

Advanced Connect and Custom Dialer in Go

The Go NATS client features a [CustomDialer](#) option which allows you to customize the connection logic against the NATS server without having to modify the internals of the client. For example, let's say that you want to make the client use the `context` package to use `DialContext` and be able to cancel connecting to NATS altogether with a deadline, you could then define a Dialer implementation as follows:

```
package main

import (
    "context"
    "log"
    "net"
    "time"

    "github.com/nats-io/nats.go"
)

type customDialer struct {
    ctx          context.Context
    nc           *nats.Conn
    connectTimeout time.Duration
    connectTimeWait time.Duration
}

func (cd *customDialer) Dial(network, address string) (net.Conn, error) {
    ctx, cancel := context.WithTimeout(cd.ctx, cd.connectTimeout)
    defer cancel()

    for {
        log.Println("Attempting to connect to", address)
        if ctx.Err() != nil {
            return nil, ctx.Err()
        }
    }
}
```

```

    }

    select {
    case <-cd.ctx.Done():
        return nil, cd.ctx.Err()
    default:
        d := &net.Dialer{}
        if conn, err := d.DialContext(ctx, network, address); err == nil {
            log.Println("Connected to NATS successfully")
            return conn, nil
        } else {
            time.Sleep(cd.connectTimeWait)
        }
    }
}
}
}

```

With the dialer implementation above, the NATS client will retry a number of times to connect to the NATS server until the context is no longer valid:

```

func main() {
    // Parent context cancels connecting/reconnecting altogether.
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    var err error
    var nc *nats.Conn
    cd := &customDialer{
        ctx:          ctx,
        connectTimeout: 10 * time.Second,
        connectTimeWait: 1 * time.Second,
    }

    opts := []nats.Option{
        nats.SetCustomDialer(cd),
        nats.ReconnectWait(2 * time.Second),
        nats.ReconnectHandler(func(c *nats.Conn) {
            log.Println("Reconnected to", c.ConnectedUrl())
        }),
        nats.DisconnectHandler(func(c *nats.Conn) {
            log.Println("Disconnected from NATS")
        }),
    },

```

```

    nats.ClosedHandler(func(c *nats.Conn) {
        log.Println("NATS connection is closed.")
    }),
    nats.NoReconnect(),
}
go func() {
    nc, err = nats.Connect("127.0.0.1:4222", opts...)
}()

```

WaitForEstablishedConnection:

```

for {
    if err != nil {
        log.Fatal(err)
    }

    // Wait for context to be canceled either by timeout
    // or because of establishing a connection...
    select {
    case <-ctx.Done():
        break WaitForEstablishedConnection
    default:
    }

    if nc == nil || !nc.IsConnected() {
        log.Println("Connection not ready")
        time.Sleep(200 * time.Millisecond)
        continue
    }
    break WaitForEstablishedConnection
}
if ctx.Err() != nil {
    log.Fatal(ctx.Err())
}

for {
    if nc.IsClosed() {
        break
    }
    if err := nc.Publish("hello", []byte("world")); err != nil {
        log.Println(err)
        time.Sleep(1 * time.Second)
    }
}

```

```
        continue
    }
    log.Println("Published message")
    time.Sleep(1 * time.Second)
}

// Disconnect and flush pending messages
if err := nc.Drain(); err != nil {
    log.Println(err)
}
log.Println("Disconnected")
}
```

```
## #1
[] cocobook-sync [] 2026-10-28 11:13:11 UTC
[] cocobook-sync [] 2026-10-28 11:13:11 UTC
```