Queues

Queues are a powerful design pattern that help you deal with common application scaling and performance challenges. Some examples of problems that Queues can help you solve are:

- Smooth out processing peaks. For example, if users can initiate resource-intensive tasks at arbitrary times, you can add these tasks to a queue instead of performing them synchronously. Then you can have worker processes pull tasks from the queue in a controlled manner. You can easily add new Queue consumers to scale up the back-end task handling as the application scales up.
- Break up monolithic tasks that may otherwise block the Node.js event loop. For example, if a user request requires CPU intensive work like audio transcoding, you can delegate this task to other processes, freeing up user-facing processes to remain responsive.
- Provide a reliable communication channel across various services. For example, you can queue tasks (jobs) in one process or service, and consume them in another. You can be notified (by listening for status events) upon completion, error or other state changes in the job life cycle from any process or service. When Queue producers or consumers fail, their state is preserved and task handling can restart automatically when nodes are restarted.

Nest provides the `@nestjs/bull` package as an abstraction/wrapper on top of Bull, a popular, well supported, high performance Node.js based Queue system implementation. The package makes it easy to integrate Bull Queues in a Nest-friendly way to your application.

Bull uses Redis to persist job data, so you'll need to have Redis installed on your system. Because it is Redis-backed, your Queue architecture can be completely distributed and platform-independent. For example, you can have some Queue producers and consumers and listeners running in Nest on one (or several) nodes, and other producers, consumers and listeners running on other Node.js platforms on other network nodes.

This chapter covers the `@nestjs/bull` package. We also recommend reading the Bull documentation for more background and specific implementation details.

Installation

To begin using it, we first install the required dependencies.

```
$ npm install --save @nestjs/bull bull
```

Once the installation process is complete, we can import the `BullModule` into the root `AppModule`.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { BullModule } from '@nestjs/bull';

@Module({
  imports: [
    BullModule.forRoot({
      redis: {
```

```
        host: 'localhost',
        port: 6379,
      },
    }),
  ],
})
export class AppModule {}
```

The `forRoot()` method is used to register a `bull` package configuration object that will be used by all queues registered in the application (unless specified otherwise). A configuration object consist of the following properties:

- `limiter: RateLimiter` - Options to control the rate at which the queue's jobs are processed. See RateLimiter for more information. Optional.
- `redis: RedisOpts` - Options to configure the Redis connection. See RedisOpts for more information. Optional.
- `prefix: string` - Prefix for all queue keys. Optional.
- `defaultJobOptions: JobOpts` - Options to control the default settings for new jobs. See JobOpts for more information. Optional.
- `settings: AdvancedSettings` - Advanced Queue configuration settings. These should usually not be changed. See AdvancedSettings for more information. Optional.

All the options are optional, providing detailed control over queue behavior. These are passed directly to the Bull `Queue` constructor. Read more about these options here.

To register a queue, import the `BullModule.registerQueue()` dynamic module, as follows:

```
BullModule.registerQueue({
  name: 'audio',
});
```

> info **Hint** Create multiple queues by passing multiple comma-separated configuration objects to the `registerQueue()` method.

The `registerQueue()` method is used to instantiate and/or register queues. Queues are shared across modules and processes that connect to the same underlying Redis database with the same credentials. Each queue is unique by its name property. A queue name is used as both an injection token (for injecting the queue into controllers/providers), and as an argument to decorators to associate consumer classes and listeners with queues.

You can also override some of the pre-configured options for a specific queue, as follows:

```
BullModule.registerQueue({
  name: 'audio',
  redis: {
    port: 6380,
```

```
  },
});
```

Since jobs are persisted in Redis, each time a specific named queue is instantiated (e.g., when an app is started/restarted), it attempts to process any old jobs that may exist from a previous unfinished session.

Each queue can have one or many producers, consumers, and listeners. Consumers retrieve jobs from the queue in a specific order: FIFO (the default), LIFO, or according to priorities. Controlling queue processing order is discussed here.

Named configurations

If your queues connect to multiple different Redis instances, you can use a technique called **named configurations**. This feature allows you to register several configurations under specified keys, which then you can refer to in the queue options.

For example, assuming that you have an additional Redis instance (apart from the default one) used by a few queues registered in your application, you can register its configuration as follows:

```
BullModule.forRoot('alternative-config', {
  redis: {
    port: 6381,
  },
});
```

In the example above, `'alternative-config'` is just a configuration key (it can be any arbitrary string).

With this in place, you can now point to this configuration in the `registerQueue()` options object:

```
BullModule.registerQueue({
  configKey: 'alternative-config',
  name: 'video'
});
```

Producers

Job producers add jobs to queues. Producers are typically application services (Nest providers). To add jobs to a queue, first inject the queue into the service as follows:

```
import { Injectable } from '@nestjs/common';
import { Queue } from 'bull';
import { InjectQueue } from '@nestjs/bull';

@Injectable()
export class AudioService {
```

```
  constructor(@InjectQueue('audio') private audioQueue: Queue) {}
}
```

> info **Hint** The `@InjectQueue()` decorator identifies the queue by its name, as provided in the `registerQueue()` method call (e.g., `'audio'`).

Now, add a job by calling the queue's `add()` method, passing a user-defined job object. Jobs are represented as serializable JavaScript objects (since that is how they are stored in the Redis database). The shape of the job you pass is arbitrary; use it to represent the semantics of your job object.

```
const job = await this.audioQueue.add({
  foo: 'bar',
});
```

Named jobs

Jobs may have unique names. This allows you to create specialized consumers that will only process jobs with a given name.

```
const job = await this.audioQueue.add('transcode', {
  foo: 'bar',
});
```

> Warning **Warning** When using named jobs, you must create processors for each unique name added to a queue, or the queue will complain that you are missing a processor for the given job. See here for more information on consuming named jobs.

Job options

Jobs can have additional options associated with them. Pass an options object after the `job` argument in the `Queue.add()` method. Job options properties are:

- `priority`: `number` - Optional priority value. Ranges from 1 (highest priority) to MAX_INT (lowest priority). Note that using priorities has a slight impact on performance, so use them with caution.
- `delay`: `number` - An amount of time (milliseconds) to wait until this job can be processed. Note that for accurate delays, both server and clients should have their clocks synchronized.
- `attempts`: `number` - The total number of attempts to try the job until it completes.
- `repeat`: `RepeatOpts` - Repeat job according to a cron specification. See RepeatOpts.
- `backoff`: `number | BackoffOpts` - Backoff setting for automatic retries if the job fails. See BackoffOpts.
- `lifo`: `boolean` - If true, adds the job to the right end of the queue instead of the left (default false).
- `timeout`: `number` - The number of milliseconds after which the job should fail with a timeout error.
- `jobId`: `number` | `string` - Override the job ID - by default, the job ID is a unique integer, but you can use this setting to override it. If you use this option, it is up to you to ensure the jobId is unique. If you attempt to add a job with an id that already exists, it will not be added.

- `removeOnComplete`: `boolean | number` - If true, removes the job when it successfully completes. A number specifies the amount of jobs to keep. Default behavior is to keep the job in the completed set.
- `removeOnFail`: `boolean | number` - If true, removes the job when it fails after all attempts. A number specifies the amount of jobs to keep. Default behavior is to keep the job in the failed set.
- `stackTraceLimit`: `number` - Limits the amount of stack trace lines that will be recorded in the stacktrace.

Here are a few examples of customizing jobs with job options.

To delay the start of a job, use the `delay` configuration property.

```
const job = await this.audioQueue.add(
  {
    foo: 'bar',
  },
  { delay: 3000 }, // 3 seconds delayed
);
```

To add a job to the right end of the queue (process the job as **LIFO** (Last In First Out)), set the `lifo` property of the configuration object to `true`.

```
const job = await this.audioQueue.add(
  {
    foo: 'bar',
  },
  { lifo: true },
);
```

To prioritize a job, use the `priority` property.

```
const job = await this.audioQueue.add(
  {
    foo: 'bar',
  },
  { priority: 2 },
);
```

Consumers

A consumer is a **class** defining methods that either process jobs added into the queue, or listen for events on the queue, or both. Declare a consumer class using the `@Processor()` decorator as follows:

```
import { Processor } from '@nestjs/bull';
```

```
@Processor('audio')
export class AudioConsumer {}
```

> info **Hint** Consumers must be registered as `providers` so the `@nestjs/bull` package can pick them up.

Where the decorator's string argument (e.g., `'audio'`) is the name of the queue to be associated with the class methods.

Within a consumer class, declare job handlers by decorating handler methods with the `@Process()` decorator.

```
import { Processor, Process } from '@nestjs/bull';
import { Job } from 'bull';

@Processor('audio')
export class AudioConsumer {
  @Process()
  async transcode(job: Job<unknown>) {
    let progress = 0;
    for (i = 0; i < 100; i++) {
      await doSomething(job.data);
      progress += 1;
      await job.progress(progress);
    }
    return {};
  }
}
```

The decorated method (e.g., `transcode()`) is called whenever the worker is idle and there are jobs to process in the queue. This handler method receives the `job` object as its only argument. The value returned by the handler method is stored in the job object and can be accessed later on, for example in a listener for the completed event.

`Job` objects have multiple methods that allow you to interact with their state. For example, the above code uses the `progress()` method to update the job's progress. See here for the complete `Job` object API reference.

You can designate that a job handler method will handle **only** jobs of a certain type (jobs with a specific `name`) by passing that `name` to the `@Process()` decorator as shown below. You can have multiple `@Process()` handlers in a given consumer class, corresponding to each job type (`name`). When you use named jobs, be sure to have a handler corresponding to each name.

```
@Process('transcode')
async transcode(job: Job<unknown>) { ... }
```

> warning **Warning** When defining multiple consumers for the same queue, the `concurrency` option in `@Process({{ '{' }} concurrency: 1 {{ '}' }})` won't take effect. The minimum `concurrency` will match the number of consumers defined. This also applies even if `@Process()` handlers use a different `name` to handle named jobs.

Request-scoped consumers

When a consumer is flagged as request-scoped (learn more about the injection scopes here), a new instance of the class will be created exclusively for each job. The instance will be garbage-collected after the job has completed.

```
@Processor({
  name: 'audio',
  scope: Scope.REQUEST,
})
```

Since request-scoped consumer classes are instantiated dynamically and scoped to a single job, you can inject a `JOB_REF` through the constructor using a standard approach.

```
constructor(@Inject(JOB_REF) jobRef: Job) {
  console.log(jobRef);
}
```

> info **Hint** The `JOB_REF` token is imported from the `@nestjs/bull` package.

Event listeners

Bull generates a set of useful events when queue and/or job state changes occur. Nest provides a set of decorators that allow subscribing to a core set of standard events. These are exported from the `@nestjs/bull` package.

Event listeners must be declared within a consumer class (i.e., within a class decorated with the `@Processor()` decorator). To listen for an event, use one of the decorators in the table below to declare a handler for the event. For example, to listen to the event emitted when a job enters the active state in the `audio` queue, use the following construct:

```
import { Processor, Process, OnQueueActive } from '@nestjs/bull';
import { Job } from 'bull';

@Processor('audio')
export class AudioConsumer {

  @OnQueueActive()
  onActive(job: Job) {
    console.log(
      `Processing job ${job.id} of type ${job.name} with data
```

```
    ${job.data}...`,
      );
    }
    ...
```

Since Bull operates in a distributed (multi-node) environment, it defines the concept of event locality. This concept recognizes that events may be triggered either entirely within a single process, or on shared queues from different processes. A **local** event is one that is produced when an action or state change is triggered on a queue in the local process. In other words, when your event producers and consumers are local to a single process, all events happening on queues are local.

When a queue is shared across multiple processes, we encounter the possibility of **global** events. For a listener in one process to receive an event notification triggered by another process, it must register for a global event.

Event handlers are invoked whenever their corresponding event is emitted. The handler is called with the signature shown in the table below, providing access to information relevant to the event. We discuss one key difference between local and global event handler signatures below.

Local event listeners	Global event listeners	Handler method signature / When fired
`@OnQueueError()`	`@OnGlobalQueueError()`	`handler(error: Error)` - An error occurred. `error` contains the triggering error.
`@OnQueueWaiting()`	`@OnGlobalQueueWaiting()`	`handler(jobId: number | string)` - A Job is waiting to be processed as soon as a worker is idling. `jobId` contains the id for the job that has entered this state.
`@OnQueueActive()`	`@OnGlobalQueueActive()`	`handler(job: Job)` - Job `job`has started.
`@OnQueueStalled()`	`@OnGlobalQueueStalled()`	`handler(job: Job)` - Job `job` has been marked as stalled. This is useful for debugging job workers that crash or pause the event loop.
`@OnQueueProgress()`	`@OnGlobalQueueProgress()`	`handler(job: Job, progress: number)` - Job `job`'s progress was updated to value `progress`.
`@OnQueueCompleted()`	`@OnGlobalQueueCompleted()`	`handler(job: Job, result: any)` Job `job` successfully completed with a result `result`.
`@OnQueueFailed()`	`@OnGlobalQueueFailed()`	`handler(job: Job, err: Error)` Job `job` failed with reason `err`.

`@OnQueuePaused()`	`@OnGlobalQueuePaused()`	`handler()` The queue has been paused.
`@OnQueueResumed()`	`@OnGlobalQueueResumed()`	`handler(job: Job)` The queue has been resumed.
`@OnQueueCleaned()`	`@OnGlobalQueueCleaned()`	`handler(jobs: Job[], type: string)` Old jobs have been cleaned from the queue. `jobs` is an array of cleaned jobs, and `type` is the type of jobs cleaned.
`@OnQueueDrained()`	`@OnGlobalQueueDrained()`	`handler()` Emitted whenever the queue has processed all the waiting jobs (even if there can be some delayed jobs not yet processed).
`@OnQueueRemoved()`	`@OnGlobalQueueRemoved()`	`handler(job: Job)` Job `job` was successfully removed.

When listening for global events, the method signatures can be slightly different from their local counterpart. Specifically, any method signature that receives `job` objects in the local version, instead receives a `jobId` (`number`) in the global version. To get a reference to the actual `job` object in such a case, use the `Queue#getJob` method. This call should be awaited, and therefore the handler should be declared `async`. For example:

```
@OnGlobalQueueCompleted()
async onGlobalCompleted(jobId: number, result: any) {
  const job = await this.immediateQueue.getJob(jobId);
  console.log('(Global) on completed: job ', job.id, ' -> result: ',
result);
}
```

> info **Hint** To access the `Queue` object (to make a `getJob()` call), you must of course inject it. Also, the Queue must be registered in the module where you are injecting it.

In addition to the specific event listener decorators, you can also use the generic `@OnQueueEvent()` decorator in combination with either `BullQueueEvents` or `BullQueueGlobalEvents` enums. Read more about events here.

Queue management

Queue's have an API that allows you to perform management functions like pausing and resuming, retrieving the count of jobs in various states, and several more. You can find the full queue API here. Invoke any of these methods directly on the `Queue` object, as shown below with the pause/resume examples.

Pause a queue with the `pause()` method call. A paused queue will not process new jobs until resumed, but current jobs being processed will continue until they are finalized.

```
await audioQueue.pause();
```

To resume a paused queue, use the `resume()` method, as follows:

```
await audioQueue.resume();
```

Separate processes

Job handlers can also be run in a separate (forked) process (source). This has several advantages:

- The process is sandboxed so if it crashes it does not affect the worker.
- You can run blocking code without affecting the queue (jobs will not stall).
- Much better utilization of multi-core CPUs.
- Less connections to redis.

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { BullModule } from '@nestjs/bull';
import { join } from 'path';

@Module({
  imports: [
    BullModule.registerQueue({
      name: 'audio',
      processors: [join(__dirname, 'processor.js')],
    }),
  ],
})
export class AppModule {}
```

Please note that because your function is being executed in a forked process, Dependency Injection (and IoC container) won't be available. That means that your processor function will need to contain (or create) all instances of external dependencies it needs.

```
@@filename(processor)
import { Job, DoneCallback } from 'bull';

export default function (job: Job, cb: DoneCallback) {
  console.log(`[${process.pid}] ${JSON.stringify(job.data)}`);
  cb(null, 'It works');
}
```

Async configuration

You may want to pass `bull` options asynchronously instead of statically. In this case, use the `forRootAsync()` method which provides several ways to deal with async configuration. Likewise, if you want to pass queue options asynchronously, use the `registerQueueAsync()` method.

One approach is to use a factory function:

```
BullModule.forRootAsync({
  useFactory: () => ({
    redis: {
      host: 'localhost',
      port: 6379,
    },
  }),
});
```

Our factory behaves like any other asynchronous provider (e.g., it can be `async` and it's able to inject dependencies through `inject`).

```
BullModule.forRootAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    redis: {
      host: configService.get('QUEUE_HOST'),
      port: configService.get('QUEUE_PORT'),
    },
  }),
  inject: [ConfigService],
});
```

Alternatively, you can use the `useClass` syntax:

```
BullModule.forRootAsync({
  useClass: BullConfigService,
});
```

The construction above will instantiate `BullConfigService` inside `BullModule` and use it to provide an options object by calling `createSharedConfiguration()`. Note that this means that the `BullConfigService` has to implement the `SharedBullConfigurationFactory` interface, as shown below:

```
@Injectable()
class BullConfigService implements SharedBullConfigurationFactory {
  createSharedConfiguration(): BullModuleOptions {
    return {
      redis: {
```

```
          host: 'localhost',
          port: 6379,
        },
      };
    }
  }
```

In order to prevent the creation of `BullConfigService` inside `BullModule` and use a provider imported from a different module, you can use the `useExisting` syntax.

```
BullModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

This construction works the same as `useClass` with one critical difference - `BullModule` will lookup imported modules to reuse an existing `ConfigService` instead of instantiating a new one.

Example

A working example is available here.