Documentation

Article

Persistent data

How to persist data between requests to your server.

Overview

If you are looking to store data between requests then the Hummingbird `persist` framework provides a key/value store. Each key is a string and the value can be any object that conforms to `Codable`.

Setup

At setup you need to choose your persist driver. Below we are using the in memory storage driver.

```
let persist = MemoryPersistDriver()
```

The persist drivers conform to `Service` from Swift Service Lifecycle and should either to added to the `Application` serivce group using `addServices(_:)` or added to an external managed `ServiceGroup`.

```
var app = Application(router: myRouter)
app.addServices(persist)
```

Usage

To create a new entry you can call `create`

```
try await persist.create(key: "mykey", value: MyValue)
```

If there is an entry for the key already then a `PersistError.duplicate` error will be thrown.

If you are not concerned about overwriting a previous key/value pair you can use

```
try await persist.set(key: "mykey", value: MyValue)
```

Both `create` and `set` have an `expires` parameter. With this parameter you can make a key/value pair expire after a certain time period. eg

```
try await persist.set(key: "sessionID", value: MyValue, expires: .hours(1))
```

To access values in the `persist` key/value store you use

```
let value = try await persist.get(key: "mykey", as: MyValueType.self)
```

This returns the value associated with the key or `nil` if that value doesn't exist. If the value is not of the expected type, this will throw `invalidConversion`.

And finally if you want to delete a key you can use

```
try await persist.remove(key: "mykey")
```

Drivers

The `persist` framework defines an API for storing key/value pairs. You also need a driver for the framework. `Hummingbird` comes with a memory based driver `MemoryPersistDriver` which will store these values in the memory of your server.

```
let persist = MemoryPersistDriver()
```

If you use the memory based driver the key/value pairs you store will be lost if your server goes down, also you will not be able to share values between server processes.

Redis

You can use Redis to store the `persists` key/value pairs with `RedisPersistDriver` from the `HummingbirdRedis` library. You would setup `persist` to use Redis as follows.

```
let redis = RedisConnectionPoolService(
    .init(hostname: redisHostname, port: 6379),
    logger: Logger(label: "Redis")
)
let persist = RedisPersistDriver(redisConnectionPoolService: redis)
```

Fluent

HummingbirdFluent also contains a `persist` driver for the storing the key/value pairs in a database. To setup the Fluent driver you need to have setup Fluent first. The first time you run with the fluent driver you should ensure you call `fluent.migrate()` after creating the `FluentPersistDriver` call has been made.

```
let fluent = Fluent(logger: Logger(label: "Fluent"))
fluent.databases.use(...)
let persist = await FluentPersistDriver(fluent: fluent)
// run migrations
if shouldMigrate {
    try await fluent.migrate()
}
```

See Also

Related Documentation

`protocol PersistDriver`

Protocol for driver supporting persistent Key/Value pairs across requests

`actor MemoryPersistDriver`

In memory driver for persist system for storing persistent cross request key/value pairs

`class FluentPersistDriver`

Fluent driver for persist system for storing persistent cross request key/value pairs

`struct RedisPersistDriver`

Redis driver for persist system for storing persistent cross request key/value pairs

`class PostgresPersistDriver`

Postgres driver for persist system for storing persistent cross request key/value pairs

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

- Migrating to Hummingbird v2

  Migration guide for converting Hummingbird v1 applications to Hummingbird v2