

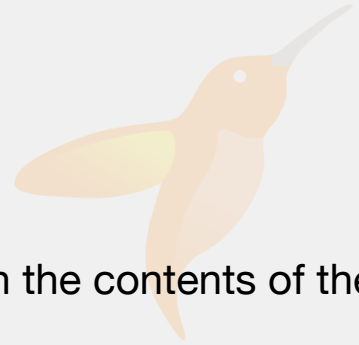
Documentation

[Hummingbird Document...](#) / Router

Article

Router

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



Overview

The default router that comes with Hummingbird uses a Trie based lookup. Routes are added using the function `on(_:_:method:use:)`. You provide the URI path, the method and the handler function. Below is a simple route which returns “Hello” in the body of the response.

```
let router = Router()
router.on("/hello", method: .GET) { request, context in
    return "Hello"
}
```

If you don't provide a path then the default is for it to be “/”.

Methods

There are shortcut functions for the most common HTTP methods. The above can be written as

```
let router = Router()
router.get("/hello") { request, context in
    return "Hello"
}
```

```
}
```

There are shortcuts for `put`, `post`, `head`, `patch` and `delete` as well.

Response generators

Route handlers are required to return a type conforming to the `ResponseGenerator` protocol. The `ResponseGenerator` protocol requires a type to be able to generate an `Response`. For example `String` has been extended to conform to `ResponseGenerator` by returning an `Response` with status `.ok`, a content-type header of `text-plain` and a body holding the contents of the `String`.

```
/// Extend String to conform to ResponseGenerator
extension String: ResponseGenerator {
    /// Generate response holding string
    public func response(from request: Request, context: some RequestContext)
        let buffer = ByteBuffer(string: self)
        return Response(
            status: .ok,
            headers: [.contentType: "text/plain; charset=utf-8"],
            body: .init(byteBuffer: buffer)
        )
}
```

In addition to `String` `ByteBuffer`, `HTTPResponseStatus` and `Optional` have also been extended to conform to `ResponseGenerator`.

It is also possible to extend `Codable` objects to generate a `Response` by conforming these objects to `ResponseEncodable`. The object will use the response encoder attached to your context to encode these objects. If an object conforms to `ResponseEncodable` then also so do arrays and dictionaries of these objects. Read more about generating Responses via `Codable` in `Response Encoding`.

Wildcards

You can use wildcards to match sections of a path component.

A single * will skip one path component

```
router.get("/files/*") { request, context in  
    return request.uri.description  
}
```

Will match

```
GET /files/test  
GET /files/test2
```

A * at the start of a route component will match all path components with the same suffix.

```
router.get("/files/*.jpg") { request, context in  
    return request.uri.description  
}
```

Will work for

```
GET /files/test.jpg  
GET /files/test2.jpg
```

A * at the end of a route component will match all path components with the same prefix.

```
router.get("/files/image.*") { request, context in  
    return request.uri.description  
}
```

Will work for

```
GET /files/image.jpg  
GET /files/image.png
```

A `**` will match and capture all remaining path components.

```
router.get("/files/**") { request, context in
    // return catchAll captured string
    return context.parameters.getCatchAll().joined(separator: "/")
}
```

The above will match routes and respond as follows

```
GET /files/image.jpg returns "image.jpg" in the response body
GET /files/folder/image.png returns "folder/image.png" in the response body
```

Parameter Capture

You can extract parameters out of the URI by prefixing the path with a colon. This indicates that this path section is a parameter. The parameter name is the string following the colon. You can get access to the URI extracted parameters from the context. This example extracts an id from the URI and uses it to return a specific user. so `"/user/56"` will return user with id 56.

```
router.get("/user/:id") { request, context in
    let id = context.parameters.get("id", as: Int.self) else { throw HTTPError
    return getUser(id: id)
}
```

In the example above if I fail to access the parameter as an `Int` then I throw an error. If you throw an `HTTPError` it will get converted to a valid HTTP response.

The parameter name in your route can also be of the form `{id}`, similar to OpenAPI specifications. With this form you can also extract parameter values from the URI that are prefixes or suffixes of a path component.

```
router.get("/files/{image}.jpg") { request, context in
    let imageName = context.parameters.get("image") else { throw HTTPError(.
    return getImage(image: imageName)
}
```

In the example above we match all paths that are a file with a jpg extension inside the files folder and then call a function with that image name.

Query parameters

The Request url query parameters are available via a number of methods from Request member `uri`. You can get the full query string using `query`. You can get the query string broken up into individual parameters and percent decoded using `queryParameters`.

```
router.get("/user") { request, context in
    // extract parameter from URL of form /user?id={userId}
    let id = request.uri.queryParameters.get("id", as: Int.self) else { thro
    return getUser(id: id)
}
```

You can also use `decodeQuery(as: context:)` to convert the query parameters into a Swift object. As with `URI.queryParameters` the values will be percent decoded.

```
struct Coordinate: Decodable {
    let x: Double
    let y: Double
}

router.get("tile") { request, context in
    // create `Coordinate` from query parameters in URL of form /tile?x={xCo
    let position = request.uri.decodeQuery(as: Coordinate.self, context: con
    return tiles.get(at: position)
}
```

Groups

Routes can be grouped together in a RouterGroup. These allow for you to prefix a series of routes with the same path and more importantly apply middleware to only those routes. The example below is a group that includes five handlers all prefixed with the path “/todos”.

```
let app = Application()
router.group("/todos")
  .put(use: createTodo)
  .get(use: listTodos)
  .get("{id}", getTodo)
  .patch("{id}", editTodo)
  .delete("{id}", deleteTodo)
```

RequestContext transformation

The RequestContext can be transformed for the routes in a route group. The Request Context you are converting to needs to conform to ChildRequestContext. This requires a parent context ie the RequestContext you are converting from and a init(context:) function to perform the conversion.

```
struct MyNewRequestContext: ChildRequestContext {
  typealias ParentContext = MyRequestContext
  init(context: ParentContext) throws {
    self.coreContext = context.coreContext
    ...
  }
}
```

Once you have defined how to perform the transform from your original RequestContext the conversion is added as follows

```
let app = Application(context: MyRequestContext.self)
router.group("/todos", context: MyNewRequestContext.self)
  .put(use: createTodo)
  .get(use: listTodos)
```

Route Collections

A [RouteCollection](#) is a collection of routes and middleware that can be added to a Router in one go. It has the same API as RouterGroup, so can have groups internal to the collection to allow for Middleware to be applied to only sub-sections of the Route Collection.

```
struct UserController<Context: RequestContext> {  
    var routes: RouteCollection<Context> {  
        let routes = RouteCollection()  
        routes.post("signup", use: signUp)  
        routes.group("login")  
            .add(middleware: BasicAuthenticationMiddleware())  
            .post(use: login)  
        return routes  
    }  
}
```

You add the route collection to your router using [addRoutes\(_:atPath:\)](#).

```
let router = Router()  
router.add("users", routes: UserController().routes)
```

Request Body

By default the request body is an AsyncSequence of ByteBuffers. You can treat it as a series of buffers or collect it into one larger buffer.

```
// process each buffer in the sequence separately  
for try await buffer in request.body {  
    process(buffer)  
}
```

```
// collect all the buffers in the sequence into a single buffer  
let buffer = try await request.body.collate(maxSize: maximumBufferSizeAllowe
```

Once you have read the sequence of buffers you cannot read it again. If you want to read the contents of a request body in middleware before it reaches the route handler, but still have it available for the route handler you can use `Request.collectBody(upTo:)`. After this point though the request body cannot be treated as a sequence of buffers as it has already been collapsed into a single buffer.

Any errors you receive while iterating the request body should always be propagated further up the callstack. It is fine to catch the errors but you should rethrow them once you are done with them, so they can be passed back to `Application` to be dealt with according.

Writing the response body

The response body is returned back to the server as a closure that will write the body. The closure is provided with a writer type conforming to `ResponseBodyWriter` and the closure uses this to write the buffers that make up the body. In most cases you don't need to know this as `ResponseBody` has initializers that take a single `ByteBuffer`, a sequence of `ByteBuffers` and an `AsyncSequence` of `ByteBuffers` which covers most of the kinds of responses.

In the situation where you need something a little more flexible you can use the closure form. Below is a `ResponseBody` that consists of 10 buffers of random data written with a one second pause between each buffer.

```
let responseBody = ResponseBody { writer in
    for _ in 0..<10 {
        try await Task.sleep(for: .seconds(1))
        let buffer = (0..
```

Once you have finished writing your response body you need to tell the writer you have finished by calling `finish(_)`. At this point you can write trailing headers by passing them to the `finish` function. NB Trailing headers are only sent if your response body is chunked and does not include a content length header.

Editing response in handler

The standard way to provide a custom response from a route handler is to return a `Response` from that handler. This method loses a lot of the automation of encoding responses, generating the correct status code etc.

Instead you can return what is called a `EditedResponse`. This includes a type that can generate a response on its own via the `ResponseGenerator` protocol and includes additional edits to the response.

```
router.post("test") { request, _ -> EditedResponse in
    return .init(
        status: .accepted,
        headers: [.contentType: "application/json"],
        response: #"{"test": "value"}"#
    )
}
```

See Also

Related Documentation

`struct Request`

Holds all the values required to process a request

`struct Response`

Holds all the required to generate a HTTP Response

`class Router`

Create rules for routing requests and then create `Responder` that will follow these rules.












`class RouteCollection`

Collection of routes

`struct RouterGroup`

Used to group together routes under a single path. Additional middleware can be added to the endpoint and each route can add a suffix to the endpoint path

Hummingbird Server

-  **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
 -  **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