Documentation

Hummingbird Document… / Result Builder Router

Article

Result Builder Router

Building your router using a result builder.

Overview

`HummingbirdRouter` provides an alternative to the standard trie based `Router` that is in the Hummingbird module. `RouterBuilder` uses a result builder to construct your router.

```
let router = RouterBuilder(context: BasicRouterRequestContext.self) {
    CORSMiddleware()
    Route(.get, "health") { _,_ in
        HTTPResponse.Status.ok
    }
    RouteGroup("user") {
        BasicAuthenticationMiddleware()
        Route(.post, "login") { request, context in
            ...
        }
    }
}
```

RequestContext

To be able to use the result builder router you need to provide a `RequestContext` that conforms to `RouterRequestContext`. This contains an additional support struct `RouterBuilderContext` required by the result builder.

```
struct MyRequestContext: RouterRequestContext {
    public var routerContext: RouterBuilderContext
    public var coreContext: CoreRequestContextStorage

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

Common Route Verbs

The common HTTP verbs: GET, PUT, POST, PATCH, HEAD, DELETE, have their own shortcut functions.

```
Route(.get, "health") { _,_ in
    HTTPResponse.Status.ok
}
```

can be written as

```
Get("health") { _,_ in
    HTTPResponse.Status.ok
}
```

Route middleware

Routes can be initialised with their own result builder as long as they end with a route `Handle` function that returns the response. This allows us to apply middleware to individual routes.

```
Post("login") {
    BasicAuthenticationMiddleware()
    Handle  { request, context in
```

```
        ...
    }
}
```

If you are not adding the handler inline you can add the function reference without the `Handle`.

```
@Sendable func processLogin(request: Request, context: MyContext) async thro
    // process login
}
RouterBuilder(context: BasicRouterRequestContext.self) {
    ...
    Post("login") {
        BasicAuthenticationMiddleware()
        processLogin
    }
}
```

RequestContext transformation

You can transform the `RequestContext` to a different type for a group of routes using `init(_:context:builder:)`. When you define the `RequestContext` type you are converting to you need to define how you initialize it from the original `RequestContext`.

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

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

```
let router = RouterBuilder(context: BasicRouterRequestContext.self) {
    RouteGroup("user", context: MyNewRequestContext.self) {
        BasicAuthenticationMiddleware()
        Route(.post, "login") { request, context in
            ...
        }
    }
}
```

Controllers

It is common practice to group routes into controller types that perform operations on a common type eg user management, CRUD operations for an asset type. By conforming your controller type to `RouterController` you can add the contained routes directly into your router eg

```
struct TodoController<Context: RouterRequestContext>: RouterController {
    var body: some RouterMiddleware<Context> {
        RouteGroup("todos") {
            Put(handler: self.put)
            Get(handler: self.get)
            Patch(handler: self.update)
            Delete(handler: self.delete)
        }
    }
}
let router = RouterBuilder(context: BasicRouterRequestContext.self) {
    TodoController()
}
```

Differences from trie router

There is one subtle difference between the result builder based `RouterBuilder` and the more traditional trie based `Router` that comes with `Hummingbird` and this is related to how middleware are processed in groups.

With the trie based `Router` a request is matched against an endpoint and then only runs the middleware applied to that endpoint.

With the result builder a request is processed by each element of the router result builder until it hits a route that matches its URI and method. If it hits a `RouteGroup` and this matches the current request uri path component then the request (with matched URI path components dropped) will be processed by the children of the `RouteGroup` including its middleware. The request path matching and middleware processing is done at the same time which means middleware only needs its parent `RouteGroup` paths to be matched for it to run.

See Also

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

- 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.

- 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