Documentation

API Collection

Jobs

Offload work your server would be doing to another server.

Overview

A Job consists of a payload and an execute method to run the job. Swift Jobs provides a framework for pushing jobs onto a queue and processing them at a later point. If the driver backing up the job queue uses persistent storage then a separate server can be used to process the jobs. The module comes with a driver that stores jobs in local memory and uses your current server to process the jobs, but there are also implementations in JobsRedis and JobsPostgres that implement the job queue using a Redis database or Postgres database.

Setting up a Job queue

Before you can start adding or processing jobs you need to setup a Jobs queue to push jobs onto. Below we create a job queue stored in local memory that will process four jobs concurrently.

```
let jobQueue = JobQueue(.memory, numWorkers: 4, logger: logger)
```

Creating a Job

Before you can start running jobs you need to define a job. A job definition requires an identifier for the job, the job parameters and the function that runs the job.

We use a struct conforming to `JobParameters` to define the job parameters and identifier.

```
struct SendEmailJobParameters: JobParameters {
    /// jobName is used to create the job identifier. It should be unique
    static let jobName = "SendEmail"
    let to: String
    let subject: String
    let body: String
}
```

Then we register the job with a job queue and also provide a closure that executes the job.

```
jobQueue.registerJob(parameters: SendEmailJobParameters.self) { parameters,
    try await myEmailService.sendEmail(to: parameters.to, subject: parameter
}
```

Now your job is ready to create. Jobs can be queued up using the function `push` on `JobQueue`.

```
let job = SendEmailJobParameters(
    to: "joe@email.com",
    subject: "Testing Jobs",
    message: "..."
)
jobQueue.push(job)
```

Processing Jobs

When you create a `JobQueue` the `numWorkers` parameter indicates how many jobs you want serviced concurrently by the job queue. If you want to activate these workers you need to add the job queue to your `ServiceGroup`.

```
let serviceGroup = ServiceGroup(
    services: [server, jobQueue],
    configuration: .init(gracefulShutdownSignals: [.sigterm, .sigint]),
    logger: logger
)
```

```
try await serviceGroup.run()
```

Or it can be added to the array of services that `Application` manages

```
let app = Application(...)
app.addServices(jobQueue)
```

If you want to process jobs on a separate server you will need to use a job queue driver that saves to some external storage eg `RedisJobQueue` or `PostgresJobQueue`.

Job Scheduler

The Jobs framework comes with a scheduler `Service` that allows you to schedule jobs to occur at regular times. Job schedules are defined using the `JobSchedule` type.

```
var jobSchedule = JobSchedule()
jobSchedule.addJob(BirthdayRemindersJob(), schedule: .daily(hour: 9))
jobSchedule.addJob(CleanupStaleSessionDataJob(), schedule: .weekly(day: .sun
```

To get your `JobSchedule` to schedule jobs on a `JobQueue` you need to create the scheduler `Service` and then add it to your `Application` service list or `ServiceGroup`.

```
var app = Application(router: router)
app.addService(jobSchedule.scheduler(on: jobQueue, named: "MyScheduler"))
```

Schedule types

A `Schedule` can be setup in a number of ways. It includes functions to trigger once every minute, hour, day, month, week day and functions to trigger on multiple minutes, hours, etc.

```
jobSchedule.addJob(TestJobParameters(), schedule: .hourly(minute: 30))
jobSchedule.addJob(TestJobParameters(), schedule: .yearly(month: 4, date: 1,
jobSchedule.addJob(TestJobParameters(), schedule: .onMinutes([0,15,30,45]))
jobSchedule.addJob(TestJobParameters(), schedule: .onDays([.saturday, .sunda
```

If these aren't flexible enough a `Schedule` can be setup using a five value crontab format. Most crontabs are supported but combinations setting both week day and date are not supported.

```
jobSchedule.addJob(TestJobParameters(), schedule: .crontab("0 12 * * *")) //
jobSchedule.addJob(TestJobParameters(), schedule: .crontab("0 */4 * * sat,su
jobSchedule.addJob(TestJobParameters(), schedule: .crontab("@daily")) // cro
```

Schedule accuracy

You can setup how accurate you want your scheduler to adhere to the schedule regardless of whether the scheduler is running or not. Obviously if your scheduler is not running it cannot schedule jobs. But you can use the `accuracy` parameter of a schedule to indicate what you want your scheduler to do once it comes back online after having been down.

Setting it to `.all` will schedule a job for every trigger point it missed eg if your scheduler was down for 6 hours and you had a hourly schedule it would push a job to the JobQueue for every one of those hours missed. Setting it to `.latest` will mean it only schedules a job for last trigger point if it was missed. If you don't set the value then it will default to `.latest`.

```
jobSchedule.addJob(TestJobParameters(), schedule: .hourly(minute: 30), accur
```

Topics

Reference

- JobsPostgres

  Postgres implementation for Hummingbird jobs framework

- JobsRedis

  Redis implementation for Hummingbird jobs framework

See Also

Related Documentation

`protocol JobParameters`

Defines job parameters and identifier

`struct JobQueue`

Job queue

`struct JobSchedule`

An array of Jobs with schedules detailing when they should be run