**Go Micro**

Go Micro is a pluggable RPC framework for distributed systems development.

The **micro** philosophy is sane defaults with a pluggable architecture. We provide defaults to get you started quickly but everything can be easily swapped out. It comes with built in support for {json,proto}-rpc encoding, consul or multicast dns for service discovery, http for communication and random hashed client side load balancing.

Plugins are available at [github.com/micro/go-plugins](https://github.com/micro/go-plugins).

**Features**

Go Micro abstracts away the details of distributed systems. Here are the main features.

* **Service Discovery** - Automatic service registration and name resolution
* **Load Balancing** - Client side load balancing built on discovery
* **Message Encoding** - Dynamic encoding based on content-type with protobuf and json support
* **Sync Streaming** - RPC based communication with support for bidirectional streaming
* **Async Messaging** - Native PubSub messaging built in for event driven architectures

Go Micro supports both the Service and Function programming models. Read on to learn more.

**Learn By Example**

An example service can be found in [**examples/service**](https://github.com/micro/examples/tree/master/service) and function in [**examples/function**](https://github.com/micro/examples/tree/master/function).

The [**examples**](https://github.com/micro/examples) directory contains examples for using things such as middleware/wrappers, selector filters, pub/sub, grpc, plugins and much more. For the complete greeter example look at [**examples/greeter**](https://github.com/micro/examples/tree/master/greeter). Other examples can be found throughout the GitHub repository.

Watch the [Golang UK Conf 2016](https://www.youtube.com/watch?v=xspaDovwk34) video for a high level overview.

**Getting started**

* [Install Protobuf](https://github.com/micro/go-micro#install-protobuf)
* [Service Discovery](https://github.com/micro/go-micro#service-discovery)
* [Writing a Service](https://github.com/micro/go-micro#writing-a-service)
* [Writing a Function](https://github.com/micro/go-micro#writing-a-function)
* [Publish & Subscribe](https://github.com/micro/go-micro#publish--subscribe)
* [Plugins](https://github.com/micro/go-micro#plugins)
* [Wrappers](https://github.com/micro/go-micro#wrappers)

**Install Protobuf**

Protobuf is required for code generation

You'll need to install:

* [protoc-gen-micro](https://github.com/micro/protoc-gen-micro)

**Service Discovery**

Service discovery is used to resolve service names to addresses.

**Consul**

[Consul](https://www.consul.io/) is used as the default service discovery system.

Discovery is pluggable. Find plugins for etcd, kubernetes, zookeeper and more in the [micro/go-plugins](https://github.com/micro/go-plugins) repo.

[Install guide](https://www.consul.io/intro/getting-started/install.html)

**Multicast DNS**

[Multicast DNS](https://en.wikipedia.org/wiki/Multicast_DNS) is a built in service discovery plugin for a zero dependency configuration.

Pass --registry=mdns to any command or the enviroment variable MICRO\_REGISTRY=mdns

MICRO\_REGISTRY=mdns go run main.go

**Writing a service**

This is a simple greeter RPC service example

Find this example at [examples/service](https://github.com/micro/examples/tree/master/service).

**Create service proto**

One of the key requirements of microservices is strongly defined interfaces. Micro uses protobuf to achieve this.

Here we define the Greeter handler with the method Hello. It takes a HelloRequest and HelloResponse both with one string arguments.

syntax = "proto3";

service Greeter {

rpc Hello(HelloRequest) returns (HelloResponse) {}

}

message HelloRequest {

string name = 1;

}

message HelloResponse {

string greeting = 2;

}

**Generate the proto**

After writing the proto definition we must compile it using protoc with the micro plugin.

protoc --proto\_path=$GOPATH/src:. --micro\_out=. --go\_out=. path/to/greeter.proto

**Write the service**

Below is the code for the greeter service.

It does the following:

1. Implements the interface defined for the Greeter handler
2. Initialises a micro.Service
3. Registers the Greeter handler
4. Runs the service

package main

import (

"context"

"fmt"

micro "github.com/micro/go-micro"

proto "github.com/micro/examples/service/proto"

)

type Greeter struct{}

func (g \*Greeter) Hello(ctx context.Context, req \*proto.HelloRequest, rsp \*proto.HelloResponse) error {

rsp.Greeting = "Hello " + req.Name

return nil

}

func main() {

// Create a new service. Optionally include some options here.

service := micro.NewService(

micro.Name("greeter"),

)

// Init will parse the command line flags.

service.Init()

// Register handler

proto.RegisterGreeterHandler(service.Server(), new(Greeter))

// Run the server

if err := service.Run(); err != nil {

fmt.Println(err)

}

}

**Run service**

go run examples/service/main.go

Output

2016/03/14 10:59:14 Listening on [::]:50137

2016/03/14 10:59:14 Broker Listening on [::]:50138

2016/03/14 10:59:14 Registering node: greeter-ca62b017-e9d3-11e5-9bbb-68a86d0d36b6

**Define a client**

Below is the client code to query the greeter service.

The generated proto includes a greeter client to reduce boilerplate code.

package main

import (

"context"

"fmt"

micro "github.com/micro/go-micro"

proto "github.com/micro/examples/service/proto"

)

func main() {

// Create a new service. Optionally include some options here.

service := micro.NewService(micro.Name("greeter.client"))

service.Init()

// Create new greeter client

greeter := proto.NewGreeterService("greeter", service.Client())

// Call the greeter

rsp, err := greeter.Hello(context.TODO(), &proto.HelloRequest{Name: "John"})

if err != nil {

fmt.Println(err)

}

// Print response

fmt.Println(rsp.Greeting)

}

**Run the client**

go run client.go

Output

Hello John

**Writing a Function**

Go Micro includes the Function programming model.

A Function is a one time executing Service which exits after completing a request.

**Defining a Function**

package main

import (

"context"

proto "github.com/micro/examples/function/proto"

"github.com/micro/go-micro"

)

type Greeter struct{}

func (g \*Greeter) Hello(ctx context.Context, req \*proto.HelloRequest, rsp \*proto.HelloResponse) error {

rsp.Greeting = "Hello " + req.Name

return nil

}

func main() {

// create a new function

fnc := micro.NewFunction(

micro.Name("greeter"),

)

// init the command line

fnc.Init()

// register a handler

fnc.Handle(new(Greeter))

// run the function

fnc.Run()

}

It's that simple.

**Publish & Subscribe**

Go-micro has a built in message broker interface for event driven architectures.

PubSub operates on the same protobuf generated messages as RPC. They are encoded/decoded automatically and sent via the broker. By default go-micro includes a point-to-point http broker but this can be swapped out via go-plugins.

**Publish**

Create a new publisher with a topic name and service client

p := micro.NewPublisher("events", service.Client())

Publish a proto message

p.Publish(context.TODO(), &proto.Event{Name: "event"})

**Subscribe**

Create a message handler. It's signature should be func(context.Context, v interface{}) error.

func ProcessEvent(ctx context.Context, event \*proto.Event) error {

fmt.Printf("Got event %+v\n", event)

return nil

}

Register the message handler with a topic

micro.RegisterSubscriber("events", ProcessEvent)

See [examples/pubsub](https://github.com/micro/examples/tree/master/pubsub) for a complete example.

**Plugins**

By default go-micro only provides a few implementation of each interface at the core but it's completely pluggable. There's already dozens of plugins which are available at [github.com/micro/go-plugins](https://github.com/micro/go-plugins). Contributions are welcome!

**Build with plugins**

If you want to integrate plugins simply link them in a separate file and rebuild

Create a plugins.go file

import (

// etcd v3 registry

\_ "github.com/micro/go-plugins/registry/etcdv3"

// nats transport

\_ "github.com/micro/go-plugins/transport/nats"

// kafka broker

\_ "github.com/micro/go-plugins/broker/kafka"

)

Build binary

// For local use

go build -i -o service ./main.go ./plugins.go

Flag usage of plugins

service --registry=etcdv3 --transport=nats --broker=kafka

**Plugin as option**

Alternatively you can set the plugin as an option to a service

import (

"github.com/micro/go-micro"

// etcd v3 registry

"github.com/micro/go-plugins/registry/etcdv3"

// nats transport

"github.com/micro/go-plugins/transport/nats"

// kafka broker

"github.com/micro/go-plugins/broker/kafka"

)

func main() {

registry := etcdv3.NewRegistry()

broker := kafka.NewBroker()

transport := nats.NewTransport()

service := micro.NewService(

micro.Name("greeter"),

micro.Registry(registry),

micro.Broker(broker),

micro.Transport(transport),

)

service.Init()

service.Run()

}

**Write plugins**

Plugins are a concept built on Go's interface. Each package maintains a high level interface abstraction. Simply implement the interface and pass it in as an option to the service.

The service discovery interface is called [Registry](https://godoc.org/github.com/micro/go-micro/registry#Registry). Anything which implements this interface can be used as a registry. The same applies to the other packages.

type Registry interface {

Register(\*Service, ...RegisterOption) error

Deregister(\*Service) error

GetService(string) ([]\*Service, error)

ListServices() ([]\*Service, error)

Watch() (Watcher, error)

String() string

}

Browse [go-plugins](https://github.com/micro/go-plugins) to get a better idea of implementation details.

**Wrappers**

Go-micro includes the notion of middleware as wrappers. The client or handlers can be wrapped using the decorator pattern.

**Handler**

Here's an example service handler wrapper which logs the incoming request

// implements the server.HandlerWrapper

func logWrapper(fn server.HandlerFunc) server.HandlerFunc {

return func(ctx context.Context, req server.Request, rsp interface{}) error {

fmt.Printf("[%v] server request: %s", time.Now(), req.Method())

return fn(ctx, req, rsp)

}

}

It can be initialised when creating the service

service := micro.NewService(

micro.Name("greeter"),

// wrap the handler

micro.WrapHandler(logWrapper),

)

**Client**

Here's an example of a client wrapper which logs requests made

type logWrapper struct {

client.Client

}

func (l \*logWrapper) Call(ctx context.Context, req client.Request, rsp interface{}, opts ...client.CallOption) error {

fmt.Printf("[wrapper] client request to service: %s method: %s\n", req.Service(), req.Method())

return l.Client.Call(ctx, req, rsp)

}

// implements client.Wrapper as logWrapper

func logWrap(c client.Client) client.Client {

return &logWrapper{c}

}