

Asynchronous Subscriptions

Asynchronous subscriptions use callbacks of some form to notify an application when a message arrives. These subscriptions are usually easier to work with, but do represent some form of internal work and resource usage, i.e. threads, by the library. Check your library's documentation for any resource usage associated with asynchronous subscriptions.

Note: For a given subscription, messages are dispatched serially, one message at a time. If your application does not care about processing ordering and would prefer the messages to be dispatched concurrently, it is the application's responsibility to move them to some internal queue to be picked up by threads/go routines.

The following example subscribes to the subject `updates` and handles the incoming messages:

Go

```
nc, err := nats.Connect("demo.nats.io")
if err != nil {
    log.Fatal(err)
}
defer nc.Close()

// Use a WaitGroup to wait for a message to arrive
wg := sync.WaitGroup{}
wg.Add(1)

// Subscribe
if _, err := nc.Subscribe("updates", func(m *nats.Msg) {
    wg.Done()
}); err != nil {
    log.Fatal(err)
}

// Wait for a message to come in
wg.Wait()
```

Java

```

Connection nc = Nats.connect("nats://demo.nats.io:4222");

// Use a latch to wait for a message to arrive
CountDownLatch latch = new CountDownLatch(1);

// Create a dispatcher and inline message handler
Dispatcher d = nc.createDispatcher((msg) -> {
    String str = new String(msg.getData(), StandardCharsets.UTF_8);
    System.out.println(str);
    latch.countDown();
});

// Subscribe
d.subscribe("updates");

// Wait for a message to come in
latch.await();

// Close the connection
nc.close();

```

JavaScript

```

const sc = StringCodec();
// this is an example of a callback subscription
// https://github.com/nats-io/nats.js/blob/master/README.md#async-vs-callbacks
nc.subscribe("updates", {
  callback: (err, msg) => {
    if (err) {
      t.error(err.message);
    } else {
      t.log(sc.decode(msg.data));
    }
  },
  max: 1,
});

// here's an iterator subscription - note the code in the
// for loop will block until the iterator completes
// either from a break/return from the iterator, an

```

```

// unsubscribe after the message arrives, or in this case
// an auto-unsubscribe after the first message is received
const sub = nc.subscribe("updates", { max: 1 });
for await (const m of sub) {
  t.log(sc.decode(m.data));
}

// subscriptions have notifications, simply wait
// the closed promise
sub.closed
  .then(() => {
    t.log("subscription closed");
  })
  .catch((err) => {
    t.err(`subscription closed with an error ${err.message}`);
  });

```

Python

```

nc = NATS()

await nc.connect(servers=["nats://demo.nats.io:4222"])

future = asyncio.Future()

async def cb(msg):
    nonlocal future
    future.set_result(msg)

await nc.subscribe("updates", cb=cb)
await nc.publish("updates", b'All is Well')
await nc.flush()

# Wait for message to come in
msg = await asyncio.wait_for(future, 1)

```

C#

```

// dotnet add package NATS.Net
using NATS.Net;

```

```

await using var client = new NatsClient();

// Subscribe to the "updates" subject and receive messages as <string> type.
// The default serializer understands all primitive types, strings,
// byte arrays, and uses JSON for complex types.
await foreach (var msg in client.SubscribeAsync<string>("updates"))
{
    Console.WriteLine($"Received: {msg.Data}");

    if (msg.Data == "exit")
    {
        // When we exit the loop, we unsubscribe from the subject
        // as a result of enumeration completion.
        break;
    }
}

```

Ruby

```

require 'nats/client'

NATS.start(servers:["nats://127.0.0.1:4222"]) do |nc|
  nc.subscribe("updates") do |msg|
    puts msg
    nc.close
  end

  nc.publish("updates", "All is Well")
end

```

C

```

static void
onMsg(natsConnection *conn, natsSubscription *sub, natsMsg *msg, void *closure)
{
    printf("Received msg: %s - %.*s\n",
           natsMsg_GetSubject(msg),
           natsMsg_GetDataLength(msg),
           natsMsg_GetData(msg));
}

```

```

    // Need to destroy the message!
    natsMsg_Destroy(msg);
}

(...)

natsConnection      *conn = NULL;
natsSubscription    *sub  = NULL;
natsStatus          s;

s = natsConnection_ConnectTo(&conn, NATS_DEFAULT_URL);
if (s == NATS_OK)
{
    // Creates an asynchronous subscription on subject "foo".
    // When a message is sent on subject "foo", the callback
    // onMsg() will be invoked by the client library.
    // You can pass a closure as the last argument.
    s = natsConnection_Subscribe(&sub, conn, "foo", onMsg, NULL);
}

(...)

// Destroy objects that were created
natsSubscription_Destroy(sub);
natsConnection_Destroy(conn);

```

📄 #1

📄 cocobook-sync 📄 2026-10-28 11:12:39 UTC

📄 cocobook-sync 📄 2026-10-28 11:12:39 UTC