

Consumers



- [Creating Pull-Based Consumers](#)
 - [Creating Push-Based Consumers](#)
 - [Listing](#)
 - [Querying](#)
 - [Stream vs Consumer sequence numbers](#)
 - [Consuming Pull-Based Consumers](#)
 - [Consuming Push-Based Consumers](#)
-

Messages are read or consumed from the Stream by Consumers. We support pull and push-based Consumers and the example scenario has both, let's walk through that.

Creating Pull-Based Consumers

The `NEW` and `DISPATCH` Consumers are pull-based, meaning the services consuming data from them have to ask the system for the next available message. This means you can easily scale your services up by adding more workers and the messages will get spread across the workers based on their availability.

Pull-based Consumers are created the same as push-based Consumers, you just don't specify a delivery target.

```
nats con ls ORDERS
```

```
No Consumers defined
```

We have no Consumers, lets add the `NEW` one:

I supply the `--sample` options on the CLI as this is not prompted for at present, everything else is prompted. The help in the CLI explains each:

```
nats con add --sample 100
```

```
? Select a Stream ORDERS
? Consumer name NEW
? Delivery target
? Start policy (all, last, 1h, msg sequence) all
? Filter Stream by subject (blank for all) ORDERS.received
? Maximum Allowed Deliveries 20
Information for Consumer ORDERS > NEW
```

Configuration:

```
    Durable Name: NEW
      Pull Mode: true
        Subject: ORDERS.received
      Deliver All: true
    Deliver Last: false
      Ack Policy: explicit
        Ack Wait: 30s
      Replay Policy: instant
Maximum Deliveries: 20
    Sampling Rate: 100
```

State:

```
Last Delivered Message: Consumer sequence: 1 Stream sequence: 1
  Acknowledgment floor: Consumer sequence: 0 Stream sequence: 0
    Pending Messages: 0
  Redelivered Messages: 0
```

This is a pull-based Consumer (empty Delivery Target), it gets messages from the first available message and requires specific acknowledgement of each and every message.

It only received messages that originally entered the Stream on `ORDERS.received`. Remember the Stream subscribes to `ORDERS.*`, this lets us select a subset of messages from the Stream.

A Maximum Delivery limit of 20 is set, this means if the message is not acknowledged it will be retried but only up to this maximum total deliveries.

Again this can all be done in a single CLI call, lets make the `DISPATCH` Consumer:

```
nats con add ORDERS DISPATCH --filter ORDERS.processed --ack explicit --pull --deliver all --
sample 100 --max-deliver 20
```

Additionally, one can store the configuration in a JSON file, the format of this is the same as `$ nats con info ORDERS DISPATCH -j | jq .config`:

```
nats con add ORDERS MONITOR --config monitor.json
```

Creating Push-Based Consumers

Our `MONITOR` Consumer is push-based, has no ack and will only get new messages and is not sampled:

```
nats con add
```

```
? Select a Stream ORDERS
? Consumer name MONITOR
? Delivery target monitor.ORDERS
? Start policy (all, last, 1h, msg sequence) last
? Acknowledgement policy none
? Replay policy instant
? Filter Stream by subject (blank for all)
? Maximum Allowed Deliveries -1
Information for Consumer ORDERS > MONITOR
```

Configuration:

```
    Durable Name: MONITOR
Delivery Subject: monitor.ORDERS
    Deliver All: false
    Deliver Last: true
    Ack Policy: none
    Replay Policy: instant
```

State:

```
Last Delivered Message: Consumer sequence: 1 Stream sequence: 3
    Acknowledgment floor: Consumer sequence: 0 Stream sequence: 2
    Pending Messages: 0
    Redelivered Messages: 0
```

Again you can do this with a single non-interactive command:

```
nats con add ORDERS MONITOR --ack none --target monitor.ORDERS --deliver last --replay instant
--filter ''
```

Additionally one can store the configuration in a JSON file, the format of this is the same as `$ nats con info ORDERS MONITOR -j | jq .config`:

```
nats con add ORDERS --config monitor.json
```

Listing

You can get a quick list of all the Consumers for a specific Stream:

```
nats con ls ORDERS
```

Consumers for Stream ORDERS:

```
DISPATCH
MONITOR
NEW
```

Querying

All details for a Consumer can be queried, lets first look at a pull-based Consumer:

```
$ nats con info ORDERS DISPATCH
Information for Consumer ORDERS > DISPATCH
```

Configuration:

```
Durable Name: DISPATCH
Pull Mode: true
Subject: ORDERS.processed
Deliver All: true
Deliver Last: false
Ack Policy: explicit
Ack Wait: 30s
Replay Policy: instant
```

```
Sampling Rate: 100
```

State:

```
Last Delivered Message: Consumer sequence: 1 Stream sequence: 1
```

```
Acknowledgment floor: Consumer sequence: 0 Stream sequence: 0
```

```
Pending Messages: 0
```

```
Redelivered Messages: 0
```

More details about the `State` section will be shown later when discussing the ack models in depth.

Stream vs Consumer sequence numbers

The two number are not directly related: the Stream sequence number is the pointer to the exact message, while the Consumer sequence number is an ever-increasing counter for consumer actions.

So for example a stream with 1 message in it would have stream sequence of 1, but if the consumer attempted 10 deliveries of that message consumer sequence would be 10 or 11.

Consuming Pull-Based Consumers

Pull-based Consumers require you to specifically ask for messages and ack them, typically you would do this with the client library `Request()` feature, but the `nats` utility has a helper:

First, we ensure we have a message:

```
nats pub ORDERS.processed "order 1"
nats pub ORDERS.processed "order 2"
nats pub ORDERS.processed "order 3"
```

We can now read them using `nats`:

```
nats con next ORDERS DISPATCH
```

```
--- received on ORDERS.processed
order 1
```

```
Acknowledged message
```

Consumer another one

```
nats con next ORDERS DISPATCH
```

```
--- received on ORDERS.processed  
order 2
```

```
Acknowledged message
```

You can prevent ACKs by supplying `--no-ack`.

To do this from code you'd send a `Request()` to `$JS.API.CONSUMER.MSG.NEXT.ORDERS.DISPATCH`:

```
nats req '$JS.API.CONSUMER.MSG.NEXT.ORDERS.DISPATCH' ''
```

```
Published [$JS.API.CONSUMER.MSG.NEXT.ORDERS.DISPATCH] : ''  
Received [ORDERS.processed] : 'order 3'
```

Here `nats req` cannot ack, but in your code you'd respond to the received message with a nil payload as an Ack to JetStream.

Consuming Push-Based Consumers

Push-based Consumers will publish messages to a subject and anyone who subscribes to the subject will get them, they support different Acknowledgement models covered later, but here on the `MONITOR` Consumer we have no Acknowledgement.

```
nats con info ORDERS MONITOR
```

Output extract

```
...  
  Delivery Subject: monitor.ORDERS  
...
```

The Consumer is publishing to that subject, so let's listen there:

```
nats sub monitor.ORDERS
```

```
Listening on [monitor.ORDERS]
```

```
[#3] Received on [ORDERS.processed]: 'order 3'
```

```
[#4] Received on [ORDERS.processed]: 'order 4'
```

Note the subject here of the received message is reported as `ORDERS.processed` this helps you distinguish what you're seeing in a Stream covering a wildcard, or multiple subjects, subject space.

This Consumer needs no ack, so any new message into the ORDERS system will show up here in real-time.

📄 #1

📄 cocobook-sync 📄 2026-10-28 11:14:39 UTC

📄 cocobook-sync 📄 2026-10-28 11:14:39 UTC