Documentation

Article

Migrating to Hummingbird v2

Migration guide for converting Hummingbird v1 applications to Hummingbird v2

Overview

In the short lifetime of the Hummingbird server framework there have been many major changes to the Swift language. Hummingbird v2 has been designed to take advantage of all the latest changes to Swift. In addition Hummingbird v1 was our first attempt at writing a server framework and we didn't necessarily get everything right, so v2 includes some changes where we feel we made the wrong design first time around. Below we cover most of the major changes in the library and how you should deal with them.

Symbol names

The first thing you will notice when moving to v2 are the symbol names. In Version 2 of Hummingbird we have removed the "HB" prefix from all the symbols.

SwiftNIO and Swift Concurrency

In the time that the Hummingbird server framework has been around there has been a seismic shift in the Swift language. When it was first in development the initial pitches for Swift Concurrency were only just being posted. It wasn't for another 9 months before we actually saw a release of Swift with any concurrency features. As features have become available we have tried to support them but the internals of Hummingbird were still SwiftNIO EventLoop based and held us back from providing full support for Concurrency.

Hummingbird v2 is now exclusively Swift concurrency based. All EventLoop based APIs have been removed.

Using EventLoop-based Libraries

If you have libraries you are calling into that still only provide EventLoop based APIs you can convert them to Swift concurrency using the `get` method from `EventLoopFuture`.

```
let value = try await eventLoopBasedFunction().get()
```

If you need to provide an `EventLoopGroup`, use either the one you provided to `Application.init` or `MultiThreadedEventLoopGroup.singleton`. And when you need an `EventLoop` use `EventLoopGroup.any`.

```
let service = MyService(eventLoopGroup: MultiThreadedEventLoopGroup.singleto
let result = try await service.doStuff(eventLoop: MultiThreadedEventLoopGrou
```

Otherwise any `EventLoopFuture` based logic you had will have to be converted to Swift concurrency. The advantage of this is, it should be a lot easier to read after.

Extending Application and Request

In Hummingbird v1 you could extend the `Application` and `Request` types to include your own custom data. This is no longer possible in version 2.

Application

In the case of the application we decided we didn't want to make `Application` this huge mega global that held everything. We have moved to a model of explicit dependency injection.

For each route controller you supply the dependencies you need at initialization, instead of extracting them from the application when you use them. This makes it clearer what dependencies you are using in each controller.

```
struct UserController {
    // The user authentication routes use fluent and session storage
    init(fluent: Fluent, sessions: SessionStorage) {
        ...
    }
}
```

Request and RequestContext

We have replaced extending of `Request` with a custom request context type that is passed along with the request. This means `Request` is just the HTTP request data (as it should be). The additional request context parameter will hold any custom data required. In situations in the past where you would use data attached to `Request`, you should now use the context.

```
router.get { request, context in
    // logger is attached to the context
    context.logger.info("The logger attached to the context includes the req
    // request decoder is attached to the context instead of the application
    let myObject = try await request.decode(as: MyObject.self, context: cont
}
```

The request context is a generic value. As long as it conforms to `RequestContext` it can hold anything you like.

```
/// Example request context with an additional data attached
struct MyRequestContext: RequestContext {
    // required by RequestContext
    var coreContext: CoreRequestContextStorage
    var additionalData: String?

    // required by RequestContext
    init(source: Source) {
        self.coreContext = .init(source: source)
        self.additionalData = nil
    }
}
```

When you create your router you pass in the request context type you'd like to use. If you don't pass one in it will default to using `BasicRequestContext` which provides enough data for the router to run but not much else.

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

> **Important**
>
> This feature is at the heart of Hummingbird 2, so we recommend reading our guide to Request Contexts.

Router

Instead of creating an application and adding routes to it, in v2 you create a router and add routes to it and then create an application using that router.

Hummingbird 1

```
let app = Application()
app.router.get { request in
    "hello"
}
```

Hummingbird 2

```
let router = Router()
router.get { request, context in
    "hello"
}
let app = Application(router: router
```

When we are passing in the router we are actually passing in a type that can build a `HTTPResponder` a protocol for a type with one function that takes a request and context and returns a response.

Router Builder

An alternative router is also provided in the HummingbirdRouter module. It uses a result builder to generate the router.

```
let router = RouterBuilder(context: MyContext.self) {
    // add logging middleware
    LogRequestsMiddleware(.info)
    // add route to return ok
    Get("health") { _,_ -> HTTPResponse.Status in
        .ok
    }
    // for all routes starting with '/user'
    RouteGroup("user") {
        // add router supplied by UserController
        UserController(fluent: fluent).routes()
    }
}
let app = Application(router: router)
```

Miscellaneous

Below is a list of other smaller changes that might catch you out

Request body streaming

In Hummingbird v1 it was assumed request bodies would be collated into one ByteBuffer and if you didn't want that to happen you had to flag the route to not collate your request body. In v2 this assumption has been reversed. It is assumed that request bodies are a stream of buffers and if you want to collate them into one buffer you need to call a method to do that.

To treat the request body as a stream of buffers

```
router.put { request, context in
    for try await buffer in request.body {
        process(buffer)
    }
}
```

To treat the request body as a single buffer.

```
router.put { request, context in
    let body = try await request.body.collate(maxSize: 1_000_000)
    process(body)
}
```

OpenAPI style URI capture parameters

In Hummingbird v1.3.0 partial path component matching and capture was introduced. For this a new syntax was introduced for parameter capture: `${parameter}` alongside the standard `:parameter` syntax. It has been decided to change the new form of the syntax to `{parameter}` to coincide with the syntax used by OpenAPI.

HummingbirdFoundation

HummingbirdFoundation has been merged into Hummingbird. It was felt the gains from separating out the code relying on Foundation were not enough for the awkwardness it created. Eventually we hope to limit our exposure to only the elements of Foundation that will be in FoundationEssentials module from the newly developed Swift Foundation.

Generic Application

`Application` is a generic type with two different type parameters. Passing around the concrete type is complex as you need to work out the type parameters. They might not be immediately obvious. Instead it is easier to pass around the opaque type `some ApplicationProtocol`.

```
func buildApplication() -> some ApplicationProtocol {
    ...
    let app = Application(router: router)
    return app
}
```

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.

- 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.