Neffos

Home

Welcome to **neffos** - a cross-platform real-time framework with expressive, elegant API written in Go.

This book contains extensive documentation for **developers and teams** working with the neffos project.

Table of Contents

About

Neffos is a new project, however, it's a work of years of experience and months of coding.

The motivation was huge, Gophers are missing a decent tool to support websockets on their applications. There are a lot of libraries created the last years but each one of them have its own limitations.

Either a low-level library that requires from the user to have years of experience, either a high-level library that limits experienced developers to customize or give performance boost to their applications.

Neffos aims to be a balance between those, give the freedom to an experienced developer to access low-level API and help teams to create and maintain real-time applications fast and easy. The neffos API is written in a way that feels like home.

Installation

Neffos is a cross-platform package.

The only requirement is the Go Programming Language, version 1.21 and above.

```
$ go get github.com/kataras/neffos@latest
```

Or inside your `go.mod` file:

```
module your_project_name

go 1.21

require (
    github.com/kataras/neffos v0.0.22
)
```

Troubleshooting

If you get a network error during installation please make sure you set a valid GOPROXY environment variable.

```
go env -w GOPROXY=direct
```

How to update

Here is the go-get command to get the latest and greatest neffos version. Master branch is usually stable enough.

```
$ go get github.com/kataras/neffos@master
```

Continue by reading our Getting Started tutorial.

Getting Started

Neffos provides a comprehensive API to work with. Identical API and keywords between server and client sides.

Here is a quick outline of the flow you'll use:

```
                                                    +----------------------------+
                                                    |.ID      ()                 |
                                                    |.IsClient()                 |
+-----------------------+                           |.Server  ()                 |
|                       |                           +----------------------------+
|       *Conn           +---------------------------+.Write(Message {...} )      |
|                       |                           |.Ask  (Message {...} )      |
+----------+------------+                           +----------------------------+
           |                                        |.Close   ()                 |
           |                                        |.IsClosed()                 |
           |                                        +----------------------------+
           |                                        |.WaitConnect  ("namespace") |
           +----------------------+                 |.Namespace    ("namespace") |
           |.Connect("namespace") |                 |.DisconnectAll()            |
           +-----+----------------+                 +----------------------------+
                 |
                 |
                 |
                 |                                                                      +--------------------------+
           +-----+----------------+                                                     | Message {                |
           |                      |                 +----------------------------+      |                          |
           |      *NSConn         +-----------------+.Emit("event", body)        +------+  Namespace: "namespace", |
           |                      |                 +----------------------------+      |  Event:     "event",     |
           +-----+----------------+                 |.Disconnect()               |      |  Body:      body,        |
                 |                                  +----------------------------+      |                          |
                 |                                  |.Ask     ("event", body)    |      | }                        |
                 |                                  |.Room     ("roomName")      |      +--------------------------+
                 +----------------------+           |.Rooms   ()                 |
                 |.JoinRoom("roomName") |           |.LeaveAll()                 |
                 +-------+--------------+           +----------------------------+
                         |
                         |
                         |                                                              +--------------------------+
                         |                                                              | Message {                |
                 +-------+--------------+                                               |                          |
                 |                      |           +----------------------------+      |  Namespace: "namespace", |
                 |        *Room         +-----------+.Emit("event", body)        +------+  Room:      "roomName",  |
                 |                      |           +----------------------------+      |  Event:     "event",     |
                 +----------------------+           |.Leave()                    |      |  Body:      body,        |
                                                    +----------------------------+      |                          |
                                                                                        | }                        |
                                                                                        +--------------------------+
```

All features are always in-sync between server and client side connections, each side gets notified of mutations.

End-developers can implement deadlines to actions like `Conn#Connect`, `Conn#Ask`, `NSConn#Disconnect`, `NSConn#Ask`, `NSConn#JoinRoom`, `Room#Leave` and etc. Methods that have to do with remote side response accept a `context.Context` as their first argument.

Dialing

1. Client connection is initialized through the `neffos.Dial` (or `neffos.dial` on javascript side) method.
2. Server can dismiss with an error any incoming client through its `server.OnConnect -> return err != nil` callback.

Send data through events

1. `Emit` sends data and fires a specific event back to the remote side.
2. A client can NOT communicate directly to the rest of the clients.
3. Server is the only one which is able to send messages to one or more clients.
4. To send data to all clients use the `Conn.Server().Broadcast` method. If its first argument is not nil then it sends data to all except that one connection ID.

Now, let's start by building a small echo application between server and a client. Client will send something and server will respond with a prefix of `"echo back: "`.

Supposedly we have both server and client side in the same project (or package).

Create a new `echo.go` file.

```go
package main
```

Import the `neffos` go package and, optionally, the `log` standard go package in order to help us print incoming messages.

```go
import (
    "log"

    "github.com/kataras/neffos"
)
```

Register events

Incoming events are being captured through callbacks. A callback declaration is a type of `func(*neffos.NSConn, neffos.Message) error`.

Note that in this example, we share the same event's callback across client and server sides, depending on your app's requirements you may want to separate those events.

```go
func onEcho(c *neffos.NSConn, msg neffos.Message) error {
    body := string(msg.Body)
    log.Println(body)


    if !c.Conn.IsClient() {
        // this block will only run on a server-side callback.
        newBody := append([]byte("echo back: "), msg.Body...)
        return neffos.Reply(newBody)
    }

    return nil
}
```

The `neffos.Reply` is just a helper function which writes back to the connection the same incoming `neffos.Message` with a custom body data.

Alternatively you may `Emit` a remote event, depending on your app's needs, this is how:

```go
c.Emit("echo", newBody)
```

Which is the same as:

```go
c.Conn.Write(neffos.Message{Namespace: "v1", Event: "echo", Body: newBody})
```

Callbacks are registered via the `Namespaces & Events` event-driven API. Let's add an `"echo"` event on a namespace called `"v1"`. Note that `Events` can be registered without a namespace as well but it's highly recommented that you put them under a non-empty namespace for future maintainability.

```go
var events = neffos.Namespaces{
    "v1": neffos.Events{
        "echo": onEcho,
    },
}
```

Using a struct value

You can also register events based on a Go structure using the `NewStruct` function which returns a compatible `neffos.ConnHandler`. Like the `neffos.Namespaces`, `neffos.Events` and `neffos.WithTimeout` can be passed on `New` and `Dial` package-level functions.

Example Code:

Please read the *comments*.

```go
type serverConn struct {
    // Dynamic field, a new "serverConn" instance is
    // created on each new connection to this namespace.

    Conn *neffos.NSConn
    // A static field is allowed, if filled before server ran then
    // is set on each new "serverConn" instance.
    SuffixResponse string
}

func (c *serverConn) OnChat(msg neffos.Message) error {
    c.Conn.Emit("ChatResponse", append(msg.Body, []byte(c.SuffixResponse)...))
    return nil
}

type clientConn struct {
    Conn *neffos.NSConn
}

func (s *clientConn) ChatResponse(msg neffos.Message) error {
    log.Printf("Echo back from server: %s", string(msg.Body))
    return nil
}
```

```go
func startServer() {
    controller := new(serverConn)
    controller.SuffixResponse = " Static Response Suffix for shake of the example"

    // This will convert a structure to neffos.Namespaces based on the struct's methods.
    // The methods can be func(msg neffos.Message) error if
    // the structure contains a *neffos.NSConn field,
    // otherwise they should be like any event callback:
    // func(nsConn *neffos.NSConn, msg neffos.Message) error.
        // If contains a field of type *neffos.NSConn then a new controller
        // is created on each new connection to this namespace
    // and static fields(if any) are set on runtime with the NSConn itself.
    // If it's a static controller (does not contain a NSConn field)
    // then it just registers its functions as regular events without performance cost.
    events := neffos.NewStruct(controller).
        // This sets for the "default" namespace,
        // alternatively you can add a `Namespace() string` to the serverConn struct
        // or leave it empty for empty namespace.
        SetNamespace("default").
        // Optionally, sets read and write deadlines on the underlying network connection.
        // After a read or write have timed out, the websocket connection is closed.
        // For example:
        // If a client or server didn't receive or sent something
        // for 20 seconds this connection will be terminated.
        SetTimeouts(20*time.Second, 20*time.Second).
        // This will convert the "OnChat" method to a "Chat" event instead.
        SetEventMatcher(neffos.EventTrimPrefixMatcher("On"))

    websocketServer := neffos.New(gobwas.DefaultUpgrader, events)

    log.Println("Listening on: ws://localhost:8080\nPress CTRL/CMD+C to interrupt.")
    log.Fatal(http.ListenAndServe(":8080", websocketServer))
}
```

Create the Client

A neffos (Go) client expects a compatible `neffos.Dialer` to dial the neffos websocket server.

Among the neffos import statement, add one of the two built-in compatible Upgraders & Dialers. For example the `neffos/gorilla` sub-package helps us to adapt the gorilla websocket implementation into neffos.

Read more about Upgraders and Dialers.

```go
import (
    // [...]
    "github.com/kataras/neffos/gorilla"
)
```

Creating a websocket client is done by the `Dial` package-level function.

The `Dial` function accepts a `context.Context` as its first parameter which you can use to manage dialing timeout.

The `url` as its third input parameter is the endpoint which our websocket server waits for incoming requests, i.e `ws://localhost:8080/echo`.

The final parameter you have to pass to create both client and server is a `ConnHandler` (`Events` or `Namespaces with Events`).

```go
ctx := context.Background()
client, err := neffos.Dial(ctx, gorilla.DefaultDialer, "ws://localhost:8080/echo", events)
```

Connect the Client to a Namespace

The client must be connected to one or more server namespaces (even if a namespace is empty `""`) in order to initialize the communication between them.

Let's connect our client to the `"v1"` namespace and emit the remote (in this case, server side's) `"echo"` event with the data of `"Greeetings!"`.

```go
c, err := client.Connect(ctx, "v1")
c.Emit("echo", []byte("Greetings!"))
```

The client's `onEcho` will be fired and it will print the message of `echo back: Greetings!`.

Note that event emitting and firing are asynchronous operations, if you exit the program immediately after it, there is not enough time to fire back the `"echo"` event that the server-side remotely fired from its side.

However, if you need to block until response is received use the `c.Ask` instead:

```go
responseMessage, err := c.Ask(ctx, "echo", []byte("Greetings!"))
```

In that case, the `responseMessage.Body` is expected to be `[]byte("echo back: Greetings!")`.

Create the Server

Creating a websocket server is done by the `neffos.New` package-level function.

A neffos server expects a compatible `neffos.Upgrader` to upgrade HTTP incoming connection to WebSocket.

```go
websocketServer := neffos.New(gorilla.DefaultUpgrader, events)
```

Run the Server

A websocket server should run inside an http endpoint, so clients can connect. We need to create an http web server and add a route which will serve and upgrade the incoming requests using the neffos server. For that, you can choose between standard package net/http and a high-performant package like iris.

The `neffos.Server` completes the `http.Handler` interface.

```go
func (*Server) ServeHTTP(http.ResponseWriter, *http.Request)
```

Assuming that we want our websocket clients to communicate with our websocket server through the `localhost:8080/echo` endpoint, follow the below guide.

If you choose `net/http`, this is how you register the neffos server to an http endpoint:

```go
import (
    // [...]
    "net/http"
)

func main() {
    // [websocketServer := neffos.New...]
    router := http.NewServeMux()
    router.Handle("/echo", websocketServer)

    http.ListenAndServe(":8080", router)
}
```

Access the `http.Request` from Conn:

```go
func(c *neffos.NSConn, msg neffos.Message) error {
    req := c.Conn.Socket().Request()
}
```

Otherwise install iris using the `go get -u github.com/kataras/iris/v12@latest` terminal command. Iris has its own adapter for neffos, it lives inside its `iris/websocket` sub-package, we need to import that as well.

```go
import (
    // [...]
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/websocket"
)

func main() {
    // [websocketServer := neffos.New...]
    app := iris.New()
    // You can either use the websocketServer.IDGenerator:
    // (http.ResponseWriter, *http.Request) string
    // as expected.
    //
    // However, if you want to use a specific Iris one:
    // func(ctx iris.Context) string
    // set it as a second input argument on the `websocket.Handler` func.
    // irisIDGenenerator = func(ctx iris.Context) string {
    //    return ctx.GetHeader("X-Username")
    // }
    // websocket.Handler(websocketServer, irisIDGenerator)

    app.Get("/echo", websocket.Handler(websocketServer))

    app.Run(iris.Addr(":8080"))
}
```

Access the `iris.Context` from Conn:

```go
import "github.com/kataras/iris/v12/websocket"

func(c *neffos.NSConn, msg neffos.Message) error {
    ctx := websocket.GetContext(c.Conn)
}
```

Putting it all together

The final program, which contains both server and client side is just **75 lines of code**.

```go
package main

import (
    "context"
    "log"
    "net/http"
    "os"

    "github.com/kataras/neffos"
    "github.com/kataras/neffos/gorilla"
)

var events = neffos.Namespaces{
    "v1": neffos.Events{
        "echo": onEcho,
    },
}

func onEcho(c *neffos.NSConn, msg neffos.Message) error {
    body := string(msg.Body)
    log.Println(body)

    if !c.Conn.IsClient() {
        newBody := append([]byte("echo back: "), msg.Body...)
        return neffos.Reply(newBody)
    }

    return nil
}

func main() {
    args := os.Args[1:]
    if len(args) == 0 {
        log.Fatalf("expected program to start with 'server' or 'client' argument")
    }
    side := args[0]

    switch side {
    case "server":
        runServer()
    case "client":
        runClient()
    default:
        log.Fatalf("unexpected argument, expected 'server' or 'client' but got '%s'",
           side)
    }
}

func runServer() {
    websocketServer := neffos.New(gorilla.DefaultUpgrader, events)
    router := http.NewServeMux()
    router.Handle("/echo", websocketServer)

    log.Println("Serving websockets on localhost:8080/echo")
    log.Fatal(http.ListenAndServe(":8080", router))
}

func runClient() {
    ctx := context.Background()
    client, err := neffos.Dial(
        ctx,
        gorilla.DefaultDialer,
        "ws://localhost:8080/echo",
        events,
    )
    if err != nil {
        panic(err)
    }

    c, err := client.Connect(ctx, "v1")
    if err != nil {
        panic(err)
    }

    c.Emit("echo", []byte("Greetings!"))

    // a channel that blocks until client is terminated,
    // i.e by CTRL/CMD +C.
    <-client.NotifyClose
}
```

Run it

```
$ go run echo.go server
> 2019/06/29 02:24:41 Serving websockets on localhost:8080/echo
> 2019/06/29 02:25:34 Greetings!
```

```
$ go run echo.go client
> 2019/06/29 02:25:34 echo back: Greetings!
```

The Javascript library is aligned with the Go's client methods.

1. Import the `neffos.js` library, as module:

```js
const neffos = require('neffos.js');
```

Or in a script HTML tag:

```
<script src="https://cdn.jsdelivr.net/npm/neffos.js@latest/dist/neffos.min.js"></script>
```

2. For example, to register events and dial:

```
var events = new Object();
events.echo = function (nsConn, msg) { // "chat" event.
    window.alert(msg.Body);
}

neffos.dial("/echo", { v1: events });
```

OR

```
neffos.dial("/echo", { v1: {
    echo: function (nsConn, msg) { /* [...] */ }
}});
```

A javascript equivalent client looks like that.

```
try {
    const conn = await neffos.dial("/echo", { v1: {
        echo: function(nsConn, msg) {
            window.alert(msg.Body);
        }
    }});

    const nsConn = await conn.connect("v1");
    nsConn.emit("echo", "Greetings!");

} catch (err) {
    console.log(err);
}
```

Upgraders & Dialers

Neffos should run behind a low-level websocket implementation for Go. The developer has full control of this, to use her/his own or wrap an existing one. Neffos comes with two built-in low-level websocket implementations for Upgrading incoming http connections or Dialing the websocket server through Go (note that neffos provides clients for Nodejs & Browser too, see the neffos.js project to learn about). Those two optional dependencies are downloaded automatically for you on the Installation process.

1. github.com/gorilla/websocket
    - When server wants to Upgrade using gorilla/websocket.Upgrader or when a client wants to dial using the gorilla/websocket.Dialer.
2. github.com/gobwas/ws
    - When server wants to Upgrade using gobwas/ws.HTTPUpgrader or when a client wants to dial using the gobwas/ws.Dialer.

Neffos provides an easy way to adapt those through the following sub-packages:

1. github.com/kataras/neffos/gorilla
    - DefaultUpgrader
    - Upgrader(websocket.Upgrader) neffos.Upgrader
    - DefaultDialer
    - Dialer(dialer *websocket.Dialer, requestHeader http.Header) neffos.Dialer
2. github.com/kataras/neffos/gobwas
    - DefaultUpgrader
    - Upgrader(upgrader ws.HTTPUpgrader) neffos.Upgrader
    - DefaultDialer
    - Dialer(dialer ws.Dialer) neffos.Dialer

However, it is possible to wrap an existing one or implement your own and pass it to a neffos server and/or neffos client.

The `neffos.New` accepts a neffos.Upgrader.

```go
type Upgrader func(w http.ResponseWriter, r *http.Request) (Socket, error)
```

The `neffos.Dial` accepts a neffos.Dialer.

```go
type Dialer func(ctx context.Context, url string) (Socket, error)
```

Both `Upgrader` and `Dialer` should return a valid neffos.Socket.

```go
Socket interface {
    // NetConn returns the underline net connection.
    NetConn() net.Conn
    // Request returns the http request value.
    Request() *http.Request
    // ReadData reads binary or text messages from the remote connection.
    ReadData(timeout time.Duration) (body []byte, err error)
    // WriteBinary sends a binary message to the remote connection.
    WriteBinary(body []byte, timeout time.Duration) error
    // WriteText sends a text message to the remote connection.
    WriteText(body []byte, timeout time.Duration) error
}
```

Usage

```go
package server

import (
    "github.com/kataras/neffos"
    "github.com/kataras/neffos/gorilla"
)

func main() {
    server := neffos.New(gorilla.DefaultUpgrader, events)
    // [...]
}
```

```go
package client

import (
    "context"

    "github.com/kataras/neffos"
    "github.com/kataras/neffos/gorilla"
)

func main() {
    client, err := neffos.Dial(context.Background(), gorilla.DefaultDialer, events)
    // [...]
}
```

Request-Response architecture

As we've seen on a part of the Getting Started section the client (or server) can also block/wait and catch a response from the remote side anywhere in the application flow, you are not limited to the asynchronous event-driven API. Both methods and styles are used in a typical, common, application.

The Conn Ask

Use this method when you have access to the connection value that you want to ask for.

The `Conn.Ask` and `NSConn.Ask` methods do exactly that.

In this section you will learn how to create a question-answer flow and how to manage incoming remote event errors, remember? Event callback can return an `error` too.

At this example, for the shake of simplicity, we will set to a client the role to ask and server to reply, but this can be converted to a bidirectional flow too, each `NSConn` and `Conn` 's method can be used by both server and client sides.

The application is fairly simple, the client will provide a date `month-day-year` and server will reply back if its a work day or if the date is not valid for work, it's a day off or the provided time format is invalid.

Create & run the Server

Let's dive in by defining our server-side.

```go
const namespace = "default"

var errDayOff = errors.New("day off")

func runServer() {
    websocketServer := neffos.New(
        gorilla.DefaultUpgrader,
        neffos.Namespaces{
            namespace: neffos.Events{
                "workday": func(c *neffos.NSConn, msg neffos.Message) error {
                    date := string(msg.Body)
                    t, err := time.Parse("01-02-2006", date)
                    if err != nil {
                        return err
                    }

                    weekday := t.Weekday()

                    if weekday == time.Saturday || weekday == time.Sunday {
                        return errDayOff
                    }


                    responseText := fmt.Sprintf("it's %s, do your job.", weekday)
                    return neffos.Reply([]byte(responseText))
                },
            },
        })

    router := http.NewServeMux()
    router.Handle("/", websocketServer)

    log.Println("Serving websockets on localhost:8080")
    log.Fatal(http.ListenAndServe(":8080", router))
}
```

Create & run the Client

Continue with our client-side.

```go
func runClient() {
    // Make the client error-aware of the `errDayoff`
    // If that's missing the `Message.Err.Error() string` would still match it
    // but comparison of err == errDayOff wouldn't be a valid one.
    neffos.RegisterKnownError(errDayOff)

    ctx := context.Background()

    client, err := neffos.Dial(ctx,
        gorilla.DefaultDialer,
        "ws://localhost:8080",

        // Empty events because client does not need an event callback in this case.
        // The client is only uses the `Ask` method which allows the caller
        // to manage the response manually even if a local event is not registered
        // at all.
        neffos.Namespaces{namespace: neffos.Events{}},
    )
    if err != nil {
        panic(err)
    }

    c, err := client.Connect(ctx, namespace)
    if err != nil {
        panic(err)
    }

    fmt.Println("Please specify a date of format: mm-dd-yyyy")

    for {
        fmt.Print(">> ")
        var date string
        fmt.Scanln(&date)

        response, err := c.Ask(ctx, "workday", []byte(date))
        if err != nil {
            if err == errDayOff {
                // >> 06-29-2019
                // it's a day off!
                fmt.Println("it's a day off!")
            } else {
                // >> 13-29-2019
                // error received: parsing time "13-29-2019": month out of range
                fmt.Printf("error received: %v\n", err)
            }

            continue
        }

        // >> 06-24-2019
        // it's Monday, do your job.
        fmt.Println(string(response.Body))
    }
}
```

Note that the error managment works the same way with the event-driven API too. Instead of `response, [err] := c.Ask(...)` the `err` error is stored at the incoming `Message.Err` field, a quick view:

```go
func onSomething(c *neffos.NSConn, msg neffos.Message) error {
    if msg.Err != nil {
        if msg.Err == errDayOff {
            // [handle errDayoff...]
        }
    }
}
```

Read more about Errors.

The Server Ask

Use this method when you:

- do NOT have access to the connection value that you want to ask for or
- want to wait for a reply from a client that may be served by other neffos server (when your app is scaled-out).

The `Server.Ask` method is like Server.Broadcast but it blocks until response, from a specific connection (when `"msg.To"` is filled) or from the first connection which will reply to this `"msg"`.

1. Accepts a context for deadline as its first input argument.
2. The second argument is the request message which should be sent to a specific namespace:event like the `Conn/NSConn.Ask`.

```go
Ask(ctx context.Context, msg Message) (Message, error)
```

Example Client Event:

```go
"onAsk": func(c *neffos.NSConn, msg neffos.Message) error {
    return neffos.Reply([]byte("I am fine"))
}
```

Usage

```go
response, err := websocketServer.Ask(context.TODO(), neffos.Message{
    // To:      toConnID,
    Namespace: namespace,
    Event:     "onAsk",
    Body:      []byte("how are you?"),
})

if err != nil {
    // [handle err...]
}
```

response.Body would be `[]byte("I am fine")`.

Errors

Neffos separates a body from an error on its send and receive process. When Emit or Reply the other side's `neffos.Message.Body` is filled with the provided data but if an event callback returns a non-nil error then the incoming `neffos.Message.Err` is set to a standard go `error` filled with the remote side's error text. However, depending on the application requirements, you may want to specify and give more meaning to those errors, i.e be able to compare them with local events like you do with `io.EOF` which may be returned from an `io.Reader`.

To support meaningful errors **in the same application**, each side that needs to compare local errors with remote ones, must mark them as "known" errors via the package-level `RegisterKnownError(err error)` function. All known errors will under-the-hoods be compared with an incoming error text and if match then this error is set to the `neffos.Message.Err` field, therefore the caller can directly compare between `knwonError == msg.Err`. For dynamic error (that its `Error() string` can differ based on a runtime condition) the `RegisterKnownError` handles a special case of error contains a `ResolveError(errorText string) bool` method.

Given the following sample:

```go
// myapp/shared/errors.go
import "errors"
import "github.com/kataras/neffos"

var MyError = errors.New("my error")

func init() {
    // Now the 'MyError' can be compared against
    // a server or client side's incoming Message.Err field.
    neffos.RegisterKnownError(MyError)
}
```

```go
// myapp/server/server.go
import "myapp/shared"
import "github.com/kataras/neffos"

func onSomething(c *neffos.NSConn, msg neffos.Message) error {
    if somethingBad {
        // send the MyError as a response
        // of this namespace's event to the client side.
        return shared.MyError
    }

    // [...]
    return nil
}
```

```go
// myapp/client/client.go
import "myapp/shared"
import "github.com/kataras/neffos"

func onSomething(c *neffos.NSConn, msg neffos.Message) error {
    if msg.Err != nil {
        if msg.Err == shared.MyError {
            // [handle MyError coming from server to client...]
        }

        // [handle other error...]
    }


    // [...]
    return nil
}
```

Read the Request-Response architecture section for a more comprehensive usage of that kind of errors.

Continue by reading about Namespaces.

Namespaces

We've already seen how to declare and connect to namespaces but details were missing. In this section you will learn how to disallow a client from connecting to a namespace and what a namespace really is.

Here is a list of what we mean with the word `Namespaces`.

1. Each connection can be connected to one or more namespaces.
2. Namespaces should be static and should be declared in both server and client-side(clients can ignore some of the server's namespaces if not willing to connect to them).
3. Client should connect to a namespace, even if its name is empty `""`.
4. Client and server are notified of each namespace connection and disconnection through their `OnNamespaceConnect, OnNamespaceConnected` and `OnNamespaceDisconnect` built-in events.
5. Client can ask server to connect to a namespace.
6. Server can ask client to connect to a namespace.
7. Server and client can accept or decline the client or server's connection to a namespace through their `OnNamespaceConnect -> return err != nil`.
8. Client can ask server to disconnect from a namespace.
9. Server can forcely disconnect a client from a namespace.

```go
// blocks until connect in both sides.
nsConn, err := conn.Connect(ctx, "namespace")
```

```go
nsConn.Emit("event", body)
// nsConn.Conn.Write(neffos.Message{
//     Namespace: "namespace",
//     Event:     "event",
//     Body:      body,
// })
```

```go
nsConn.Disconnect(ctx)
```

Continue by reading about Rooms.

Rooms

Neffos supports any number of rooms under a connected namespace.

1. Each connection connected to a namespace can join to one or more rooms.
2. Rooms are dynamic.
3. `Server#Broadcast(Message {Namespace: ..., Room: ...})` sends data to all clients connected to a specific `Namespace:Room`.
4. Client and server can ask remote side to join to a room, get notified by `OnRoomJoin` and `OnRoomJoined`.
5. Client and server can ask remote side to leave from a room, get notified by `OnRoomLeave` and `OnRoomLeft`.
6. Client and server can disallow remote request for room join through their `OnRoomJoin -> return err != nil`.
7. Client and server can disallow remote request for room leave through their `OnRoomLeave -> return err != nil`.

```go
// blocks until join in both sides.
room, err := nsConn.JoinRoom(ctx, "room")
```

```go
room.Emit("event", body)
// room.NSConn.Conn.Write(neffos.Message{
//     Namespace: "namespace",
//     Room:      "room",
//     Event:     "event",
//     Body:      body,
// })
```

```go
room.Leave(ctx)
```

Timeouts

Neffos provides a wrapper of `neffos.Namespaces` and `neffos.Events` structures to deal with custom Write and Read deadtimes/timeouts, it's the neffos.WithTimeout.

Its outline looks like this:

```go
// WithTimeout completes the `ConnHandler` interface.
// Can be used to register namespaces and events or just events on an empty namespace
// with Read and Write timeouts.
//
// See `New` and `Dial`.
type WithTimeout struct {
    ReadTimeout  time.Duration
    WriteTimeout time.Duration

    Namespaces Namespaces
    Events     Events
}
```

It's easy to set timeouts, instead of `var events = neffos.Namespaces{"namespace": neffos.Events: ...}` you just wrap it with `neffos.WithTimeout` like this:

```go
var events = neffos.WithTimeout {
    ReadTimeout:  60 * time.Second,
    WriteTimeout: 60 * time.Second,

    Namespaces: neffos.Namespaces {
        "namespace": neffos.Events {
            "onMyEvent": func(c *neffos.NSConn, msg neffos.Message) error {
                // [...]
            },
        },
    },
}
```

or for empty namespace:

```go
var events = neffos.WithTimeout {
    ReadTimeout:  60 * time.Second,
    WriteTimeout: 60 * time.Second,

    Events: neffos.Events {
        "onMyEvent": func(c *neffos.NSConn, msg neffos.Message) error {
            // [...]
        },
    },

}
```

```go
websocketServer := neffos.New(gobwas.DefaultUpgrader, events)
```

```go
websocketClient, err := neffos.Dial(
    context.Background(),
    gobwas.DefaultDialer,
    "ws://localhost:8080/echo",
    events,
)
```

Of cource you could provide those timeouts based on what `Upgrader` and `Dialer` implementations are being used on Server and Client respectfully, i.e the gorilla's one has `ReadTimeout` and `WriteTimeout` optional fields that you can set on a custom `gorilla/websocket.Upgrader` structure but if you want a universal way to set your own timeouts in the connection handler level(namespaces, events) use the `neffos.WithTimeout` instead.

Broadcast

Neffos has an entirely different and much more efficient approach to broadcasting compared to alternatives that usually are hinder the whole connections list as a result not a single new client can connect or disconnect as long a message is broadcasting which it is unacceptable for high-scale systems. Neffos broadcasting system does **not require blocking**.

As we've seen on the Getting Started section, a client cannot directly communicate with the rest of the clients, they can communicate only through server. Therefore the only broadcaster is the Server itself. To send a message to all connections we use the `Server#Broadcast(exceptConnID, neffos.Message{...})`.

Message broadcasting

When we want to **broadcast an incoming message except the client who sent this message** through an event we use the following snippet:

```
neffos.Events {
    "onSomething": func(c *neffos.NSConn, msg neffos.Message) error {
        if !c.Conn.IsClient() {
            c.Conn.Server().Broadcast(c, msg)
            return nil
        }

        // [...]
    },
}
```

Now, as you can imagine we can pass any `neffos.Message` as an input argument to the `Server#Broadcast` method. The Message data structure looks like this:

```
type Message struct {
    Namespace string
    Room string
    Event string

    Body []byte
    Err error

    To string

    // [...]
}
```

To fire a particular remote event with a specific message body to a set of connections that are joined to a particular room inside a "default" namespace:

```
// [websocketServer := neffos.New...]

websocketServer.Broadcast(nil, neffos.Message{
    Namespace: "default", // <-
    Event: "toThatEvent", // <-
    Room: "room1", // <-
    Body: []byte("the data"),
})
```

Let's see how we can broadcast a message to a **specific connection**. To send a message to a specific connection you need to know its ID and set it to the `Message.To` field, let's assume that ID is `"X-User-1"`.

```
// [websocketServer := neffos.New...]

websocketServer.Broadcast(nil, neffos.Message{
    Namespace: "default", // <-
    Event: "toThatEvent", // <-
    Body: []byte("the data"),
    To: "X-User-1", // <-
})
```

> The first parameter sets whether a message should be broadcasted to all **except this**, i.e `neffos.Exclude("connectionID")` or a `Connection instance`. So, if it's nil it broadcasts to all connected connections based on their connected Namespace and Room (if any).

Another way to send a message to a particular client when you have access to the connection is by its `Connection#Emit` or `Connection#Write`.

There is a `Server.SyncBroadcaster` option field. If set to true, changes the default behavior and synchronizes `Server.Broadcast` calls when `Server.StackExchange` is nil (or `Server.UseStackExchange` is not used). When [[StackExchange|scale-out]] is used then this field is ignored, published messages are correctly handled by Redis or Nats.

Perform connection actions outside of events

Except `Server#Broadcast` you can use the `Server#Do` method, **which blocks new connections**, to execute actions on connections from the server side.

```go
// Do loops through all connected connections and fires the "fn", with this method
// callers can do whatever they want on a connection outside of a event's callback,
// but make sure that these operations are not taking long time to complete
// because it delays the new incoming connections.
// If "async" is true then this method does not block the flow of the program.
Do(fn func(*Conn), async bool)
```

For example to force-disconnect all clients from a namespace outside of an event callback you can use the following code:

```go
// [websocketServer := neffos.New...]

websocketServer.Do(func(c *neffos.Conn){
    c.Namespace("default").Disconnect(context.Background())
}, false)

log.Printf("%d clients are disconnected from 'default' namespace by the server.",
    websocketServer.GetTotalConnections())
```

> `Server#GetTotalConnections` method returns the total number of active client connections.

Or to force-close the connections entirely:

```go
// [websocketServer := neffos.New...]

websocketServer.Do(func(c *neffos.Conn){
    c.Close()
}, false)

log.Println("all clients are terminated by the server.")
```

Authentication

You can check for authentication and force-close a client at any state of your application, on `Server.OnConnect` or through event callbacks for example. The recommended way is to do that on the HTTP handshake, before upgrade to the websocket subprotocol, but it really depends on your app's flow and requirements.

Let's see three small examples here, two with `net/http` and one more real-world example using JWT authentication and Iris.

The first one is simple enough, it does the authentication before the websocket server's handler execution, through a middleware using the std `net/http` library.

Let's assume that the `isAuthenticated(r *http.Request) bool` function is our method to check if a client request is authenticated or not.

```go
// [websocketServer := neffos.New...]

router := http.NewServeMux()
router.HandleFunc("/websocket_endpoint", func(w http.ResponseWriter, r *http.Request){
    if !isAuthenticated(r) {
        statusCode := http.StatusUnauthorized // 401.
        http.Error(w, http.StatusText(statusCode), statusCode)
        return
    }

    // if succeed continue to our websocket app.
    websocketServer.ServeHTTP(w,r)
})

http.ListenAndServe(":8080", router)
```

The second one is simple too, we just move the authentication process to the `Server.OnConnect` instead. You can get the original `http.Request` by `Conn.Socket().Request()`.

Remember, if a `Server.OnConnect` event returns a non-nil error then the server immediately closes the connection before everything else. The client-side will receive this error through its `client, err := neffos.Dial(...)` for Go client and `catch` callback of the javascript's `neffos.dial(...)` function(see below).

```go
// [websocketServer := neffos.New...]
websocketServer.OnConnect = func(c *neffos.Conn) error {
    r := c.Socket().Request()
    if !isAuthenticated(r) {
        // this error will return back to
        // the client's `neffos.Dial` fucntion.
        return errors.New("not authenticated")
    }

    // if succeed...
    log.Printf("[%s] connected to the server", c.ID())
    return nil
}
```

```go
// [client, err := neffos.Dial...]
if err != nil {
    // err.Error() == "not authenticated"
    // [handle unauthenticated error...]
}
```

Finally, let's continue with a more real-world example using JWT and Iris. Iris offers a community-driven jwt middleware to use.

1. Install it by executing the following shell command:

```sh
$ go get github.com/iris-contrib/middleware/jwt
# and if you don't have Iris installed yet:
$ go get github.com/kataras/iris/v12@latest
```

2. This example extracts the token through a `"token"` url parameter. Authenticated clients should be designed to set that with a signed token. This middleware has two methods, the first one is the `Serve` method - it is an `iris.Handler` and the second one is the `CheckJWT(iris.Context) bool`.

```go
import (
    // [...]
    "github.com/iris-contrib/middleware/jwt"
)

// [...]

j := jwt.New(jwt.Config{
    // Extract by the "token" url.
    Extractor: jwt.FromParameter("token"),

    ValidationKeyGetter: func(token *jwt.Token) (interface{}, error) {
        return []byte("My Secret"), nil
    },
    SigningMethod: jwt.SigningMethodHS256,
})
```

3. To register it before websocket upgrade process, you simply prepend the jwt `j` middleware to the registered websocket endpoint route, before the `websocket.Handler(websocketServer)`.

```go
import (
    "github.com/kataras/iris/v12"
    "github.com/kataras/iris/v12/websocket"
    "github.com/kataras/neffos"

    "github.com/iris-contrib/middleware/jwt"
)

func main(){
    app := iris.New()
    // [websocketServer := neffos.New....]
    app.Get("/echo", j.Serve, websocket.Handler(websocketServer))
    app.Run(iris.Addr(":8080"))
}
```

3.1. To register it on an event you can use jwt middleware's `CheckJWT(iris.Context) error` method and return any error to the event callback.

```go
// [websocketServer := neffos.New....]
websocketServer.OnConnect = func(c *neffos.Conn) error {
    ctx := websocket.GetContext(c)
    if err := j.CheckJWT(ctx); err != nil {
        // will send the above error on the client
        // and will not allow it to connect
        // to the websocket server at all.
        return err
    }

    log.Printf("[%s] connected to the server", c.ID())
    return nil
}
```

4. To get the claims/the payload inside a websocket event callback:

- when you have access to the jwt middleware you can just use its `Get` method.

```go
ctx := websocket.GetContext(...)
user :=  j.Get(ctx)
```

- otherwise use the `ctx.Values().Get("jwt")` which is the key that the `*jwt.Token` value is stored on authenticated requests.

```go
func (nsConn *neffos.NSConn, msg neffos.Message) error {
    ctx := websocket.GetContext(nsConn.Conn)
    user := ctx.Values().Get("jwt").(*jwt.Token)

    log.Printf("This is an authenticated request\n")
    log.Printf("Claim content:")
    log.Printf("%#+v\n", user.Claims)
    // [...]
}
```

5. And the client-side, at this case a browser, might look something like that.

```
var token = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIx"+
"MjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0Ijoz"+
"MjEzMjF9.8waEX7-vPKACa-Soi1pQvW3Rl8QY-SUFcHKTLZI4mvU";

var wsURL = "ws://localhost:8080/echo?token=" + token;
// [...]
try {
    var conn = await neffos.dial(wsURL, { namespace: events...});
} catch(err) {
    // [handle unauthenticated error...]
}
```

> Of course in production, your server side must contain routes that their job will be to create the tokens for each of your users and then the client-side must store it and send it as HTTP Header on each request to the server, the default jwt's behavior to extract a token value is by the `Authentication: Bearer $TOKEN` header.

Scale out

Scale out through StackExchange interface and its completors is a feature that was implemented at neffos v0.0.6.

Made after a user feature request* with the five following commits:

1. New feature: Scale-out using Redis built-in StackExchange implementation
2. Add the ability to register and use more than one StackExchange
3. Add nats StackExchange and scale-out example
4. Add nats support for the new Server.Ask method
5. Add redis support for the new Server.Ask method
6. Export nats subject prefix

Scaling out a neffos app can be achieved by adding just three extra lines of code(!)

1. Import `github.com/kataras/neffos/stackexchange/redis` or `nats`
2. Initialize the StackExchange, e.g. `exc := redis.NewStackExchange(...)`
3. Right after the server := neffos.New add `server.UseStackExchange(exc)`.

Each StackExchange contains its own configuration.

How it works

1. Each client connected to a neffos namespace is subscribed a channel/subject and unsubscribes itself on namespace disconnection.
2. Each time a neffos Broadcast is called the message directly goes to the registered StackExchange.Publish method, which takes care to publish the message to the correct channel/subject, so subscribers can be notified about this message.
3. Each time a subscriber notified for a new message its Conn.Write is responsible to dispatch that message to the client side.
4. The same pattern is used for one to one messaging and Server.Ask.

Use this feature when you have multiple neffos servers hosted independently for a particular neffos app of yours and you want your app to be able to communicate with all of those neffos servers as one.

Next steps

- Scale out using Redis
- Scale out using Nats

Scale out using Redis

This section explains aspects of setting up a Redis server for scaling out.

Set up

1. Download & Run Redis from https://redis.io/download
   - or for Windows OS https://github.com/ServiceStack/redis-windows
2. In the Go import statements add `github.com/kataras/neffos/stackexchange/redis` and use it.

Use

The first input argument is the optional redis client configuration.

The second one is the redis **channel prefix** which messages will be published and connections will be subscribed to.

Setting a channel prefix isolates one neffos app from others that use different channel prefixes.

```go
import "github.com/kataras/neffos/stackexchange/redis"

// [server := neffos.New...]

exc, err := redis.NewStackExchange(redis.Config{}, "MyChatApp")
if err != nil {
        // [...]
}
server.UseStackExchange(exc)
```

Configure options as needed

Options can be set in the `Config` structure.

Name	Description	Def
Network	Protocol to use	"
Addr	Network address of a single redis instance	"127.0.0.1:63
Clusters	A list of network addresses for redis clusters	
Password	A password to connect	en
DialTimeout	Max time to connect	
MaxActive	Keep open at least the given number of connections to the redis instance	

Redis Clustering

Redis Clustering is a method for achieving high availability by using multiple Redis servers. Clustering is supported by the `Clusters []string` configuration field.

References

- What is Redis Clustering
- Redis documentation

Scale out using Nats

This section explains aspects of setting up a Nats server for scaling out.

Set up

1. Download & Run Nats from https://nats-io.github.io/docs/nats_server/installation.html
2. In the Go import statements add `github.com/kataras/neffos/stackexchange/nats` and use it.

Use

The first and only required input argument is the URL connection string of the nats server(s). Servers addresses are separated by comma `,` . If it's empty then it defaults to `"nats://127.0.0.1:4222"` .

The second one is optional and can be used to register various nats client options, such as authentication, see the available nats.go options.

The nats StackExchange's client configuration defaults to nats.go DefaultOptions.

```go
import "github.com/kataras/neffos/stackexchange/nats"

// [server := neffos.New...]

exc, err := nats.NewStackExchange(":4222")
if err != nil {
        // [...]
}
server.UseStackExchange(exc)
```

If you use the same nats server instance for multiple neffos apps, set its `exc.SubjectPrefix` to different values across your apps.

References

- What is Nats
- Nats Server Configuration

Encoding

With neffos you can send any type of data, remember? The `neffos.Message.Body` field is just a `[]byte`. At short, the `neffos.Message.Body` is the raw data client/server sends, users of the neffos package can use any format to unmarshal on read and marshal to send, such as `protocol-buffers`, `encoding/json`, `encoding/xml` and etc.

In this section you will learn how to send `JSON` data to Rooms using the `Emit` with `neffos.Marshal` and how to read those data inside an event callback's with the `neffos.Message.Unmarshal` method. The `neffos.DefaultMarshaler/Unnmarshaler` will be used on `neffos.Marshal` and `neffos.Message.Unmarshal` if the value you are trying to send/read does not complete the `neffos.MessageObjectMarshaler` on send and `neffos.MessageObjectUnmarshaler` on read, therefore the default encoding that neffos using for struct values is the JSON one. Below you will find a brief outline of the above:

```go
package neffos

var (
    DefaultMarshaler = json.Marshal
    DefaultUnmarshaler = json.Unmarshal
)

type (
    MessageObjectMarshaler interface {
        Marshal() ([]byte, error)
    }

    MessageObjectUnmarshaler interface {
        Unmarshal(body []byte) error
    }
)

type Message struct {
    // [...]
}

func (m *Message) Unmarshal(outPtr interface{}) error {
    if unmarshaler, ok := outPtr.(MessageObjectUnmarshaler); ok {
        return unmarshaler.Unmarshal(m.Body)
    }

    return DefaultUnmarshaler(m.Body, outPtr)
}
```

Create our userMessage structure

```go
// file: main.go

// userMessage implements the `neffos.MessageBodyMarshaler` and
// `neffos.MessageBodyUnmarshaler` interfaces.
type userMessage struct {
    From string `json:"from"`
    Text string `json:"text"`
}

// Defaults to `DefaultUnmarshaler & DefaultMarshaler`
// that are calling the json.Unmarshal & json.Marshal respectfully
// if the instance's Marshal and Unmarshal methods are missing,
// however for the example shake we complete
// the MessageBodyMarshaler and MessageBodyUnmarshaler too,
// these can help you customize the out and in format.
func (u *userMessage) Marshal() ([]byte, error) {
    return json.Marshal(u)
}

func (u *userMessage) Unmarshal(b []byte) error {
    return json.Unmarshal(b, u)
}
```

Define the server and client events

```go
// file: main.go

var serverAndClientEvents = neffos.Namespaces{
    namespace: neffos.Events{
        neffos.OnNamespaceConnected: func(c *neffos.NSConn, msg neffos.Message) error {
            log.Printf("[%s] connected to namespace [%s].", c, msg.Namespace)
            return nil
        },
        neffos.OnNamespaceDisconnect: func(c *neffos.NSConn, msg neffos.Message) error {
            log.Printf("[%s] disconnected from namespace [%s].", c, msg.Namespace)
            return nil
        },

        neffos.OnRoomJoined: func(c *neffos.NSConn, msg neffos.Message) error {
            text := fmt.Sprintf("[%s] joined to room [%s].", c, msg.Room)
            log.Printf("\n%s", text)

            // notify others.
            if !c.Conn.IsClient() {
                c.Conn.Server().Broadcast(c, neffos.Message{
                    Namespace: msg.Namespace,
                    Room:      msg.Room,
                    Event:     "notify",
                    Body:      []byte(text),
                })
            }


            return nil
        },
        neffos.OnRoomLeft: func(c *neffos.NSConn, msg neffos.Message) error {
            text := fmt.Sprintf("[%s] left from room [%s].", c, msg.Room)
            log.Printf("\n%s", text)

            // notify others.
            if !c.Conn.IsClient() {
                c.Conn.Server().Broadcast(c, neffos.Message{
                    Namespace: msg.Namespace,
                    Room:      msg.Room,
                    Event:     "notify",
                    Body:      []byte(text),
                })
            }

            return nil
        },

        "chat": func(c *neffos.NSConn, msg neffos.Message) error {
            if !c.Conn.IsClient() {
                c.Conn.Server().Broadcast(c, msg)
            } else {
                var userMsg userMessage
                err := msg.Unmarshal(&userMsg)
                if err != nil {
                    log.Fatal(err)
                }
                fmt.Printf("%s >> [%s] says: %s\n",
                    msg.Room, userMsg.From, userMsg.Text)
            }
            return nil
        },
        // client-side only event to catch any server messages comes
        // from the custom "notify" event.
        "notify": func(c *neffos.NSConn, msg neffos.Message) error {
            if !c.Conn.IsClient() {
                return nil
            }

            fmt.Println(string(msg.Body))
            return nil
        },
    },
}
```

Create and run our Server

```go
// file: main.go

func startServer() {
    server := neffos.New(gobwas.DefaultUpgrader, serverAndClientEvents)
    server.IDGenerator = func(w http.ResponseWriter, r *http.Request) string {
        if userID := r.Header.Get("X-Username"); userID != "" {
            return userID
        }

        return neffos.DefaultIDGenerator(w, r)
    }

    server.OnUpgradeError = func(err error) {
        log.Printf("ERROR: %v", err)
    }

    server.OnConnect = func(c *neffos.Conn) error {
        if c.WasReconnected() {
            log.Printf("[%s] connection is a result of a client-side re-connection,
               with tries: %d", c.ID(), c.ReconnectTries)
        }

        log.Printf("[%s] connected to the server.", c)

        // if returns non-nil error then it refuses the client to
        // connect to the server.
        return nil
    }

    server.OnDisconnect = func(c *neffos.Conn) {
        log.Printf("[%s] disconnected from the server.", c)
    }

    log.Printf("Listening on: %s\nPress CTRL/CMD+C to interrupt.", addr)
    http.Handle("/", http.FileServer(http.Dir("./browser")))
    http.Handle(endpoint, server)
    log.Fatal(http.ListenAndServe(addr, nil))
}
```

Create our (Go) Client side

```go
// file: main.go

func startClient() {
    scanner := bufio.NewScanner(os.Stdin)

    fmt.Print("Please specify a username: ")
    if !scanner.Scan() {
        return
    }
    username := scanner.Text()

    // init the websocket connection by dialing the server.
    client, err := neffos.Dial(
        // Optional context cancelation and deadline for dialing.
        nil,
        // The underline dialer, can be also a gobwas.Dialer/DefautlDialer
        // or a gorilla.Dialer/DefaultDialer.
        // Here we wrap a custom gobwas dialer in order to send the username among,
        // on the handshake state,
        // see `startServer().server.IDGenerator`.
        gobwas.Dialer(gobwas.Options{
            Header: gobwas.Header{"X-Username": []string{username}}}),
        // The endpoint, i.e ws://localhost:8080/path.
        addr+endpoint,
        // The namespaces and events, can be optionally shared with the server's.
        serverAndClientEvents)

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

    defer client.Close()

    go func() {
        <-client.NotifyClose
        os.Exit(0)
    }()

    // connect to the "default" namespace.
    c, err := client.Connect(nil, namespace)
    if err != nil {
        log.Fatal(err)
    }
    askRoom:
    fmt.Print("Please specify a room to join, i.e room1: ")
    if !scanner.Scan() {
        log.Fatal(scanner.Err())
    }
    roomToJoin := scanner.Text()

    room, err := c.JoinRoom(nil, roomToJoin)
    if err != nil {
        log.Fatal(err)
    }

    fmt.Fprint(os.Stdout, ">> ")

    for {
        if !scanner.Scan() {
            log.Printf("ERROR: %v", scanner.Err())
            break
        }

        text := scanner.Text()

        if text == "exit" {
            break
        }

        if text == "leave" {
            room.Leave(nil)
            goto askRoom
        }

        // username is the connection's ID ==
        // room.String() returns -> NSConn.String() returns ->
        // Conn.String() returns -> Conn.ID()
        // which generated by server-side via `Server#IDGenerator`.
        userMsg := userMessage{From: username, Text: text}
        room.Emit("chat", neffos.Marshal(userMsg))

        fmt.Fprint(os.Stdout, ">> ")
    }
}
```

Create our (Javascript browserify) Client side

In this example, we make an exception and we include the browser-side as well, so you can have a small taste of it.

Read more about neffos.js.

```javascript
// file: browser/app.js

const neffos = require('neffos.js');

var scheme = document.location.protocol == "https:" ? "wss" : "ws";
var port = document.location.port ? ":" + document.location.port : "";
var wsURL = scheme + "://" + document.location.hostname + port + "/echo";

var outputTxt = document.getElementById("output");

function addMessage(msg) {
    outputTxt.innerHTML += msg + "\n";
}

function handleError(reason) {
    console.log(reason);
    window.alert(reason);
}

class UserMessage {
    constructor(from, text) {
        this.from = from;
        this.text = text;
    }
}


async function handleNamespaceConnectedConn(nsConn) {
    const roomToJoin = prompt("Please specify a room to join, i.e room1: ");
    nsConn.joinRoom(roomToJoin);

    let inputTxt = document.getElementById("input");
    let sendBtn = document.getElementById("sendBtn");

    sendBtn.disabled = false;
    sendBtn.onclick = function () {
        const input = inputTxt.value;
        inputTxt.value = "";

        switch (input) {
            case "leave":
                nsConn.room(roomToJoin).leave();
                // or room.leave();
                break;
            default:
                const userMsg = new UserMessage(nsConn.conn.ID, input);
                nsConn.emit("chat", neffos.marshal(userMsg));
                addMessage("Me: " + input);
        }
    };
}
```

```javascript
async function runExample() {
    // You can omit the "default" and simply define only Events,
    // the namespace will be an empty string"",
    // however if you decide to make any changes on this example
    // make sure the changes are reflecting inside the ../server.go file as well.
    try {
        const username = prompt("Please specify a username: ");

        const conn = await neffos.dial(wsURL, {
            default: { // "default" namespace.
                _OnNamespaceConnected: function (nsConn, msg) {
                    addMessage("connected to namespace: " + msg.Namespace);
                    handleNamespaceConnectedConn(nsConn);

                },
                _OnNamespaceDisconnect: function (nsConn, msg) {
                    addMessage("disconnected from namespace: " + msg.Namespace);
                },
                _OnRoomJoined: function (nsConn, msg) {
                    addMessage("joined to room: " + msg.Room);
                },
                _OnRoomLeft: function (nsConn, msg) {
                    addMessage("left from room: " + msg.Room);
                },
                notify: function (nsConn, msg) {
                    addMessage(msg.Body);
                },
                chat: function (nsConn, msg) { // "chat" event.
                    const userMsg = msg.unmarshal()
                    addMessage(userMsg.from + ": " + userMsg.text);
                }
            }
        }, {
            headers: {
                'X-Username': username
            },
            // if > 0 then on network failures it tries to reconnect every 5 seconds,
            // defaults to 0 (disabled).
            reconnect: 5000
        });

        conn.connect("default");

    } catch (err) {
        handleError(err);
    }
}

runExample();
```

Third-party Requirements

- NPM

How to run

Open a terminal window instance and execute:

```
$ cd ./browser
# build the browser-side client: ./browser/bundle.js which
# ./browser/index.html imports.
$ npm install && npm run-script build
$ cd ../
$ go run main.go server # start the neffos websocket server.
```

Open some web browser windows and navigate to http://localhost:8080, each window will ask for a username and a room to join, each window(client connection) and server get notified for namespace connected/disconnected, room joined/left and chat events.

To start the go client side just open a new terminal window and execute:

```
$ go run main.go client
```

It will ask you for username and a room to join as well, it acts exactly the same as the `./browser/app.js` browser-side application.

Read the full source code of this example by navigating to the repository's _examples/basic directory.

You can continue by learning how to send and receive Protobufs.

Protobufs

Let's not waste time here, if you've already read the Encoding section, we can start right away.

Install protocol buffers

Install the protocol-buffers generator by downloading a github release of it and add it to your $PATH env variable, you can find the releases at this github page.

Details about protocol-buffers for Go can be found at evelopers.google.com/protocol-buffers/docs/gotutorial.

Define the data structure with protobuf

Let's start by defining what structure looks like.

Create a new file on your package named `user_message.proto` and fill it.

```
syntax="proto3";

package main;

message UserMessage {
    string Username =1;
    string Text = 2;
}
```

Generate proto for Go

Protobufs requires code generation.

Start a session terminal and execute:

```
$ protoc --go_out=. user_message.proto
```

This will create a `user_message.pb.go` which extends the data structure in protobuf form for Go.

Javascript support

- https://github.com/protocolbuffers/protobuf/tree/master/js#commonjs-imports
- https://github.com/protobufjs/protobuf.js#nodejs (with browserify)
    - We use that in _examples/protobuf/browser repository example.
- https://github.com/protobufjs/protobuf.js#browsers (alternative)

Proto Marshal and Unmarshal

```go
userMsg := &UserMessage{
    Username: "my username",
    Text: "text from terminal",
}

// send data to the "chat" event.
body, err := proto.Marshal(userMsg)
if err != nil {
    // [handle err...]
}
c.EmitBinary("chat", body)
```

```go
// assuming that we are on the other side's "chat" event's callback,
// here is how you can read protobuf data.
var userMsg UserMessage
if err := proto.Unmarshal(msg.Body, &userMsg); err != nil {
    return err
}

// userMsg.Username == "my username"
// userMsg.Text == "text from terminal"
```

Read the complete source code by navigating to the repository's _examples/protobuf directory.

Working with Native Messages

When app expects to accept and send only raw/native websocket messages almost all features of neffos are disabled because no custom message format can pass the "ack" step.

When only allow native messages is a fact?

- When the registered namespace is just one and it's empty and contains only one registered event which is the `OnNativeMessage`.
    - When `Events{...}` is used instead of `Namespaces{ "namespaceName": Events{...}}` then the namespace is empty "".

Native messages event can be used side by side with any other event if the application expects some clients to make use of neffos and some others to use the native websocket API provided by browsers and software libraries. However, it's not recommended to enable this feature unless it's really necessary, you use neffos for its features and performance, otherwise why bother?

Let's start by creating a simple websocket client using just the browser's websocket API.

```html
<html>

<body style="padding:10px;">
    <input type="text" id="messageTxt" />
    <button type="button" id="sendBtn">Send</button>
    <div id="messages"
    style="width: 375px;margin:10 0 0 0px;border-top: 1px solid black;">
    </div>

    <script type="text/javascript">
        var messageTxt = document.getElementById("messageTxt");
        var messages = document.getElementById("messages");
        var sendBtn = document.getElementById("sendBtn")

        // You see? No neffos at all here.
        w = new WebSocket("ws://localhost:8080/endpoint");
        w.onopen = function () {
            console.log("Websocket connection enstablished");
        };
        w.onclose = function () {
            appendMessage("<div><center><h3>Disconnected</h3></center></div>");
        };
        w.onmessage = function (message) {
            appendMessage("<div>" + message.data + "</div>");
        };

        sendBtn.onclick = function () {
            myText = messageTxt.value;
            messageTxt.value = "";


            appendMessage("<div style='color: red'> me: " + myText + "</div>");
            w.send(myText);
        };

        messageTxt.addEventListener("keyup", function (e) {
            if (e.keyCode === 13) {
                e.preventDefault();

                sendBtn.click();
            }
        });

        function appendMessage(messageDivHTML) {
            messages.insertAdjacentHTML('afterbegin', messageDivHTML);
        }
    </script>
</body>

</html>
```

Enable native messages feature is done by defining a `neffos.OnNativeMessage` built-in event.

```go
var pongMessage = []byte("pong")

var events = neffos.Events{
    neffos.OnNativeMessage: func(c *neffos.NSConn, msg neffos.Message) error {
        // ^^^^^^^^^^^^^^^
        log.Printf("Got: %s", string(msg.Body))

        if !c.Conn.IsClient() {
            return c.Conn.Socket().WriteText(pongMessage, 0)
            //                     ^^^^^^^^^
        }

        return nil
    },
}
```

Now let's make a neffos server that can accept and send to a client like the above.

```go
func runServer() {
    websocketServer := neffos.New(gorilla.DefaultUpgrader, events)

    router := http.NewServeMux()
    router.Handle("/endpoint", websocketServer)
    router.Handle("/", http.FileServer(http.Dir("./browser")))

    log.Println("Serving websockets on localhost:8080/endpoint")
    log.Fatal(http.ListenAndServe(":8080", router))
}
```

It's also possible to create a go-client side using any third-party websocket library or the standard `net/x/websocket`, however in this case we will create a neffos one:

```go
func runClient() {
    ctx := context.Background()

    client, err := neffos.Dial(ctx,
        gorilla.DefaultDialer,
        "ws://localhost:8080/endpoint",
        events)

    if err != nil {
        panic(err)
    }

    // Connect does nothing at this case,
    // it just returns a connection to work with its methods.
    c, err := client.Connect(ctx, "")
    if err != nil {
        panic(err)
    }

    c.Conn.Socket().WriteText([]byte("ping"), 0)

    <-client.NotifyClose
}
```

The full source code of the example above is available at _examples/native-messages.

Project & Community

Proud to be involved in developing, releasing and nurturing open source projects, helping to build and sustain developer-based communities around them.

Let's build the future of real-time cross-platform communication together.

Contributing

To get involved, share your ideas, and collaborate with us, head to our Contributing home page.

Tell the world

Using neffos in your company or a projects of yours? Add your site to our list!

Improve neffos

- Ticket system - report bugs and make feature requests
- Chat to us - in our #neffos-framework channel on Gitter
- neffos wiki - fork it to contribute tips and documentation