

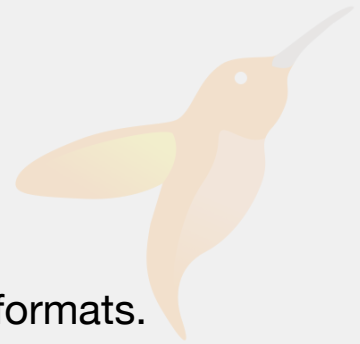
Documentation

[Hummingbird Document...](#) / Request Decoding

Article

Request Decoding

Decoding of Requests with JSON content and other formats.



Overview

Hummingbird uses `Codable` to decode requests. It defines what decoder to use via the `requestDecoder` parameter of your `RequestContext`. By default this is set to decode JSON, using `JSONDecoder` that comes with Swift Foundation.

Requests are converted to Swift objects using the `decode(as:context:)` method in the following manner.

```
struct User: Decodable {
    let email: String
    let firstName: String
    let surname: String
}

router.post("user") { request, context -> HTTPResponse.Status in
    // decode user from request
    let user = try await request.decode(as: User.self, context: context)
    // create user and if ok return `.ok` status
    try await createUser(user)
    return .ok
}
```

Like the standard `Codable` decode functions `Request.decode(as:context:)` can throw an error if decoding fails. The decode function is also async as the request body is an

asynchronous sequence of `ByteBuffers`. We need to collate the request body into one buffer before we can decode it.

Date decoding

As mentioned above the default is to use `JSONDecoder` for decoding Request bodies. This default is also set to use ISO 8601 dates in the form `YYYY-MM-DDThh:mm:ssZ`. If you are generating requests for a Hummingbird server in a Swift app using `JSONEncoder` you can output ISO 8601 dates by setting `JSONEncoder.dateEncodingStrategy` to `.iso8601`.

Setting up a custom decoder

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

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

Decoding based on Request headers

Because the full request is supplied to the `RequestDecoder`. You can make decoding decisions based on headers in the request. In the example below we are decoding using either the `JSONDecoder` or `URLEncodedFormDecoder` based on the “content-type” header.

```
struct MyRequestDecoder: RequestDecoder {
    func decode<T>(_ type: T.Type, from request: Request, context: some Requ
```

```
guard let header = request.headers[.contentType] else { throw HTTPError
guard let mediaType = MediaType(from: header) else { throw HTTPError
switch mediaType {
case .applicationJson:
    return try await JSONDecoder().decode(type, from: request, conte
case .applicationUrlEncoded:
    return try await URLEncodedFormDecoder().decode(type, from: requ
default:
    throw HTTPError(.badRequest)
}
}
}
```

See Also

Related Documentation

`protocol RequestDecoder`

protocol for decoder deserializing from a Request body

`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.

Response Encoding

Writing Responses using JSON 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
