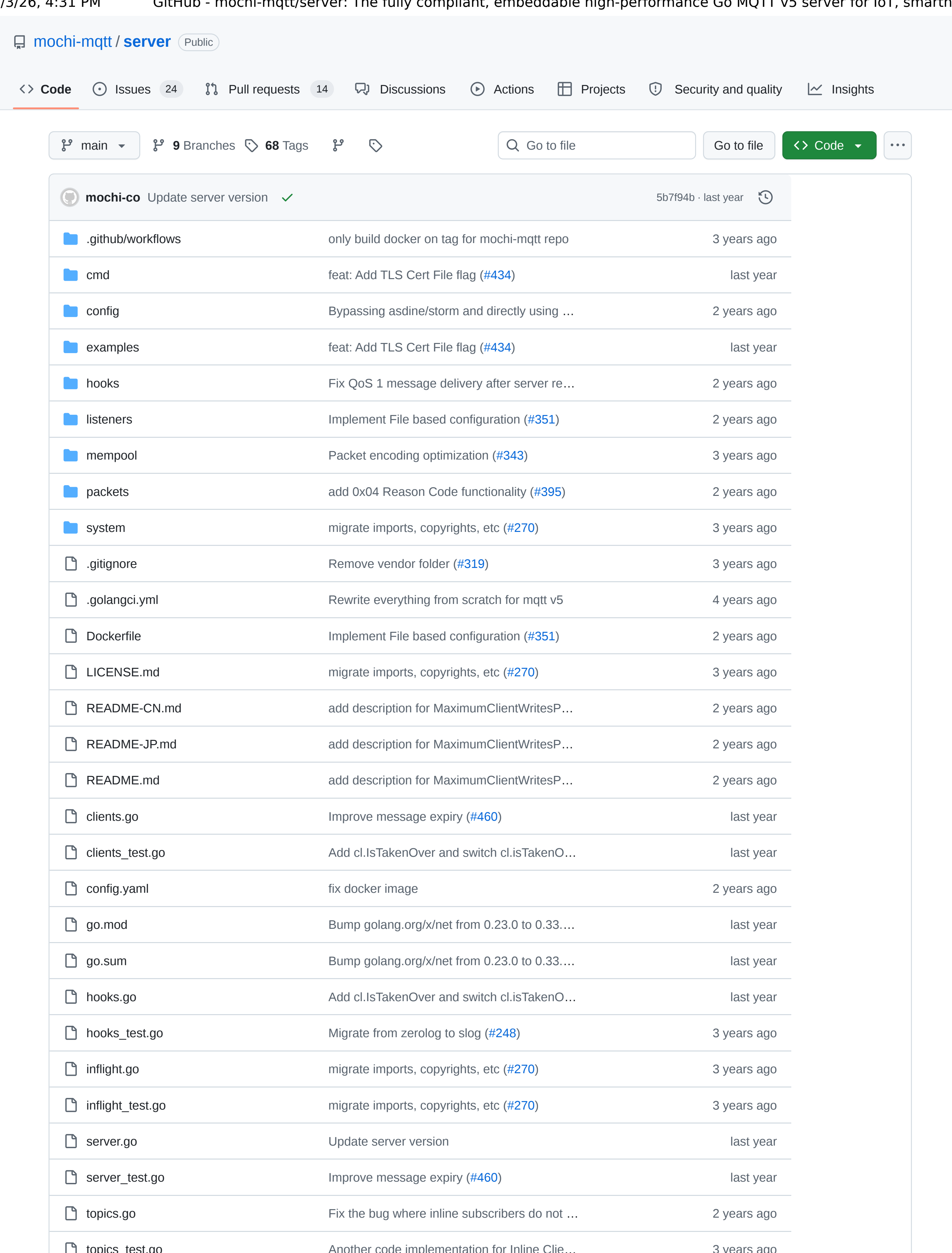


mochi-mqtt / **server** Public

Code | Issues 24 | Pull requests 14 | Discussions | Actions | Projects | Security and quality | Insights

main | **9** Branches | **68** Tags | Go to file | Go to file | Code

mochi-co Update server version ✓ 5b7f94b · last year

Name	Last commit message	Last commit date
.github/workflows	only build docker on tag for mochi-mqtt repo	3 years ago
cmd	feat: Add TLS Cert File flag (#434)	last year
config	Bypassing asdine/storm and directly using …	2 years ago
examples	feat: Add TLS Cert File flag (#434)	last year
hooks	Fix QoS 1 message delivery after server re…	2 years ago
listeners	Implement File based configuration (#351)	2 years ago
mempool	Packet encoding optimization (#343)	3 years ago
packets	add 0x04 Reason Code functionality (#395)	2 years ago
system	migrate imports, copyrights, etc (#270)	3 years ago
.gitignore	Remove vendor folder (#319)	3 years ago
.golangci.yml	Rewrite everything from scratch for mqtt v5	4 years ago
Dockerfile	Implement File based configuration (#351)	2 years ago
LICENSE.md	migrate imports, copyrights, etc (#270)	3 years ago
README-CN.md	add description for MaximumClientWritesP…	2 years ago
README-JP.md	add description for MaximumClientWritesP…	2 years ago
README.md	add description for MaximumClientWritesP…	2 years ago
clients.go	Improve message expiry (#460)	last year
clients_test.go	Add cl.IsTakenOver and switch cl.isTakenO…	last year
config.yaml	fix docker image	2 years ago
go.mod	Bump golang.org/x/net from 0.23.0 to 0.33.…	last year
go.sum	Bump golang.org/x/net from 0.23.0 to 0.33.…	last year
hooks.go	Add cl.IsTakenOver and switch cl.isTakenO…	last year
hooks_test.go	Migrate from zerolog to slog (#248)	3 years ago
inflight.go	migrate imports, copyrights, etc (#270)	3 years ago
inflight_test.go	migrate imports, copyrights, etc (#270)	3 years ago
server.go	Update server version	last year
server_test.go	Improve message expiry (#460)	last year
topics.go	Fix the bug where inline subscribers do not …	2 years ago
topics_test.go	Another code implementation for Inline Clie…	3 years ago

README MIT license

Mochi-MQTT Server

English | 简体中文 | 日本語 | Translators Wanted!

🎆 **mochi-co/mqtt is now part of the new mochi-mqtt organisation.** Read about this announcement here.

Mochi-MQTT is a fully compliant, embeddable high-performance Go MQTT v5 (and v3.1.1) broker/server

Mochi MQTT is an embeddable fully compliant MQTT v5 broker server written in Go, designed for the development of telemetry and internet-of-things projects. The server can be used either as a standalone binary or embedded as a library in your own applications, and has been designed to be as lightweight and fast as possible, with great care taken to ensure the quality and maintainability of the project.

What is MQTT?

MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks (Learn more). Mochi MQTT fully implements version 5.0.0 of the MQTT protocol.

Mochi-MQTT Features

- Full MQTTv5 Feature Compliance, compatibility for MQTT v3.1.1 and v3.0.0:
  - User and MQTTv5 Packet Properties
  - Topic Aliases
  - Shared Subscriptions
  - Subscription Options and Subscription Identifiers
  - Message Expiry
  - Client Session Expiry
  - Send and Receive QoS Flow Control Quotas
  - Server-side Disconnect and Auth Packets
  - Will Delay Intervals
  - Plus all the original MQTT features of Mochi MQTT v1, such as Full QoS(0,1,2), $SYS topics, retained messages, etc.
- Developer-centric:
  - Most core broker code is now exported and accessible, for total developer control.
  - Full-featured and flexible Hook-based interfacing system to provide easy 'plugin' development.
  - Direct Packet Injection using special inline client, or masquerade as existing clients.
- Performant and Stable:
  - Our classic trie-based Topic-Subscription model.
  - Client-specific write buffers to avoid issues with slow-reading or irregular client behaviour.
  - Passes all Paho Interoperability Tests for MQTT v5 and MQTT v3.
  - Over a thousand carefully considered unit test scenarios.
- TCP, Websocket (including SSL/TLS), and $SYS Dashboard listeners.
- Built-in Redis, Badger, Pebble and Bolt Persistence using Hooks (but you can also make your own).
- Built-in Rule-based Authentication and ACL Ledger using Hooks (also make your own).

Compatibility Notes

Because of the overlap between the v5 specification and previous versions of mqtt, the server can accept both v5 and v3 clients, but note that in cases where both v5 an v3 clients are connected, properties and features provided for v5 clients will be downgraded for v3 clients (such as user properties).

Support for MQTT v3.0.0 and v3.1.1 is considered hybrid-compatibility. Where not specifically restricted in the v3 specification, more modern and safety-first v5 behaviours are used instead - such as expiry for inflight and retained messages, and clients - and quality-of-service flow control limits.

When is this repo updated?

Unless it's a critical issue, new releases typically go out over the weekend.

Roadmap

- Please open an issue to request new features or event hooks!
- Cluster support.
- Enhanced Metrics support.

Quick Start

Running the Broker with Go

Mochi MQTT can be used as a standalone broker. Simply checkout this repository and run the cmd/main.go entrypoint in the cmd folder which will expose tcp (:1883), websocket (:1882), and dashboard (:8080) listeners.

```
cd cmd
go build -o mqtt && ./mqtt
```

Using Docker

You can now pull and run the official Mochi MQTT image from our Docker repo:

```
docker pull mochimqtt/server
or
docker run -v $(pwd)/config.yaml:/config.yaml mochimqtt/server
```

For most use cases, you can use File Based Configuration to configure the server, by specifying a valid `yaml` or `json` config file.

A simple Dockerfile is provided for running the cmd/main.go Websocket, TCP, and Stats server, using the `allow-all` auth hook.

```
docker build -t mochi:latest .
docker run -p 1883:1883 -p 1882:1882 -p 8080:8080 -v $(pwd)/config.yaml:/config.yaml mochi:latest
```

File Based Configuration

You can use File Based Configuration with either the Docker image (described above), or by running the build binary with the `--config=config.yaml` or `--config=config.json` parameter.

Configuration files provide a convenient mechanism for easily preparing a server with the most common configurations. You can enable and configure built-in hooks and listeners, and specify server options and compatibilities:

```yaml
listeners:
  - type: "tcp"
    id: "tcp12"
    address: ":1883"
  - type: "ws"
    id: "ws1"
    address: ":1882"
  - type: "sysinfo"
    id: "stats"
    address: ":1880"
hooks:
  auth:
    allow_all: true
options:
  inline_client: true
```

Please review the examples found in examples/config for all available configuration options.

There are a few conditions to note:

1. If you use file-based configuration, the supported hook types for configuration are currently limited to auth, storage, and debug. Each type of hook can only have one instance.
2. You can only use built in hooks with file-based configuration, as the type and configuration structure needs to be known by the server in order for it to be applied.
3. You can only use built in listeners, for the reasons above.

If you need to implement custom hooks or listeners, please do so using the traditional manner indicated in cmd/main.go.

Developing with Mochi MQTT

Importing as a package

Importing Mochi MQTT as a package requires just a few lines of code to get started.

```
import (
  "log"

  mqtt "github.com/mochi-mqtt/server/v2"
  "github.com/mochi-mqtt/server/v2/hooks/auth"
  "github.com/mochi-mqtt/server/v2/listeners"
)

func main() {
  // Create signals channel to run server until interrupted
  sigs := make(chan os.Signal, 1)
  done := make(chan bool, 1)
  signal.Notify(sigs, syscall.SIGINT, syscall.SIGTERM)
  go func() {
    <-sigs
    done <- true
  }()

  // Create the new MQTT Server.
  server := mqtt.New(nil)

  // Allow all connections.
  _ = server.AddHook(new(auth.AllowHook), nil)

  // Create a TCP listener on a standard port.
  tcp := listeners.NewTCP(listeners.Config{ID: "t1", Address: ":1883"})
  err := server.AddListener(tcp)
  if err != nil {
    log.Fatal(err)
  }


  go func() {
    err := server.Serve()
    if err != nil {
      log.Fatal(err)
    }
  }()

  // Run server until interrupted
  <-done

  // Cleanup
}
```

Examples of running the broker with various configurations can be found in the examples folder.

Network Listeners

The server comes with a variety of pre-packaged network listeners which allow the broker to accept connections on different protocols. The current listeners are:

Listener	Usage
listeners.NewTCP	A TCP listener
listeners.NewUnixSock	A Unix Socket listener
listeners.NewNet	A net.Listener listener
listeners.NewWebsocket	A Websocket listener
listeners.NewHTTPStats	An HTTP $SYS info dashboard
listeners.NewHTTPHealthCheck	An HTTP healthcheck listener to provide health check responses for e.g. cloud infrastructure

> Use the `listeners.Listener` interface to develop new listeners. If you do, please let us know!

A `*listeners.Config` may be passed to configure TLS.

Examples of usage can be found in the examples folder or cmd/main.go.

Server Options and Capabilities

A number of configurable options are available which can be used to alter the behaviour or restrict access to certain features in the server.

```
server := mqtt.New(&mqtt.Options{
  Capabilities: mqtt.Capabilities{
    MaximumSessionExpiryInterval: 3600,
    MaximumClientWritesPending: 3,
    Compatibilities: mqtt.Compatibilities{
      ObscureNotAuthorized: true,
    },
  },
  ClientNetWriteBufferSize: 4096,
  ClientNetReadBufferSize: 4096,
  SysTopicResendInterval: 10,
  InlineClient: false,
})
```

Review the mqtt.Options, mqtt.Capabilities, and mqtt.Compatibilities structs for a comprehensive list of options. `ClientNetWriteBufferSize` and `ClientNetReadBufferSize` can be configured to adjust memory usage per client, based on your needs. The size of `Capabilities.MaximumClientWritesPending` will affect the memory usage of the server. If the number of IoT devices online at the same time is large, and the set value is very large, even if there is no data transmission, the memory usage of the server will increase a lot. The default value is 1024*8, and this parameter can be adjusted according to the actual situation.

Default Configuration Notes

Some choices were made when deciding the default configuration that need to be mentioned here:

- By default, the value of `server.Options.Capabilities.MaximumMessageExpiryInterval` is set to 86400 (24 hours), in order to prevent exposing the broker to DOS attacks on hostile networks when using the out-of-the-box configuration (as an infinite expiry would allow an infinite number of retained/inflight messages to accumulate). If you are operating in a trusted environment, or you have capacity for a larger retention period, you may wish to override this (set to `0` for no expiry).

Event Hooks

A universal event hooks system allows developers to hook into various parts of the server and client life cycle to add and modify functionality o the broker. These universal hooks are used to provide everything from authentication, persistent storage, to debugging tools.

Hooks are stackable - you can add multiple hooks to a server, and they will be run in the order they were added. Some hooks modify values, and these modified values will be passed to the subsequent hooks before being returned to the runtime code.

Type	Import	Info
Access Control	mochi-mqtt/server/hooks/auth . AllowHook	Allow access to all connecting clients and read/write to all topics.
Access Control	mochi-mqtt/server/hooks/auth . Auth	Rule-based access control ledger.

Type	Import	Info
Persistence	mochi-mqtt/server/hooks/storage/bolt	Persistent storage using BoltDB (deprecated).
Persistence	mochi-mqtt/server/hooks/storage/badger	Persistent storage using BadgerDB.
Persistence	mochi-mqtt/server/hooks/storage/pebble	Persistent storage using PebbleDB.
Persistence	mochi-mqtt/server/hooks/storage/redis	Persistent storage using Redis.
Debugging	mochi-mqtt/server/hooks/debug	Additional debugging output to visualise packet flow.

Many of the internal server functions are now exposed to developers, so you can make your own Hooks by using the above as examples. If you do, please Open an issue and let everyone know!

Access Control

Allow Hook

By default, Mochi MQTT uses a DENY-ALL access control rule. To allow connections, this must overwritten using an Access Control hook. The simplest of these hooks is the `auth.AllowAll` hook, which provides ALLOW-ALL rules to all connections, subscriptions, and publishing. It's also the simplest hook to use:

```
server := mqtt.New(nil)
_ = server.AddHook(new(auth.AllowHook), nil)
```

> Don't do this if you are exposing your server to the internet or untrusted networks - it should really be used for development, testing, and debugging only.

Auth Ledger

The Auth Ledger hook provides a sophisticated mechanism for defining access rules in a struct format. Auth ledger rules come in two forms: Auth rules (connection), and ACL rules (publish subscribe).

Auth rules have 4 optional criteria and an assertion flag:

Criteria	Usage
Client	client id of the connecting client
Username	username of the connecting client
Password	password of the connecting client
Remote	the remote address or ip of the client
Allow	true (allow this user) or false (deny this user)

ACL rules have 3 optional criteria and an filter match:

Criteria	Usage
Client	client id of the connecting client
Username	username of the connecting client
Remote	the remote address or ip of the client
Filters	an array of filters to match

Rules are processed in index order (0,1,2,3), returning on the first matching rule. See hooks/auth/ledger.go to review the structs.

```
server := mqtt.New(nil)
err := server.AddHook(new(auth.Hook), &auth.Options{
    Ledger: &auth.Ledger{
    Auth: auth.AuthRules{ // Auth disallows all by default
      {Username: "peach", Password: "password1", Allow: true},
      {Username: "melon", Password: "password2", Allow: true},
      {Remote: "127.0.0.1:*", Allow: true},
```

```
      {Remote: "localhost:*", Allow: true},
    },
    ACL: auth.ACLRules{ // ACL allows all by default
      {Remote: "127.0.0.1:*"}, // local superuser allow all
      {
        // user melon can read and write to their own topic
        Username: "melon", Filters: auth.Filters{
          "melon/#":   auth.ReadWrite,
          "updates/#": auth.WriteOnly, // can write to updates, but can't read updates from others
        },
      },
      {
        // Otherwise, no clients have publishing permissions
        Filters: auth.Filters{
          "#":         auth.ReadOnly,
          "updates/#": auth.Deny,
        },
      },
    },
  }
})
```

The ledger can also be stored as JSON or YAML and loaded using the Data field:

```
err := server.AddHook(new(auth.Hook), &auth.Options{
    Data: data, // build ledger from byte slice: yaml or json
})
```

See examples/auth/encoded/main.go for more information.

Persistent Storage

Redis

A basic Redis storage hook is available which provides persistence for the broker. It can be added to the server in the same fashion as any other hook, with several options. It uses github.com/go-redis/redis/v8 under the hook, and is completely configurable through the Options value.

```
err := server.AddHook(new(redis.Hook), &redis.Options{
  Options: &rv8.Options{
    Addr:     "localhost:6379", // default redis address
    Password: "",               // your password
    DB:       0,                // your redis db
  },
})
if err != nil {
  log.Fatal(err)
}
```

For more information on how the redis hook works, or how to use it, see the examples/persistence/redis/main.go or hooks/storage/redis code.

Pebble DB

There's also a Pebble Db storage hook if you prefer file-based storage. It can be added and configured in much the same way as the other hooks (with somewhat less options).

```
err := server.AddHook(new(pebble.Hook), &pebble.Options{
  Path: pebblePath,
  Mode: pebble.NoSync,
})
if err != nil {
  log.Fatal(err)
}
```

For more information on how the pebble hook works, or how to use it, see the examples/persistence/pebble/main.go or hooks/storage/pebble code.

Badger DB

Similarly, for file-based storage, there is also a BadgerDB storage hook available. It can be added and configured in much the same way as the other hooks.

```
err := server.AddHook(new(badger.Hook), &badger.Options{
  Path: badgerPath,
})
if err != nil {
  log.Fatal(err)
}
```

For more information on how the badger hook works, or how to use it, see the examples/persistence/badger/main.go or hooks/storage/badger code.

There is also a BoltDB hook which has been deprecated in favour of Badger, but if you need it, check examples/persistence/bolt/main.go.

Developing with Event Hooks

Many hooks are available for interacting with the broker and client lifecycle. The function signatures for all the hooks and `mqtt.Hook` interface can be found in hooks.go.

> The most flexible event hooks are OnPacketRead, OnPacketEncode, and OnPacketSent - these hooks be used to control and modify all incoming and outgoing packets.

Function	Usage
OnStarted	Called when the server has successfully started.
OnStopped	Called when the server has successfully stopped.
OnConnectAuthenticate	Called when a user attempts to authenticate with the server. An implementation of this method MUST be used to allow or deny access to the server (see hooks/auth/allow_all or basic). It can be used in custom hooks to check connecting users against an existing user database. Returns true if allowed.
OnACLCheck	Called when a user attempts to publish or subscribe to a topic filter. As above.
OnSysInfoTick	Called when the $SYS topic values are published out.
OnConnect	Called when a new client connects, may return an error or packet code to halt the client connection process.
OnSessionEstablish	Called immediately after a new client connects and authenticates and immediately before the session is established and CONNACK is sent.
OnSessionEstablished	Called when a new client successfully establishes a session (after OnConnect)
OnDisconnect	Called when a client is disconnected for any reason.
OnAuthPacket	Called when an auth packet is received. It is intended to allow developers to create their own mqtt v5 Auth Packet handling mechanisms. Allows packet modification.
OnPacketRead	Called when a packet is received from a client. Allows packet modification.
OnPacketEncode	Called immediately before a packet is encoded to be sent to a client. Allows packet modification.
OnPacketSent	Called when a packet has been sent to a client.
OnPacketProcessed	Called when a packet has been received and successfully handled by the broker.
OnSubscribe	Called when a client subscribes to one or more filters. Allows packet modification.
OnSubscribed	Called when a client successfully subscribes to one or more filters.
OnSelectSubscribers	Called when subscribers have been collected for a topic, but before shared subscription subscribers have been selected. Allows receipient modification.
OnUnsubscribe	Called when a client unsubscribes from one or more filters. Allows packet modification.
OnUnsubscribed	Called when a client successfully unsubscribes from one or more filters.

Function	Usage
OnPublish	Called when a client publishes a message. Allows packet modification.
OnPublished	Called when a client has published a message to subscribers.
OnPublishDropped	Called when a message to a client is dropped before delivery, such as if the client is taking too long to respond.
OnRetainMessage	Called then a published message is retained.
OnRetainPublished	Called then a retained message is published to a client.
OnQosPublish	Called when a publish packet with Qos >= 1 is issued to a subscriber.
OnQosComplete	Called when the Qos flow for a message has been completed.
OnQosDropped	Called when an inflight message expires before completion.
OnPacketIDExhausted	Called when a client runs out of unused packet ids to assign.
OnWill	Called when a client disconnects and intends to issue a will message. Allows packet modification.
OnWillSent	Called when an LWT message has been issued from a disconnecting client.
OnClientExpired	Called when a client session has expired and should be deleted.
OnRetainedExpired	Called when a retained message has expired and should be deleted.
StoredClients	Returns clients, eg. from a persistent store.
StoredSubscriptions	Returns client subscriptions, eg. from a persistent store.
StoredInflightMessages	Returns inflight messages, eg. from a persistent store.
StoredRetainedMessages	Returns retained messages, eg. from a persistent store.
StoredSysInfo	Returns stored system info values, eg. from a persistent store.

If you are building a persistent storage hook, see the existing persistent hooks for inspiration and patterns. If you are building an auth hook, you will need `OnACLCheck` and `OnConnectAuthenticate`.

Inline Client (v2.4.0+)

It's now possible to subscribe and publish to topics directly from the embedding code, by using the `inline client` feature. Currently, the inline client does not support shared subscriptions. The Inline Client is an embedded client which operates as part of the server, and can be enabled in the server options:

```
server := mqtt.New(&mqtt.Options{
  InlineClient: true,
})
```

Once enabled, you will be able to use the `server.Publish`, `server.Subscribe`, and `server.Unsubscribe` methods to issue and received messages from broker-adjacent code.

> See direct examples for real-life usage examples.

Inline Publish

To publish basic message to a topic from within the embedding application, you can use the `server.Publish(topic string, payload []byte, retain bool, qos byte) error` method.

```
err := server.Publish("direct/publish", []byte("packet scheduled message"), false, 0)
```

> The Qos byte in this case is only used to set the upper qos limit available for subscribers, as per MQTT v5 spec.

Inline Subscribe

To subscribe to a topic filter from within the embedding application, you can use the `server.Subscribe(filter string, subscriptionId int, handler InlineSubFn) error` method with a callback function. Note that only QoS 0 is supported for inline subscriptions. If you wish to have multiple callbacks for the same filter, you can use the MQTTv5 `subscriptionId` property to differentiate.

```
callbackFn := func(cl *mqtt.Client, sub packets.Subscription, pk packets.Packet) {
    server.Log.Info("inline client received message from subscription", "client", cl.ID, "subscriptionId", sub.Identi
}
server.Subscribe("direct/#", 1, callbackFn)
```

Inline Unsubscribe

You may wish to unsubscribe if you have subscribed to a filter using the inline client. You can do this easily with the `server.Unsubscribe(filter string, subscriptionId int) error` method:

```
server.Unsubscribe("direct/#", 1)
```

Packet Injection

If you want more control, or want to set specific MQTT v5 properties and other values you can create your own publish packets from a client of your choice. This method allows you to inject MQTT packets (no just publish) directly into the runtime as though they had been received by a specific client.

Packet injection can be used for any MQTT packet, including ping requests, subscriptions, etc. And because the Clients structs and methods are now exported, you can even inject packets on behalf of a connected client (if you have a very custom requirements).

Most of the time you'll want to use the Inline Client described above, as it has unique privileges: it bypasses all ACL and topic validation checks, meaning it can even publish to $SYS topics. In this case, you can create an inline client from scratch which will behave the same as the built-in inline client.

```
cl := server.NewClient(nil, "local", "inline", true)
server.InjectPacket(cl, packets.Packet{
  FixedHeader: packets.FixedHeader{
    Type: packets.Publish,
  },
  TopicName: "direct/publish",
  Payload: []byte("scheduled message"),
})
```

> MQTT packets still need to be correctly formed, so refer our the test packets catalogue and MQTTv5 Specification for inspiration.

See the hooks example to see this feature in action.

Testing

Unit Tests

Mochi MQTT tests over a thousand scenarios with thoughtfully hand written unit tests to ensure each function does exactly what we expect. You can run the tests using go:

```
go run --cover ./...
```

Paho Interoperability Test

You can check the broker against the Paho Interoperability Test by starting the broker using `examples/paho/main.go`, and then running the mqtt v5 and v3 tests with `python3 client_test5.py` from the *interoperability* folder.

> Note that there are currently a number of outstanding issues regarding false negatives in the paho suite, and as such, certain compatibility modes are enabled in the `paho/main.go` example.

Performance Benchmarks

Mochi MQTT performance is comparable with popular brokers such as Mosquitto, EMQX, and others.

Performance benchmarks were tested using MQTT-Stresser on a Apple Macbook Air M2, using `cmd/main.go` default settings. Taking into account bursts of high and low throughput, the median scores are the most useful. Higher is better.

> The values presented in the benchmark are not representative of true messages per second throughput. They rely on an unusual calculation by mqtt-stresser, but are usable as they are consistent across all brokers. Benchmarks are provided as a general performance expectation guideline only. Comparisons are performed using out-of-the-box default configurations.

`mqtt-stresser -broker tcp://localhost:1883 -num-clients=2 -num-messages=10000`

Broker	publish fastest	median	slowest	receive fastest	median	slowest
Mochi v2.2.10	124,772	125,456	124,614	314,461	313,186	311,910
Mosquitto v2.0.15	155,920	155,919	155,918	185,485	185,097	184,709
EMQX v5.0.11	156,945	156,257	155,568	17,918	17,783	17,649
Rumqtt v0.21.0	112,208	108,480	104,753	135,784	126,446	117,108

`mqtt-stresser -broker tcp://localhost:1883 -num-clients=10 -num-messages=10000`

Broker	publish fastest	median	slowest	receive fastest	median	slowest
Mochi v2.2.10	41,825	31,663	23,008	144,058	65,903	37,618
Mosquitto v2.0.15	42,729	38,633	29,879	23,241	19,714	18,806
EMQX v5.0.11	21,553	17,418	14,356	4,257	3,980	3,756
Rumqtt v0.21.0	42,213	23,153	20,814	49,465	36,626	19,283

Million Message Challenge (hit the server with 1 million messages immediately):

`mqtt-stresser -broker tcp://localhost:1883 -num-clients=100 -num-messages=10000`

Broker	publish fastest	median	slowest	receive fastest	median	slowest
Mochi v2.2.10	13,532	4,425	2,344	52,120	7,274	2,701
Mosquitto v2.0.15	3,826	3,395	3,032	1,200	1,150	1,118
EMQX v5.0.11	4,086	2,432	2,274	434	333	311
Rumqtt v0.21.0	78,972	5,047	3,804	4,286	3,249	2,027

> Not sure what's going on with EMQX here, perhaps the docker out-of-the-box settings are not optimal, so take it with a pinch of salt as we know for a fact it's a solid piece of software.

Contribution Guidelines

Contributions and feedback are both welcomed and encouraged! Open an issue to report a bug, ask a question, or make a feature request. If you open a pull request, please try to follow the following guidelines:

- Try to maintain test coverage where reasonably possible.
- Clearly state what the PR does and why.
- Please remember to add your SPDX FileContributor tag to files where you have made a meaningful contribution.

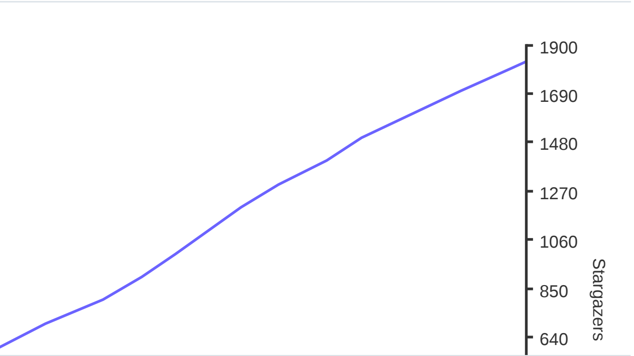
SPDX Annotations are used to clearly indicate the license, copyright, and contributions of each file in a machine-readable format. If you are adding a new file to the repository, please ensure it has the following SPDX header:

```
// SPDX-License-Identifier: MIT
// SPDX-FileCopyrightText: 2023 mochi-mqtt
// SPDX-FileContributor: Your name or alias <optional@email.address>
```

```
package name
```

Please ensure to add a new `SPDX-FileContributor` line for each contributor to the file. Refer to other files for examples. Please remember tc do this, your contributions to this project are valuable and appreciated - it's important to receive credit!

Stargazers over time



Releases 65

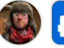
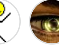
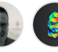
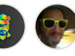
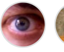
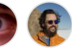
v2.7.9 Latest
on Mar 1, 2025

+ 64 releases

Packages

No packages published

Contributors 35

+ 21 contributors

Languages

- **Go** 100.0%