

Documentation

[Hummingbird Document...](#) / Response Encoding

Article

Response Encoding

Writing Responses using JSON and other formats.



Overview

Hummingbird uses Codable to encode responses. If your router handler returns a type conforming to [ResponseEncodable](#) this will get converted to a [Response](#) using the encoder [responseEncoder](#) parameter of your [RequestContext](#). By default this is set to create a JSON Response using JSONEncoder that comes with Swift Foundation.

```
struct User: ResponseEncodable {  
    let email: String  
    let name: String  
}  
  
router.get("user") { request, _ -> User in  
    let user = User(email: "js@email.com", name: "John Smith")  
    return user  
}
```

With the above code and the default JSON encoder you will get a response with header `content-type` set to `application/json; charset=utf-8` and body

```
{"email":"js@email.com","name":"John Smith"}
```

Date encoding

As mentioned above the default is to use `JSONEncoder` for encoding Response bodies. This default is also set to use ISO 8601 dates in the form `YYYY-MM-DDThh:mm:ssZ`. If you are decoding responses from a Hummingbird server in a Swift app using `JSONDecoder` you can parse dates using ISO 8601 by setting `JSONDecoder.dateDecodingStrategy` to `.iso8601`.

Setting up a custom encoder

If you want to use a different format, a different JSON encoder or want to support multiple formats, you need to setup you own `responseEncoder` in a custom request context. Your response encoder needs to conform to the `ResponseEncoder` protocol which has one requirement `encode(_ :from:context:)`. For instance Hummingbird also includes a encoder for URL encoded form data. Below you can see a custom request context setup to use `URLEncodedFormEncoder` for response encoding. The router is then initialized with this context. Read [Request Contexts](#) to find out more about request contexts.

```
struct URLEncodedRequestContext: RequestContext {
    var responseEncoder: URLEncodedFormEncoder { .init() }
    ...
}
let router = Router(context: URLEncodedRequestContext.self)
```

Encoding based on Request headers

Because the original request is supplied to the `ResponseEncoder`. You can make encoding decisions based on headers in the request. In the example below we are encoding using either the `JSONEncoder` or `URLEncodedFormEncoder` based on the “accept” header from the request.

```
struct MyResponseEncoder: ResponseEncoder {
    func encode(_ value: some Encodable, from request: Request, context: some RequestContext) throws -> Response {
        guard let header = request.headers[values: .accept].first else { throw HTTPError(.unsupportedMediaType) }
        guard let mediaType = MediaType(from: header) else { throw HTTPError(.unsupportedMediaType) }
        switch mediaType {
            case .applicationJson:
                // ...
            case .applicationFormUrlencoded:
                // ...
        }
    }
}
```

```
        return try JSONEncoder().encode(value, from: request, context: c
    case .applicationUrlEncoded:
        return try URLEncodedFormEncoder().encode(value, from: request,
    default:
        throw HTTPError(.badRequest)
    }
}
```

See Also

Related Documentation

`protocol ResponseEncoder`

protocol for encoders generating a `Response`

`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.

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.

Request Contexts

Controlling contextual data provided to middleware and route handlers

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