> Response
    ) async throws -> Response {
        var context = context
        context.additionalData = getData(request)
        return try await next(request, context)
    }
}
```

Now anything run after `MyMiddleware` can access the `additionalData` set in `MyMiddleware`.

Using RequestContextSource

You can also use the RequestContext to store information from the `RequestContextSource`. If you are running a Hummingbird server then this contains the Swift NIO `Channel` that generated the request. Below is an example of extracting the remote IP from the Channel and passing it to an endpoint.

```
/// RequestContext that includes a copy of the Channel that created it
struct AppRequestContext: RequestContext {
    var coreContext: CoreRequestContextStorage
    let channel: Channel

    init(source: Source) {
        self.coreContext = .init(source: source)
        self.channel = source.channel
    }

    /// Extract Remote IP from Channel
    var remoteAddress: SocketAddress? { self.channel.remoteAddress }
}

let router = Router(context: AppRequestContext.self)
router.get("ip") { _, context in
    guard let ip = context.remoteAddress else { throw HTTPError(.badRequest)
    return "Your IP is \(ip)"
}
```

Authentication Middleware

The most obvious example of this is passing user authentication information forward. The authentication framework from HummingbirdAuth makes use of this. If you want to use the authentication and sessions middleware your context will also need to conform to `AuthRequestContext`.

```
public struct MyRequestContext: AuthRequestContext {
    public var coreContext: CoreRequestContextStorage
    // required by AuthRequestContext
```

```
    public var identity: User?

    public init(source: Source) {
        self.coreContext = .init(source: source)
        self.identity = nil
    }
}
```

HummingbirdAuth does provide `BasicAuthRequestContext`: a default implementation of `AuthRequestContext`.

See Also

Related Documentation

`protocol RequestContext`

Protocol that all request contexts should conform to. A RequestContext is a statically typed metadata container for information that is associated with a `Request`, and is therefore instantiated alongside the request.

`protocol AuthRequestContext`

Protocol that all request contexts should conform to if they want to support authentication middleware

`struct BasicRequestContext`

Implementation of a basic request context that supports everything the Hummingbird library needs

`struct CoreRequestContextStorage`

Request context values required by Hummingbird itself.

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.

- Middleware

  Processing requests and responses outside of request 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