Jennifer Module API Reference

The public surface of every shipped module, generated from the modules' own doc comments.

Made with <3 by mplx <jennifer@mplx.dev>

2026-08-16

Module API reference

One page per shipped module, generated from the module's own doc comments. This is the exact, always-current public surface (every exported function, struct, enum, and constant). For prose, rationale, and runnable examples, see the module guides.

The whole reference as a single PDF: `jennifer-module-api.pdf` (jennifer-module-api.pdf).

Module	Summary
acme	An ACME (RFC 8555) client: obtain and renew TLS certificates from Let's Encrypt and compatible CAs.
amqp	An AMQP 0-9-1 client over net for RabbitMQ and compatible brokers.
ansi	Terminal styling as explicit string wrappers.
args	A declarative command-line argument parser, at the common surface of Python's argparse plus the features an argparse user misses immediately: typed flags (long + short), defaults, required args, ...
barcode	Generate scannable barcodes as images (not printer commands - the complement to label, which emits printer-native barcode commands).
bloom	A Bloom filter: a compact, probabilistic set.
cron	Parse and evaluate cron expressions - the five-field schedule spec minute hour day-of-month month day-of-week.
csv	RFC 4180 comma-separated values: parse text into rows of fields and format rows back into text, with a quoting-aware hand-written scanner.
discord	Post messages to a Discord channel through a channel Webhook, on top of the http client - a sibling of gotify / slack.
docblock	A Jennifer doc-comment parser.
dot	Graphviz DOT graph description: build a graph of nodes and edges with attributes and render it to .dot text for an external Graphviz tool to lay out and draw (dot -Tsvg graph.dot > graph.svg).
dotenv	Read .env configuration files - the KEY=VALUE lines that keep secrets and settings out of source - with layered profiles and ${VAR} interpolation.
feed	Web syndication feeds: build and parse both RSS 2.0 and Atom 1.0 through one module.
flatdb	A file-backed JSON document store.
font	A pure-Jennifer TrueType / OpenType (SFNT) font parser: read a .ttf / .otf from bytes and expose its metrics, character map, and glyph outlines.
gotify	Push notifications to a Gotify server (https://gotify.net), on top of the http client.
gpio	Raspberry-Pi (and any Linux SBC) GPIO over sysfs.
graphql	A thin GraphQL client over http / rest.
html	Build an HTML element tree and render it to a correctly escaped HTML5 string.
http	An HTTP/1.1 client over the net system library.
ical	Build and parse iCalendar (RFC 5545): a Calendar holding Events (VEVENT) and Todos (VTODO), encoded to a VCALENDAR and parsed back.
idna	Internationalized domain names: convert a Unicode domain to its ASCII-compatible (xn--) form and back, over a Punycode (RFC 3492) core.
imap	An IMAP4rev1 client (RFC 3501): tagged commands and untagged "*" responses over the net system library, with plaintext / implicit TLS / STARTTLS and auth by LOGIN, XOAUTH2, CRAM-MD5, or SCRAM-SHA...
influxdb	An InfluxDB client over http for both major API generations: write measurements as line-protocol points and run queries, getting back a parsed result.

ipnet	IP addresses and CIDR networks, IPv4 and IPv6.
jsonl	JSON Lines (JSONL / NDJSON): newline-delimited JSON, one independent value per line.
jsonrpc	JSON-RPC 2.0 (https://www.jsonrpc.org/specification) over HTTP: a client that calls remote methods, and a transport-agnostic server handle that dispatches an incoming request to the entry...
jwt	JSON Web Tokens (RFC 7519): sign, verify, and decode compact JWTs.
kvstore	A backend selector for a key/value store with per-key TTL: one Store value is one of three backends - a memcache connection, a redis connection, or the in-process kv library - and dispatche...
label	Describe and print labels for industrial label printers.
ldap	An LDAP v3 client and a lightweight, read-only directory server (RFC 4511), over TCP with the wire messages built and parsed as ASN.1 BER (the asn1 library) on the net transport.
log	Leveled, structured logging.
markdown	A lightweight Markdown renderer for a small CommonMark subset: ATX headings, bold / italic emphasis, inline code, links, images, fenced code blocks, unordered / ordered lists (nested by indenta...
mcp	Model Context Protocol (https://modelcontextprotocol.io) - the stateless JSON-RPC 2.0 surface an LLM host and an MCP peer speak.
memcache	A client for a memcached server, speaking its classic text protocol over the net system library.
mikrotik	A MikroTik RouterOS API client over net (not SSH).
mime	Build and parse MIME messages (RFC 5322 headers + RFC 2045/2046 bodies): a header-and-boundary structure that is exactly the text orchestration a .j module does well.
mqtt	An MQTT 3.1.1 publish/subscribe client over the net system library - the same "protocol clients are modules, net is the transport" line the other network clients follow.
multipart	Build and parse multipart/form-data bodies (RFC 7578) - the file-upload counterpart to mime's email multipart.
ntp	A simple SNTP client (RFC 4330 / 5905): query a time server over UDP and get back its time plus the local clock offset and round-trip delay.
oauth	A generic OAuth2 client: the get-a-token half of OAuth2 (the use-a-token half is sasl XOAUTH2).
orm	A minimal relational mapper over the sql library.
password	Password generation, validation, and complexity scoring against a policy schema.
pdf	Generate PDF documents - text, lines, rectangles - the way html / label generate their formats.
plot	Data plotting to SVG: turn numbers into a self-contained <svg> chart string you can write to a .svg file or drop into HTML.
pop	A POP3 receive client (RFC 1939): the line-oriented status dialogue ("+OK" / "-ERR") over the net system library, with plaintext, implicit TLS, or STLS, and auth by USER / PASS, XOAUTH2, CRAM-MD5...
prometheus	A Prometheus metrics module in two halves.
ratelimit	A rate limiter over a selectable key/value backend (kvstore.Store - memcache, redis, or the in-process kv).
redis	A Redis client speaking RESP2 (the REdis Serialization Protocol) over the net system library.
resque	Background jobs on Redis, wire-compatible with Resque.
rest	An ergonomic REST layer over the http client and json.
ringbuffer	A fixed-capacity ring buffer of strings: a bounded FIFO that overwrites the oldest entry when full.

s3	An S3-compatible object-storage client: get / put / delete objects and list a bucket, signing every request with AWS Signature Version 4.
sasl	The SASL authentication mechanisms shared by the mail clients (smtp / pop / imap).
screen	Terminal user interfaces: an explicit screen, not a GUI framework.
semver	Strict Semantic Versioning 2.0.0 (https://semver.org): parse, compare, increment, and range-match version numbers - the full surface a package registry or dependency resolver needs.
session	Server-side sessions over a selectable key/value backend.
slack	Post messages to a Slack channel through an Incoming Webhook, on top of the http client - a sibling of gotify / discord.
smtp	An SMTP send client: the line-oriented command/response dialogue of RFC 5321, over the net system library (plaintext, implicit TLS, or STARTTLS) with SASL AUTH: PLAIN, LOGIN, XOAUTH2, CRAM-MD5, a...
snmp	An SNMP v1 / v2c client and agent (RFC 1157 / RFC 3416), over UDP with community-string authentication.
sqlmigrate	A version-tracked schema-migration runner over the sql library.
statsd	A StatsD client (over UDP): emit metric:value|type lines for a StatsD / Datadog / Telegraf agent to aggregate.
telegram	A Telegram Bot API client over the http client + json.
tengine	A small text template engine for lightweight CMS-style rendering - a subset of Go's text/template (the Go / Hugo style), evaluated directly over a json.Value data tree.
totp	Time-based one-time passwords (RFC 6238 TOTP over RFC 4226 HOTP) - the six-digit two-factor codes authenticator apps show.
transport	The shared connection-transport security mode, used by every network-client module that opens a socket (smtp, pop, imap, redis, amqp, mqtt).
uri	URL parsing, building, and query-string handling (RFC 3986).
validate	Declarative validation of a map of string to string (a form body, a query, a config) against a rule set, returning a structured error list instead of ad-hoc per-field if checks.
vcard	Build and parse vCard (RFC 6350, vCard 4.0): a Card of contact fields encoded to a VCARD and parsed back.
web	An ergonomic HTTP framework over the httpd server engine.
webapi	A JSON-API conventions layer over the bundled web module.
webhook	Sign and verify HMAC-signed webhooks - the GitHub-style X-Hub-Signature-256 convention.
websocket	An RFC 6455 WebSocket client over net.

acme API reference

An ACME (RFC 8555) client: obtain and renew TLS certificates from Let's Encrypt and compatible CAs. It drives the full flow - account registration, a new order, HTTP-01 or DNS-01 challenge, CSR finalize, and certificate download - over `http` + `json` , with every request a JWS signed by the account key (`RS256` for an RSA key, `ES256` for an EC key). Keys and the CSR come from the `crypto` library, so this needs the default `jennifer` binary.

The building blocks are separate calls, not one `obtain()` , because proving domain control is the caller's job: between `order` and `finalize` you must publish the challenge response - serve `keyAuthorization` at `/.well-known/acme-challenge/<token>` (HTTP-01), or set the `dnsRecord` TXT at `_acme-challenge.<domain>` (DNS-01) - then `accept` the challenge. See the demo for the orchestration.

Test against a CA's **staging** endpoint first (Let's Encrypt staging issues untrusted certs with far higher rate limits).

Import with `import "acme.j" as acme;` . See the `acme` guide for prose and examples.

Functions

acme.accept(client as Client, challengeUrl as string)

Tell the CA a challenge response is in place (a POST of `{}` to the challenge URL). Do this only after publishing the HTTP-01 / DNS-01 response.

Parameters

- `client` `{Client}` - a registered client
- `challengeUrl` `{string}` - the challenge URL (from `challenge`)

Returns `{null}` - nothing

Throws

- `{Error}` - on an accept error

acme.authorization(client as Client, authzUrl as string)

Fetch an authorization: its domain, status, and the challenges the CA offers.

Parameters

- `client` `{Client}` - a registered client
- `authzUrl` `{string}` - an authorization URL (from `order`)

Returns `{Authorization}` - the authorization

acme.challenge(authz as Authorization, kind as string)

The challenge of a given type within an authorization (`"http-01"` / `"dns-01"`).

Parameters

- `authz` `{Authorization}` - an authorization from `authorization`
- `kind` `{string}` - the challenge type to select

Returns `{Challenge}` - the matching challenge

Throws

- `{Error}` - when no challenge of that type is offered

acme.connect(directoryUrl as string, accountKey as bytes)

Build a client from a CA directory URL and an account private key (PEM `bytes` , RSA or EC - from `crypto.rsaGenerateKey` / `crypto.ecGenerateKey`). Fetches the directory to learn the CA's endpoints. Does not yet register.

Parameters

- `directoryUrl` `{string}` - the CA's ACME directory URL
- `accountKey` `{bytes}` - the account private key, PEM-encoded

Returns `{Client}` - the configured client (kid empty until `register`)

Throws

- `{Error}` - on a transport error or a directory missing required endpoints

acme.dnsRecord(client as Client, token as string)

The TXT record value for a **DNS-01** challenge: the base64url (unpadded) SHA-256 of the key authorization. Publish it at `_acme-challenge.<domain>` .

Parameters

- `client` `{Client}` - the client
- `token` `{string}` - the challenge token

Returns `{string}` - the `_acme-challenge` TXT value

acme.downloadCertificate(client as Client, order as Order)

Download the issued certificate chain (PEM) for a valid order.

Parameters

- `client` `{Client}` - a registered client
- `order` `{Order}` - a `valid` order (with a `certificate` URL)

Returns `{string}` - the PEM certificate chain (leaf first)

Throws

- `{Error}` - when the order has no certificate yet

acme.fetchOrder(client as Client, orderUrl as string)

Fetch an order's current state by URL (a POST-as-GET). Use it to poll after `finalize` .

Parameters

- `client` `{Client}` - a registered client
- `orderUrl` `{string}` - the order URL

Returns `{Order}` - the order's current state

acme.finalize(client as Client, order as Order, csrDer as bytes, intervalMs as int, maxTries as int)

Finalize an order by submitting a CSR, then poll the order until it is `valid` (the certificate is ready) or `invalid` . `csrDer` is a DER PKCS#10 request (`crypto.csr(certKey, domains)`) for the same domains as the order.

Parameters

- `client` `{Client}` - a registered client
- `order` `{Order}` - the (ready) order
- `csrDer` `{bytes}` - the DER CSR
- `intervalMs` `{int}` - milliseconds between status polls
- `maxTries` `{int}` - maximum number of polls

Returns `{Order}` - the finalized order (`certificate` set when valid)

Throws

- `{Error}` - on a finalize error or if issuance does not complete

acme.keyAuthorization(client as Client, token as string)

The key authorization for a challenge token: `token.thumbprint` , where the thumbprint is the RFC 7638 SHA-256 of the account key's JWK (base64url, unpadded). This is the exact body an **HTTP-01** challenge serves at `/.well-known/acme-challenge/<token>` .

Parameters

- `client` `{Client}` - the client (its account key defines the thumbprint)
- `token` `{string}` - the challenge token

Returns `{string}` - the key authorization

acme.order(client as Client, domains as list of string)

Create an order for one or more domains. The returned order is `pending` with one authorization URL per domain.

Parameters

- `client` `{Client}` - a registered client
- `domains` `{list of string}` - the domains to certify (the first is the CN)

Returns `{Order}` - the new order

Throws

- `{Error}` - on an order error

acme.pollAuthorization(client as Client, authzUrl as string, intervalMs as int, maxTries as int)

Poll an authorization until it leaves `pending` (becomes `valid` or `invalid`), waiting `intervalMs` between attempts up to `maxTries` .

Parameters

- `client` `{Client}` - a registered client
- `authzUrl` `{string}` - the authorization URL
- `intervalMs` `{int}` - milliseconds between polls
- `maxTries` `{int}` - maximum number of polls

Returns `{Authorization}` - the settled authorization

Throws

- `{Error}` - if it is still pending after `maxTries`

acme.register(client as Client, email as string)

Register (or look up) the account with the CA and return a client carrying the account URL (`kid`). Agrees to the CA's terms of service. Idempotent: the CA returns the existing account for a known key.

Parameters

- `client` `{Client}` - a client from `connect`
- `email` `{string}` - a contact email (empty for none)

Returns `{Client}` - a client with `kid` set

Throws

- `{Error}` - on a registration error

Structs

acme.Authorization

A domain-control authorization: the challenges the CA offers for it.

Field	Type	Description
domain	string	the identifier being authorized
status	string	pending / valid / invalid
challenges	list of Challenge	the offered challenges

acme.Challenge

One challenge within an authorization.

Field	Type	Description
kind	string	the challenge type (http-01 / dns-01 / tls-alpn-01)
url	string	the challenge URL (POST to it to accept / tell the CA it is ready)
token	string	the challenge token
status	string	pending / valid / invalid

acme.Client

An ACME client: the CA's directory endpoints plus the account key and (once registered) the account URL used as the JWS `kid` . Value-semantic; `register` returns a copy with `kid` filled in.

Field	Type	Description
directory	string	the directory URL the client was built from
newNonce	string	the CA's new-nonce endpoint
newAccount	string	the CA's new-account endpoint
newOrder	string	the CA's new-order endpoint
accountKey	bytes	the account private key, PEM-encoded
alg	string	the JWS algorithm for the key (RS256 or ES256)
kid	string	the account URL (empty until register)

acme.Order

An ACME order for one or more identifiers (domains).

Field	Type	Description
url	string	the order URL (poll this for status)
status	string	pending / ready / processing / valid / invalid
authorizations	list of string	the per-identifier authorization URLs
finalize	string	the finalize URL (POST the CSR here)
certificate	string	the certificate URL (empty until issued)

amqp API reference

An AMQP 0-9-1 client over `net` for RabbitMQ and compatible brokers. `connect` runs the connection + channel handshake (protocol header, `Connection.Start` / `Start-Ok` with SASL PLAIN auth, `Tune` / `Tune-Ok` , `Open` / `Open-Ok` , `Channel.Open`); `declareQueue` declares a classic queue and `declareQuorumQueue` a replicated quorum queue; `publish` sends a message (method + content-header + body frames); `get` pulls the next message with `Basic.Get` (a synchronous pull, no async delivery loop); `ack` acknowledges it; `close` shuts the connection down cleanly.

The binary frame and method encoding is built by hand from `bytes` and the bitwise operators - the largest protocol module here. Needs the default `jennifer` binary (`net`); a protocol error or dropped connection throws `Error{kind: "amqp"}` . Uses one channel (1); heartbeats are disabled.

Import with `import "amqp.j" as amqp;` . See the `amqp` guide for prose and examples.

Functions

amqp.ack(c as Conn, deliveryTag as int)

Acknowledge a delivered message by its tag.

Parameters

- `c {Conn}` - the connection
- `deliveryTag {int}` - the delivery tag from a got Message

amqp.bindQueue(c as Conn, queue as string, exchange as string, routingKey as string)

Bind a queue to an exchange with a routing key. For a "fanout" exchange the routing key is ignored; for "direct" it is matched exactly; for "topic" it is a pattern (* / #).

Parameters

- `c {Conn}` - the connection
- `queue {string}` - the queue name
- `exchange {string}` - the exchange name
- `routingKey {string}` - the binding routing key / pattern

Throws

- `{Error}` - kind "amqp" on failure

amqp.cancelConsume(c as Conn, consumerTag as string)

Cancel a subscription started with consume (Basic.Cancel). After this the broker sends no further deliveries for that consumer tag.

Parameters

- `c {Conn}` - the connection
- `consumerTag {string}` - the consumer tag returned by consume

Throws

- `{Error}` - kind "amqp" on failure

amqp.close(c as Conn)

Close the connection cleanly (Connection.Close) and shut the socket.

Parameters

- `c` `{Conn}` - the connection

amqp.confirmSelect(c as Conn)

Put the channel into publisher-confirm mode (Confirm.Select). After this, each publish is confirmed by the broker; call waitConfirm once per publish to block for that confirmation. Idempotent.

Parameters

- `c` `{Conn}` - the connection

Throws

- `{Error}` - kind "amqp" on failure

amqp.connect(opts as Options)

Connect to a broker and open a channel.

Parameters

- `opts` `{Options}` - the connection options

Returns `{Conn}` - the open connection

Throws

- `{Error}` - kind "amqp" on a handshake failure

amqp.consume(c as Conn, queue as string, autoAck as bool)

Start a server-pushed subscription with Basic.Consume and return the consumer tag (the broker generates one). Follow with receiveDelivery in a loop - wrap it in a `spawn` so the pushed deliveries do not block the rest of the program. When `autoAck` is false, ack (or nack) each Delivery by its deliveryTag.

Parameters

- `c` `{Conn}` - the connection
- `queue` `{string}` - the queue name
- `autoAck` `{bool}` - whether the broker should auto-acknowledge each delivery

Returns `{string}` - the consumer tag (pass to cancelConsume)

Throws

- `{Error}` - kind "amqp" on failure

amqp.declareExchange(c as Conn, name as string, exType as string)

Declare an exchange (idempotent, durable). `exType` is "direct", "topic", or "fanout".

Parameters

- `c` `{Conn}` - the connection
- `name` `{string}` - the exchange name
- `exType` `{string}` - the exchange type ("direct" / "topic" / "fanout")

Throws

- `{Error}` - kind "amqp" on failure

amqp.declareQueue(c as Conn, name as string, durable as bool)

Declare a queue (idempotent). `durable` survives a broker restart.

Parameters

- `c` `{Conn}` - the connection

- `name {string}` - the queue name ("" for a server-generated name)
- `durable {bool}` - whether the queue is durable

Returns `{QueueInfo}` - the queue name and message / consumer counts

Throws

- `{Error}` - kind "amqp" on failure

amqp.declareQuorumQueue(c as Conn, name as string)

Declare a quorum queue: a Raft-replicated queue type for high availability and data safety. Sets the `x-queue-type=quorum` argument. Quorum queues are always durable (there is no durable flag) and cannot be server-named, so `name` must be non-empty. Re-declaring must use the same type, so do not also declare the same name as a classic queue.

Parameters

- `c {Conn}` - the connection
- `name {string}` - the queue name (non-empty)

Returns `{QueueInfo}` - the queue name and message / consumer counts

Throws

- `{Error}` - kind "amqp" on an empty name or a declare failure

amqp.get(c as Conn, queue as string, autoAck as bool)

Pull the next message from a queue with Basic.Get. When `autoAck` is false, ack the returned message with `ack` .

Parameters

- `c {Conn}` - the connection
- `queue {string}` - the queue name
- `autoAck {bool}` - whether the broker should auto-acknowledge

Returns `{Message}` - the message (its `empty` field is true when none was ready)

Throws

- `{Error}` - kind "amqp" on failure

amqp.nack(c as Conn, deliveryTag as int, requeue as bool)

Negatively acknowledge a delivered message (Basic.Nack). With `requeue` true the broker re-queues the message for redelivery; with false it is dropped (or dead-lettered if configured). Nacks a single message (not "multiple").

Parameters

- `c {Conn}` - the connection
- `deliveryTag {int}` - the delivery tag from a Message / Delivery
- `requeue {bool}` - whether the broker should re-queue the message

amqp.options(host as string, user as string, password as string)

Default options for a broker: port 5672, vhost "/".

Parameters

- `host {string}` - the broker host
- `user {string}` - the username
- `password {string}` - the password

Returns `{Options}` - the options

amqp.publish(c as Conn, exchange as string, routingKey as string, body as bytes)

Publish a message body to an exchange with a routing key. Use exchange "" for the default exchange (routing key = queue name).

Parameters

- `c {Conn}` - the connection
- `exchange {string}` - the exchange name ("" for the default)
- `routingKey {string}` - the routing key
- `body {bytes}` - the message body

Throws

- `{Error}` - kind "amqp" on failure

amqp.publishText(c as Conn, exchange as string, routingKey as string, text as string)

Publish a text message (UTF-8). Convenience over publish.

Parameters

- `c {Conn}` - the connection
- `exchange {string}` - the exchange name ("" for the default)
- `routingKey {string}` - the routing key
- `text {string}` - the message text

Throws

- `{Error}` - kind "amqp" on failure

amqp.publishWith(c as Conn, exchange as string, routingKey as string, body as bytes, props as Properties)

Publish a message body carrying message properties (content-type, persistence, correlation-id, reply-to). Otherwise identical to publish. Build the property set with `Properties{...}` ; an unset field ("" / false) is omitted from the wire.

Parameters

- `c {Conn}` - the connection
- `exchange {string}` - the exchange name ("" for the default)
- `routingKey {string}` - the routing key
- `body {bytes}` - the message body
- `props {Properties}` - the message properties

Throws

- `{Error}` - kind "amqp" on failure

amqp.receiveDelivery(c as Conn)

Block for the next server-pushed message (Basic.Deliver + content header + body frame(s)) and return it as a Delivery. Only valid after consume. This is the cooperative receive loop: there is no callback - the app calls receiveDelivery repeatedly, typically from inside its own `spawn` , and acts on each Delivery.

Parameters

- `c {Conn}` - the connection

Returns `{Delivery}` - the next delivered message

Throws

- `{Error}` - kind "amqp" on a non-delivery frame or a dropped connection

amqp.waitConfirm(c as Conn)

Block for the broker's confirmation of the next outstanding publish (only valid after confirmSelect). Returns true on Basic.Ack (the broker took responsibility for the message) and false on Basic.Nack (the broker could not).

Parameters

- `c {Conn}` - the connection

Returns `{bool}` - true when confirmed, false when nacked

Throws

- `{Error}` - kind "amqp" on an unexpected reply

amqp.withPort(o as Options, port as int)

Copy options with a different port.

Parameters

- `o {Options}` - the options
- `port {int}` - the port

Returns `{Options}` - a fresh options

amqp.withVhost(o as Options, vhost as string)

Copy options with a different virtual host.

Parameters

- `o {Options}` - the options
- `vhost {string}` - the virtual host

Returns `{Options}` - a fresh options

Structs

amqp.Conn

An open connection (single channel).

Field	Type	Description
socket	net.Conn	the underlying connection
channel	int	the channel number (always 1)
frameMax	int	the negotiated maximum frame size (bytes)

amqp.Delivery

A message pushed by the broker via Basic.Deliver (returned by receiveDelivery).

Field	Type	Description
consumerTag	string	the consumer tag the delivery is for
deliveryTag	int	the delivery tag (pass to ack / nack)
redelivered	bool	true if the broker has delivered this message before
exchange	string	the exchange the message came from

routingKey	string	the routing key
body	bytes	the message body

amqp.Message

A message pulled with get.

Field	Type	Description
empty	bool	true when the queue had no message (the other fields are zero)
deliveryTag	int	the delivery tag (pass to ack)
exchange	string	the exchange the message came from
routingKey	string	the routing key
body	bytes	the message body

amqp.Options

Connection options.

Field	Type	Description
host	string	the broker host
port	int	the broker port (5672 by default)
user	string	the username
password	string	the password
vhost	string	the virtual host ("/" by default)
security	transport.Security	transport.Security.None (plaintext AMQP, the default) or .Tls (AMQPS - TLS on connect, verifying the broker certificate); .Starttls is rejected (AMQP has no in-band upgrade)

amqp.Properties

Message properties carried on a publish (all optional; a "" string or a false flag omits that property from the wire, keeping the content header minimal).

Field	Type	Description
contentType	string	the MIME content-type (e.g. "application/json"); "" omits it
persistent	bool	true sets delivery-mode 2 (persistent); false omits it (transient)
correlationId	string	the correlation id (request / reply matching); "" omits it
replyTo	string	the reply-to queue / routing key; "" omits it

amqp.QueueInfo

Queue metadata returned by declareQueue.

Field	Type	Description
name	string	the queue name (the server's, for a server-named queue)
messageCount	int	the number of ready messages
consumerCount	int	the number of consumers

ansi API reference

Terminal styling as explicit string wrappers. The first module built on Jennifer's module system: pure Jennifer, no Go. Colour is gated on stdout being a terminal (with the NO_COLOR / FORCE_COLOR override), so wrapped output stays clean when it is redirected to a file or a pipe.

Import with `import "ansi.j" as ansi;` . See the `ansi` guide for prose and examples.

Functions

ansi.bgColor(s as string, name as string)

Wrap a string in the named background colour.

Parameters

- `s {string}` - the text to colourize
- `name {string}` - the background colour name (e.g. "red", "blue")

Returns `{string}` - the wrapped text, or s unchanged when colour is off

Throws

- `{Error}` - when name is not a known background colour

ansi.black(s as string)

Wrap a string in black foreground colour.

Parameters

- `s {string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.blue(s as string)

Wrap a string in blue foreground colour.

Parameters

- `s {string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.bold(s as string)

Wrap a string in the bold text style.

Parameters

- `s {string}` - the text to style

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.color(s as string, name as string)

Wrap a string in the named foreground colour.

Parameters

- `s {string}` - the text to colourize
- `name {string}` - the colour name (e.g. "red", "green", "cyan")

Returns `{string}` - the wrapped text, or s unchanged when colour is off

Throws

- `{Error}` - when name is not a known colour

ansi.cyan(s as string)

Wrap a string in cyan foreground colour.

Parameters

- `s` `{string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.dim(s as string)

Wrap a string in the dim text style.

Parameters

- `s` `{string}` - the text to style

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.gray(s as string)

Wrap a string in gray foreground colour.

Parameters

- `s` `{string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.green(s as string)

Wrap a string in green foreground colour.

Parameters

- `s` `{string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.italic(s as string)

Wrap a string in the italic text style.

Parameters

- `s` `{string}` - the text to style

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.magenta(s as string)

Wrap a string in magenta foreground colour.

Parameters

- `s` `{string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.red(s as string)

Wrap a string in red foreground colour.

Parameters

- `s` `{string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.reverse(s as string)

Wrap a string in the reverse (inverted) text style.

Parameters

- `s` `{string}` - the text to style

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.rgb(s as string, r as int, g as int, b as int)

Wrap a string in a 24-bit truecolor foreground.

Parameters

- `s` `{string}` - the text to colourize
- `r` `{int}` - the red channel (0-255)
- `g` `{int}` - the green channel (0-255)
- `b` `{int}` - the blue channel (0-255)

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.strip(s as string)

Remove every SGR escape - the inverse of the wrappers, regardless of whether colour is currently enabled.

Parameters

- `s` `{string}` - the text to strip

Returns `{string}` - the text with all SGR escapes removed

ansi.style(s as string, name as string)

Wrap a string in the named text style.

Parameters

- `s` `{string}` - the text to style
- `name` `{string}` - the style name (e.g. "bold", "italic", "underline")

Returns `{string}` - the wrapped text, or s unchanged when colour is off

Throws

- `{Error}` - when name is not a known style

ansi.underline(s as string)

Wrap a string in the underline text style.

Parameters

- `s` `{string}` - the text to style

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.white(s as string)

Wrap a string in white foreground colour.

Parameters

- `s` `{string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

ansi.yellow(s as string)

Wrap a string in yellow foreground colour.

Parameters

- `s` `{string}` - the text to colourize

Returns `{string}` - the wrapped text, or s unchanged when colour is off

args API reference

A declarative command-line argument parser, at the common surface of Python's `argparse` plus the features an argparse user misses immediately: typed flags (long + short), defaults, required args, `choices` , `count` / `append` actions, positionals with `nargs` , subcommands, and `--version` . It is the structured layer over `os.ARGS` (`os.hasFlag` / `os.flag` are primitive lookups). Pure Jennifer over `strings` + `convert` + `lists` + `maps` , so it runs on **both binaries** .

Build a value-semantic `Parser` with the copy-returning builder pattern (`args.parser` then `args.flag` / `args.intFlag` / `args.positional` / ...), then `args.parse($p, os.ARGS)` -> a `Result` . Read values back with the typed accessors (`args.asString` / `asInt` / `asFloat` / `asBool` / `asList` / `args.count`). An unknown flag, a missing required arg, a bad-type value, or a `choices` violation throws a catchable `Error{kind: "args"}` ; `-h` / `--help` (and `--version`) set the result's `done` flag with `helpText` to print, rather than exiting the process, so it composes with `try` / `catch` .

Import with `import "args.j" as args;` . See the `args` guide for prose and examples.

Functions

args.asBool(r as Result, name as string)

The bool value of a flag (true when the stored value is "true").

Parameters

- `r` `{Result}` - the parse result
- `name` `{string}` - the flag name

Returns `{bool}` - the value

args.asFloat(r as Result, name as string)

The float value of an argument.

Parameters

- `r` `{Result}` - the parse result
- `name` `{string}` - the argument name

Returns `{float}` - the value (0.0 if unset)

args.asInt(r as Result, name as string)

The int value of an argument (parsed from its stored string).

Parameters

- `r` `{Result}` - the parse result
- `name` `{string}` - the argument name

Returns `{int}` - the value (0 if unset)

args.asList(r as Result, name as string)

The list value of an append flag or variadic positional.

Parameters

- `r` `{Result}` - the parse result
- `name` `{string}` - the argument name

Returns `{list of string}` - the collected values ([] if none)

args.asString(r as Result, name as string)

The string value of an argument (its provided value or default; "" if unset).

Parameters

- `r` `{Result}` - the parse result
- `name` `{string}` - the argument name

Returns `{string}` - the value

args.boolFlag(p as Parser, long as string, short as string, help as string)

Add a boolean flag (presence sets it true; the argparse `store_true` action).

Parameters

- `p` `{Parser}` - the parser
- `long` `{string}` - the long name
- `short` `{string}` - the short name ("" for none)
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.choices(p as Parser, allowed as list of string)

Constrain the most-recently-added argument to a set of allowed values.

Parameters

- `p` `{Parser}` - the parser
- `allowed` `{list of string}` - the permitted values

Returns `{Parser}` - the updated parser

Throws

- `{Error}` - kind "args" if no argument has been added yet

args.command(p as Parser, name as string, help as string, sub as Parser)

Add a subcommand with its own parser (argparse subparsers).

Parameters

- `p` `{Parser}` - the parent parser
- `name` `{string}` - the subcommand word
- `help` `{string}` - the subcommand's one-line help
- `sub` `{Parser}` - the subcommand's own parser

Returns `{Parser}` - the updated parser

args.count(r as Result, name as string)

The tally of a count flag.

Parameters

- `r` `{Result}` - the parse result
- `name` `{string}` - the flag name

Returns `{int}` - the number of occurrences (0 if none)

args.countFlag(p as Parser, long as string, short as string, help as string)

Add a repeatable counting flag: each occurrence increments a tally (`-vvv` -> 3), the argparse `count` action. Read with `args.count` .

Parameters

- `p {Parser}` - the parser
- `long {string}` - the long name
- `short {string}` - the short name ("" for none)
- `help {string}` - the help text

Returns `{Parser}` - the updated parser

args.dispatch(r as Result, handlers as map of string to func)

Dispatch the parsed subcommand to its handler. `handlers` maps a subcommand name to a `func` value `func(r as Result)` ; the handler for `r.command` is called with the whole `Result` (so it reads its own args with `args.asString` / `asInt` / ...), and its return value is passed back - so a handler may return an exit code. A `Result` with `done` set (a handled `--help` / `--version`) dispatches nothing and returns `null` ; a caller usually checks `r.done` and prints `r.helpText` before calling this. A missing handler for the selected subcommand, or a selection of none, is a catchable error.

The handlers are ordinary `func` values, not names - a func value called here runs in the entry program's own context, so it resolves its own imports.

Parameters

- `r {Result}` - the parse result
- `handlers {map of string to func}` - subcommand name -> its handler

Throws

- `{Error}` - kind "args" when no subcommand was selected or none matches

args.flag(p as Parser, long as string, short as string, deflt as string, help as string)

Add a string-valued optional flag (`--long` / `-short`), with a default.

Parameters

- `p {Parser}` - the parser
- `long {string}` - the long name (used without the leading "--")
- `short {string}` - the single-char short name ("" for none)
- `deflt {string}` - the default when the flag is absent
- `help {string}` - the flag's help text

Returns `{Parser}` - the updated parser

args.floatFlag(p as Parser, long as string, short as string, deflt as float, help as string)

Add a float-valued optional flag.

Parameters

- `p {Parser}` - the parser
- `long {string}` - the long name
- `short {string}` - the short name ("" for none)
- `deflt {float}` - the default value
- `help {string}` - the help text

Returns `{Parser}` - the updated parser

args.has(r as Result, name as string)

Whether an argument was actually supplied on the command line (as opposed to taking its default).

Parameters

- `r` `{Result}` - the parse result
- `name` `{string}` - the argument name

Returns `{bool}` - true if supplied

args.intFlag(p as Parser, long as string, short as string, deflt as int, help as string)

Add an int-valued optional flag.

Parameters

- `p` `{Parser}` - the parser
- `long` `{string}` - the long name
- `short` `{string}` - the short name ("" for none)
- `deflt` `{int}` - the default value
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.listFlag(p as Parser, long as string, short as string, help as string)

Add a repeatable value flag: each occurrence appends to a list (the argparse `append` action). Read with `args.asList` .

Parameters

- `p` `{Parser}` - the parser
- `long` `{string}` - the long name
- `short` `{string}` - the short name ("" for none)
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.parse(p as Parser, argv as list of string)

Parse `argv` (the full `os.ARGS` , whose first element is the program name and is skipped) against the parser.

Parameters

- `p` `{Parser}` - the specification
- `argv` `{list of string}` - the argument vector (pass `os.ARGS`)

Returns `{Result}` - the parsed values; check `.done` for --help / --version

Throws

- `{Error}` - kind "args" on an unknown flag, missing required arg, bad type, or bad choice

args.parser(prog as string, help as string)

Start a parser with a program name and one-line description.

Parameters

- `prog` `{string}` - the program name (shown in usage)
- `help` `{string}` - the one-line description

Returns `{Parser}` - an empty parser

args.positional(p as Parser, name as string, help as string)

Add a single required positional argument.

Parameters

- `p` `{Parser}` - the parser
- `name` `{string}` - the positional's name (its result key)
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.positionalList(p as Parser, name as string, help as string)

Add a variadic positional collecting zero or more values into a list (`nargs "*"`). Read with `args.asList` .

Parameters

- `p` `{Parser}` - the parser
- `name` `{string}` - the positional's name
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.positionalList1(p as Parser, name as string, help as string)

Add a variadic positional requiring one or more values (`nargs "+"`). Read with `args.asList` .

Parameters

- `p` `{Parser}` - the parser
- `name` `{string}` - the positional's name
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.positionalN(p as Parser, name as string, n as int, help as string)

Add a positional taking exactly n values into a list (`nargs N`).

Parameters

- `p` `{Parser}` - the parser
- `name` `{string}` - the positional's name
- `n` `{int}` - the exact number of values
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.positionalOpt(p as Parser, name as string, deflt as string, help as string)

Add an optional positional (`nargs "?"` , 0 or 1) with a default.

Parameters

- `p` `{Parser}` - the parser
- `name` `{string}` - the positional's name
- `deflt` `{string}` - the default when absent
- `help` `{string}` - the help text

Returns `{Parser}` - the updated parser

args.required(p as Parser)

Mark the most-recently-added argument as required.

Parameters

- `p` `{Parser}` - the parser

Returns `{Parser}` - the updated parser

Throws

- `{Error}` - kind "args" if no argument has been added yet

args.usage(p as Parser)

The generated usage / help text for a parser (what `--help` prints).

Parameters

- `p {Parser}` - the parser

Returns `{string}` - the multi-line help text

args.version(p as Parser, ver as string)

Enable a `--version` action printing `ver` .

Parameters

- `p {Parser}` - the parser
- `ver {string}` - the version string

Returns `{Parser}` - the updated parser

Structs

args.Arg

One argument definition (a flag or a positional). Built by the `flag` / `positional` family; not usually constructed directly.

Field	Type	Description
name	string	the canonical key (long flag name, or positional name)
short	string	the single-character short flag ("" for none)
kind	string	"flag" or "positional"
typ	string	the value type: "string" / "int" / "float" / "bool"
action	string	"store" (default), "count" (repeat -> int), or "append" (repeat -> list)
fallback	string	the default value in string form (when hasDefault)
hasDefault	bool	whether fallback applies when the arg is absent
required	bool	whether the arg must be supplied
nargs	string	positional arity: "" (one), "?" (0-1), "*" (0+), "+" (1+), or an integer count
choices	list of string	the allowed values ([] = any)
help	string	the per-argument help text

args.Command

A subcommand: a name plus its own `Parser` (its own flags and positionals).

Field	Type	Description
name	string	the subcommand word (e.g. "add")
help	string	the subcommand's one-line help
parser	Parser	the parser for the subcommand's own arguments

args.Parser

A command-line specification: the program name, description, arguments, and any subcommands.

Value-semantic - every builder returns an updated copy.

Field	Type	Description
prog	string	the program name shown in usage
help	string	the one-line description
version	string	the version string for --version ("" = no --version)
args	list of Arg	the declared flags and positionals, in order
commands	list of Command	the declared subcommands

args.Result

The outcome of a parse. Read it with the typed accessors rather than poking the maps directly. When `done` is true the parser handled `-h` / `--help` / `--version` : print `helpText` and stop.

Field	Type	Description
command	string	the chosen subcommand name ("" if none)
values	map of string to string	store-action values (string form)
lists	map of string to list of string	append-action and variadic-positional values
counts	map of string to int	count-action tallies
present	map of string to bool	which args were supplied on the command line
helpText	string	the text to print when done is set (--help / --version)
done	bool	true when --help / --version was handled (caller should stop)

barcode API reference

Generate scannable barcodes as images (not printer commands - the complement to `label` , which emits printer-native barcode commands). `encode(data, symbology, opts) -> Symbol` builds a device-independent representation - a black/white module matrix for 2D codes, a run of bar widths for 1D codes - and the renderers turn a `Symbol` into output: `svg` (resolution-independent, embeds in HTML / email), `png` (a monochrome PNG hand-encoded over `compress` + `crc` , no image library), `terminal` (Unicode half-block art), and `matrix` (the raw cells).

Symbologies: 2D `qr` (Reed-Solomon over GF(256), EC levels L/M/Q/H, automatic version selection 1-40, data-mask scoring, and numeric / alphanumeric / byte mode chosen for compactness) and `datamatrix` (ECC200, square symbols 10x10 to 26x26); 1D `code128` , `code93` , `ean13` , `ean8` , `upca` , `upce` , `itf` , `code39` , and `gs1-128` (FNC1 + Application Identifiers). A 1D SVG also carries a human-readable text line (Options.humanReadable). Pure `.j` over `compress` (zlib) + `crc` (CRC-32) + `encoding` + the bitwise operators; runs on both binaries.

Import with `import "barcode.j" as barcode;` . See the `barcode` guide for prose and examples.

Functions

barcode.defaults()

Sensible default rendering options (scale 8, quiet 4, EC level M, black on white). Scale 8 (8 px per module / narrow bar) keeps a code comfortably scannable off a screen by a phone camera, where a smaller render invites focus and moire trouble; drop `opts.scale` for a more compact image.

Returns `{Options}` - the defaults

barcode.encode(data as string, symbology as string, opts as Options)

Encode data as a symbol of the given symbology. 2D: "qr", "datamatrix". 1D: "code128", "code93", "ean13", "ean8", "upca", "upce", "itf", "code39", "gs1-128".

Parameters

- `data` `{string}` - the payload
- `symbology` `{string}` - the symbology
- `opts` `{Options}` - rendering / EC options (uses `ecLevel` for QR)

Returns `{Symbol}` - the encoded symbol

Throws

- `{Error}` - kind "barcode" on an unknown symbology or invalid data

barcode.matrix(symbol as Symbol)

The raw cells of a 2D symbol.

Parameters

- `symbol` `{Symbol}` - a matrix symbol

Returns `{list of list of bool}` - the module grid (true = dark)

barcode.png(symbol as Symbol, opts as Options)

Render a symbol as a monochrome (grayscale) PNG.

Parameters

- `symbol` `{Symbol}` - the symbol
- `opts` `{Options}` - scale / quiet / height

Returns `{bytes}` - the PNG image

barcode.svg(symbol as Symbol, opts as Options)

Render a symbol as an SVG string.

Parameters

- `symbol` `{Symbol}` - the symbol
- `opts` `{Options}` - scale / quiet / height / colours

Returns `{string}` - the SVG document

barcode.terminal(symbol as Symbol)

Render a symbol as Unicode half-block art for a terminal (2D only).

Parameters

- `symbol` `{Symbol}` - a matrix symbol

Returns `{string}` - the terminal rendering

Structs

barcode.Options

Rendering options.

Field	Type	Description
scale	int	pixels (PNG) or units (SVG) per module / narrow bar
height	int	bar height for 1D codes (module units)
quiet	int	quiet-zone width in modules / narrow bars
ecLevel	string	QR error-correction level: "L", "M", "Q", or "H"
foreground	string	the dark colour (SVG only; PNG is always black), e.g. "#000000"
background	string	the light colour (SVG only; PNG is always white), e.g. "#ffffff"
humanReadable	bool	render the data as a text line under a 1D barcode (SVG only; on by default)

barcode.Symbol

A device-independent encoded symbol.

Field	Type	Description
kind	SymbolKind	Matrix (2D) or Linear (1D)
size	int	the matrix dimension (2D; 0 for 1D)
matrix	list of list of bool	the 2D module grid (true = dark)
bars	list of int	1D bar/space run widths, starting with a bar
text	string	the encoded data

Enums

barcode.SymbolKind

The kind of an encoded symbol: `Matrix` (a 2D module grid, e.g. QR) or `Linear` (a 1D run of bar / space widths).

bloom API reference

A Bloom filter: a compact, probabilistic set. `add` records a string; `mightContain` tests membership with **no false negatives** (a member always reports true) but possible **false positives** (a non-member may report true, with a probability that grows as the filter fills). The bit array is packed into `bytes` ; the `hashes` positions per item come from double-hashing one SHA-256 digest (`pos_i = (h1 + i*h2) mod size`), so a single hash yields all k positions.

Value-semantic: `add` returns a fresh filter (the bit array is copied), so chain adds (`$f = bloom.add($f, x)`). Over `hash` + `strings` + `binary` ; runs on both binaries.

Import with `import "bloom.j" as bloom;` . See the `bloom` guide for prose and examples.

Functions

bloom.add(f as Filter, item as string)

Add an item to the filter. Returns a fresh filter.

Parameters

- `f` `{Filter}` - the filter
- `item` `{string}` - the item to add

Returns `{Filter}` - a filter with the item recorded

bloom.addAll(f as Filter, items as list of string)

Add every item of a list. Returns a fresh filter.

Parameters

- `f` `{Filter}` - the filter
- `items` `{list of string}` - the items to add

Returns `{Filter}` - a filter with all items recorded

bloom.deserialize(b as bytes)

Reconstruct a filter from bytes produced by `serialize` . The result is identical (same size, k, and bits), so membership is preserved.

Parameters

- `b` `{bytes}` - the encoded filter

Returns `{Filter}` - the reconstructed filter

Throws

- `{Error}` - kind "bloom" if the byte layout is malformed

bloom.merge(a as Filter, b as Filter)

Alias for `union` .

Parameters

- `a` `{Filter}` - the first filter
- `b` `{Filter}` - the second filter

Returns `{Filter}` - a filter holding the members of both

Throws

- `{Error}` - kind "bloom" if the two filters differ in size or k

bloom.mightContain(f as Filter, item as string)

Test whether an item might be in the filter. False positives are possible; false negatives never happen (a previously added item always returns true).

Parameters

- `f` `{Filter}` - the filter
- `item` `{string}` - the item to test

Returns `{bool}` - true if the item might be present, false if it is definitely absent

bloom.new(size as int, hashes as int)

Create an empty filter with `size` bits and `hashes` hash functions (k).

Parameters

- `size` `{int}` - the number of bits (must be >= 1)
- `hashes` `{int}` - the number of hash positions per item (must be >= 1)

Returns `{Filter}` - the empty filter

Throws

- `{Error}` - kind "bloom" if size or hashes is < 1

bloom.optimal(n as int, fpr as float)

Create a filter sized for n expected elements at a target false-positive rate `fpr` . Picks the optimal bit count m = ceil(-(n *ln(fpr))/(ln(2)^2)) and hash count k = round((m/n)* ln(2)) (clamped to k >= 1), then returns an empty Filter of that shape.

Parameters

- n `{int}` - the expected number of elements (must be >= 1)
- `fpr` `{float}` - the target false-positive rate, in the open interval (0, 1)

Returns `{Filter}` - an empty filter sized for the requested load

Throws

- `{Error}` - kind "bloom" if n < 1 or fpr is not in (0, 1)

bloom.serialize(f as Filter)

Serialize a filter to bytes: a 4-byte big-endian `size` , a 4-byte big-endian `hashes` , then the raw bit array. `deserialize` reverses it exactly.

Parameters

- `f` `{Filter}` - the filter to encode

Returns `{bytes}` - the encoded filter

bloom.union(a as Filter, b as Filter)

Union two filters of the same size and k with a bitwise OR. An item present in either input is present in the result. Returns a fresh filter.

Parameters

- a `{Filter}` - the first filter
- b `{Filter}` - the second filter

Returns `{Filter}` - a filter holding the members of both

Throws

- `{Error}` - kind "bloom" if the two filters differ in size or k

Structs

bloom.Filter

A Bloom filter.

Field	Type	Description
bits	bytes	the packed bit array
size	int	the number of bits
hashes	int	the number of hash positions per item (k)

cron API reference

Parse and evaluate cron expressions - the five-field schedule spec `minute hour day-of-month month day-of-week`. `parse` turns an expression into a `Schedule`, `matches` tests whether a `time.Time` fires it, and `next` finds the next fire at or after a given time. Each field takes `*`, single values, `a-b` ranges, `a,b,c` lists, and `/n` steps (`a-b/n`, or a wildcard with a step). Day-of-week is `0-7` (both `0` and `7` are Sunday). When both day-of-month and day-of-week are restricted, a day matching **either** fires (the standard cron rule). The month field also accepts the three-letter names `JAN` - `DEC` and the weekday field `SUN` - `SAT` (case-insensitive), anywhere a number works. A whole expression may be a nickname macro (`@daily`, `@hourly`, ...); `@reboot` parses to a Schedule that never fires (startup only).

A pure calculator over `time` - no clock, no sleeping. A scheduler is the caller's loop (`spawn` + `time.sleep` until `cron.next`). Both binaries.

Import with `import "cron.j" as cron;`. See the `cron` guide for prose and examples.

Functions

cron.matches(schedule as Schedule, t as time.Time)

Report whether a time fires the schedule (minute granularity - seconds are ignored).

Parameters

- `schedule` `{Schedule}` - the schedule
- `t` `{time.Time}` - the instant to test

Returns `{bool}` - true if the schedule fires at `t`

cron.next(schedule as Schedule, after as time.Time)

Find the next time at or after `after` that fires the schedule. Searches minute by minute (skipping non-matching days whole), up to a five-year horizon.

Parameters

- `schedule` `{Schedule}` - the schedule
- `after` `{time.Time}` - the earliest acceptable fire time (its zone is kept)

Returns `{time.Time}` - the next fire time (seconds zeroed)

Throws

- `{Error}` - kind "cron" if nothing matches within the horizon

cron.parse(expr as string)

Parse a five-field cron expression into a Schedule. The whole expression may instead be a nickname macro (`@yearly` / `@annually`, `@monthly`, `@weekly`, `@daily` / `@midnight`, `@hourly`, `@reboot`). The month field accepts `JAN` - `DEC` and the weekday field `SUN` - `SAT` (case-insensitive) anywhere a number works.

Parameters

- `expr` `{string}` - `minute hour day-of-month month day-of-week`, or a `@macro`

Returns `{Schedule}` - the parsed schedule

Throws

- `{Error}` - kind "cron" on the wrong field count or an out-of-range value

Structs

cron.Schedule

A parsed cron schedule: the allowed values per field. `weekdays` are ISO weekdays (Monday = 1 ... Sunday = 7), normalized from the cron `0-7` form.

Field	Type	Description
minutes	list of int	allowed minutes (0-59)
hours	list of int	allowed hours (0-23)
daysOfMonth	list of int	allowed days of the month (1-31)
months	list of int	allowed months (1-12)
weekdays	list of int	allowed ISO weekdays (1-7, Monday = 1)
domStar	bool	whether the day-of-month field was "*"
dowStar	bool	whether the day-of-week field was "*"
reboot	bool	true only for the @reboot macro (a startup-only schedule that never fires a time: matches is always false and next throws)

csv API reference

RFC 4180 comma-separated values: parse text into rows of fields and format rows back into text, with a quoting-aware hand-written scanner. Pure Jennifer - no Go, no system library. The delimiter is configurable, so the same code reads and writes TSV and other single-character-separated formats. Records separate on LF or CRLF (a bare CR outside quotes also ends a record); a field is quoted with `"` when it contains the delimiter, a quote, or a newline, and an embedded quote doubles to `""` . format() joins records with LF and adds no trailing newline, so parse(format(rows)) round-trips the data.

Import with `import "csv.j" as csv;` . See the `csv` guide for prose and examples.

Functions

csv.closeReader(reader as Reader)

Close a streaming reader (closes the underlying file handle).

Parameters

- `reader {Reader}` - the reader

csv.closeWriter(writer as Writer)

Close a streaming writer (closes the underlying file handle).

Parameters

- `writer {Writer}` - the writer

csv.dialect(delimiter as string)

A Dialect with the given delimiter and the standard defaults: quote `"` , no comment prefix, no trimming. Set the other fields to customise.

Parameters

- `delimiter {string}` - the single-character field delimiter

Returns `{Dialect}` - the dialect

csv.format(rows as list of list of string)

Join rows into standard comma-delimited CSV. **Not safe for a spreadsheet target** - use `formatSafe` when the output may be opened in Excel / Sheets (see `formatWith` on the CWE-1236 formula-injection risk).

Parameters

- `rows {list of list of string}` - the rows of fields to format

Returns `{string}` - the formatted comma-separated text

csv.formatDialect(rows as list of list of string, d as Dialect)

Format rows under a Dialect (custom delimiter and quote character). Records are joined with LF and no trailing newline, like formatWith. The comment and trim fields are parse-only and ignored here.

Parameters

- `rows {list of list of string}` - the rows of fields to format
- `d {Dialect}` - the dialect controlling delimiter/quote

Returns `{string}` - the formatted text

csv.formatSafe(rows as list of list of string)

Injection-safe standard comma-delimited CSV (see formatSafeWith). The export-to-spreadsheet path; prefer it over format for spreadsheet targets.

Parameters

- `rows` `{list of list of string}` - the rows of fields to format

Returns `{string}` - the formatted, injection-neutralised comma-separated text

csv.formatSafeWith(rows as list of list of string, delim as string)

Like formatWith, but neutralises spreadsheet formula injection (CWE-1236) before quoting: any field whose first character is = , + , - , @ , a tab, or a CR is prefixed with a single apostrophe so Excel / Google Sheets import it as literal text instead of executing it. Use this - not formatWith - for any CSV a spreadsheet will open.

Parameters

- `rows` `{list of list of string}` - the rows of fields to format
- `delim` `{string}` - the single-character field delimiter

Returns `{string}` - the formatted, injection-neutralised text

csv.formatWith(rows as list of list of string, delim as string)

Join rows into text with a single-character delimiter, quoting each field as needed. Records are separated by LF with no trailing newline.

Not safe for a spreadsheet target. A field beginning = + - @ (or a tab / CR) is a formula that Excel / Google Sheets will execute on open (CWE-1236). If the output is opened as a spreadsheet, use `formatSafe` / `formatSafeWith` , which neutralise such fields.

Parameters

- `rows` `{list of list of string}` - the rows of fields to format
- `delim` `{string}` - the single-character field delimiter

Returns `{string}` - the formatted delimiter-separated text

csv.fromRecords(header as list of string, records as list of map of string to string)

The inverse of toRecords: emit the header row followed by one row per record, taking fields in header order (a key absent from a record writes ""). The explicit header fixes the column order, which map iteration does not.

Parameters

- `header` `{list of string}` - the column names, in output order
- `records` `{list of map of string to string}` - the records to emit

Returns `{list of list of string}` - the header row followed by one row per record

csv.parse(s as string)

Scan standard comma-delimited CSV.

Parameters

- `s` `{string}` - the CSV text to parse

Returns `{list of list of string}` - the parsed rows of fields

csv.parseDialect(text as string, d as Dialect)

Parse CSV text under a Dialect: custom delimiter and quote, comment-line skipping, and optional unquoted-field trimming. Quoted fields keep their content (including whitespace and embedded newlines) verbatim.

Parameters

- `text` `{string}` - the CSV text to parse
- `d` `{Dialect}` - the dialect controlling delimiter/quote/comment/trim

Returns `{list of list of string}` - the parsed rows of fields

csv.parseWith(s as string, delim as string)

Scan CSV text with a single-character delimiter and return its rows, each a list of string fields. A quoted field may span the delimiter, newlines, and doubled quotes; an empty input yields no rows.

Parameters

- `s` `{string}` - the CSV text to parse
- `delim` `{string}` - the single-character field delimiter

Returns `{list of list of string}` - the parsed rows of fields

csv.readRow(reader as Reader)

Read one complete CSV record from the reader, returning its fields. A record whose quoted field spans physical lines is read to completion (further lines are pulled until the quotes balance). Returns the empty list `[]` at end-of-file (guard the loop with `readerEof`) and for a blank line.

Parameters

- `reader` `{Reader}` - the reader

Returns `{list of string}` - the record's fields, or `[]` at EOF / on a blank line

csv.reader(file as fs.File)

Wrap an open, read-mode `fs.File` as a comma-delimited streaming reader.

Parameters

- `file` `{fs.File}` - an open read-mode file handle

Returns `{Reader}` - the reader

csv.readerEof(reader as Reader)

Whether the reader has reached end-of-file. Loop `while (not readerEof($r))` .

Parameters

- `reader` `{Reader}` - the reader

Returns `{bool}` - true once no further record remains

csv.readerWith(file as fs.File, delim as string)

Wrap an open, read-mode `fs.File` as a streaming reader with a custom single-character delimiter.

Parameters

- `file` `{fs.File}` - an open read-mode file handle
- `delim` `{string}` - the single-character field delimiter

Returns `{Reader}` - the reader

csv.toRecords(rows as list of list of string)

Treat the first row as a header and map each later row into a `map of string to string` keyed by the header names. Every record carries every header key (a short row fills missing fields with ""); fields past the header width are dropped. An empty input yields no records.

Parameters

- `rows` `{list of list of string}` - the rows, with the first as the header

Returns `{list of map of string to string}` - one record per non-header row

csv.writeRow(writer as Writer, fields as list of string)

Append one record to the writer: the fields, quoted as needed, followed by LF.

Parameters

- `writer` `{Writer}` - the writer
- `fields` `{list of string}` - the record's fields

csv.writer(file as fs.File)

Wrap an open write/append-mode `fs.File` as a comma-delimited streaming writer.

Parameters

- `file` `{fs.File}` - an open write- or append-mode file handle

Returns `{Writer}` - the writer

csv.writerWith(file as fs.File, delim as string)

Wrap an open write/append-mode `fs.File` as a streaming writer with a custom single-character delimiter.

Parameters

- `file` `{fs.File}` - an open write- or append-mode file handle
- `delim` `{string}` - the single-character field delimiter

Returns `{Writer}` - the writer

Structs

csv.Dialect

A CSV dialect: the field delimiter, the quote character, an optional comment-line prefix (a physical line whose record starts with it is skipped on parse; "" disables comments), and whether unquoted fields are whitespace-trimmed on parse. Build one with `dialect(delimiter)` for the common defaults.

Field	Type	Description
delimiter	string	the single-character field delimiter
quote	string	the quote character (default ")
comment	string	the comment-line prefix, or "" for none
trim	bool	trim leading/trailing whitespace from unquoted fields

csv.Reader

A streaming CSV reader over an open `fs.File` , for tables too large to hold in memory. The wrapped file is a handle - it shares its read position across value copies - so successive `readRow` calls advance the same stream.

Field	Type	Description
file	fs.File	the underlying open file handle
delim	string	the single-character field delimiter

csv.Writer

A streaming CSV writer over an open `fs.File` . Each `writeRow` call appends one quoted-as-needed record terminated by LF, so a large table is written without building the whole text in memory.

Field	Type	Description
file	fs.File	the underlying open write/append-mode file handle
delim	string	the single-character field delimiter

discord API reference

Post messages to a Discord channel through a channel Webhook, on top of the `http` client - a sibling of `gotify` / `slack` . `send(webhookUrl, content)` posts a plain message; the `message` builder assembles a richer payload from embeds (`embed` , plus `embedField` / `embedFooter` / `embedAuthor`) and optional webhook identity overrides (`username` / `avatar`), and `sendMessage` posts it. Needs the default `jennifer` binary (uses `net` via `http`). The webhook URL is a secret belonging to the caller - read it from the environment or a config file; never commit it.

Import with `import "discord.j" as discord;` . See the `discord` guide for prose and examples.

Functions

discord.avatar(m as Message, url as string)

Set a per-message avatar override (the image Discord shows instead of the webhook's configured avatar). Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `url` `{string}` - the avatar image URL

Returns `{Message}` - a message with the avatar override set

discord.content(m as Message, content as string)

Set the top-level message content. Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `content` `{string}` - the content text

Returns `{Message}` - a message with the content set

discord.embed(m as Message, title as string, description as string, color as int)

Add an embed (a titled, coloured card). `color` is a decimal RGB integer (e.g. 3066993 for green). At least one of `title` / `description` should be non-empty. Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `title` `{string}` - the embed title
- `description` `{string}` - the embed body (Discord markdown)
- `color` `{int}` - the left-bar colour as a decimal RGB integer

Returns `{Message}` - a message with the embed appended

discord.embedAuthor(m as Message, name as string)

Set the author name on the most recent embed. Throws if the message has no embed yet. Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `name` `{string}` - the author name

Returns `{Message}` - a message whose latest embed gained the author

discord.embedField(m as Message, name as string, value as string, inline as bool)

Add a field (a name / value pair, optionally inline) to the most recent embed. Throws if the message has no embed yet. Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `name` `{string}` - the field name (bold heading)
- `value` `{string}` - the field value (Discord markdown)
- `inline` `{bool}` - true to let Discord pack this field beside others

Returns `{Message}` - a message whose latest embed gained the field

discord.embedFooter(m as Message, text as string)

Set the footer text on the most recent embed. Throws if the message has no embed yet. Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `text` `{string}` - the footer text

Returns `{Message}` - a message whose latest embed gained the footer

discord.message()

Start an empty rich message (no content, no embeds, default identity).

Returns `{Message}` - a fresh message

discord.render(m as Message)

Render a message to its JSON payload string.

Parameters

- `m` `{Message}` - the message

Returns `{string}` - the JSON payload Discord expects

discord.send(webhookUrl as string, content as string)

Post a plain-text message to a Discord Webhook.

Parameters

- `webhookUrl` `{string}` - the channel Webhook URL
- `content` `{string}` - the message content (Discord markdown)

Returns `{http.Response}` - Discord answers 204 No Content on success

discord.sendMessage(webhookUrl as string, m as Message)

Post a built rich message to a Discord Webhook.

Parameters

- `webhookUrl` `{string}` - the channel Webhook URL
- `m` `{Message}` - the message to send

Returns `{http.Response}` - Discord answers 204 No Content on success

discord.username(m as Message, name as string)

Set a per-message username override (the name Discord shows instead of the webhook's configured name). Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `name` `{string}` - the display name to show

Returns `{Message}` - a message with the username override set

Structs

discord.Embed

One embed under construction: the title / description / colour card plus the optional field list, footer, and author. The `fields` entries and the `footer` / `author` strings are pre-rendered JSON fragments ("" to omit).

Field	Type	Description
title	string	the embed title
description	string	the embed body (Discord markdown)
color	int	the left-bar colour as a decimal RGB integer
fields	list of string	pre-rendered field JSON fragments
footer	string	"" or a pre-rendered footer JSON object
author	string	"" or a pre-rendered author JSON object

discord.Message

A rich message under construction: a top-level `content` string, a list of embeds, and optional webhook-identity overrides (`username` / `avatar` , each "" to omit).

Field	Type	Description
content	string	the message content ("" to omit; embeds must then be present)
embeds	list of Embed	the message embeds
username	string	"" or a per-message username override
avatar	string	"" or a per-message avatar image URL override

docblock API reference

A Jennifer doc-comment parser. Read Jennifer source and return the documentation embedded in it as structured, typed values. It produces data; it does not render (turning docs into HTML is a separate consumer). A doc comment opens with a doc-block marker, and its body is a summary line, an optional description, and tag lines; it immediately precedes the construct it documents (`func` , `def struct` , `def enum` , `def const`) or, when it carries a module tag, is the file preamble. `export` is read from the construct keyword, not a tag. Types are written verbatim in Jennifer syntax inside braces. The module reports, never enforces: signature mismatches (a documented name that names no real parameter, a parameter with no doc) and orphaned comments surface as Diagnostic values, and the caller decides what is fatal.

Import with `import "docblock.j" as docblock;` . See the `docblock` guide for prose and examples.

Functions

docblock.parse(source as string)

Read Jennifer source and return a FileDoc: the module preamble, one doc per func / struct / const, and any diagnostics (mismatched or orphaned doc comments). It reports; it does not fail on a documentation error.

Parameters

- `source {string}` - the Jennifer source text to parse

Returns `{FileDoc}` - the extracted documentation tree

Structs

docblock.ConstDoc

The documentation of one constant.

Field	Type	Description
name	string	the constant name
exported	bool	whether the constant is exported
type	string	the declared type, verbatim in Jennifer syntax
summary	string	the summary (its first paragraph)
description	string	the description (the paragraphs after the summary)
since	string	the documented since-version
deprecated	string	the deprecation note, or "" if not deprecated
see	list of string	the cross-references
internal	bool	whether the constant is marked internal

docblock.Diagnostic

A reported documentation problem (a mismatch or an orphaned comment).

Field	Type	Description
severity	string	the level, currently always "warning"
line	int	the source line the doc comment documents
message	string	the human-readable description of the problem

docblock.EnumDoc

The documentation of one enum (sum type). Enum variants are described in the summary / description prose

rather than with per-variant tags, so an `EnumDoc` carries no field list.

Field	Type	Description
name	string	the enum name
exported	bool	whether the enum is exported
summary	string	the summary (its first paragraph)
description	string	the description (the paragraphs after the summary)
since	string	the documented since-version
deprecated	string	the deprecation note, or "" if not deprecated
see	list of string	the cross-references
internal	bool	whether the enum is marked internal

docblock.FileDoc

The full documentation extracted from one source file.

Field	Type	Description
module	ModuleDoc	the module preamble documentation
funcs	list of FuncDoc	the documented methods
structs	list of StructDoc	the documented structs
enums	list of EnumDoc	the documented enums (sum types)
consts	list of ConstDoc	the documented constants
diagnostics	list of Diagnostic	the reported documentation problems

docblock.FuncDoc

The documentation of one method.

Field	Type	Description
name	string	the method name
exported	bool	whether the method is exported
summary	string	the summary (its first paragraph)
description	string	the description (the paragraphs after the summary)
params	list of ParamDoc	the documented parameters
returns	ReturnDoc	the documented return value
throws	list of ThrowDoc	the documented thrown errors
examples	list of string	the documented examples
since	string	the documented since-version
deprecated	string	the deprecation note, or "" if not deprecated
see	list of string	the cross-references
internal	bool	whether the method is marked internal

docblock.ModuleDoc

The module preamble documentation (the doc comment carrying a module tag).

Field	Type	Description
summary	string	the summary (its first paragraph)

description	string	the description (the paragraphs after the summary)
author	string	the documented author
version	string	the documented version
license	string	the documented license
see	list of string	the cross-references

docblock.ParamDoc

One documented parameter or struct field.

Field	Type	Description
name	string	the parameter or field name
type	string	the declared type, verbatim in Jennifer syntax
description	string	the prose description

docblock.ReturnDoc

A documented return value.

Field	Type	Description
type	string	the returned type, verbatim in Jennifer syntax
description	string	the prose description

docblock.StructDoc

The documentation of one struct.

Field	Type	Description
name	string	the struct name
exported	bool	whether the struct is exported
summary	string	the summary (its first paragraph)
description	string	the description (the paragraphs after the summary)
fields	list of ParamDoc	the documented fields
since	string	the documented since-version
deprecated	string	the deprecation note, or "" if not deprecated
see	list of string	the cross-references
internal	bool	whether the struct is marked internal

docblock.ThrowDoc

A documented thrown error.

Field	Type	Description
type	string	the thrown type, verbatim in Jennifer syntax
description	string	the prose description

dot API reference

Graphviz DOT graph description: build a graph of nodes and edges with attributes and render it to `.dot` text for an external Graphviz tool to lay out and draw (`dot -Tsvg graph.dot > graph.svg`). Pure Jennifer - no Go, no system library, both binaries - it emits the text description only; graph layout is Graphviz's job and is deliberately not reimplemented. Value semantics throughout: every builder returns a fresh Graph, so the graph you pass in is never mutated.

Import with `import "dot.j" as dot;` . See the `dot` guide for prose and examples.

Functions

dot.digraph(name as string)

A new, empty directed graph. Renders as `digraph` with `a -> b` edges.

Parameters

- `name {string}` - the graph name

Returns `{Graph}` - an empty directed graph

dot.edge(g as Graph, src as string, dst as string)

Add a bare edge between two node ids.

Parameters

- `g {Graph}` - the graph
- `src {string}` - the source node id
- `dst {string}` - the destination node id

Returns `{Graph}` - a copy with the edge appended

dot.edgeAttr(g as Graph, key as string, value as string)

Set a default edge attribute, rendered once as `edge [ ... ]` before the edges - so every edge inherits, e.g. `color` = gray.

Parameters

- `g {Graph}` - the graph
- `key {string}` - the attribute name
- `value {string}` - the attribute value

Returns `{Graph}` - a copy with the default set

dot.edgeWith(g as Graph, src as string, dst as string, attrs as map of string to string)

Add an edge with attributes (`{"label": "calls", "style": "dashed"}`).

Parameters

- `g {Graph}` - the graph
- `src {string}` - the source node id
- `dst {string}` - the destination node id
- `attrs {map of string to string}` - the edge's DOT attributes

Returns `{Graph}` - a copy with the edge appended

dot.graph(name as string)

A new, empty undirected graph. Renders as `graph` with a `--` b edges.

Parameters

- `name` `{string}` - the graph name

Returns `{Graph}` - an empty undirected graph

dot.graphAttr(g as Graph, key as string, value as string)

Set a graph-level attribute (rendered as `key="value";`), e.g. `rankdir` = LR or a graph `label` .

Parameters

- `g` `{Graph}` - the graph
- `key` `{string}` - the attribute name
- `value` `{string}` - the attribute value

Returns `{Graph}` - a copy with the attribute set

dot.node(g as Graph, id as string)

Add a bare node (no attributes). Adding a node is optional - an id first seen on an edge is created implicitly by Graphviz - but an explicit node carries a label, shape, or colour.

Parameters

- `g` `{Graph}` - the graph
- `id` `{string}` - the node id

Returns `{Graph}` - a copy with the node appended

dot.nodeAttr(g as Graph, key as string, value as string)

Set a default node attribute, rendered once as `node [ ... ]` before the nodes - so every node inherits, e.g. `shape` = box.

Parameters

- `g` `{Graph}` - the graph
- `key` `{string}` - the attribute name
- `value` `{string}` - the attribute value

Returns `{Graph}` - a copy with the default set

dot.nodeWith(g as Graph, id as string, attrs as map of string to string)

Add a node with attributes (`{"label": "Parser", "shape": "box"}`).

Parameters

- `g` `{Graph}` - the graph
- `id` `{string}` - the node id
- `attrs` `{map of string to string}` - the node's DOT attributes

Returns `{Graph}` - a copy with the node appended

dot.render(g as Graph)

Render the graph to DOT text (a trailing newline included), ready to write to a `.dot` file or pipe to `dot` .

Parameters

- `g` `{Graph}` - the graph to render

Returns `{string}` - the DOT source

Structs

dot.Edge

A directed or undirected edge between two node ids, with its own attributes.

Field	Type	Description
from	string	the source node id
to	string	the destination node id
attrs	map of string to string	DOT attributes, in insertion order

dot.Graph

A whole graph: its kind (directed = digraph), name, ordered nodes and edges, and default attribute blocks for the graph, its nodes, and its edges.

Field	Type	Description
directed	bool	true renders a digraph with -> edges, false a graph with --
name	string	the graph name (quoted in the output)
nodes	list of Node	explicit nodes, in insertion order
edges	list of Edge	edges, in insertion order
graphAttrs	map of string to string	key="value"; graph-level attributes
nodeAttrs	map of string to string	the node [...] default block
edgeAttrs	map of string to string	the edge [...] default block

dot.Node

A single graph node: an id (its key in the graph) plus an ordered attribute map (label, shape, color, ...).

Field	Type	Description
id	string	the node identifier
attrs	map of string to string	DOT attributes, in insertion order

dotenv API reference

Read `.env` configuration files - the `KEY=VALUE` lines that keep secrets and settings out of source - with layered profiles and `${VAR}` interpolation.

Single-file primitives: `parse` turns text into a `map of string to string` , `read` parses a file, and `load` parses a file and sets each variable in the process environment (via `os.setEnv` , unconditional override). Layered loaders: `readCascade` / `resolve` / `loadCascade` / `autoload` merge `.env` -> `.env.local` -> `.env.<profile>` -> `.env.<profile>.local` from one explicit directory, later overriding earlier, with a **real OS env var always winning** over a file value (a committed file cannot clobber a deployment secret). The profile comes from `JENNIFER_ENV` (empty = base files only).

Values handle # comments (whole-line and inline on unquoted values), blank lines, a leading `export` , single-quoted (fully literal) and double-quoted (expand `\n` / `\t` / `\r` , may span multiple physical lines) values, and `${VAR}` interpolation in unquoted and double-quoted values (backward-reference only: earlier keys -> real OS env -> empty; single quotes never interpolate; no `$(...)` / backtick command substitution). Over `fs` + `strings` + `os` + `path` + `regex` + `maps` ; pure `.j` , both binaries.

Import with `import "dotenv.j" as dotenv;` . See the `dotenv` guide for prose and examples.

Functions

dotenv.autoload(dir as string)

Convenience: `loadCascade(dir, JENNIFER_ENV)` - the layered, real-env-wins load with the profile taken from the `JENNIFER_ENV` environment variable (empty = base files only). The one env var this module reads to pick a profile.

Parameters

- `dir {string}` - the base directory (usually `os.cwd()`)

Returns `{map of string to string}` - the merged file variables

Throws

- `{Error}` - kind "dotenv" on an invalid profile label or variable name

dotenv.load(path as string)

Read a `.env` file and set each variable in the process environment (via `os.setEnv` , **unconditional override**), returning the parsed map. This is the low-level primitive; for a real-env-wins layered load use `loadCascade` / `autoload` .

Parameters

- `path {string}` - the file path

Returns `{map of string to string}` - the variables that were set

Throws

- `{Error}` - on a filesystem error, or an invalid variable name

dotenv.loadCascade(dir as string, profile as string)

Merge the `.env` layers (`readCascade`) then `os.setEnv` each variable **only when it is not already set** in the real environment - a real env var is never clobbered by a committed file. A variable set to the **empty string** counts as unset (there is no `os.hasEnv`), so a file value still fills it. Returns the file map (all keys, whether or not they were set). To force-override instead, use `readCascade` + your own `os.setEnv` loop.

Parameters

- `dir {string}` - the base directory
- `profile {string}` - the profile label, or "" for base files only

Returns `{map of string to string}` - the merged file variables

Throws

- `{Error}` - kind "dotenv" on an invalid profile label or variable name

dotenv.parse(text as string)

Parse `.env` text into a map. Blank lines and # comment lines are skipped, a leading `export` is stripped, and each KEY=VALUE becomes an entry (later duplicates win). Double-quoted values may span multiple physical lines; `${VAR}` references are interpolated (earlier keys -> real OS env -> ""). A line with no = or an empty key is ignored.

Parameters

- `text {string}` - the `.env` file contents

Returns `{map of string to string}` - the parsed variables

Throws

- `{Error}` - kind "dotenv" on an unterminated multi-line double-quoted value

dotenv.read(path as string)

Read and parse a `.env` file, without touching the environment.

Parameters

- `path {string}` - the file path

Returns `{map of string to string}` - the parsed variables

Throws

- `{Error}` - on a filesystem error (a positioned `fs` error)

dotenv.readCascade(dir as string, profile as string)

Merge the `.env` layers from one directory into a map, **without touching the environment** . Reads, later overriding earlier and skipping absent files: `.env` -> `.env.local` -> `.env.<profile>` -> `.env.<profile>.local` . A `${VAR}` in a later file resolves against keys from earlier files. An empty `profile` loads only the base files (there is no `.env.default`).

Parameters

- `dir {string}` - the single base directory (no search / no walk-up)
- `profile {string}` - the profile label, or "" for base files only

Returns `{map of string to string}` - the merged file variables

Throws

- `{Error}` - kind "dotenv" on an invalid profile label; an absent layer is skipped (not an error) but an unreadable one raises the `fs` error

dotenv.resolve(dir as string, profile as string)

The **effective** configuration map: `readCascade` overlaid by the real OS environment, so a real env var always wins over a file value. The result holds exactly the file map's keys (each taking the OS-env value when that variable is set), for a program that reads a config map instead of calling `os.getEnv` . Touches nothing in the environment. A variable set to the **empty string** in the environment counts as unset (there is no `os.hasEnv`), so a file value still fills it.

Parameters

- `dir` `{string}` - the base directory
- `profile` `{string}` - the profile label, or "" for base files only

Returns `{map of string to string}` - the effective values (real env wins)

feed API reference

Web syndication feeds: build and parse both RSS 2.0 and Atom 1.0 through one module. A feed is a value-semantic `Feed` (title, link, updated, entries) of `Entry` (title, link, id, published / updated, summary, content). The format is chosen when you `build` (`"rss"` or `"atom"`) and detected from the root element when you `parse` , so the same `Feed` shape serves both - a feed reader, podcast client, news aggregator, or changelog-to-feed generator.

Parsing rides the `xml` library (entities, CDATA, Atom's namespaces); building emits escaped XML through the same. Dates go through `time` (RFC 822 `pubDate` for RSS, RFC 3339 `updated` for Atom); an unset date is the Unix epoch and is omitted on build. `fetch` pulls a feed over the `http` module.

Import with `import "feed.j" as feed;` . See the `feed` guide for prose and examples.

Functions

feed.add(f as Feed, e as Entry)

A copy of `f` with `e` appended to its entries.

Parameters

- `f` `{Feed}` - the source feed
- `e` `{Entry}` - the entry to append

Returns `{Feed}` - the feed with the entry added

feed.build(f as Feed, format as string)

Render `f` to feed XML in the named format: `"rss"` (RSS 2.0) or `"atom"` (Atom 1.0). Values are XML-escaped; unset dates are omitted.

Parameters

- `f` `{Feed}` - the feed to render
- `format` `{string}` - `"rss"` or `"atom"`

Returns `{string}` - the feed document as XML text

Throws

- `{Error}` - when `format` is not `"rss"` or `"atom"`

feed.entry(title as string, link as string)

A new `Entry` with the given title and link (empty id / summary / content, unset dates).

Parameters

- `title` `{string}` - the entry title
- `link` `{string}` - the entry URL

Returns `{Entry}` - the new entry

feed.entryAuthor(e as Entry, author as string)

A copy of `e` with its author set.

Parameters

- `e` `{Entry}` - the source entry
- `author` `{string}` - the item author (a name or email)

Returns `{Entry}` - the updated entry

feed.entryCategory(e as Entry, category as string)

A copy of e with `category` appended to its category tags.

Parameters

- `e` `{Entry}` - the source entry
- `category` `{string}` - the category / tag to add

Returns `{Entry}` - the entry with the category added

feed.entryContent(e as Entry, c as string)

A copy of e with its full content set (Atom).

Parameters

- `e` `{Entry}` - the source entry
- `c` `{string}` - the content text

Returns `{Entry}` - the updated entry

feed.entryEnclosure(e as Entry, url as string, length as int, type as string)

A copy of e with a media enclosure set (the podcast / audio / video file).

Parameters

- `e` `{Entry}` - the source entry
- `url` `{string}` - the media URL
- `length` `{int}` - the media size in bytes (0 when unknown)
- `type` `{string}` - the MIME type (e.g. "audio/mpeg")

Returns `{Entry}` - the entry with the enclosure set

feed.entryId(e as Entry, id as string)

A copy of e with its stable id set.

Parameters

- `e` `{Entry}` - the source entry
- `id` `{string}` - the identifier (RSS guid / Atom id)

Returns `{Entry}` - the updated entry

feed.entryPublished(e as Entry, t as time.Time)

A copy of e with its published instant set.

Parameters

- `e` `{Entry}` - the source entry
- `t` `{time.Time}` - the first-published instant

Returns `{Entry}` - the updated entry

feed.entrySummary(e as Entry, s as string)

A copy of e with its summary set.

Parameters

- `e` `{Entry}` - the source entry
- `s` `{string}` - the summary / description text

Returns `{Entry}` - the updated entry

feed.entryUpdated(e as Entry, t as time.Time)

A copy of `e` with its updated instant set.

Parameters

- `e` `{Entry}` - the source entry
- `t` `{time.Time}` - the last-updated instant

Returns `{Entry}` - the updated entry

feed.feed(title as string, link as string)

A new empty `Feed` with the given title and site link (no entries, unset date).

Parameters

- `title` `{string}` - the feed title
- `link` `{string}` - the feed's site URL

Returns `{Feed}` - the new feed

feed.feedAuthor(f as Feed, author as string)

A copy of `f` with its author set.

Parameters

- `f` `{Feed}` - the source feed
- `author` `{string}` - the feed author (a name or email)

Returns `{Feed}` - the updated feed

feed.feedCategory(f as Feed, category as string)

A copy of `f` with `category` appended to its category tags.

Parameters

- `f` `{Feed}` - the source feed
- `category` `{string}` - the category / tag to add

Returns `{Feed}` - the feed with the category added

feed.feedUpdated(f as Feed, t as time.Time)

A copy of `f` with its `updated` instant set.

Parameters

- `f` `{Feed}` - the source feed
- `t` `{time.Time}` - the last-updated instant

Returns `{Feed}` - the updated feed

feed.fetch(url as string)

Fetch a feed over HTTP and parse it: `http.get` the URL, then `parse` the body. Needs the `http` module (the default `jennifer` ; a friendly network error on `jennifer-tiny`).

Parameters

- `url` `{string}` - the feed URL

Returns `{Feed}` - the fetched, parsed feed

Throws

- `{Error}` - on a transport error, a non-2xx status, or malformed feed XML

feed.hasEnclosure(e as Entry)

Report whether an entry carries a media enclosure (a non-empty enclosure url).

Parameters

- e `{Entry}` - the entry to test

Returns `{bool}` - true when the entry has an attached media file

feed.kind(text as string)

The feed format of `text` , by its root element: `"rss"` or `"atom"` .

Parameters

- `text` `{string}` - the feed XML

Returns `{string}` - `"rss"` or `"atom"`

Throws

- `{Error}` - when the XML is malformed or the root is neither

feed.parse(text as string)

Parse feed XML into a `Feed` , detecting RSS vs Atom from the root element. Malformed dates degrade to the epoch sentinel rather than failing the parse.

Parameters

- `text` `{string}` - the feed XML (RSS 2.0 or Atom 1.0)

Returns `{Feed}` - the parsed feed

Throws

- `{Error}` - when the XML is malformed or the root is neither feed format

Structs

feed.Enclosure

A media enclosure attached to an entry: the podcast / audio / video file. Maps to RSS `<enclosure url length type>` and Atom `<link rel="enclosure">` . An entry with an empty `url` has no enclosure (omitted on build).

Field	Type	Description
url	string	the media URL
length	int	the media size in bytes (0 when unknown)
type	string	the MIME type (e.g. "audio/mpeg")

feed.Entry

One feed item. `link` is the item URL, `id` its stable identifier (RSS `guid` / Atom `id`), `summary` a short description (RSS `description` / Atom `summary`) and `content` the full body (Atom `content` ; RSS carries only the one description, mapped to `summary`). `author` is the item author, `categories` its tags, and `enclosure` an attached media file (podcast episode). An unset date is the Unix epoch.

Field	Type	Description
title	string	the entry title

link	string	the entry URL
id	string	the stable identifier (RSS guid / Atom id)
published	time.Time	the first-published instant (epoch if unset)
updated	time.Time	the last-updated instant (epoch if unset)
summary	string	a short summary / description
content	string	the full content (Atom only)
author	string	the item author (RSS author / dc:creator, Atom author/name)
categories	list of string	the item's category tags
enclosure	Enclosure	an attached media file (empty url = none)

feed.Feed

A syndication feed: channel-level metadata plus its entries.

Field	Type	Description
title	string	the feed title
link	string	the feed's site URL
updated	time.Time	the feed's last-build / updated instant (epoch if unset)
entries	list of Entry	the feed items, in document order
author	string	the feed author (RSS managingEditor, Atom author/name)
categories	list of string	the feed's category tags

flatdb API reference

A file-backed JSON document store. Load a JSON file once into a value-semantic handle, query and edit it in memory through JSON Pointer, and write it back with a crash-atomic whole-file replace (temp file + rename). It is deliberately NOT a database engine: crash-atomic snapshotting of small data. Atomicity is whole-file (temp + rename); there is no isolation (one process, reload-the-whole-file, no concurrent transactions) and durability is only as strong as the OS's buffering of the rename. For a real database, use a client over `net` (e.g. `redis`), not this. It is a thin file-lifecycle + ergonomics layer over the `json` write surface, not a re-implementation of it. Runs on both binaries (pure `json` + `fs` , no network).

Import with `import "flatdb.j" as flatdb;` . See the `flatdb` guide for prose and examples.

Functions

flatdb.append(db as DB, pointer as string, value as json.Value)

Push value onto the list addressed by pointer, returning a new DB. The list must already exist (create it first with `set($db, ptr, json.list())`).

Parameters

- `db` `{DB}` - the store to edit
- `pointer` `{string}` - the JSON Pointer to a list
- `value` `{json.Value}` - the value to append

Returns `{DB}` - a new store with the element appended

Throws

- `{Error}` - when pointer does not resolve to an existing list

flatdb.get(db as DB, pointer as string)

Return the sub-document at pointer (the whole document for "").

Parameters

- `db` `{DB}` - the store to read
- `pointer` `{string}` - the JSON Pointer

Returns `{json.Value}` - the node at pointer

Throws

- `{Error}` - when pointer does not resolve

flatdb.has(db as DB, pointer as string)

Report whether pointer resolves to an existing node.

Parameters

- `db` `{DB}` - the store to read
- `pointer` `{string}` - the JSON Pointer

Returns `{bool}` - true when the node exists

flatdb.keys(db as DB, pointer as string)

List the keys of the object at pointer, in document order.

Parameters

- `db` `{DB}` - the store to read

- `pointer {string}` - the JSON Pointer to an object

Returns `{list of string}` - the object's keys

Throws

- `{Error}` - when pointer does not resolve to an object

flatdb.length(db as DB, pointer as string)

Return the element count of a list, or entry count of an object, at pointer.

Parameters

- `db {DB}` - the store to read
- `pointer {string}` - the JSON Pointer to a list or object

Returns `{int}` - the element or entry count

Throws

- `{Error}` - when pointer does not resolve to a list or object

flatdb.open(path as string)

Load the JSON document at path into a DB. A missing file yields an empty document (an empty object), so open never fails on a first run.

Parameters

- `path {string}` - the backing file path

Returns `{DB}` - the loaded store

flatdb.openString(text as string)

Load a DB from an in-memory JSON string instead of a file - for a database fetched over the network (`http.get(url, {}).body`), embedded in the program, or built elsewhere. The returned DB has **no backing path, so it is read-only** : every reader verb works and the mutating verbs still return a fresh in-memory DB, but `save` throws (there is nowhere to write). Whitespace-only text yields an empty document, matching `open` on a missing file. This keeps flatdb transport-agnostic: it never imports `http` / `net` , so it stays `fs` -only and builds on both binaries; the caller brings the bytes.

Parameters

- `text {string}` - the JSON document

Returns `{DB}` - a read-only store

flatdb.remove(db as DB, pointer as string)

Drop the key or element at pointer, returning a new DB.

Parameters

- `db {DB}` - the store to edit
- `pointer {string}` - the JSON Pointer to remove

Returns `{DB}` - a new store with the node removed

Throws

- `{Error}` - when pointer does not resolve

flatdb.save(db as DB)

Write the document back to its own backing file, crash-atomically (temp + rename). A read-only DB (from `openString` , no backing file) has nowhere to write - use `saveAs` to give it a path.

Parameters

- `db` `{DB}` - the store to persist

Throws

- `{Error}` - kind "flatdb" if the DB is read-only (no backing file), or on a filesystem write or rename failure

flatdb.saveAs(db as DB, path as string)

Write the document to `path` (not necessarily its own) and return a **fresh DB bound to that path** - the passed-in `db` is unchanged (value semantics, like `set` / `append` / `remove`). Two uses: the first dump to disk of a read-only `openString` DB, and copying / forking an on-disk store to a new file (a snapshot or a new version). Which store is "current" for the next `save` is whichever handle you keep - reassign (`$db = flatdb.saveAs($db, path);`) to make the new file current, or hold both to keep the original too. Same crash-atomic temp+rename as `save` .

Parameters

- `db` `{DB}` - the store whose data to write
- `path` `{string}` - the destination file path

Returns `{DB}` - a new DB bound to `path` (its data equals `db` 's)

Throws

- `{Error}` - kind "flatdb" if `path` is empty, or on a filesystem failure

flatdb.set(db as DB, pointer as string, value as json.Value)

Write value at pointer (upsert an object key / replace a list index), returning a new DB. Strict: intermediate containers must already exist.

Parameters

- `db` `{DB}` - the store to edit
- `pointer` `{string}` - the JSON Pointer to write
- `value` `{json.Value}` - the value to write (build scalars with `json.decode` , objects and lists with `json.map` / `json.list`)

Returns `{DB}` - a new store with the write applied

Throws

- `{Error}` - when an intermediate container does not exist

Structs

flatdb.DB

The value the caller holds: the file path plus the decoded document. A module holds no mutable state and `spawn` deep-copies scope, so a store cannot be a shared open connection - it is a value. Mutating verbs return a fresh DB; `save` is the only side effect.

Field	Type	Description
path	string	the backing file path (empty for a read-only DB from openString)
data	json.Value	the decoded in-memory document

font API reference

A pure-Jennifer TrueType / OpenType (SFNT) font parser: read a `.ttf` / `.otf` from `bytes` and expose its metrics, character map, and glyph outlines. No Go, no dependency - just the `bytes` type and the bitwise operators for the big-endian tables - so it runs on **both binaries** .

Both outline backends ship: the **TrueType glyf** backend (simple and composite glyphs, quadratic curves) and the **CFF / PostScript** backend for OpenType `OTTO` fonts (a Type2 charstring interpreter with global / local subroutines and CID-keyed FDArray / FDSelect, so CJK fonts outline too), detected on parse; a CFF glyph's cubic curves render as native `C` segments in `glyphPath` and are approximated as quadratics in the `Glyph` struct. It parses the core tables - `head` , `cmap` (formats 4 and 12), `maxp` / `hhea` / `hmtx` , `OS/2` (vertical metrics), the legacy `kern` table, `loca` / `glyf` or `CFF` , and `name` .

Import with `import "font.j" as font;` . See the `font` guide for prose and examples.

Functions

font.advance(f as Font, cp as int)

The horizontal advance width of the glyph for a codepoint, in font units.

Parameters

- `f {Font}` - the font
- `cp {int}` - the Unicode codepoint

Returns `{int}` - the advance width

font.advanceGid(f as Font, gid as int)

The horizontal advance width of a glyph id, in font units (the raw hmtx metric, for building a PDF `W` widths array keyed by glyph / CID).

Parameters

- `f {Font}` - the font
- `gid {int}` - the glyph id

Returns `{int}` - the advance width

font.advances(f as Font, gids as list of int)

Batch advance-width lookup: the advances (font units) for a list of glyph ids, in one call. Prefer this over `advanceGid` per glyph (see `glyphIds`).

Parameters

- `f {Font}` - the font
- `gids {list of int}` - the glyph ids

Returns `{list of int}` - the advance widths, in order

font.ascender(f as Font)

The typographic ascender in font units (OS/2 `sTypoAscender` , else the hhea ascent). The distance from the baseline to the top of the em box.

Parameters

- `f {Font}` - the font

Returns `{int}` - the ascender

font.bbox(f as Font)

The font bounding box in font units as [xMin, yMin, xMax, yMax] (the `head` table's global glyph bounds, for a PDF `FontBBox`).

Parameters

- `f {Font}` - the font

Returns `{list of int}` - [xMin, yMin, xMax, yMax]

font.capHeight(f as Font)

The capital height in font units (OS/2 `sCapHeight`): the height of a flat capital such as `H`. 0 when the font has no OS/2 v2+ table.

Parameters

- `f {Font}` - the font

Returns `{int}` - the cap height, or 0 when unavailable

font.data(f as Font)

The raw font file bytes (for embedding the font in a container such as a PDF `FontFile2`).

Parameters

- `f {Font}` - the font

Returns `{bytes}` - the font file contents

font.descender(f as Font)

The typographic descender in font units (OS/2 `sTypoDescender`, else hhea). Conventionally negative (below the baseline).

Parameters

- `f {Font}` - the font

Returns `{int}` - the descender

font.glyph(f as Font, cp as int)

The full outline of the glyph for a codepoint: its contours (on / off-curve points), advance, and bounding box. A codepoint the font lacks maps to glyph 0 (`.notdef`).

Parameters

- `f {Font}` - the font
- `cp {int}` - the Unicode codepoint

Returns `{Glyph}` - the glyph outline

font.glyphId(f as Font, cp as int)

The glyph id (index) a codepoint maps to through the font's character map, or 0 (`.notdef`) when the font lacks the codepoint. The identifier a PDF embeds under Identity encoding.

Parameters

- `f {Font}` - the font
- `cp {int}` - the Unicode codepoint

Returns `{int}` - the glyph id

font.glyphIds(f as Font, cps as list of int)

Batch glyph-id lookup: map a list of codepoints to their glyph ids in one call. Prefer this over calling `glyphId` per

character - the font is value-semantic (its raw bytes are copied on every call), so a per-character loop copies the whole font each time; this copies it once for the whole batch.

Parameters

- `f {Font}` - the font
- `cps {list of int}` - the codepoints

Returns `{list of int}` - the glyph ids, in order

font.glyphPath(f as Font, cp as int)

The glyph outline for a codepoint as an SVG path `d` string, in font-unit coordinates (y-up, as fonts store them - flip y for screen rendering). Quadratic segments render as `Q` commands. An empty glyph yields `""` .

Parameters

- `f {Font}` - the font
- `cp {int}` - the Unicode codepoint

Returns `{string}` - the SVG path data

font.isCff(f as Font)

Whether the font uses CFF / PostScript outlines (an OpenType `OTTO`) rather than TrueType `glyf` outlines.

Parameters

- `f {Font}` - the font

Returns `{bool}` - true for a CFF font

font.kern(f as Font, left as int, right as int)

The kerning adjustment between two codepoints in font units (negative pulls them closer). Reads the legacy `kern` table (version 0, horizontal format-0 subtables); 0 when the font has no `kern` table or no pair entry. Modern fonts carry kerning in GPOS instead, which this does not read.

Parameters

- `f {Font}` - the font
- `left {int}` - the left codepoint
- `right {int}` - the right codepoint

Returns `{int}` - the kerning adjustment, or 0

font.lineGap(f as Font)

The typographic line gap in font units (OS/2 `sTypoLineGap` , else hhea): the recommended extra leading between lines. Line height = ascender - descender + lineGap.

Parameters

- `f {Font}` - the font

Returns `{int}` - the line gap

font.name(f as Font)

The font family name.

Parameters

- `f {Font}` - the font

Returns `{string}` - the family name

font.numGlyphs(f as Font)

The number of glyphs in the font.

Parameters

- `f` `{Font}` - the font

Returns `{int}` - the glyph count

font.open(path as string)

Load and parse a font from a file path.

Parameters

- `path` `{string}` - the .ttf file path

Returns `{Font}` - the parsed font

Throws

- `{Error}` - on a read or parse error

font.parse(b as bytes)

Parse a TrueType / OpenType (SFNT) font from its bytes - a `glyf` (TrueType) or `CFF` (OpenType/PostScript) outline font.

Parameters

- `b` `{bytes}` - the font file contents

Returns `{Font}` - the parsed font

Throws

- `{Error}` - on a malformed font or an unrecognised container

font.unitsPerEm(f as Font)

The font's units-per-em (the coordinate scale of every outline / metric).

Parameters

- `f` `{Font}` - the font

Returns `{int}` - units per em

font.xHeight(f as Font)

The x-height in font units (OS/2 `sxHeight`): the height of a lowercase `x` . 0 when the font has no OS/2 v2+ table.

Parameters

- `f` `{Font}` - the font

Returns `{int}` - the x-height, or 0 when unavailable

Structs

font.Contour

One closed contour of a glyph: its ordered points.

Field	Type	Description
points	list of Point	the contour points

font.Font

A parsed font. Holds the raw bytes plus the table offsets and header values needed to answer metric / cmap / outline queries; glyph outlines are decoded on demand.

Field	Type	Description
data	bytes	the raw font file
unitsPerEm	int	the em square size (the coordinate scale)
numGlyphs	int	the number of glyphs
longLoca	bool	whether the loca table uses 32-bit offsets
numHMetrics	int	the number of long horizontal metrics
loca	int	the loca table offset
glyf	int	the glyf table offset
hmtx	int	the hmtx table offset
cmapSub	int	the chosen cmap subtable offset
cmapFmt	int	the chosen cmap subtable format (4 or 12)
family	string	the font family name
ascender	int	the typographic ascender (OS/2 sTypoAscender, else hhea)
descender	int	the typographic descender (negative; OS/2 sTypoDescender, else hhea)
lineGap	int	the typographic line gap (OS/2 sTypoLineGap, else hhea)
capHeight	int	the capital height (OS/2 sCapHeight; 0 when unavailable)
xHeight	int	the x-height (OS/2 sxHeight; 0 when unavailable)
kern	int	the kern-table offset (0 when the font has no kern table)
cff	int	the CFF-table offset (0 for a TrueType/glyf font)
xMin	int	the font bounding-box minimum x (head xMin)
yMin	int	the font bounding-box minimum y (head yMin)
xMax	int	the font bounding-box maximum x (head xMax)
yMax	int	the font bounding-box maximum y (head yMax)

font.Glyph

A glyph outline: its contours, advance width, and bounding box, all in font units. An empty glyph (e.g. a space) has no contours.

Field	Type	Description
advance	int	the horizontal advance width
xMin	int	the bounding-box minimum x
yMin	int	the bounding-box minimum y
xMax	int	the bounding-box maximum x
yMax	int	the bounding-box maximum y
contours	list of Contour	the glyph contours

font.Point

A point in a glyph contour, in font-unit coordinates.

Field	Type	Description

x	int	the x coordinate
y	int	the y coordinate
onCurve	bool	whether the point is on the curve (else a quadratic control point)

gotify API reference

Push notifications to a Gotify server (https://gotify.net), on top of the `http` client. Hold a value-semantic Config (server URL + application token) and call push; it POSTs the message form to URL/message with the X-Gotify-Key header, per Gotify's push-message contract. Needs the default `jennifer` binary (uses `net` via `http`). The URL and token belong to the caller - read them from the environment or a config file; never commit them.

Import with `import "gotify.j" as gotify;` . See the `gotify` guide for prose and examples.

Functions

gotify.push(cfg as Config, title as string, message as string, priority as int)

Send a notification to the Gotify server.

Parameters

- `cfg` `{Config}` - the server URL + token
- `title` `{string}` - the notification title
- `message` `{string}` - the notification body
- `priority` `{int}` - the Gotify priority (higher is more urgent)

Returns `{http.Response}` - 2xx on success; a bad token surfaces as a 4xx value

gotify.pushExtras(cfg as Config, title as string, message as string, priority as int, extras as Extras)

Send a notification carrying `extras` (markdown rendering and/or a click URL). POSTs a JSON body (not the form) so the nested `extras` object reaches Gotify; the plain `push` stays form-encoded and unchanged.

Parameters

- `cfg` `{Config}` - the server URL + token
- `title` `{string}` - the notification title
- `message` `{string}` - the notification body
- `priority` `{int}` - the Gotify priority (higher is more urgent)
- `extras` `{Extras}` - the extras to attach (a zero Extras adds none)

Returns `{http.Response}` - 2xx on success; a bad token surfaces as a 4xx value

gotify.pushMarkdown(cfg as Config, title as string, message as string, priority as int)

Send a notification whose body renders as **markdown** (sets the `client::display` content-type to `text/markdown`). Convenience over `pushExtras` with `Extras{markdown: true}` .

Parameters

- `cfg` `{Config}` - the server URL + token
- `title` `{string}` - the notification title
- `message` `{string}` - the markdown notification body
- `priority` `{int}` - the Gotify priority (higher is more urgent)

Returns `{http.Response}` - 2xx on success; a bad token surfaces as a 4xx value

gotify.pushWith(cfg as Config, title as string, message as string, priority as int, clickUrl as string)

Send a notification that opens `clickUrl` when tapped (sets `client::notification.click.url`).

Convenience over `pushExtras` with `Extras{clickUrl: ...}` .

Parameters

- `cfg` `{Config}` - the server URL + token
- `title` `{string}` - the notification title
- `message` `{string}` - the notification body
- `priority` `{int}` - the Gotify priority (higher is more urgent)
- `clickUrl` `{string}` - the URL to open when the notification is tapped

Returns `{http.Response}` - 2xx on success; a bad token surfaces as a 4xx value

Structs

gotify.Config

A Gotify target: value-semantic, passed to each push (no module state).

Field	Type	Description
url	string	the server URL, no trailing slash
token	string	the application token

gotify.Extras

Optional Gotify message extras (value-semantic, passed to `pushExtras`). A zero `Extras` (markdown false, empty clickUrl) adds nothing, so a push with it matches the plain `push` . Only the requested keys reach the wire.

Field	Type	Description
markdown	bool	render the message body as markdown (sets the client::display content-type to text/markdown)
clickUrl	string	URL to open when the notification is tapped (sets client::notification.click.url); "" for none

gpio API reference

Raspberry-Pi (and any Linux SBC) GPIO over sysfs. /sys/class/gpio is plain files, so `fs` is the whole backend: export a pin, set its direction, read / write its value, unexport it. "Blink an LED from a five-line script." It is stateless and pin-keyed (sysfs derives every path from the pin number, so no handle is needed). The sysfs root defaults to /sys/class/gpio; set the JENNIFER_GPIO_BASE environment variable to point elsewhere (a differently mounted sysfs, or a mock tree under test). This is the sysfs backend, which is deprecated in favour of the /dev/gpiochip character device but stays available on the hobbyist Pi kernels this targets; the API is kept stable so the backend could later be swapped for a chardev system library with no change to scripts. GPIO is a Linux-platform feature: off a GPIO-capable host every call reports a clear "base directory not found" error rather than crashing.

Import with `import "gpio.j" as gpio;` . See the `gpio` guide for prose and examples.

Functions

gpio.read(pin as int)

Return a pin's current value (0 or 1).

Parameters

- `pin` `{int}` - the GPIO pin number

Returns `{int}` - the pin's current value, 0 or 1

Throws

- `{Error}` - when the sysfs GPIO tree is absent

gpio.release(pin as int)

Unexport a pin.

Parameters

- `pin` `{int}` - the GPIO pin number

Throws

- `{Error}` - when the sysfs GPIO tree is absent

gpio.setup(pin as int, direction as string)

Export a pin and set its direction (gpio.IN or gpio.OUT).

Parameters

- `pin` `{int}` - the GPIO pin number
- `direction` `{string}` - the pin direction, gpio.IN or gpio.OUT (the "in" / "out" strings)

Throws

- `{Error}` - when direction is not gpio.IN/gpio.OUT, or the sysfs GPIO tree is absent

gpio.write(pin as int, value as int)

Set an output pin's value (0 or 1).

Parameters

- `pin` `{int}` - the GPIO pin number
- `value` `{int}` - the value to write, 0 or 1

Throws

- `{Error}` - when value is not 0/1, or the sysfs GPIO tree is absent

graphql API reference

A thin GraphQL client over `http` / `rest` . Point a `graphql.Client` at one endpoint URL, then `graphql.query(client, query, variables)` POSTs `{"query": ..., "variables": ...}` and returns the decoded JSON response as a `json.Value` (the result is under `/data`).

The query is an **opaque string** the caller supplies - GraphQL syntax is the server's job, not this module's - and a mutation is just a query string, so no separate verb is needed. The one GraphQL-specific rule this client gets right: a GraphQL execution error is an **HTTP 200 with a top-level errors array** , not a non-2xx status, so `query` inspects the payload and raises a positioned `Error` (kind `"graphql"`) carrying the server's messages rather than trusting the status line. A genuine transport / auth failure (a non-2xx status) is also raised as a `graphql` error with the status and body.

For a document that defines several named operations, `queryNamed` / `tryQueryNamed` add the `operationName` selector. To handle GraphQL errors yourself instead of catching a throw - branching on an error `code` , reading partial data - `tryQuery` / `tryQueryNamed` return the raw envelope (both `/data` and `/errors`) and only raise on an HTTP-level failure; `graphql.hasErrors` and `graphql.errorMessages` inspect it.

The `Client` wraps a `rest.Client` , so it carries the endpoint URL, per-request headers, and (via `rest` 's `http.TlsOptions`) TLS settings for a self-signed or private-CA host. Default `jennifer` binary only (net-backed via `http` / `rest`).

Import with `import "graphql.j" as graphql;` . See the `graphql` guide for prose and examples.

Functions

graphql.basic(c as Client, user as string, pass as string)

Return a copy of the client that sends HTTP Basic `Authorization` .

Parameters

- `c {Client}` - the client
- `user {string}` - the username
- `pass {string}` - the password

Returns `{Client}` - a new client with the header set

graphql.bearer(c as Client, token as string)

Return a copy of the client that sends `Authorization: Bearer <token>` .

Parameters

- `c {Client}` - the client
- `token {string}` - the bearer token

Returns `{Client}` - a new client with the header set

graphql.client(endpoint as string)

A GraphQL client for one endpoint URL. The full endpoint is POSTed to verbatim (no path is appended), so pass the complete GraphQL URL (e.g. `https://host/graphql`).

Parameters

- `endpoint {string}` - the GraphQL endpoint URL

Returns `{Client}` - a client with no auth and default (verifying) TLS

graphql.errorMessages(resp as json.Value)

Join the `message` field of each entry in a response's `errors` array into one `"; "` -separated string (this is the text `query` puts in the raised `Error`). For structured error data - `extensions.code`, `path`, `locations` - read the `errors` array of the envelope directly with the `json` accessors, e.g. `json.asString($resp, "/errors/0/extensions/code")`.

Parameters

- `resp` `{json.Value}` - a decoded GraphQL response with an `errors` array

Returns `{string}` - the joined error messages

graphql.hasErrors(resp as json.Value)

Report whether a decoded response carries a non-empty top-level `errors` array - the GraphQL error signal, which the spec delivers on an HTTP 200. Use this on the envelope returned by `tryQuery` to branch without a `try` / `catch`.

Parameters

- `resp` `{json.Value}` - a decoded GraphQL response

Returns `{bool}` - true if the response reports one or more GraphQL errors

graphql.header(c as Client, name as string, value as string)

Return a copy of the client with an arbitrary request header set.

Parameters

- `c` `{Client}` - the client
- `name` `{string}` - the header name
- `value` `{string}` - the header value

Returns `{Client}` - a new client with the header set

graphql.insecure(c as Client)

Return a copy of the client that skips TLS certificate verification. Insecure - for a trusted network or a local test server only.

Parameters

- `c` `{Client}` - the client

Returns `{Client}` - a new client that accepts any server certificate

graphql.query(c as Client, query as string, variables as json.Value)

Run a GraphQL operation (query or mutation) and return the decoded JSON response as a `json.Value`; the result is at `/data`. The `variables` is a `json.Value` object (use an empty `json.map()` when the operation takes none).

Raises a positioned `Error` (kind `"graphql"`) when the server reports GraphQL execution errors (an HTTP 200 with a top-level `errors` array - the messages are collected into the error), or when the request fails at the HTTP level (a non-2xx status, carrying the status and body). A successful return therefore has no `errors`, so `/data` is present and complete. To handle GraphQL errors yourself (e.g. branch on an error `code`), use `tryQuery`.

Parameters

- `c` `{Client}` - the client
- `query` `{string}` - the GraphQL query or mutation document
- `variables` `{json.Value}` - the variables object (empty object for none)

Returns `{json.Value}` - the full decoded response (data under `/data`)

Throws

- `{Error}` - kind "graphql" on GraphQL errors or a non-2xx HTTP status

graphql.queryNamed(c as Client, query as string, variables as json.Value, operationName as string)

Like `query` , but for a document that defines several named operations: the `operationName` selects which one to run (some servers require it when more than one is present).

Parameters

- `c {Client}` - the client
- `query {string}` - the GraphQL document
- `variables {json.Value}` - the variables object (empty object for none)
- `operationName {string}` - the name of the operation to execute

Returns `{json.Value}` - the full decoded response (data under `/data`)

Throws

- `{Error}` - kind "graphql" on GraphQL errors or a non-2xx HTTP status

graphql.tryQuery(c as Client, query as string, variables as json.Value)

Run an operation and return the decoded response **without raising on GraphQL errors** - the envelope comes back with both `/data` (possibly partial) and `/errors` , so you can inspect the errors yourself (`graphql.hasErrors($resp)` , `graphql.errorMessages($resp)` , or `json.asString($resp, "/errors/0/extensions/code")` for structured data). An HTTP-level failure (a non-2xx status) still raises, since there is no GraphQL envelope to return.

Parameters

- `c {Client}` - the client
- `query {string}` - the GraphQL query or mutation document
- `variables {json.Value}` - the variables object (empty object for none)

Returns `{json.Value}` - the full decoded response (data under `/data` , errors under `/errors`)

Throws

- `{Error}` - kind "graphql" only on a non-2xx HTTP status

graphql.tryQueryNamed(c as Client, query as string, variables as json.Value, operationName as string)

`tryQuery` (no raise on GraphQL errors) for a document with several named operations, selected by `operationName` .

Parameters

- `c {Client}` - the client
- `query {string}` - the GraphQL document
- `variables {json.Value}` - the variables object (empty object for none)
- `operationName {string}` - the name of the operation to execute

Returns `{json.Value}` - the full decoded response (data under `/data` , errors under `/errors`)

Throws

- `{Error}` - kind "graphql" only on a non-2xx HTTP status

graphql.withCA(c as Client, pem as bytes)

Return a copy of the client that trusts a private-CA / self-signed certificate (PEM). Certificate verification stays on, against this CA.

Parameters

- `c` `{Client}` - the client
- `pem` `{bytes}` - the CA certificate in PEM form

Returns `{Client}` - a new client with the CA trusted

Structs

graphql.Client

A GraphQL client: an endpoint URL plus headers and TLS options, carried in a wrapped `rest.Client` . Build one with `graphql.client` , then layer auth / TLS with the builders (each returns a new `Client` ; value semantics, no mutation).

Field	Type	Description
rest	rest.Client	the underlying REST client (endpoint, headers, TLS)

html API reference

Build an HTML element tree and render it to a correctly escaped HTML5 string. Pure Jennifer over `strings` and `lists` - a writer, not a parser, so it has no dependency on an XML parser: serialization is a handful of string operations. The shared output layer any HTML-emitting consumer reuses (a Markdown renderer, a documentation generator, a view layer). Text nodes are escaped on render (`&` `<` `>`); attribute values also escape `"` ; a `raw` node passes through verbatim for already-trusted markup. Void elements (`br` , `img` , ...) render without a closing tag and drop children.

Import with `import "html.j" as html;` . See the `html` guide for prose and examples.

Functions

html.attr(name as string, value as string)

Build one name/value attribute for an element.

Parameters

- `name` `{string}` - the attribute name
- `value` `{string}` - the attribute value

Returns `{Attr}` - the attribute

html.attrOf(node as Node, name as string)

The value of an element's attribute by name, or `""` if it has no such attribute. (Read `.attrs` directly for the full list.)

Parameters

- `node` `{Node}` - the element node
- `name` `{string}` - the attribute name

Returns `{string}` - the attribute value, or ""

html.boolAttr(name as string)

Build a boolean (valueless) HTML attribute - `disabled` , `checked` , `selected` , `required` , `readonly` , `multiple` , `autofocus` , and the like. It renders as the bare name (`<input disabled>`), not `name=""` . The name is validated exactly as `attr` validates it.

Parameters

- `name` `{string}` - the attribute name

Returns `{Attr}` - the boolean attribute

html.element(tag as string, attrs as list of Attr, children as list of Node)

Build an element node from a tag, its attributes, and its children. Pass `[]` for either when there are none.

Parameters

- `tag` `{string}` - the element tag name
- `attrs` `{list of Attr}` - the element's attributes
- `children` `{list of Node}` - the element's child nodes

Returns `{Node}` - the element node

html.escape(s as string)

Return s with the text-context HTML metacharacters replaced by entities (`&` first, so an escaped entity is not re-escaped). Public because escaping a bare string for HTML text is useful without building a node.

Parameters

- `s` `{string}` - the string to escape

Returns `{string}` - the escaped string

html.findAll(node as Node, selector as string)

Every element matching an XPath-ish `selector` relative to `node` : `/` -separated steps, each a tag name, `*` (any element), or `name[k]` (the k-th such child, 1-based). Steps match **direct element children** ; text nodes are skipped.

Parameters

- `node` `{Node}` - the node to search under
- `selector` `{string}` - the selector path (e.g. "body/ul/li")

Returns `{list of Node}` - the matching element nodes (empty if none)

html.get(node as Node, selector as string)

The first element matching `selector` (see `findAll`), or an empty element node (`.tag == ""`) if there is no match.

Parameters

- `node` `{Node}` - the node to search under
- `selector` `{string}` - the selector path

Returns `{Node}` - the first match, or an empty element node

html.has(node as Node, selector as string)

Whether any element matches `selector` (see `findAll`).

Parameters

- `node` `{Node}` - the node to search under
- `selector` `{string}` - the selector path

Returns `{bool}` - true if at least one element matches

html.hasAttr(node as Node, name as string)

Whether an element has an attribute by name (including a valueless boolean one).

Parameters

- `node` `{Node}` - the element node
- `name` `{string}` - the attribute name

Returns `{bool}` - true if present

html.parse(src as string)

Parse an HTML string into a `Node` tree. The result is a synthetic `#root` element whose `children` are the document's top-level nodes; walk it with `.children` / `get` / `findAll` , and re-serialize any node with `render` . Tolerant of real-world HTML (void / self-closing tags, unquoted attributes, mismatched nesting, comments, DOCTYPE, `script` / `style` raw text).

Parameters

- `src` `{string}` - the HTML source

Returns `{Node}` - the `#root` element containing the parsed top-level nodes

Throws

- `{Error}` - kind "html" if the document exceeds the depth or node budget

html.raw(s as string)

Build a node whose content is emitted verbatim - for already-trusted markup only, since it is not escaped.

Parameters

- `s` `{string}` - the verbatim markup

Returns `{Node}` - the raw node

html.render(node as Node)

Serialize a node and its subtree to an HTML5 string.

Parameters

- `node` `{Node}` - the node to render

Returns `{string}` - the rendered HTML

html.renderAll(nodes as list of Node)

Serialize a list of sibling nodes (a fragment) in order.

Parameters

- `nodes` `{list of Node}` - the sibling nodes

Returns `{string}` - the rendered HTML fragment

html.safeUrl(url as string)

Return a URL safe to place in an `href` / `src` attribute, or `"#"` if its scheme is not one of `http` / `https` / `mailto`. Whitespace and control characters are ignored while reading the scheme (so `"java\tscript:..."` is still caught), and a relative reference (no scheme) is returned unchanged. This is the anti-XSS gate for building links from untrusted input.

Parameters

- `url` `{string}` - the URL to check

Returns `{string}` - the URL if its scheme is allowed, else `"#"`

html.text(s as string)

Build a text node; its content is HTML-escaped on render.

Parameters

- `s` `{string}` - the text content

Returns `{Node}` - the text node

html.unescape(s as string)

Decode the common HTML entities in a string - `<` / `>` / `&quot;` / `&#39;` / `&apos;` / `&` to their characters - the inverse of `escape`. `&` is decoded last, so `&lt;` yields the literal `<`, not <. A string with no & is returned unchanged.

Scope is deliberately the metacharacter entities only: the ones any standard text-context escaper (including this module's `escape` / `escapeAttr`) emits, so a round-trip is exact. It is not a general HTML5 entity decoder - the ~2000 named references for authoring (` ` / `&copy;` / `&mdash;` / ...) and numeric refs (`&#8212;`) are out of scope, since decoding those means shipping the full named table plus numeric parsing, not the metacharacter round-trip this serves.

Parameters

- `s` `{string}` - the text to decode

Returns `{string}` - the decoded text

Structs

html.Attr

One HTML attribute. A normal attribute has a name and a value and renders as `name="value"` ; a boolean attribute (`boolean: true` , built by `boolAttr`) carries no value and renders as the bare name (`disabled` , `checked` , ...).

Field	Type	Description
name	string	the attribute name
value	string	the attribute value (escaped on render; empty for a boolean attribute)
boolean	bool	true for a valueless boolean attribute (rendered as the bare name)

html.Node

A node is one of three kinds, tagged by `kind` : "element" (tag + attrs + children), "text" (escaped content), or "raw" (verbatim content). The constructors below are the intended way to build one.

Field	Type	Description
kind	NodeKind	the node kind (Element, Text, or Raw)
tag	string	the element tag name (element nodes only)
attrs	list of Attr	the element's attributes (element nodes only)
children	list of Node	the element's child nodes (element nodes only)
text	string	the content of a text or raw node

Enums

html.NodeKind

The kind of an HTML node: `Element` (a tag with attributes and children), `Text` (escaped text content), or `Raw` (verbatim, already-trusted markup).

http API reference

An HTTP/1.1 client over the `net` system library. Build a request (method, URL, headers, body), send it, and read the response back into a `Response` (status, headers, body). `http://` connects in the clear; `https://` connects with TLS (`net.connectTLS`). The one-shot verbs (`get` / `post` / `request` / ...) use one connection per request (`Connection: close`); `connect` / `exchange` reuse a persistent connection to one origin (keep-alive) so a request loop pays a single handshake. `send` adds a request policy on top of the one-shot path: automatic 3xx redirect-following, retry / backoff on 429 / 5xx, and a cookie jar across the redirect chain. Because it uses `net` , this module needs the default `jennifer` binary. A text `Response` body is decoded as UTF-8 (a binary body raises an error; use `requestBytes` / `getBytes` for that). Chunked and Content-Length framing are both handled.

Import with `import "http.j" as http;` . See the `http` guide for prose and examples.

Functions

http.basic(user as string, pass as string)

Build an `Authorization` value for HTTP Basic auth (base64 of "user:password"), to pass as a request header. Mirrors `rest.basic` .

Parameters

- `user` `{string}` - the username
- `pass` `{string}` - the password

Returns `{string}` - the "Basic <base64>" header value

http.close(session as Session)

Close a session's connection. Idempotent-safe to call once; do not use the session afterwards.

Parameters

- `session` `{Session}` - the session to close

http.connect(url as string, options as Options)

Open a persistent connection to the origin of `url` (its scheme / host / port; the path is ignored - `exchange` supplies per-request paths). The socket is reused across `exchange` calls to the same origin, so a request loop pays one handshake instead of N.

Parameters

- `url` `{string}` - a URL whose origin to connect to
- `options` `{Options}` - the request options (timeout, cap, tls)

Returns `{Session}` - an open session

Throws

- `{Error}` - kind "http" (or a net error) if the connection cannot be opened

http.defaultOptions()

The zero `Options` (default timeout / cap, verified TLS, no redirects, no retries), for inline use: `http.send(m, u, {}, "", http.defaultOptions())` . Set fields on the result to opt into behaviour.

Returns `{Options}` - the default options

http.delete(url as string, headers as map of string to string)

Issue a DELETE request.

Parameters

- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)

Returns `{Response}` - the parsed response

http.exchange(session as Session, method as string, path as string, headers as map of string to string, body as string)

Send one request over a session's reused connection and return the response plus the updated session (reassign it: `def x as http.Exchange init http.exchange($s, ...); $s = $x.session;`). `path` is a request target on the session's origin (e.g. "/items?page=2"). Cookies from the response are folded into the session jar and replayed on later requests. If the connection was closed (by a prior `Connection: close` or a to-EOF response), it is transparently reopened. Redirects are **not** followed here (that can cross origins and break the socket) - use `send` for redirect-following; `exchange` returns the 3xx.

Parameters

- `session` `{Session}` - the session (from `connect` or a prior `exchange`)
- `method` `{string}` - the HTTP method
- `path` `{string}` - the request path on the session's origin
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for none)

Returns `{Exchange}` - the response and the session to thread onward

Throws

- `{Error}` - kind "http" on a malformed response or a cap breach, "read timed out" on timeout

http.get(url as string, headers as map of string to string)

Issue a GET request.

Parameters

- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)

Returns `{Response}` - the parsed response

http.getBytes(url as string, headers as map of string to string)

GET a URL and return its body as raw `bytes` (the download shortcut).

Parameters

- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)

Returns `{BytesResponse}` - the response with a raw bytes body

http.head(url as string, headers as map of string to string)

Issue a HEAD request (status and headers, no body).

Parameters

- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)

Returns `{Response}` - the parsed response (empty body)

http.header(resp as Response, name as string)

Read a response header by name, case-insensitively.

Parameters

- `resp` `{Response}` - the response to read from
- `name` `{string}` - the header name (case-insensitive)

Returns `{string}` - the header value, or "" if absent

http.options(url as string, headers as map of string to string)

Issue an OPTIONS request (capability probe; read the `Allow` header).

Parameters

- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)

Returns `{Response}` - the parsed response

http.patch(url as string, contentType as string, body as string, headers as map of string to string)

Issue a PATCH request (a partial update) with `contentType` and `body` .

Parameters

- `url` `{string}` - the absolute request URL
- `contentType` `{string}` - the Content-Type header value
- `body` `{string}` - the request body
- `headers` `{map of string to string}` - extra request headers ({} for none)

Returns `{Response}` - the parsed response

http.post(url as string, contentType as string, body as string, headers as map of string to string)

Issue a POST request with `contentType` and `body` .

Parameters

- `url` `{string}` - the absolute request URL
- `contentType` `{string}` - the Content-Type header value
- `body` `{string}` - the request body
- `headers` `{map of string to string}` - extra request headers ({} for none)

Returns `{Response}` - the parsed response

http.put(url as string, contentType as string, body as string, headers as map of string to string)

Issue a PUT request with `contentType` and `body` .

Parameters

- `url` `{string}` - the absolute request URL
- `contentType` `{string}` - the Content-Type header value
- `body` `{string}` - the request body
- `headers` `{map of string to string}` - extra request headers ({} for none)

Returns `{Response}` - the parsed response

http.request(method as string, url as string, headers as map of string to string, body as string)

Send one HTTP request and return the response (with the default idle timeout).

Parameters

- `method` `{string}` - the HTTP method (e.g. "GET", "POST")
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for no body)

Returns `{Response}` - the parsed response

Throws

- `{Error}` - kind "http" if the response is malformed, or a "read timed out" error on timeout

http.requestBytes(method as string, url as string, headers as map of string to string, body as string)

Send one request and return a raw-bytes `BytesResponse` (default idle timeout, default 64 MiB cap, full TLS verification). The binary counterpart to `request` ; for a large download or a self-signed host use `requestWithBytes` .

Parameters

- `method` `{string}` - the HTTP method (e.g. "GET")
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for no body)

Returns `{BytesResponse}` - the response with a raw bytes body

Throws

- `{Error}` - kind "http" if the response is malformed or exceeds the cap, or a "read timed out" error on timeout

http.requestRawBody(method as string, url as string, headers as map of string to string, body as bytes, timeoutMs as int, maxBytes as int)

Send one request with a **raw bytes body** and return the text `Response` . Unlike `requestWith` (whose string body is UTF-8-encoded onto the wire), the body here is written byte-for-byte, so a binary payload - a `multipart/form-data` file upload, a protobuf - is transmitted intact. The response is still parsed as text (use `requestWithBytes` if the *response* may be non-UTF-8).

Parameters

- `method` `{string}` - the HTTP method (e.g. "POST")
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers (set your own Content-Type)
- `body` `{bytes}` - the raw request body
- `timeoutMs` `{int}` - the per-read idle timeout in milliseconds (0 = none)
- `maxBytes` `{int}` - the response-body cap (0 = 64 MiB default, negative = unlimited)

Returns `{Response}` - the parsed response

Throws

- `{Error}` - kind "http" on a malformed response or a cap breach; "read timed out" on timeout

http.requestRawBodyTls(method as string, url as string, headers as map of string to string, body as bytes, timeoutMs as int, maxBytes as int, tls as TlsOptions)

`requestRawBody` with explicit TLS options for an `https://` server (a self-signed or private-CA host). For `http://` the options are ignored.

Parameters

- `method` `{string}` - the HTTP method
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers
- `body` `{bytes}` - the raw request body
- `timeoutMs` `{int}` - the per-read idle timeout in milliseconds (0 = none)
- `maxBytes` `{int}` - the response-body cap (0 = default, negative = unlimited)
- `tls` `{TlsOptions}` - certificate-verification options for the TLS handshake

Returns `{Response}` - the parsed response

Throws

- `{Error}` - kind "http" on a malformed response or a cap breach; "read timed out" on timeout

http.requestTls(method as string, url as string, headers as map of string to string, body as string, tls as TlsOptions)

Send one request with explicit TLS options (default idle timeout and body cap). The `https` -with-a-self-signed-cert shortcut over `requestWithTls` .

Parameters

- `method` `{string}` - the HTTP method (e.g. "GET", "POST")
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for no body)
- `tls` `{TlsOptions}` - certificate-verification options for the TLS handshake

Returns `{Response}` - the parsed response

Throws

- `{Error}` - kind "http" if the response is malformed, or a "read timed out" error on timeout

http.requestWith(method as string, url as string, headers as map of string to string, body as string, timeoutMs as int, maxBytes as int)

Send one HTTP request with an explicit idle timeout and body cap, returning the response. `timeoutMs` bounds each read (a stalled server fails rather than hanging); 0 disables it. `maxBytes` caps the response body: 0 uses the 64 MiB default (`MAX_BODY_BYTES` , which protects against an untrusted server streaming to OOM), a negative value lifts the cap (for a trusted large download), and a positive value sets an exact ceiling. `request` and the verb shortcuts use `DEFAULT_TIMEOUT_MS` and the default cap.

Parameters

- `method` `{string}` - the HTTP method (e.g. "GET", "POST")
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for no body)
- `timeoutMs` `{int}` - the per-read idle timeout in milliseconds (0 = none)
- `maxBytes` `{int}` - the response-body cap (0 = 64 MiB default, negative = unlimited, positive = exact)

Returns `{Response}` - the parsed response

Throws

- `{Error}` - kind "http" if the response is malformed or exceeds `maxBytes` , or a "read timed out" error on timeout

http.requestWithBytes(method as string, url as string, headers as map of string to string, body as string, timeoutMs as int, maxBytes as int, tls as TlsOptions)

Send one request and return the body as **raw bytes** (a `BytesResponse`) - the byte-safe path for downloading binary content (a `.tar.gz` , an image) that a UTF-8 text `Response` cannot hold. Explicit per-read idle timeout and body cap; pass a negative `maxBytes` for an unbounded download (a large release archive), or `TlsOptions` for a self-signed / private-CA `https://` host.

Parameters

- `method` `{string}` - the HTTP method (e.g. "GET")
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for no body)
- `timeoutMs` `{int}` - the per-read idle timeout in milliseconds (0 = none)
- `maxBytes` `{int}` - the response-body cap (0 = 64 MiB default, negative = unlimited, positive = exact)
- `tls` `{TlsOptions}` - certificate-verification options for the TLS handshake

Returns `{BytesResponse}` - the response with a raw bytes body

Throws

- `{Error}` - kind "http" if the response is malformed or exceeds `maxBytes` , or a "read timed out" error on timeout

http.requestWithTls(method as string, url as string, headers as map of string to string, body as string, timeoutMs as int, maxBytes as int, tls as TlsOptions)

Like `requestWith` , but with explicit TLS options for an `https://` URL (a self-signed or private-CA server). For `http://` the options are ignored.

Parameters

- `method` `{string}` - the HTTP method (e.g. "GET", "POST")
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for no body)
- `timeoutMs` `{int}` - the per-read idle timeout in milliseconds (0 = none)
- `maxBytes` `{int}` - the response-body cap (0 = 64 MiB default, negative = unlimited, positive = exact)
- `tls` `{TlsOptions}` - certificate-verification options for the TLS handshake

Returns `{Response}` - the parsed response

Throws

- `{Error}` - kind "http" if the response is malformed or exceeds `maxBytes` , or a "read timed out" error on timeout

http.send(method as string, url as string, headers as map of string to string, body as string, options as Options)

Send a request with a policy: follow up to `options.maxRedirects` 3xx redirects, retry a 429 / 5xx up to `options.maxRetries` with exponential backoff (honouring a numeric `Retry-After`), and carry cookies across the redirect chain. A `303` (and a `301` / `302` on a `POST`) becomes a bodyless `GET`; `307` / `308` preserve the method and body. With a zero `Options` it is a one-shot request, exactly like `request`.

Parameters

- `method` `{string}` - the HTTP method
- `url` `{string}` - the absolute request URL
- `headers` `{map of string to string}` - request headers ({} for none)
- `body` `{string}` - the request body ("" for none)
- `options` `{Options}` - the request policy

Returns `{Response}` - the final response (after any redirects / retries)

Throws

- `{Error}` - kind "http" on a malformed response or a body-cap breach, "read timed out" on timeout

Structs

http.BytesResponse

A response with a **raw bytes body** - the byte-safe counterpart to `Response`, for downloading binary payloads (a `.tar.gz`, an image, any non-text content) that a UTF-8 string body cannot hold. Returned by `requestBytes` / `requestWithBytes` / `getBytes`. `headers` keys are lowercased (read `$r.headers["content-type"]` directly, or with `http.header` after wrapping - the map is shared shape with `Response`).

Field	Type	Description
status	int	the numeric status code
statusText	string	the reason phrase from the status line
headers	map of string to string	response headers, keys lowercased
body	bytes	the response body, exactly as received (no decoding)

http.Exchange

The result of an `exchange`: the response, and the updated session to thread into the next `exchange` (it carries the reused socket and any new cookies).

Field	Type	Description
response	Response	the response
session	Session	the session to use for the next request

http.Options

Per-request policy for `send` and a persistent `Session`. The zero value (from `http.defaultOptions()` or `def o as http.Options;`) is the safe default: the standard timeout and body cap, TLS fully verified, and **no** redirect-following or retrying (so a bare `send` behaves like `request`). Opt into a behaviour by setting its field.

Field	Type	Description
timeoutMs	int	per-read idle timeout (0 = 30s default, as request)
maxBytes	int	response-body cap (0 = 64 MiB default, negative = unlimited)
maxRedirects	int	how many 3xx redirects to follow (0 = none; the 3xx is returned)

maxRetries	int	how many times to retry a 429 / 5xx (0 = none), with exponential backoff honouring Retry-After
backoffMs	int	base backoff for the first retry (0 = 250ms), doubled each attempt and capped at 30s
tls	TlsOptions	TLS verification options for https:// (zero = full verification)
allowCrossOriginRedirect	bool	keep credential headers / cookies across a redirect to a different origin (default false = drop them, the browser rule)

http.Response

An HTTP response. `headers` keys are lowercased (HTTP header names are case-insensitive); use `http.header` for a case-insensitive read.

Field	Type	Description
status	int	the numeric status code (e.g. 200, 404)
statusText	string	the reason phrase from the status line
headers	map of string to string	response headers, keys lowercased
body	string	the response body decoded as UTF-8 text

http.Session

A persistent HTTP connection to one origin (scheme + host + port), holding the reused socket, a cookie jar, and the request options. Value-semantic, but the `conn` handle is shared across copies (a `net.Conn`), so the live socket survives being threaded through `exchange` . Build with `connect` ; retire with `close` .

Field	Type	Description
conn	net.Conn	the underlying socket (shared across copies)
scheme	string	"http" or "https"
host	string	the origin host
port	int	the origin port
open	bool	whether conn is currently usable (false after the server closed it)
jar	map of string to string	the accumulated cookies (name -> value)
options	Options	the request options (timeout, cap, tls)

http.TlsOptions

TLS options for an `https://` request, mirroring `net.TLSOptions` . The zero value (`skipVerify` false, empty `caCert`) full-verifies the server certificate against the URL host, which is what a plain `request` / verb shortcut uses - so an `https://` call with no options behaves exactly as before.

Field	Type	Description
skipVerify	bool	accept any certificate (self-signed, wrong host, expired). Opt-in; disables authentication and exposes the connection to a man-in-the-middle. Use only for a trusted LAN endpoint you cannot give a proper CA.
caCert	bytes	a PEM certificate to trust in addition to the system roots, for a private CA or a pinned self-signed cert. The safer alternative to skipVerify: the server is still authenticated, just against this cert.

ical API reference

Build and parse iCalendar (RFC 5545): a `Calendar` holding `Event` s (`VEVENT`) and `Todo` s (`VTODO`), encoded to a `VCALENDAR` and parsed back. Pure Jennifer over `strings` / `lists` + `time` - no Go, no system engine.

Beyond the basics (`UID` / `SUMMARY` / `DESCRIPTION` / `LOCATION`), an event supports **recurrence** (`RRULE` / `RDATE` / `EXDATE` , with an `occurrences` expander for `FREQ` / `INTERVAL` / `COUNT` / `UNTIL`), **all-day** dates (`VALUE=DATE`) and a named-zone **TZID** , an `ORGANIZER` and `ATTENDEE` s, and `VALARM` alarms. Times go through `time` : UTC values write as `20240615T130000Z` ; a TZID event stores its wall clock as a floating value paired with the zone name (because `time` models fixed-offset zones only, add a matching `VTIMEZONE` for a strict consumer). Text values are RFC 5545-escaped and content lines folded at 75 octets, so `parse(encode(cal))` round-trips.

Import with `import "ical.j" as ical;` . See the `ical` guide for prose and examples.

Functions

ical.add(cal as Calendar, ev as Event)

A copy of the calendar with an event appended (value-semantic).

Parameters

- `cal {Calendar}` - the calendar
- `ev {Event}` - the event to add

Returns `{Calendar}` - a fresh calendar with the event appended

ical.addAlarm(ev as Event, a as Alarm)

A copy of the event with a `VALARM` appended.

Parameters

- `ev {Event}` - the event
- `a {Alarm}` - the alarm

Returns `{Event}` - a fresh event with the alarm added

ical.addAttendee(ev as Event, a as Attendee)

A copy of the event with an `ATTENDEE` appended.

Parameters

- `ev {Event}` - the event
- `a {Attendee}` - the attendee

Returns `{Event}` - a fresh event with the attendee added

ical.addExdate(ev as Event, t as time.Time)

A copy of the event with an excluded recurrence instant (`EXDATE`) appended.

Parameters

- `ev {Event}` - the event
- `t {time.Time}` - the excluded instant

Returns `{Event}` - a fresh event with the EXDATE added

ical.addRdate(ev as Event, t as time.Time)

A copy of the event with an extra recurrence instant (`RDATE`) appended.

Parameters

- `ev` `{Event}` - the event
- `t` `{time.Time}` - the extra instant

Returns `{Event}` - a fresh event with the RDATE added

ical.addTodo(cal as Calendar, td as Todo)

A copy of the calendar with a to-do appended.

Parameters

- `cal` `{Calendar}` - the calendar
- `td` `{Todo}` - the to-do to add

Returns `{Calendar}` - a fresh calendar with the to-do appended

ical.alarm(action as string, trigger as string, description as string)

An alarm (`VALARM`).

Parameters

- `action` `{string}` - the action ("DISPLAY" / "AUDIO" / "EMAIL")
- `trigger` `{string}` - the trigger value (e.g. `-PT15M`)
- `description` `{string}` - the reminder text ("" for none)

Returns `{Alarm}` - the alarm

ical.attendee(address as string, cn as string, role as string)

An attendee.

Parameters

- `address` `{string}` - the calendar address (e.g. `mailto:bob@example.com`)
- `cn` `{string}` - the common name ("" for none)
- `role` `{string}` - the role ("" for none; e.g. "REQ-PARTICIPANT")

Returns `{Attendee}` - the attendee

ical.calendar()

A calendar with the default Jennifer `PRODID` and no events.

Returns `{Calendar}` - the empty calendar

ical.calendarWith(prodid as string)

A calendar with a caller-supplied `PRODID` .

Parameters

- `prodid` `{string}` - the product identifier

Returns `{Calendar}` - the empty calendar

ical.describe(ev as Event, description as string)

A copy of the event with its description set (value-semantic).

Parameters

- `ev` `{Event}` - the event
- `description` `{string}` - the description text

Returns `{Event}` - a fresh event with the description set

ical.describeTodo(td as Todo, description as string)

A copy of the to-do with its `DESCRIPTION` set.

Parameters

- `td` `{Todo}` - the to-do
- `description` `{string}` - the description

Returns `{Todo}` - a fresh to-do with the description set

ical.encode(cal as Calendar)

Render a calendar to iCalendar text (RFC 5545): a `VCALENDAR` wrapping one `VEVENT` per event (and a `VTODO` per to-do), CRLF line endings, escaped text, folded long lines. Optional properties (`DESCRIPTION` / `LOCATION` / recurrence / organizer / attendees / alarms) are emitted only when set.

Parameters

- `cal` `{Calendar}` - the calendar to encode

Returns `{string}` - the iCalendar text (CRLF-terminated)

ical.event(uid as string, start as time.Time, end as time.Time, summary as string)

An event. `DTSTAMP` defaults to the start; the optional fields (`DESCRIPTION` / `LOCATION` / recurrence / organizer / attendees / alarms) are empty until set with the `describe` / `locate` / `recur` / ... builders.

Parameters

- `uid` `{string}` - the unique identifier
- `start` `{time.Time}` - the start instant
- `end` `{time.Time}` - the end instant
- `summary` `{string}` - the title

Returns `{Event}` - the event

ical.locate(ev as Event, location as string)

A copy of the event with its location set (value-semantic).

Parameters

- `ev` `{Event}` - the event
- `location` `{string}` - the location text

Returns `{Event}` - a fresh event with the location set

ical.occurrences(ev as Event, max as int)

Expand an event's recurrence into up to `max` occurrence instants, in order: the `RRULE` series from `DTSTART` (honouring `FREQ` / `INTERVAL` / `COUNT` / `UNTIL`), plus any `RDATE` s, minus any `EXDATE` s. A non-recurring event yields just its start. Only the frequency rules are expanded - `BYDAY` / `BYMONTH` and other `BY*` parts are not applied (the base cadence is still produced), so this covers the common "every N days / weeks / months / years" case.

Parameters

- `ev` `{Event}` - the event
- `max` `{int}` - the maximum number of occurrences to return

Returns `{list of time.Time}` - the occurrence instants (at most `max`)

ical.parse(text as string)

Parse iCalendar text into a `Calendar` . Unfolds folded lines and reads the `PRODID` , each `VEVENT` (`UID` / `DTSTAMP` / `DTSTART` / `DTEND` / `SUMMARY` / `DESCRIPTION` / `LOCATION` , the all-day `VALUE=DATE` and `TZID` parameters, `RRULE` / `RDATE` / `EXDATE` , `ORGANIZER` / `ATTENDEE` , and nested `VALARM` s), and each `VTODO` . A `VTIMEZONE` is parsed and skipped. An event with no `DTSTART` is skipped; a missing `DTEND` defaults to the start.

Parameters

- `text` `{string}` - the iCalendar text

Returns `{Calendar}` - the parsed calendar

ical.recur(ev as Event, rrule as string)

A copy of the event with a recurrence rule (`RRULE`). Pass a raw RRULE value (e.g. `"FREQ=WEEKLY;COUNT=10"`) or build one with `rule` .

Parameters

- `ev` `{Event}` - the event
- `rrule` `{string}` - the RRULE value

Returns `{Event}` - a fresh recurring event

ical.rule(freq as string, interval as int, count as int)

Build a simple `RRULE` value from a frequency, interval, and count. `INTERVAL` is omitted when 1 and `COUNT` when 0 (i.e. unbounded).

Parameters

- `freq` `{string}` - the frequency ("DAILY" / "WEEKLY" / "MONTHLY" / "YEARLY")
- `interval` `{int}` - the interval (every N periods; 1 = every period)
- `count` `{int}` - the number of occurrences (0 = unbounded)

Returns `{string}` - the RRULE value

ical.todo(uid as string, stamp as time.Time, summary as string)

A to-do (`VTODO`). `DTSTAMP` defaults to `stamp` ; `DUE` / `STATUS` / `DESCRIPTION` are set with the builders.

Parameters

- `uid` `{string}` - the unique identifier
- `stamp` `{time.Time}` - the `DTSTAMP`
- `summary` `{string}` - the title

Returns `{Todo}` - the to-do

ical.withAllDay(ev as Event, isAllDay as bool)

A copy of the event marked (or unmarked) as an all-day event. All-day `DTSTART` / `DTEND` encode as `VALUE=DATE` (a bare `YYYYMMDD`).

Parameters

- `ev` `{Event}` - the event
- `isAllDay` `{bool}` - whether the event is all-day

Returns `{Event}` - a fresh event with the all-day flag set

ical.withDue(td as Todo, due as time.Time)

A copy of the to-do with its `DUE` instant set.

Parameters

- `td` `{Todo}` - the to-do
- `due` `{time.Time}` - the due instant

Returns `{Todo}` - a fresh to-do with the due date set

ical.withOrganizer(ev as Event, address as string)

A copy of the event with its `ORGANIZER` set.

Parameters

- `ev` `{Event}` - the event
- `address` `{string}` - the calendar address (e.g. `mailto:a@example.com`)

Returns `{Event}` - a fresh event with the organizer set

ical.withStatus(td as Todo, status as string)

A copy of the to-do with its `STATUS` set (e.g. "NEEDS-ACTION" / "COMPLETED").

Parameters

- `td` `{Todo}` - the to-do
- `status` `{string}` - the status

Returns `{Todo}` - a fresh to-do with the status set

ical.withZone(ev as Event, tzid as string)

A copy of the event with its time-zone id (`TZID`) set, so `DTSTART` / `DTEND` encode as local time in that zone (`DTSTART;TZID=America/New_York:...`) instead of UTC. The named zone is preserved verbatim; add a matching `VTIMEZONE` for a strict consumer.

Parameters

- `ev` `{Event}` - the event
- `tzid` `{string}` - the IANA time-zone name (e.g. "Europe/London"); "" restores UTC

Returns `{Event}` - a fresh event with the time zone set

Structs

ical.Alarm

An event alarm (`VALARM`).

Field	Type	Description
action	string	the ACTION ("DISPLAY" / "AUDIO" / "EMAIL")
trigger	string	the TRIGGER value (e.g. -PT15M = 15 minutes before)
description	string	the DESCRIPTION (the reminder text; "" when unset)

ical.Attendee

An event attendee (`ATTENDEE`).

Field	Type	Description
address	string	the calendar address (e.g. mailto:bob@example.com)
cn	string	the common name (CN parameter; "" when unset)
role	string	the participation role (ROLE; e.g. "REQ-PARTICIPANT"; "" when unset)

ical.Calendar

A calendar: a product identifier, its events, and its to-dos.

Field	Type	Description
prodid	string	the PRODID product identifier
events	list of Event	the calendar's events
todos	list of Todo	the calendar's to-dos (VTODOs)

ical.Event

A single calendar event (a VEVENT).

Field	Type	Description
uid	string	the globally-unique UID
stamp	time.Time	the DTSTAMP (creation / last-modified instant)
start	time.Time	the DTSTART start instant
end	time.Time	the DTEND end instant
summary	string	the SUMMARY (title)
description	string	the DESCRIPTION ("" when unset)
location	string	the LOCATION ("" when unset)
allDay	bool	whether the event is an all-day event (VALUE=DATE)
tzid	string	the TZID time-zone name for a local DTSTART / DTEND ("" = UTC)
rrule	string	the RRULE recurrence rule value ("" when non-recurring)
rdates	list of time.Time	extra recurrence instants (RDATE)
exdates	list of time.Time	excluded recurrence instants (EXDATE)
organizer	string	the ORGANIZER calendar address ("" when unset)
attendees	list of Attendee	the ATTENDEEs
alarms	list of Alarm	the event's VALARMs

ical.Todo

A to-do item (a VTODO).

Field	Type	Description
uid	string	the globally-unique UID
stamp	time.Time	the DTSTAMP
summary	string	the SUMMARY
due	time.Time	the DUE instant (valid only when hasDue)
hasDue	bool	whether a DUE is set
description	string	the DESCRIPTION ("" when unset)
status	string	the STATUS (e.g. "NEEDS-ACTION" / "COMPLETED"; "" when unset)

idna API reference

Internationalized domain names: convert a Unicode domain to its ASCII-compatible (`xn--`) form and back, over a Punycode (RFC 3492) core. So `münchen.de` goes on the wire as `xn--mnchen-3ya.de` (DNS, SMTP envelopes, URL hosts are ASCII-only). Pure Jennifer over `strings` , `convert` , and `encoding` - no networking, TinyGo-clean. This is the Punycode transformation plus lowercasing, not full IDNA2008 (which layers nameprep / mapping tables on top); it covers the common cases. It needs `convert.toCodepoint` / `convert.fromCodepoint` for the bootstring arithmetic on rune values.

Import with `import "idna.j" as idna;` . See the `idna` guide for prose and examples.

Functions

idna.isAscii(domain as string)

Report whether a domain is already all-ASCII (needs no conversion).

Parameters

- `domain` `{string}` - the domain name to test

Returns `{bool}` - true when every character is ASCII

idna.toAscii(domain as string)

Convert a domain to its ASCII-compatible form, label by label.

Parameters

- `domain` `{string}` - the (possibly Unicode) domain name

Returns `{string}` - the ASCII-compatible domain (Unicode labels get an `xn--` prefix)

idna.toUnicode(domain as string)

Convert an ASCII-compatible domain back to Unicode, label by label.

Parameters

- `domain` `{string}` - the ASCII-compatible domain name

Returns `{string}` - the Unicode domain (`xn--` labels get Punycode-decoded)

imap API reference

An IMAP4rev1 client (RFC 3501): tagged commands and untagged "*" responses over the `net` system library, with plaintext / implicit TLS / STARTTLS and auth by LOGIN, XOAUTH2, CRAM-MD5, or SCRAM-SHA-1 / SCRAM-SHA-256. A practical subset covering both reading and basic folder management, not the full protocol: SELECT a folder, SEARCH it (filtered), FETCH whole messages or named headers, and manage messages - STORE flags, COPY, CREATE a folder, mark + EXPUNGE (so, delete and move). It is not read-only. Retrieved messages come back as strings for the `mime` module to parse. Uses `net` , so it needs the default `jennifer` binary. A session is stateful: `connect` , `selectFolder` , `search` / `fetch` / `fetchHeaders` , optional `addFlags` / `expunge` , `logout` . A "NO" / "BAD" completion throws a catchable `Error` (kind "imap"). One fixed command tag is used, which is safe for this synchronous client (one command in flight at a time). Message literals (`{N}`) are framed over bytes by their byte count, so an 8-bit / multi-byte UTF-8 literal is read byte-exact.

Import with `import "imap.j" as imap;` . See the `imap` guide for prose and examples.

Functions

imap.addFlags(session as Session, uid as int, flags as string)

Add IMAP flags / keywords to a message (`UID STORE +FLAGS.SILENT`). Use a keyword like "$cl_1" to colour the message in Thunderbird, or the system flag "\\Deleted" to mark it for `expunge` . The folder is selected read-write by `selectFolder` , so the store is permitted.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID
- `flags {string}` - the space-separated flags, e.g. "$cl_1" or "\\Deleted"

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.append(session as Session, folder as string, message as string)

Upload `message` (a full RFC 5322 message - headers, a blank line, then the body, e.g. built with the `mime` module) into `folder` (APPEND). The folder must already exist. Use this to save a copy to "Sent" after sending, or to store a message a client composed. The message is sent as a byte-counted literal, so any 8-bit / multi-byte content is uploaded exactly.

Parameters

- `session {Session}` - the open session
- `folder {string}` - the destination folder name
- `message {string}` - the full RFC 5322 message

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap"), e.g. a missing folder

imap.appendWith(session as Session, folder as string, flags as string, message as string)

Like `append` , but sets initial flags on the stored message (APPEND with a flag list) - e.g. `"\\Seen"` for a copy to Sent (already read) or `"\\Draft"` for a saved draft. `flags` is a space-separated flag string, as `addFlags` .

Parameters

- `session {Session}` - the open session
- `folder {string}` - the destination folder name

- `flags {string}` - the space-separated initial flags, e.g. "\\Seen"
- `message {string}` - the full RFC 5322 message

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.connect(opts as Options)

Open a session: greeting, optional STARTTLS, then LOGIN (or XOAUTH2).

Parameters

- `opts {Options}` - the connection and auth parameters

Returns `{Session}` - the open session

Throws

- `{Error}` - on a bad greeting or a "NO" / "BAD" login completion (kind "imap")

imap.copy(session as Session, uid as int, folder as string)

Copy a message (by UID) into another folder (`UID COPY`). The source copy stays until it is deleted, so the standard "move" is copy + addFlags(..., "\\Deleted") + expunge - or the atomic `move` below. The destination folder must already exist - COPY to a missing one is a "NO [TRYCREATE]" error.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID
- `folder {string}` - the destination folder name

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.createFolder(session as Session, folder as string)

Create a folder (CREATE). A server answers "NO" if it already exists (often with an "[ALREADYEXISTS]" response code), so wrap this in try / catch for a create-if-missing.

Parameters

- `session {Session}` - the open session
- `folder {string}` - the folder name to create

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap"), including when the folder already exists

imap.criteria()

An empty `Criteria` (matches all messages). Set the fields you want to filter on, then pass it to `search` .

Returns `{Criteria}` - a zero-value criteria

imap.done(session as Session)

Leave IDLE: send `DONE` , drain any pending pushes, and read the tagged completion of the `IDLE` command, returning the session to command mode so ordinary calls (`fetch` , `search` , ...) work again. Pair every `idle` with a `done` .

Parameters

- `session {Session}` - the idling session

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.expunge(session as Session)

Permanently remove every message flagged "\\Deleted" in the selected folder (EXPUNGE). Sequence numbers shift as messages are removed, so mark all the target messages with `addFlags(..., "\\Deleted")` first and call this once.

Parameters

- `session {Session}` - the open session

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.fetch(session as Session, uid as int)

Retrieve a message (its full body) as a raw string for `mime.parse` , addressed by its stable UID (`UID FETCH ... BODY.PEEK[]`). Get a UID from `search` .

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID

Returns `{string}` - the raw message body

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.fetchAll(opts as Options, folder as string)

Connect, select `folder` , retrieve every message, and log out.

Parameters

- `opts {Options}` - the connection and auth parameters
- `folder {string}` - the folder name

Returns `{list of string}` - the raw body of every message

Throws

- `{Error}` - on a bad greeting or a "NO" / "BAD" completion (kind "imap")

imap.fetchHeaders(session as Session, uid as int, fields as string)

Retrieve only the named header fields of a message (space-separated, e.g. "SUBJECT DATE") as a raw header block for `mime.parse` - far cheaper than fetching the whole body when you only need a few headers. Addressed by UID.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID
- `fields {string}` - the space-separated header names, e.g. "SUBJECT DATE"

Returns `{string}` - the raw header block (the fields plus the terminating blank line)

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.fetchMessage(session as Session, uid as int)

Fetch a message by UID and parse it into a `mime.Part` tree, ready for `mime.attachments` /

`mime.textBodies` / `mime.data` . Convenience for the common `mime.parse(imap.fetch(...))` pattern; import `mime` too to walk the result.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID

Returns `{mime.Part}` - the parsed message tree

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.fetchPartial(session as Session, uid as int, offset as int, length as int)

Retrieve a **byte range** of a message's body (`UID FETCH ... BODY.PEEK[]<offset.length>`), so a large message can be pulled in bounded chunks instead of one huge literal. `offset` is the 0-based start, `length` the number of octets (the server returns fewer at end-of-body). Both must be >= 0.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID
- `offset {int}` - the 0-based byte offset into the body
- `length {int}` - the number of octets to retrieve

Returns `{string}` - the requested byte range of the body

Throws

- `{Error}` - kind "imap" on a bad range or a "NO" / "BAD" completion

imap.flags(session as Session, uid as int)

Return the flags currently set on a message (by UID) as a space-separated string (e.g. "\\Seen $cl_1"), or "" when none. Useful to confirm a STORE actually persisted: a server that does not allow a custom keyword answers OK but drops it, so the keyword will be absent here.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID

Returns `{string}` - the flags, space-separated (no surrounding parentheses)

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.folders(session as Session, pattern as string)

List the folders matching `pattern` (the IMAP `LIST "" pattern`). Use `"*"` for every folder at any depth, `"%"` for the top level only, or scope to a subtree with a prefix (all of Archive is the pattern `Archive/` then `*`). The reference is empty, so put any prefix in the pattern. (Named `folders` , not `list` , since `list` is a reserved type keyword.)

Parameters

- `session {Session}` - the open session
- `pattern {string}` - the folder pattern (* = any depth, % = one level)

Returns `{list of Folder}` - the matching folders (name, delimiter, flags)

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.idle(session as Session)

Enter IDLE (RFC 2177): send `IDLE` and wait for the server's "+ idling" continuation, leaving the session in the idling state so the server pushes mailbox changes (call `supportsIdle` first - a server that lacks the extension answers with a tagged "NO"/"BAD", surfaced here as an `Error`). Follow with `receiveNotification` / `pollNotification` to read pushes, then `done` to return to command mode. A server drops an idle session after about 29 minutes, so a long-lived client must periodically `done` and `idle` again.

Parameters

- `session {Session}` - the open session

Throws

- `{Error}` - kind "imap" when the server does not enter IDLE (e.g. a tagged "NO"/"BAD")

imap.logout(session as Session)

End the session and close the connection.

Parameters

- `session {Session}` - the open session

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.move(session as Session, uid as int, folder as string)

Move a message (by UID) into another folder in one atomic step (`UID MOVE` , RFC 6851) - the server copies then removes and expunges the source, so unlike the copy + `\\Deleted` + expunge dance there is no window with a duplicate and no manual expunge. The destination folder must already exist. `MOVE` is widely but not universally supported; a server lacking it answers "BAD", so fall back to `copy` + `addFlags(..., "\\Deleted")` + `expunge` when catching that.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID
- `folder {string}` - the destination folder name

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap"), including an unsupported MOVE

imap.pollNotification(session as Session, timeoutMs as int)

Like `receiveNotification` , but wait at most `timeoutMs` milliseconds (armed with `net.setDeadline`): return the next push if one arrives in time, else the empty sentinel (`kind == ""`) so a poll loop can do other work between checks. Requires an `idle` first.

Parameters

- `session {Session}` - the idling session
- `timeoutMs {int}` - the maximum time to wait, in milliseconds

Returns `{Notification}` - the next push, or the empty sentinel on a timeout

Throws

- `{Error}` - kind "imap" on a read failure other than a timeout

imap.receiveNotification(session as Session)

Block until the next server push arrives while idling and return it as a typed `Notification` ("exists" new-mail /

"expunge" / "recent"). Returns the empty sentinel (`kind == ""`) only if the peer closes or the server ends IDLE. No callbacks - wrap this in a loop, and in a `spawn` to run it beside other work. Requires an `idle` first.

Parameters

- `session {Session}` - the idling session

Returns `{Notification}` - the next push, or the empty sentinel when IDLE ends

Throws

- `{Error}` - kind "imap" on a read failure other than a timeout

imap.removeFlags(session as Session, uid as int, flags as string)

Remove IMAP flags / keywords from a message (`UID STORE -FLAGS.SILENT`), the inverse of addFlags. Removing a flag that is not set is a harmless no-op, e.g. removeFlags(session, uid, "$cl_1") clears that tag, "\\Deleted" un-marks a pending delete.

Parameters

- `session {Session}` - the open session
- `uid {int}` - the message UID
- `flags {string}` - the space-separated flags to clear, e.g. "$cl_1"

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.search(session as Session, criteria as Criteria)

Return the **UIDs** of the messages in the selected folder matching `criteria` (`UID SEARCH`). A UID is stable across an expunge - it keeps addressing the same message after others are removed - so the returned ids are the correct input to every other verb here and the basis for "fetch only what is new since last run". The server-side fields become one round-trip (no bodies); if any client-side condition is set - a `subjectRegex` / `fromRegex` , `hasAttachments` , or a `since` / `before` carrying a time-of-day - the candidates are then refined by fetching just their headers / structure / INTERNALDATE. An empty `imap.criteria()` returns every message (`UID SEARCH ALL`).

Parameters

- `session {Session}` - the open session
- `criteria {Criteria}` - the filter (build with `imap.criteria()` + fields)

Returns `{list of int}` - the matching message UIDs

Throws

- `{Error}` - kind "imap" on a "NO" / "BAD" completion, an inverted date range (`since` after `before`), or a control character in a substring field

imap.selectFolder(session as Session, name as string)

Select a folder (e.g. "INBOX") and return its message count.

Parameters

- `session {Session}` - the open session
- `name {string}` - the folder name

Returns `{int}` - the number of messages in the folder

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.status(session as Session, folder as string)

Query a folder's counts without selecting it (STATUS) - message / unseen / recent totals plus UIDNEXT / UIDVALIDITY. Handy for a folder badge or a "new mail?" poll on a folder other than the selected one.

Parameters

- `session {Session}` - the open session
- `folder {string}` - the folder name

Returns `{Status}` - the folder counts

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

imap.supportsIdle(session as Session)

Report whether the server advertises the IDLE extension (RFC 2177) in its `CAPABILITY` reply. Call before `idle` : a server without IDLE answers the `IDLE` command with a tagged "NO"/"BAD", which `idle` surfaces as an `Error` .

Parameters

- `session {Session}` - the open session

Returns `{bool}` - true when IDLE is advertised

Throws

- `{Error}` - on a "NO" / "BAD" completion (kind "imap")

Structs

imap.Criteria

A message-search filter. Every field is optional; a zero-value `Criteria` (from `imap.criteria()`) matches all messages, so `search($s, imap.criteria())` is the old `SEARCH ALL` . Fields split into two groups by where they run:

Field	Type	Description
subject	string	SUBJECT substring ("" ignores it)
from	string	FROM substring
to	string	TO substring
text	string	TEXT substring (whole message)
since	time.Time	only messages on/after this instant (a zero time ignores it)
before	time.Time	only messages strictly before this instant (a zero time ignores it)
seen	bool	only \Seen messages
unseen	bool	only unseen messages
flagged	bool	only \Flagged messages
answered	bool	only \Answered messages
largerThan	int	LARGER than n bytes (0 ignores it)
smallerThan	int	SMALLER than n bytes (0 ignores it)
subjectRegex	string	RE2 pattern on the Subject header (client-side, "" ignores it)
fromRegex	string	RE2 pattern on the From header (client-side)

hasAttachments	bool	keep only messages with an attachment (client-side heuristic)

imap.Folder

One folder as reported by `list` .

Field	Type	Description
name	string	the folder name (ASCII, or modified-UTF-7 for non-ASCII names)
delimiter	string	the hierarchy separator (e.g. "/" or "."), "" for a flat namespace
flags	list of string	the folder attribute flags (e.g. "\HasChildren", "\Noselect")

imap.Notification

One server-pushed mailbox change delivered during IDLE (RFC 2177). A `kind` of "" is the idle-gap sentinel (`imap.pollNotification` timed out, or IDLE ended) carrying no push.

Field	Type	Description
kind	string	the push kind: "exists" (new message count), "expunge" (a message was removed), "recent" (recent-count change), or "" (no push / idle gap)
number	int	the sequence number or count the push carried (0 for the sentinel)

imap.Options

The parameters for opening an IMAP session.

Field	Type	Description
host	string	the server hostname
port	int	the server port (e.g. 993 for implicit TLS)
security	transport.Security	the transport: transport.Security.None (plaintext) / .Tls (implicit) / .Starttls
user	string	the login username
pass	string	the login password, or the OAuth2 access token when auth is "xoauth2"
auth	string	the auth mechanism: "" (default - LOGIN), "auto" (probe CAPABILITY and pick the strongest mechanism, falling back to LOGIN), "xoauth2", "cram" (CRAM-MD5), "scram-sha-1", or "scram-sha-256"

imap.Session

An open IMAP session.

Field	Type	Description
conn	net.Conn	the underlying connection

imap.Status

Folder status counts, as reported by `status` . A field the server did not return stays 0.

Field	Type	Description
messages	int	total messages (MESSAGES)
recent	int	messages with the \Recent flag (RECENT)
unseen	int	messages without \Seen (UNSEEN)
uidnext	int	the UID that will be assigned to the next message (UIDNEXT)
uidvalidity	int	the folder's UID-validity value (UIDVALIDITY)

influxdb API reference

An InfluxDB client over `http` for both major API generations: write measurements as line-protocol points and run queries, getting back a parsed result. The push counterpart to a scrape.

Two selectable backends under one `Client` (stance 1: one module, one version discriminator). A **1.x** client (`client` / `clientWith`) writes to `/write?db=...` with optional Basic auth and queries InfluxQL over `/query` (`query`). A **2.x / 3.x** client (`client2`) writes to `/api/v2/write?org=...&bucket=...` with `Authorization: Token <token>` auth and queries Flux over `/api/v2/query` (`queryFlux`). `write` dispatches on the client's `version` , so the same call serves both; the line protocol is identical across versions, so `line` is reused verbatim.

A `Point` is built with value-semantic builders (`point` / `tag` / `field` / `intField` / `stringField` / `boolField` / `at`), each returning a fresh `Point` , then rendered to a line-protocol line by `line` (or sent by `write`). Field types are carried as pre-rendered fragments, so one point can mix float / int / string / bool fields despite Jennifer's homogeneous maps. Needs the default `jennifer` binary (`http` over `net`); a failed request throws `Error{kind: "influxdb"}` (a 2.x client's token is redacted from it).

Import with `import "influxdb.j" as influxdb;` . See the `influxdb` guide for prose and examples.

Functions

influxdb.at(p as Point, unixNanos as int)

Set an explicit timestamp in nanoseconds since the Unix epoch. Returns a fresh point.

Parameters

- `p` `{Point}` - the point
- `unixNanos` `{int}` - the timestamp in nanoseconds

Returns `{Point}` - a timestamped point

influxdb.atTime(p as Point, t as time.Time)

Set the timestamp from a `time.Time` . Returns a fresh point.

Parameters

- `p` `{Point}` - the point
- `t` `{time.Time}` - the timestamp

Returns `{Point}` - a timestamped point

influxdb.boolField(p as Point, key as string, value as bool)

Add a boolean field. Returns a fresh point.

Parameters

- `p` `{Point}` - the point
- `key` `{string}` - the field key
- `value` `{bool}` - the field value

Returns `{Point}` - a point with the field added

influxdb.client(url as string, db as string)

Open a client to a database with no authentication.

Parameters

- `url` `{string}` - the base URL (e.g. "http://localhost:8086")

- `db` `{string}` - the database name

Returns `{Client}` - a ready client

influxdb.client2(url as string, org as string, bucket as string, token as string)

Open an InfluxDB 2.x / 3.x client: token auth, addressing data by organization + bucket. `write` posts to `/api/v2/write` and `queryFlux` runs Flux over `/api/v2/query` .

Parameters

- `url` `{string}` - the base URL (e.g. "http://localhost:8086")
- `org` `{string}` - the organization
- `bucket` `{string}` - the target bucket
- `token` `{string}` - the API token (sent as `Authorization: Token <token>`)

Returns `{Client}` - a ready 2.x client

influxdb.clientWith(url as string, db as string, user as string, password as string)

Open a 1.x client with HTTP Basic-auth credentials.

Parameters

- `url` `{string}` - the base URL
- `db` `{string}` - the database name
- `user` `{string}` - the username
- `password` `{string}` - the password

Returns `{Client}` - a ready client

influxdb.field(p as Point, key as string, value as float)

Add a float field. Returns a fresh point.

Parameters

- `p` `{Point}` - the point
- `key` `{string}` - the field key
- `value` `{float}` - the field value

Returns `{Point}` - a point with the field added

influxdb.intField(p as Point, key as string, value as int)

Add an integer field (rendered with the line-protocol `i` suffix). Returns a fresh point.

Parameters

- `p` `{Point}` - the point
- `key` `{string}` - the field key
- `value` `{int}` - the field value

Returns `{Point}` - a point with the field added

influxdb.line(p as Point)

Render a point to one line-protocol line.

Parameters

- `p` `{Point}` - the point

Returns `{string}` - the line-protocol line

Throws

- `{Error}` - kind "influxdb" if the point has no fields

influxdb.point(measurement as string)

Start a point for a measurement (no tags or fields yet).

Parameters

- `measurement` `{string}` - the measurement name

Returns `{Point}` - a fresh point

influxdb.query(c as Client, influxql as string)

Run an InfluxQL statement against a 1.x client's database and parse the result. (The 2.x query language is Flux - see `queryFlux` .)

Parameters

- `c` `{Client}` - the client
- `influxql` `{string}` - the InfluxQL statement (e.g. `SELECT * FROM cpu`)

Returns `{Result}` - the parsed series

Throws

- `{Error}` - kind "influxdb" on a request failure or a query error

influxdb.queryFlux(c as Client, flux as string)

Run a Flux query against a 2.x client's organization and return the raw response body: InfluxDB answers a Flux query with annotated CSV (RFC 4180 with `#datatype` / `#group` / `#default` header rows), not JSON, so this returns it verbatim - parse it with the `csv` module. Posts the script to `/api/v2/query?org=...` under `application/vnd.flux` .

Parameters

- `c` `{Client}` - a 2.x client (from `client2`)
- `flux` `{string}` - the Flux script

Returns `{string}` - the annotated-CSV response body

Throws

- `{Error}` - kind "influxdb" on a non-2xx response (token redacted)

influxdb.stringField(p as Point, key as string, value as string)

Add a string field (double-quoted, escaped). Returns a fresh point.

Parameters

- `p` `{Point}` - the point
- `key` `{string}` - the field key
- `value` `{string}` - the field value

Returns `{Point}` - a point with the field added

influxdb.tag(p as Point, key as string, value as string)

Add a tag (indexed string metadata). Returns a fresh point.

Parameters

- `p` `{Point}` - the point
- `key` `{string}` - the tag key
- `value` `{string}` - the tag value

Returns `{Point}` - a point with the tag added

influxdb.write(c as Client, points as list of Point)

Write points to the client's target (line protocol, nanosecond precision). Dispatches on the client's `version` : a 1.x client posts to `/write?db=...` , a 2.x client to `/api/v2/write?org=...&bucket=...` .

Parameters

- `c` `{Client}` - the client
- `points` `{list of Point}` - the points to write

Throws

- `{Error}` - kind "influxdb" on a non-2xx response (2.x token redacted)

Structs

influxdb.Client

A client for either API generation. A 1.x client carries `db` and optional Basic-auth `user` / `password` ; a 2.x client carries `org` / `bucket` and a `token` . `version` selects which set is used (unused fields stay "").

Field	Type	Description
url	string	the base URL (e.g. "http://localhost:8086")
version	Version	the API generation (V1 or V2)
db	string	the 1.x database name ("" for a 2.x client)
user	string	the 1.x Basic-auth username ("" for no auth)
password	string	the 1.x Basic-auth password
org	string	the 2.x organization ("" for a 1.x client)
bucket	string	the 2.x bucket ("" for a 1.x client)
token	string	the 2.x API token ("" for a 1.x client)

influxdb.Point

A line-protocol point under construction. Tags and fields are held as pre-rendered, escaped `key=value` fragments so a point can mix field types.

Field	Type	Description
measurement	string	the measurement name
tags	list of string	escaped key=value tag fragments
fields	list of string	rendered key=value field fragments
timestamp	int	the timestamp in nanoseconds (when timed)
timed	bool	whether an explicit timestamp is set

influxdb.Result

A parsed query result: the flattened series across every statement.

Field	Type	Description
series	list of Series	the result series

influxdb.Series

One result series: its measurement name, its GROUP BY tag set, the column names, and the rows (each cell stringified).

Field	Type	Description
name	string	the measurement name
tags	map of string to string	the series tag set ({} if none)
columns	list of string	the column names (e.g. ["time", "value"])
values	list of list of string	the rows, one stringified cell per column

Enums

influxdb.Version

The API generation a `Client` targets: `influxdb.Version.V1` (InfluxDB 1.x - `/write?db=...` + InfluxQL, optional Basic auth) or `influxdb.Version.V2` (InfluxDB 2.x / 3.x - `/api/v2/write?org=...&bucket=...` + Flux, token auth). The backend selector `write` dispatches on; the zero value is `V1` .

ipnet API reference

IP addresses and CIDR networks, IPv4 and IPv6. Parse an address or a `address/prefix` block, test membership, and compute netmask / broadcast - for allow-lists and subnet math. An `Address` holds its raw bytes (4 for IPv4, 16 for IPv6, network byte order); a `Network` pairs a base address with a prefix length. Pure Jennifer over `strings` + `convert` and the bitwise operators; both binaries. `parseAddress` folds an IPv4-mapped IPv6 literal (`::ffff:a.b.c.d`) down to a v4 `Address` so it can't slip past a v4 allow-list (see `unmap`).

Import with `import "ipnet.j" as ipnet;` . See the `ipnet` guide for prose and examples.

Functions

ipnet.aggregate(nets as list of Network)

Collapse a list of networks into the minimal equivalent set: contained blocks are dropped and adjacent sibling pairs are merged into their shorter parent (e.g. 192.168.0.0/25 + 192.168.0.128/25 -> 192.168.0.0/24). IPv4 and IPv6 entries are aggregated independently; the result is sorted ascending.

Parameters

- `nets {list of Network}` - the networks to aggregate

Returns `{list of Network}` - the minimal covering set

ipnet.broadcast(net as Network)

The broadcast (last) address of a network - every host bit set. For IPv4 this is the broadcast address; for IPv6 it is the last address in the block.

Parameters

- `net {Network}` - the network

Returns `{Address}` - the last address in the network

ipnet.compare(a as Address, b as Address)

Order two addresses: -1 if a < b , 0 if equal, 1 if a > b . Different versions order v4 before v6 (a stable, if arbitrary, total order for sorting).

Parameters

- a `{Address}` - the first address
- b `{Address}` - the second address

Returns `{int}` - -1, 0, or 1

ipnet.contains(net as Network, addr as Address)

Whether an address falls within a network (same version and matching prefix bits). A version mismatch is simply false.

Parameters

- `net {Network}` - the network
- `addr {Address}` - the address to test

Returns `{bool}` - true if the address is in the network

ipnet.equal(a as Address, b as Address)

Whether two addresses are equal (same version and bytes).

Parameters

- a `{Address}` - the first address
- b `{Address}` - the second address

Returns `{bool}` - true if equal

ipnet.firstUsable(net as Network)

The first usable host address of a network. For IPv4 this is the address after the network base (the base is the network address), except a /31 (RFC 3021 point-to-point, both addresses usable) and a /32 (single host), where it is the base itself. For IPv6 it is the base + 1 (skipping the subnet-router anycast), except /127 and /128 where it is the base.

Parameters

- `net {Network}` - the network

Returns `{Address}` - the first usable address

ipnet.hostCount(net as Network)

The total number of addresses in a network (2 ^ host-bits). Throws when the block is too large to hold in an `int` (an IPv4 prefix is always fine; an IPv6 prefix must be >= 66). For usable-host semantics see `firstUsable` / `lastUsable` .

Parameters

- `net {Network}` - the network

Returns `{int}` - the count of addresses in the block

Throws

- `{Error}` - kind "ipnet" when the block exceeds the int range

ipnet.hosts(net as Network)

Every usable host address of a network, in ascending order (from `firstUsable` to `lastUsable` inclusive). Capped at 65536 addresses - a larger block throws, so materializing a whole /8 cannot exhaust memory; walk those with `next` instead.

Parameters

- `net {Network}` - the network

Returns `{list of Address}` - the usable host addresses

Throws

- `{Error}` - kind "ipnet" when the block has more than 65536 addresses

ipnet.isGlobal(addr as Address)

Whether an address is a normal globally-routable unicast address - i.e. its `scope` is `Global` (not private, loopback, link-local, multicast, unspecified, or a reserved / documentation range). Best-effort, per the well-known special-purpose registries.

Parameters

- `addr {Address}` - the address

Returns `{bool}` - true if globally routable

ipnet.isLinkLocal(addr as Address)

Whether an address is link-local (169.254.0.0/16 or fe80::/10).

Parameters

- `addr {Address}` - the address

Returns `{bool}` - true if link-local

ipnet.isLoopback(addr as Address)

Whether an address is a loopback address (127.0.0.0/8 or ::1).

Parameters

- `addr {Address}` - the address

Returns `{bool}` - true if loopback

ipnet.isMulticast(addr as Address)

Whether an address is multicast (224.0.0.0/4 or ff00::/8).

Parameters

- `addr {Address}` - the address

Returns `{bool}` - true if multicast

ipnet.isPrivate(addr as Address)

Whether an address is in private (RFC 1918) or IPv6 unique-local (fc00::/7) space.

Parameters

- `addr {Address}` - the address

Returns `{bool}` - true if private / ULA

ipnet.isUnspecified(addr as Address)

Whether an address is the unspecified address (0.0.0.0 or ::).

Parameters

- `addr {Address}` - the address

Returns `{bool}` - true if unspecified

ipnet.lastUsable(net as Network)

The last usable host address of a network. For IPv4 this is the address before the broadcast address, except a /31 or /32 (where the last address is usable). IPv6 has no broadcast address, so the last address of the block is usable.

Parameters

- `net {Network}` - the network

Returns `{Address}` - the last usable address

ipnet.netmask(net as Network)

The netmask of a network as an address (e.g. 255.255.255.0 for a /24).

Parameters

- `net {Network}` - the network

Returns `{Address}` - the netmask address

ipnet.networkString(net as Network)

Render a network as `address/prefix` .

Parameters

- `net {Network}` - the network

Returns `{string}` - the CIDR text

ipnet.next(addr as Address)

The next address after `addr` (numerically +1). Throws at the last address of the version (255.255.255.255 / all-ones IPv6) - there is no successor.

Parameters

- `addr {Address}` - the address

Returns `{Address}` - the following address

Throws

- `{Error}` - kind "ipnet" at the last address

ipnet.overlaps(a as Network, b as Network)

Whether two networks share any address. Different versions never overlap.

Parameters

- a `{Network}` - the first network
- b `{Network}` - the second network

Returns `{bool}` - true if the blocks intersect

ipnet.parse(cidr as string)

Parse a CIDR block `address/prefix` into a `Network` with host bits zeroed. The prefix range is taken from the **literal** , so an IPv6 v4-mapped block (`::ffff:0:0/96`) is accepted as a /96 and then folded to its v4 equivalent (`0.0.0.0/0` , i.e. the prefix is translated down by the 96-bit v4-mapped offset) - mirroring how `parseAddress` folds a v4-mapped address. A block with a prefix shorter than 96 stays a genuine v6 network (it spans beyond the v4-mapped range).

Parameters

- `cidr {string}` - the CIDR text (e.g. "10.0.0.0/8" or "2001:db8::/32")

Returns `{Network}` - the network

Throws

- `{Error}` - kind "ipnet" on malformed input or an out-of-range prefix

ipnet.parseAddress(s as string)

Parse a bare IP address (IPv4 dotted-quad or IPv6, with `::` compression and embedded IPv4 supported).

Parameters

- `s {string}` - the address text

Returns `{Address}` - the parsed address

Throws

- `{Error}` - kind "ipnet" on malformed input

ipnet.prev(addr as Address)

The previous address before `addr` (numerically -1). Throws at the first address (0.0.0.0 / ::).

Parameters

- `addr {Address}` - the address

Returns `{Address}` - the preceding address

Throws

- `{Error}` - kind "ipnet" at the first address

ipnet.scope(addr as Address)

Classify an address into its `Scope` . A v4-mapped IPv6 address is folded to v4 first (via `unmap`), so `::ffff:10.0.0.1` classifies as `Private` .

Parameters

- `addr {Address}` - the address

Returns `{Scope}` - the address's scope

ipnet.split(net as Network, newPrefix as int)

Divide a network into the equal subnets of a longer prefix (e.g. a /24 into four /26s). Returns them in ascending order. Capped at 65536 subnets (a `newPrefix` no more than 16 bits longer than the network prefix).

Parameters

- `net {Network}` - the network to split
- `newPrefix {int}` - the subnet prefix length (>= net.prefix, <= version max)

Returns `{list of Network}` - the subnets

Throws

- `{Error}` - kind "ipnet" on a bad prefix or more than 65536 subnets

ipnet.subnetOf(child as Network, parent as Network)

Whether `child` is wholly contained in `parent` (a subnet of it, or equal). Different versions are never a subnet.

Parameters

- `child {Network}` - the candidate subnet
- `parent {Network}` - the enclosing network

Returns `{bool}` - true if child lies inside parent

ipnet.toString(addr as Address)

Render an address to its canonical string (IPv4 dotted-quad, or RFC 5952 canonical IPv6).

Parameters

- `addr {Address}` - the address

Returns `{string}` - the canonical text

ipnet.unmap(addr as Address)

Fold an IPv4-mapped IPv6 address (`::ffff:a.b.c.d` , the `::ffff:0:0/96` range) down to the plain IPv4 `Address` it represents; any other address is returned unchanged. Without this, `::ffff:127.0.0.1` stays a version-6 `Address` and silently misses a version-4 network in a `contains` allow-list / deny-list check - a bypass of exactly the kind the leading-zero and embedded-IPv4 guards already close (OM-010). `parseAddress` applies it, so a v4-mapped literal parses straight to a v4 `Address` . The deprecated IPv4-compatible form (`::a.b.c.d`) is intentionally left alone: it is ambiguous with low addresses like `::1` and unmapping it would be wrong.

Parameters

- `addr {Address}` - the address to normalize

Returns `{Address}` - a v4 Address if `addr` was v4-mapped, else `addr` unchanged

ipnet.version(addr as Address)

The IP version of an address (4 or 6).

Parameters

- `addr {Address}` - the address

Returns `{int}` - 4 or 6

Structs

ipnet.Address

An IP address as raw bytes.

Field	Type	Description
version	int	the IP version: 4 or 6
octets	bytes	the address bytes (4 for IPv4, 16 for IPv6), network byte order

ipnet.Network

A CIDR network: a base address (host bits zeroed) plus a prefix length.

Field	Type	Description
addr	Address	the network base address
prefix	int	the prefix length (0..32 for IPv4, 0..128 for IPv6)

Enums

ipnet.Scope

The address scope: which well-known category an address falls in. A total, disjoint classification - every address is exactly one `Scope` - so the `is*` predicates are thin wrappers over it and a caller can `match` on the whole set. `Reserved` covers the special-purpose ranges that are neither a normal global unicast address nor one of the named categories (documentation, benchmarking, shared CGNAT space, and other reserved blocks).

jsonl API reference

JSON Lines (JSONL / NDJSON): newline-delimited JSON, one independent value per line. A thin framing layer over `json` - each record is a `json.Value` , so `encode` / `decode` compose `json.encode` / `json.decode` with a `\n` split / join, and the file helpers add `fs` . Pure Jennifer; both binaries.

`encode(records)` writes one compact JSON value per line (trailing newline); `decode(text)` parses each non-blank line back into a `json.Value` , so `decode(encode(records))` round-trips. Whole-file `readFile` / `writeFile` / `appendFile` cover the common case; a streaming `Reader` reads one record at a time for files too large to hold in memory.

Import with `import "jsonl.j" as jsonl;` . See the `jsonl` guide for prose and examples.

Functions

jsonl.appendFile(path as string, records as list of json.Value)

Encode records and append them to a file (created if missing) - the common JSONL pattern of adding rows to a growing log.

Parameters

- `path` `{string}` - the file path
- `records` `{list of json.Value}` - the records to append

jsonl.closeReader(reader as Reader)

Close a streaming reader.

Parameters

- `reader` `{Reader}` - the reader

jsonl.closeWriter(writer as Writer)

Close a streaming writer (closes the underlying file handle).

Parameters

- `writer` `{Writer}` - the writer

jsonl.decode(text as string)

Decode JSONL text into records, one `json.Value` per non-blank line. Blank and whitespace-only lines are skipped, and a trailing `\r` (CRLF input) is trimmed, so `decode(encode(records))` round-trips.

Parameters

- `text` `{string}` - the JSONL text

Returns `{list of json.Value}` - the parsed records

jsonl.encode(records as list of json.Value)

Encode records as JSONL: one compact JSON value per line, each terminated by a newline. An empty list yields the empty string.

Parameters

- `records` `{list of json.Value}` - the records to encode

Returns `{string}` - the JSONL text

jsonl.hasMore(reader as Reader)

Whether the reader is not yet at end-of-file. This is a coarse check: it can report `true` when only trailing blank lines remain, which carry no record. Prefer looping on `readRecord(...).done` - the reliable end signal.

Parameters

- `reader {Reader}` - the reader

Returns `{bool}` - true if the file still has unread bytes

jsonl.openReader(path as string)

Open a JSONL file for streaming.

Parameters

- `path {string}` - the file path

Returns `{Reader}` - the reader

jsonl.readFile(path as string)

Read and decode a whole JSONL file.

Parameters

- `path {string}` - the file path

Returns `{list of json.Value}` - the records

jsonl.readRecord(reader as Reader)

Read and decode the next record, skipping blank lines. Returns a `Record` : `{value, done: false}` for a record, or `{done: true}` once the stream is exhausted (including when only trailing blank lines remained). Loop until `done` rather than guarding with `hasMore` , which cannot see trailing blanks.

Parameters

- `reader {Reader}` - the reader

Returns `{Record}` - the next record, or a `done` marker at end

jsonl.readValue(reader as Reader)

Read and decode the next record, skipping blank lines, and return the `json.Value` directly. At end-of-stream this returns a JSON `null` ; guard the loop with `not fs.eof(...)` (JSONL written by `writeValue` has one value per line and no trailing blanks) or use `readRecord` when you need the explicit `done` flag to distinguish end-of-stream from a genuine `null` record.

Parameters

- `reader {Reader}` - the reader

Returns `{json.Value}` - the next record, or a JSON `null` at end

jsonl.reader(file as fs.File)

Wrap an already-open, read-mode `fs.File` as a streaming reader. Use this when the caller owns the file handle (openReader opens the path for you). The file is a handle, so successive `readValue` calls advance the same stream.

Parameters

- `file {fs.File}` - an open read-mode file handle

Returns `{Reader}` - the reader

jsonl.writeFile(path as string, records as list of json.Value)

Encode records and write them to a file, replacing any existing content.

Parameters

- `path` `{string}` - the file path
- `records` `{list of json.Value}` - the records to write

jsonl.writeValue(writer as Writer, value as json.Value)

Append one record to the writer: the compact JSON encoding followed by a newline.

Parameters

- `writer` `{Writer}` - the writer
- `value` `{json.Value}` - the record to append

jsonl.writer(file as fs.File)

Wrap an open write/append-mode `fs.File` as a streaming JSONL writer.

Parameters

- `file` `{fs.File}` - an open write- or append-mode file handle

Returns `{Writer}` - the writer

Structs

jsonl.Reader

A line-at-a-time reader over an open file, for JSONL too large to hold in memory. Build with `openReader` ; the wrapped `fs.File` shares its read position across value copies (a handle), so successive `readRecord` calls advance the same stream.

Field	Type	Description
file	fs.File	the underlying open file handle

jsonl.Record

One streaming read: a decoded record, or `done` set once the stream is exhausted. `done` is the reliable end signal - a raw `not eof` check is not, because trailing blank lines leave the file un-exhausted yet carry no record.

Field	Type	Description
value	json.Value	the decoded record (undefined when done)
done	bool	true once no further record remains

jsonl.Writer

A streaming JSONL writer over an open `fs.File` . Each `writeValue` call appends one compact JSON value terminated by a newline, so a growing log is written without building the whole text in memory.

Field	Type	Description
file	fs.File	the underlying open write/append-mode file handle

jsonrpc API reference

JSON-RPC 2.0 (https://www.jsonrpc.org/specification) over HTTP: a **client** that calls remote methods, and a transport-agnostic **server** `handle` that dispatches an incoming request to the entry program's methods **by name** , via `meta.callMain` - the method name arrives on the wire, so it is resolved as a runtime string, not a `func` value. Built on `json` for the wire format and `http` for the client transport, so it needs the default `jennifer` binary.

`params` and a call's result are `json.Value` s: the caller builds params with the `json` write API (`json.list` / `json.map` + `json.append` / `json.set`) and reads the result with the `json` accessors. Any client-side failure - a JSON-RPC error reply, a transport error (connection refused / timeout), or a malformed reply - surfaces as a single catchable `Error{kind: "jsonrpc"}` .

Server security. `handle` dispatches each request's `method` to a *top-level* `func` of that name in the entry program, so **every** top-level method taking one `json.Value` parameter is remotely reachable - there is no separate route registry. Name RPC handlers deliberately (a shared prefix, a dedicated dispatch file) and do not co-locate a `handle` -served program with privileged one-argument helpers. Authentication is the transport's job: gate on a header / token before calling `handle` . A handler that throws yields a generic `-32603` reply (the thrown message is *not* put on the wire), so raise errors freely without leaking internals.

Import with `import "jsonrpc.j" as jsonrpc;` . See the `jsonrpc` guide for prose and examples.

Functions

jsonrpc.call(client as Client, method as string, params as json.Value)

Call a remote method and return its result. Any failure - a JSON-RPC error reply, a transport error (connection refused / timeout / malformed HTTP), or a malformed reply (no `result` , no `error` , or an id that does not match the request) - throws a catchable `Error{kind: "jsonrpc"}` .

Parameters

- `client {Client}` - the endpoint client
- `method {string}` - the method name
- `params {json.Value}` - the params (a `json.list` array or a `json.map` object; pass `json.list()` for a method that takes no arguments)

Returns `{json.Value}` - the `result` member of the reply

jsonrpc.client(endpoint as string)

Build a client for a JSON-RPC endpoint.

Parameters

- `endpoint {string}` - the endpoint URL

Returns `{Client}` - a configured client

jsonrpc.clientWith(endpoint as string, headers as map of string to string)

Build a client with extra request headers (e.g. `Authorization`).

Parameters

- `endpoint {string}` - the endpoint URL
- `headers {map of string to string}` - headers sent with every request

Returns `{Client}` - a configured client

jsonrpc.handle(requestBody as string)

Dispatch a JSON-RPC request body and return the reply body (transport- agnostic: wire it to `httpd` / `net` however you serve). Each request's `method` names a top-level method `func NAME(params as json.Value)` in the program that imported this module; it is called with the request's params and must return a `json.Value` **or a scalar** (int / float / string / bool / null) as its result. A missing method is a -32601 reply, a thrown error a generic -32603 reply. A single request yields a single reply; a notification (no `id`) yields no reply (`""`); a batch (a JSON array) yields an array reply with the notification entries omitted (`""` when the batch is all notifications).

Security. Every top-level one- `json.Value` -argument method is reachable by name (there is no route allow-list); the module-level doc explains how to scope handlers and where authentication belongs. A thrown handler error's message is *not* echoed to the client (only the generic -32603 text is), so internal detail does not leak.

Parameters

- `requestBody` `{string}` - the raw request JSON

Returns `{string}` - the reply JSON, or `""` when no reply is owed

jsonrpc.notify(client as Client, method as string, params as json.Value)

Send a notification: a request with no `id` , for which the server returns no reply. A delivered notification says nothing about whether the method ran, but a transport error (connection refused / timeout) still throws `Error{kind: "jsonrpc"}` - it means the request never reached the server.

Parameters

- `client` `{Client}` - the endpoint client
- `method` `{string}` - the method name
- `params` `{json.Value}` - the params (see `call`)

Structs

jsonrpc.Client

A client bound to a JSON-RPC HTTP endpoint. Value-semantic; build with `client` / `clientWith` .

Field	Type	Description
endpoint	string	the JSON-RPC endpoint URL (http:// or https://)
headers	map of string to string	extra request headers (auth, ...)

jwt API reference

JSON Web Tokens (RFC 7519): sign, verify, and decode compact JWTs. Claims are a `json.Value` object; the token is the usual `header.payload.signature` of base64url segments. Ten algorithms across four families: HMAC (`HS256` / `HS384` / `HS512` , over a shared secret), RSA PKCS#1 v1.5 (`RS256` / `RS384` / `RS512`), ECDSA (`ES256` / `ES384` / `ES512`), and Ed25519 (`EdDSA`). The key is always `bytes` : an HMAC secret, a PEM-encoded RSA / EC key, or a raw Ed25519 key from `crypto.signKeypair` .

`verify` takes the algorithm you *expect* and rejects a token whose header disagrees - this closes the classic JWT algorithm-confusion attack (a forged `HS256` token verified with an RSA public key as the HMAC secret). It also enforces the `exp` (expiry) and `nbf` (not-before) time claims when present.

RS* / ES* need the `crypto` library's RSA / ECDSA surface, which is on the default `jennifer` binary; HS* and `EdDSA` run on both binaries.

Import with `import "jwt.j" as jwt;` . See the `jwt` guide for prose and examples.

Functions

jwt.decode(token as string)

Decode a JWT's claims **without verifying** the signature or time claims - for inspecting a token you do not trust yet (e.g. to read its `kid` / issuer before fetching a key). Never trust these claims for authorization; use `verify` for that.

Parameters

- `token` `{string}` - the compact JWT

Returns `{json.Value}` - the payload claims, unverified

Throws

- `{Error}` - on a malformed token

jwt.header(token as string)

Read a JWT's header (also without verifying) - useful for the `alg` and `kid` fields when selecting a verification key.

Parameters

- `token` `{string}` - the compact JWT

Returns `{json.Value}` - the token header

Throws

- `{Error}` - on a malformed token

jwt.sign(claims as json.Value, key as bytes, alg as string)

Sign `claims` into a compact JWT with the given algorithm. The header is `{"alg": alg, "typ": "JWT"}` ; the payload is the encoded claims. `key` is the HMAC secret, a PEM RSA / EC private key, or an Ed25519 private key, by family.

Parameters

- `claims` `{json.Value}` - the claims object (the token payload)
- `key` `{bytes}` - the signing key (HMAC secret / PEM / Ed25519 private)
- `alg` `{string}` - the JOSE algorithm (e.g. "HS256", "RS256", "ES256", "EdDSA")

Returns `{string}` - the signed `header.payload.signature` token

Throws

- `{Error}` - on an unsupported algorithm or a key the algorithm rejects

jwt.verify(token as string, key as bytes, alg as string)

Verify a JWT and return its claims. Checks that the token's header algorithm equals `alg` (rejecting algorithm-confusion), that the signature is valid for `key` , and - when present - that the `exp` (expiry) and `nbf` (not-before) NumericDate claims allow the token now.

Parameters

- `token` `{string}` - the compact JWT
- `key` `{bytes}` - the verification key (HMAC secret / PEM public / Ed25519 public)
- `alg` `{string}` - the algorithm the caller requires (e.g. "HS256", "RS256")

Returns `{json.Value}` - the verified claims

Throws

- `{Error}` - on a malformed token, an algorithm mismatch, a bad signature, or an expired / not-yet-valid token

jwt.verifyJwks(token as string, jwksJson as string, alg as string)

Verify a JWT against a **JWKS** (a JSON Web Key Set), selecting the key by the header's `kid` . Reads `kid` from the (unverified) header, finds the matching JWK in the set's `keys` array, converts it to a PEM public key with `crypto.jwkToPem` , and verifies. For asymmetric algorithms only (RS* / ES*): a JWKS carries public keys, so an HMAC `alg` is a `jwt` error (use `verifyWithKeys` with the shared secret instead).

`alg` is **required** and enforced against the header, exactly as `verify` / `verifyWithKeys` - selecting the key by `kid` does not select the algorithm, so the algorithm-confusion protection stays. Needs the default `jennifer` binary (`crypto.jwkToPem` is net-of- `crypto/x509` , off the TinyGo build).

Parameters

- `token` `{string}` - the compact JWT
- `jwksJson` `{string}` - the JWKS JSON (`{"keys":[{...}, ...]}`)
- `alg` `{string}` - the algorithm the caller requires (e.g. "RS256", "ES256")

Returns `{json.Value}` - the verified claims

Throws

- `{Error}` - on a missing / unknown `kid` , an HMAC `alg` , a malformed JWK, or any verify failure

jwt.verifyLeeway(token as string, key as bytes, alg as string, leeway as int)

Verify a JWT exactly like `verify` , but widen the `exp` / `nbf` time-claim checks by `leeway` seconds on each side, tolerating a small clock difference between the token's issuer and this verifier. A token is accepted while `now` < `exp` + `leeway` and `now` >= `nbf` - `leeway` ; the signature, the header algorithm, and the `crit` check are enforced exactly as in `verify` . `leeway` must be non-negative. `verify(token, key, alg)` is `verifyLeeway(token, key, alg, 0)` .

Parameters

- `token` `{string}` - the compact JWT
- `key` `{bytes}` - the verification key (HMAC secret / PEM public / Ed25519 public)
- `alg` `{string}` - the algorithm the caller requires (e.g. "HS256", "RS256")
- `leeway` `{int}` - clock-skew tolerance in seconds (must be >= 0)

Returns `{json.Value}` - the verified claims

Throws

- `{Error}` - on a malformed token, an algorithm mismatch, a bad signature, an out-of-window token, or a negative leeway

jwt.verifyWith(token as string, key as bytes, alg as string, policy as Policy)

Verify a JWT like `verify` , then additionally enforce the application-level claim policy: when `policy.iss` is non-empty the token's `iss` must equal it, and when `policy.aud` is non-empty the token's `aud` (a string or an array of strings) must include it. `exp` / `nbf` and the signature are checked exactly as in `verify` ; issuer and audience are application policy the library cannot assume, hence the explicit opt-in.

Parameters

- `token {string}` - the compact JWT
- `key {bytes}` - the verification key (HMAC secret / PEM public / Ed25519 public)
- `alg {string}` - the algorithm the caller requires (e.g. "HS256", "RS256")
- `policy {Policy}` - the expected issuer / audience (empty string skips a check)

Returns `{json.Value}` - the verified claims

Throws

- `{Error}` - on any `verify` failure, or a mismatched issuer / audience

jwt.verifyWithKeys(token as string, keysByKid as map of string to string, alg as string)

Verify a JWT, selecting the verification key by the header's `kid` (key id). Reads `kid` from the (unverified) header, looks it up in `keysByKid` , and verifies with the matching value - a shared secret for HS\ *, or, for RS*\ / ES*, either a PEM public key **or a JWK** (a JSON object with a `"kty"` member, converted to a PEM via `crypto.jwkToPem`). A token with no `kid` header, or a `kid` absent from the map, is a `jwt` error (never a silent skip). The map value is **text** (decoded UTF-8), so it holds an HS* secret, PEM, or JWK - a raw binary Ed25519 key (from `crypto.signKeypair`) is not text-representable, so `EdDSA` is not supported on this kid-based path; pass its key to `verify` directly.

`alg` is **still required** and enforced against the header, exactly as in `verify` : selecting the key by `kid` does not select the algorithm, so this keeps the algorithm-confusion protection (an attacker cannot swap in `HS256` and have your RSA public PEM read as an HMAC secret). Pin `alg` to what the issuer uses.

For a raw **JWKS** (a JSON set of `{"kty","kid","n","e"|"x","y"}` keys), use `verifyJwks` , which resolves the `kid` and converts the JWK to a key via `crypto.jwkToPem` . This `keysByKid` form stays useful for a `kid -> secret` (HMAC) map, or when you have already resolved keys to PEM out of band.

Parameters

- `token {string}` - the compact JWT
- `keysByKid {map of string to string}` - `kid` -> HMAC secret, PEM key text, or a JWK (RS* / ES*)
- `alg {string}` - the algorithm the caller requires (e.g. "HS256", "RS256")

Returns `{json.Value}` - the verified claims

Throws

- `{Error}` - on a missing / unknown `kid` , or any `verify` failure

kvstore API reference

A backend selector for a key/value store with per-key TTL: one `Store` value is one of three backends - a `memcache` connection, a `redis` connection, or the in-process `kv` library - and dispatches a uniform set / get / delete / touch / incrWindow surface to whichever was chosen. This is the shared backend layer under the `session` and `ratelimit` modules, so each works over any of the three without duplicating the dispatch. The verb set mirrors `memcache` , so a fixed-window counter (atomic incr + TTL) works identically on all three.

`Store` is a sum type (enum), not a tagged struct: each variant carries only its own backend handle, and the dispatch is a `match` the compiler checks for exhaustiveness - there is no invalid "kind says memcache but the handle is empty" state, and a new backend cannot be silently forgotten in one verb.

The `memcache` / `redis` backends are distributed (state lives on the server, shared across processes); the **local** backend (the `kv` library) keeps state in this process - either in memory (`inProcessStore` , reset each run) or persisted to a file (`fileStore` , survives across `jennifer run` invocations - handy for development and CLI tools with no server). Pick a distributed backend when more than one process must see the same state.

Import with `import "kvstore.j" as kvstore;` . See the `kvstore` guide for prose and examples.

Functions

kvstore.delete(store as Store, key as string)

Remove `key` ; returns whether it existed.

Parameters

- `store` `{Store}` - the backend store
- `key` `{string}` - the key to remove

Returns `{bool}` - whether the key existed

kvstore.fileStore(path as string)

A store backed by the in-process `kv` library, **persisted to path** : its contents load on open and survive across `jennifer run` invocations (unlike `inProcessStore` , which resets each run). No server; single-process (safe for `spawn` within one process, but not for concurrent separate processes on one file - use a distributed backend for that). Practical for development and persistent CLI state.

Parameters

- `path` `{string}` - the file to persist the store to

Returns `{Store}` - a file-backed local store

kvstore.get(store as Store, key as string)

Return the value at `key` , or "" when absent / expired.

Parameters

- `store` `{Store}` - the backend store
- `key` `{string}` - the key to read

Returns `{string}` - the value, or "" when absent / expired

kvstore.inProcessStore()

A store backed by the in-process `kv` library (this process only, no server). Opens a fresh `kv.Store` ; keep the returned `Store` for the program's lifetime.

Returns `{Store}` - an in-process store

kvstore.incrWindow(store as Store, key as string, ttl as int)

Atomically increment the counter at `key` and return the new value, creating it (at 1, with a `ttl` -second expiry) on the first hit. The atomic-counter-plus-TTL primitive a fixed-window rate limiter is built from - portable across all three backends (redis `INCR` + `EXPIRE` ; memcache / kv `incr` else `add`).

Parameters

- `store` `{Store}` - the store
- `key` `{string}` - the counter key
- `ttl` `{int}` - the TTL to arm on the first hit (seconds)

Returns `{int}` - the counter's new value

kvstore.memcacheStore(mc as memcache.Session)

A store backed by a memcache connection (distributed).

Parameters

- `mc` `{memcache.Session}` - the memcache connection

Returns `{Store}` - a memcache-backed store

kvstore.redisStore(rc as redis.Session)

A store backed by a redis connection (distributed).

Parameters

- `rc` `{redis.Session}` - the redis connection

Returns `{Store}` - a redis-backed store

kvstore.set(store as Store, key as string, value as string, ttl as int)

Store `value` at `key` with a `ttl` -second expiry (0 = no expiry), replacing any existing value.

Parameters

- `store` `{Store}` - the backend store
- `key` `{string}` - the key to write
- `value` `{string}` - the value to store
- `ttl` `{int}` - the expiry in seconds (0 = no expiry)

kvstore.touch(store as Store, key as string, ttl as int)

Re-arm `key` 's expiry to `ttl` seconds; returns whether it existed.

Parameters

- `store` `{Store}` - the backend store
- `key` `{string}` - the key to re-arm
- `ttl` `{int}` - the new expiry in seconds (0 = no expiry)

Returns `{bool}` - whether the key existed

Enums

kvstore.Store

A store bound to one backend (a sum type over the three). Value-semantic; the live connection / handle a variant wraps is shared across copies (a `net.Conn` or a `kv.Store` handle). Build with `memcacheStore` / `redisStore` / `inProcessStore` / `fileStore` .

label API reference

Describe and print labels for industrial label printers. One module, one way to describe a label, with the printer language as a **selectable backend** (a `Device` dialect), not a module per printer. A three-stage pipeline keeps the stages independent: **build** a device-independent `Label` in millimetres, **render** it to a chosen dialect string, then **emit** that string anywhere (a file, a database, or the thin `send` convenience over `net` to a printer's `:9100` raw port). Dialects: `"zpl"` (Zebra Programming Language, raster - needs the target `dpi`) and `"cab"` (cab JScript, millimetre-native); each dialect encoder lives in its own file (`label_zpl.inc.j` / `label_cab.inc.j`) spliced in via `include` , so a new dialect is a new file plus a branch in `render` . Build and render are pure and run on both binaries; only `send` needs the default `jennifer` binary.

Import with `import "label.j" as label;` . See the `label` guide for prose and examples.

Functions

label.barcode(label as Label, x as float, y as float, type as string, opts as BarcodeOptions, data as string)

Place a barcode field, returning a new Label. `type` is a linear symbology - "code128", "ean13", "ean8", "itf" (Interleaved 2 of 5), "code39", "gs1-128" - or a 2D symbology - "datamatrix", "qr". `opts` refines it (size, check digit, 2D error level, human-readable line); pass a zero-value `BarcodeOptions` for the defaults. ITF is numeric-only and even-length (its digits are paired): non-numeric data is rejected and odd-length data is padded with a leading zero (so a 13-digit body becomes ITF-14). GS1-128 data uses the parenthesised Application Identifier form, e.g. `(00)3006...` .

Parameters

- `label` `{Label}` - the label to extend
- `x` `{float}` - the x origin in millimetres
- `y` `{float}` - the y origin in millimetres
- `type` `{string}` - the barcode symbology ("code128"/"ean13"/"ean8"/"itf"/"code39"/"gs1-128"/"datamatrix"/"qr")
- `opts` `{BarcodeOptions}` - size / check-digit / error-level / text refinements
- `data` `{string}` - the barcode data

Returns `{Label}` - a new Label with the barcode field added

Throws

- `{Error}` - kind "label" for an unknown type or invalid ITF data

label.box(label as Label, x as float, y as float, w as float, h as float, thickness as float)

Place a rectangular box (outline), returning a new Label.

Parameters

- `label` `{Label}` - the label to extend
- `x` `{float}` - the x origin in millimetres
- `y` `{float}` - the y origin in millimetres
- `w` `{float}` - the box width in millimetres
- `h` `{float}` - the box height in millimetres
- `thickness` `{float}` - the line thickness in millimetres

Returns `{Label}` - a new Label with the box added

label.cab()

Build a cab JScript render target with default print-setup.

Returns `{Device}` - a cab device

label.cabWith(setup as CabSetup)

Build a cab JScript render target carrying explicit cab print-setup.

Parameters

- `setup` `{CabSetup}` - the cab-only print-setup (job name, heat, orientation, sensor, geometry)

Returns `{Device}` - a cab device

label.image(label as Label, x as float, y as float, name as string)

Place an image referenced by name, returning a new Label. The image must be pre-stored on the printer (cab: the `images/` folder; ZPL: a stored graphic); `name` is the stored name in that dialect's convention. Printed at native size. (Embedding a bitmap in the job is a planned follow-on.)

Parameters

- `label` `{Label}` - the label to extend
- `x` `{float}` - the x origin in millimetres
- `y` `{float}` - the y origin in millimetres
- `name` `{string}` - the stored image name

Returns `{Label}` - a new Label with the image field added

label.new(width as float, height as float)

Start a new, empty label of the given size in millimetres (quantity 1).

Parameters

- `width` `{float}` - the label width in millimetres
- `height` `{float}` - the label height in millimetres

Returns `{Label}` - a fresh, empty label

label.quantity(label as Label, n as int)

Set the number of copies to print, returning a new Label.

Parameters

- `label` `{Label}` - the label to update
- `n` `{int}` - the number of copies

Returns `{Label}` - a new Label with the quantity set

label.render(label as Label, device as Device)

Render a label to a dialect command string.

Parameters

- `label` `{Label}` - the label to render
- `device` `{Device}` - the target dialect (and dpi for raster dialects)

Returns `{string}` - the printer command stream

Throws

- `{Error}` - kind "label" for an unknown dialect

label.send(host as string, port as int, rendered as string)

Send a rendered command stream to a printer's raw `:9100` port over TCP.

Parameters

- `host` `{string}` - the printer host or IP
- `port` `{int}` - the raw print port (usually 9100)
- `rendered` `{string}` - the rendered command stream from `render`

Throws

- `{Error}` - on a network failure (a positioned `net` error)

label.text(label as Label, x as float, y as float, opts as TextOptions, content as string)

Place a text field, returning a new Label.

Parameters

- `label` `{Label}` - the label to extend
- `x` `{float}` - the x origin in millimetres
- `y` `{float}` - the y origin in millimetres
- `opts` `{TextOptions}` - the text options (font height)
- `content` `{string}` - the text to print

Returns `{Label}` - a new Label with the text field added

label.zpl(dpi as int)

Build a ZPL render target for a printer of the given resolution.

Parameters

- `dpi` `{int}` - the printer dots-per-inch

Returns `{Device}` - a ZPL device

Structs

label.BarcodeOptions

Optional refinements for a barcode. A zero-value struct (`def o as label.BarcodeOptions;`) means no options: default size, no added check digit, default error correction, human-readable line shown.

Field	Type	Description
height	float	bar height (linear) or module size (2D) in millimetres; 0 uses the default
checkDigit	string	append an auto-computed check digit: "" (none), "mod10", "mod11", "mod16", "mod36", or "mod43"
errorLevel	string	2D error-correction level: "" (default), "L", "M", "Q", or "H"
hideText	bool	true suppresses a linear barcode's human-readable line
moduleWidth	float	a linear barcode's narrow-element width in millimetres; 0 uses the dialect default (cab; zpl uses its own default module width)
ratio	float	the wide:narrow bar ratio for a ratio-based code (Interleaved 2 of 5 / Code 39); 0 uses the default (3)

label.CabSetup

cab-only print-setup that has no ZPL equivalent, carried on the render target so it can be reproduced without leaking into the device-independent `Label` . A zero-value struct emits none of it (a bare `J` , no `H` / `O` line, and an `S` line derived from the label size). Every field is optional; the cab encoder emits a command only when the

corresponding field is set. ZPL ignores it all.

Field	Type	Description
jobName	string	the J job name; "" emits a bare J
heat	int	the H heat/contrast level
speed	int	the H print speed
mode	string	the trailing H tokens verbatim, e.g. "T,R0"; "" and heat/speed 0 omit the H line
orientation	string	the O orientation token, e.g. "R"; "" omits the O line
sensor	string	the S photocell/sensor type, e.g. "l1" (die-cut labels with gap); "" emits no prefix
xOffset	float	the S horizontal origin offset in millimetres
yOffset	float	the S vertical origin offset in millimetres
height	float	the S label height in millimetres (transport direction); 0 derives the whole S line from the label size
pitch	float	the S label pitch in millimetres (label height + gap between labels)
width	float	the S label width in millimetres
columnPitch	float	the S horizontal distance to the next column in millimetres (multi-up dies)
columns	int	the S number of labels across; <= 1 emits a single-up geometry

label.Device

The render target: which dialect to emit and, for raster dialects, the printer resolution. Build one with `label.zpl(dpi)` , `label.cab()` , or `label.cabWith(setup)` rather than a raw literal.

Field	Type	Description
dialect	string	"zpl" or "cab"
dpi	int	the printer dots-per-inch (used by "zpl"; ignored by "cab")
cab	CabSetup	cab-only print-setup (ignored by "zpl")

label.Field

One field on a label. `kind` selects which attributes matter: "text" uses `h` as the font height and `data` as the content; "barcode" uses `barcodeType` , `h` (bar height, or module size for a 2D code), and `data` ; "box" uses `w` , `h` , `thickness` ; "image" uses `data` as the stored name.

Field	Type	Description
kind	string	"text", "barcode", "box", or "image"
x	float	the x origin in millimetres
y	float	the y origin in millimetres
w	float	the box width in millimetres (0 otherwise)
h	float	box height / barcode height (or 2D module size) / text font height, in millimetres
thickness	float	the box line thickness in millimetres (0 otherwise)
barcodeType	string	the symbology for a barcode field (empty otherwise)
data	string	the text content, barcode data, or image name
checkDigit	string	a barcode's auto-computed check digit ("" | "mod10" | ...)
errorLevel	string	a 2D barcode's error-correction level ("" | "L" | "M" | "Q" | "H")
hideText	bool	suppress a linear barcode's human-readable line

rotation	int	text rotation in degrees counter-clockwise (0, 90, 180, 270)
points	int	text font size in points; 0 means use h as a millimetre height
bold	bool	bold text face
moduleWidth	float	a linear barcode's narrow-element width in millimetres (0 = dialect default)
ratio	float	a ratio-based barcode's wide:narrow ratio (0 = dialect default)

label.Label

A device-independent label: its physical size in millimetres, the number of copies to print, and its fields. Value-semantic - every builder returns a new Label.

Field	Type	Description
width	float	the label width in millimetres
height	float	the label height in millimetres
quantity	int	the number of copies to print
fields	list of Field	the placed fields

label.TextOptions

Options for a text field. A zero-value struct (`def o as label.TextOptions;`) means an unrotated, non-bold field sized by `height` . Set `points` for a point-sized font (it wins over `height`).

Field	Type	Description
height	float	the font height in millimetres (used when points is 0)
points	int	the font size in points; when > 0 it is used instead of height
rotation	int	rotation in degrees counter-clockwise: 0, 90, 180, or 270
bold	bool	true selects a bold face

ldap API reference

An LDAP v3 client and a lightweight, read-only directory server (RFC 4511), over TCP with the wire messages built and parsed as ASN.1 BER (the `asn1` library) on the `net` transport. The client binds (simple or SASL SCRAM), searches with RFC 4515 string filters, follows paged results, and unbinds; LDAPS and StartTLS reuse `net` 's TLS. The server answers simple bind and search for an in-memory directory you build in a few lines - enough to back any LDAP client (an authentication portal such as Authelia, or an app that authenticates its users against LDAP): it validates `userPassword` on bind and evaluates search filters against the entries so a client can find users and read their groups.

The server is read-only over the LDAP protocol (it answers bind and search but rejects add / modify / delete on the wire), yet the in-memory directory is fully mutable from your own code: after building it you can addEntry / modifyEntry / deleteEntry / setAttribute at runtime, so a web interface or a database sync can update the directory a running server is serving. The Directory is shared mutable state (a kv-backed store shared across the serve loop's connection tasks), so a change is visible to the live server immediately. No schema enforcement, no ACLs.

Import with `import "ldap.j" as ldap;` . See the `ldap` guide for prose and examples.

Functions

ldap.add(conn as Conn, dn as string, attrs as map of string to list of string)

add creates an entry from a DN and an attribute map, returning the Result (check .code).

Parameters

- `conn {Conn}` - an open, bound connection
- `dn {string}` - the DN of the entry to create
- `attrs {map of string to list of string}` - the entry's attributes (name -> values)

Returns `{Result}` - the operation result (check .code)

Throws

- `{Error}` - on a transport or protocol error

ldap.addEntry(dir as Directory, e as Entry)

addEntry inserts an entry, replacing any existing entry with the same DN.

Parameters

- `dir {Directory}` - the directory to modify
- `e {Entry}` - the entry to insert (or replace a same-DN entry with)

ldap.allOf(filters as list of asn1.Value)

allOf builds a conjunction: (&(f1)(f2)...).

Parameters

- `filters {list of asn1.Value}` - the sub-filters that must all match

Returns `{asn1.Value}` - the combined filter, for search

ldap.anyOf(filters as list of asn1.Value)

anyOf builds a disjunction: (|(f1)(f2)...).

Parameters

- `filters {list of asn1.Value}` - the sub-filters, any of which may match

Returns `{asn1.Value}` - the combined filter, for search

ldap.approx(attr as string, value as string)

approx builds an approximate-match (attr~=value) filter; the server treats it as equality.

Parameters

- `attr {string}` - the attribute name
- `value {string}` - the value to match approximately

Returns `{asn1.Value}` - the encoded filter, for search

ldap.bind(conn as Conn, dn as string, password as string)

bind performs a simple bind and returns the Result (check .code against ldap.SUCCESS / ldap.INVALID_CREDENTIALS). An empty dn and password is an anonymous bind.

Parameters

- `conn {Conn}` - an open connection
- `dn {string}` - the bind DN ("" for anonymous)
- `password {string}` - the password ("" for anonymous)

Returns `{Result}` - the bind result (check .code)

Throws

- `{Error}` - on a transport or protocol error

ldap.bindSasl(conn as Conn, user as string, password as string, algo as string)

bindSasl performs a SASL SCRAM bind (algo "sha1" -> SCRAM-SHA-1, "sha256" -> SCRAM-SHA-256), verifying the server signature. Returns the Result on success or throws on a protocol / verification failure.

Parameters

- `conn {Conn}` - an open connection
- `user {string}` - the SCRAM authentication username
- `password {string}` - the user's password
- `algo {string}` - the SCRAM hash: "sha1" or "sha256"

Returns `{Result}` - the bind result on success

Throws

- `{Error}` - on a protocol or server-signature-verification failure

ldap.change(operation as int, name as string, values as list of string)

change builds one modify change (operation is ldap.MOD_ADD / MOD_DELETE / MOD_REPLACE).

Parameters

- `operation {int}` - ldap.MOD_ADD, MOD_DELETE, or MOD_REPLACE
- `name {string}` - the attribute to change
- `values {list of string}` - the values for the change (empty deletes the attribute)

Returns `{Change}` - the change, for modify

ldap.close(conn as Conn)

close closes the connection without sending an unbind.

Parameters

- `conn {Conn}` - the connection to close

ldap.connect(address as string, security as transport.Security)

connect opens an LDAP connection. security selects the transport: transport.Security.None is plaintext (port defaults to 389), transport.Security.Tls is LDAPS / implicit TLS (port defaults to 636), and transport.Security.Starttls connects in plaintext then upgrades in-band via the StartTLS extended operation.

Parameters

- `address {string}` - the server "host:port" (port defaults to 389, or 636 for Tls)
- `security {transport.Security}` - the transport: None, Tls, or Starttls

Returns `{Conn}` - the open connection

Throws

- `{Error}` - on a connection or StartTLS failure

ldap.delete(conn as Conn, dn as string)

delete removes an entry by DN, returning the Result.

Parameters

- `conn {Conn}` - an open, bound connection
- `dn {string}` - the DN of the entry to remove

Returns `{Result}` - the operation result (check .code)

Throws

- `{Error}` - on a transport or protocol error

ldap.deleteEntry(dir as Directory, dn as string)

deleteEntry removes the entry with the given DN if present.

Parameters

- `dir {Directory}` - the directory to modify
- `dn {string}` - the DN of the entry to remove (matched case-insensitively)

ldap.directory(entries as list of Entry)

directory builds a mutable in-memory directory the server answers for - the lightweight, ephemeral default. Use openDirectory for one that persists.

Parameters

- `entries {list of Entry}` - the initial entries to seed the directory with

Returns `{Directory}` - the in-memory directory

ldap.entry(dn as string, attrs as map of string to list of string)

entry builds a directory entry from a DN and an attribute map (name -> values).

Parameters

- `dn {string}` - the entry's distinguished name
- `attrs {map of string to list of string}` - the entry's attributes (name -> values)

Returns `{Entry}` - the constructed entry

ldap.equals(attr as string, value as string)

equals builds an equality filter: (attr=value).

Parameters

- `attr {string}` - the attribute name
- `value {string}` - the value to match exactly

Returns `{asn1.Value}` - the encoded filter, for search

ldap.firstValue(entry as Entry, name as string)

firstValue returns an entry's first value for an attribute, or "" if absent.

Parameters

- `entry {Entry}` - the entry to read
- `name {string}` - the attribute name, matched case-insensitively

Returns `{string}` - the first value, or "" if the attribute has none

ldap.getEntry(dir as Directory, dn as string)

getEntry returns the entry with the given DN, or an empty Entry (dn == "") if absent.

Parameters

- `dir {Directory}` - the directory to read
- `dn {string}` - the DN to look up (matched case-insensitively)

Returns `{Entry}` - the matching entry, or an empty Entry (dn == "") if absent

ldap.greaterOrEqual(attr as string, value as string)

greaterOrEqual builds an (attr>=value) filter.

Parameters

- `attr {string}` - the attribute name
- `value {string}` - the lower bound to compare against

Returns `{asn1.Value}` - the encoded filter, for search

ldap.group(dn as string, members as list of string)

group builds a groupOfNames entry (objectClass + cn from the DN's RDN + member) from members.

Parameters

- `dn {string}` - the group's distinguished name (its RDN value becomes cn)
- `members {list of string}` - the member DNs

Returns `{Entry}` - the constructed groupOfNames entry

ldap.hasEntry(dir as Directory, dn as string)

hasEntry reports whether an entry with the given DN exists.

Parameters

- `dir {Directory}` - the directory to read
- `dn {string}` - the DN to test (matched case-insensitively)

Returns `{bool}` - true if an entry with that DN exists

ldap.lessOrEqual(attr as string, value as string)

lessOrEqual builds an (attr<=value) filter.

Parameters

- `attr {string}` - the attribute name
- `value {string}` - the upper bound to compare against

Returns `{asn1.Value}` - the encoded filter, for search

ldap.listEntries(dir as Directory)

listEntries returns a snapshot of every entry currently in the directory.

Parameters

- `dir {Directory}` - the directory to read

Returns `{list of Entry}` - a snapshot of all current entries

ldap.listen(address as string)

listen opens a listening socket for the server ("host:port"; port defaults to 389).

Parameters

- `address {string}` - the listen address "host:port" (port defaults to 389)

Returns `{net.Listener}` - the open listener, for serveOn

Throws

- `{Error}` - if the socket cannot be opened

ldap.modify(conn as Conn, dn as string, changes as list of Change)

modify applies a list of changes to an entry, returning the Result.

Parameters

- `conn {Conn}` - an open, bound connection
- `dn {string}` - the DN of the entry to modify
- `changes {list of Change}` - the changes to apply, from change

Returns `{Result}` - the operation result (check .code)

Throws

- `{Error}` - on a transport or protocol error

ldap.modifyDn(conn as Conn, dn as string, newRdn as string, deleteOldRdn as bool, newSuperior as string)

modifyDn renames or moves an entry: newRdn is the new relative DN, deleteOldRdn drops the old RDN attribute, and newSuperior ("" to keep the parent) moves the entry under a new parent DN.

Parameters

- `conn {Conn}` - an open, bound connection
- `dn {string}` - the DN of the entry to rename or move
- `newRdn {string}` - the new relative DN (e.g. "cn=bob")
- `deleteOldRdn {bool}` - whether to drop the old RDN attribute value
- `newSuperior {string}` - the new parent DN, or "" to keep the current parent

Returns `{Result}` - the operation result (check .code)

Throws

- `{Error}` - on a transport or protocol error

ldap.modifyEntry(dir as Directory, e as Entry)

modifyEntry replaces the entry with the same DN (an alias for addEntry).

Parameters

- `dir {Directory}` - the directory to modify
- `e {Entry}` - the replacement entry, keyed by its DN

ldap.negate(f as asn1.Value)

negate builds a negation: (!(f)).

Parameters

- `f {asn1.Value}` - the sub-filter to invert

Returns `{asn1.Value}` - the negated filter, for search

ldap.openDirectory(path as string)

openDirectory opens a file-backed directory that persists across restarts (a kv.openFile store at path). A new file starts empty; an existing one restores its entries. Seed a fresh store on first run by checking listEntries and addEntry-ing when it is empty. Edits (addEntry / modifyEntry / ...) persist.

Parameters

- `path {string}` - the backing file path for the kv store

Returns `{Directory}` - the file-backed directory

ldap.parseFilter(s as string)

parseFilter compiles an RFC 4515 filter string (e.g. "(&(a=1)(b=2))") into a filter value.

Parameters

- `s {string}` - the RFC 4515 filter text

Returns `{asn1.Value}` - the compiled filter, for search

Throws

- `{Error}` - on a malformed filter string

ldap.password(plain as string, scheme as string)

password hashes a plaintext password into an LDAP userPassword value. scheme is "plain", "sha", "sha256", "ssha", "ssha256" (salted variants use a random 8-byte salt), "pbkdf2" (PBKDF2-SHA256), or "pbkdf2-sha512". Store the result as an entry's userPassword.

Parameters

- `plain {string}` - the plaintext password to hash
- `scheme {string}` - one of plain / sha / sha256 / ssha / ssha256 / pbkdf2 / pbkdf2-sha512

Returns `{string}` - the encoded userPassword value (with its scheme prefix)

Throws

- `{Error}` - on an unknown scheme

ldap.passwordModify(conn as Conn, userDn as string, oldPassword as string, newPassword as string)

passwordModify changes a password via the RFC 3062 extended operation. Pass userDn "" for the currently-bound user, oldPassword "" to skip the old-password check (an administrative reset), and newPassword "" to have the server generate one. Returns the Result.

Parameters

- `conn {Conn}` - an open, bound connection
- `userDn {string}` - the target user DN, or "" for the currently-bound user
- `oldPassword {string}` - the current password, or "" to skip the check (admin reset)
- `newPassword {string}` - the new password, or "" to let the server generate one

Returns `{Result}` - the operation result (check .code)

Throws

- `{Error}` - on a transport or protocol error

ldap.present(attr as string)

present builds a presence filter: (attr=*).

Parameters

- `attr {string}` - the attribute that must be present

Returns `{asn1.Value}` - the encoded filter, for search

ldap.search(conn as Conn, baseDn as string, scope as int, filter as asn1.Value, attributes as list of string)

search runs a search and returns the matching entries. scope is one of ldap.SCOPE_BASE / SCOPE_ONE / SCOPE_SUB; filter comes from parseFilter or the filter constructors; attributes is the list to return (empty = all user attributes).

Parameters

- `conn {Conn}` - an open, bound connection
- `baseDn {string}` - the search base DN
- `scope {int}` - ldap.SCOPE_BASE, SCOPE_ONE, or SCOPE_SUB
- `filter {asn1.Value}` - the filter, from parseFilter or a filter constructor
- `attributes {list of string}` - the attributes to return (empty = all user attributes)

Returns `{list of Entry}` - the matching entries

Throws

- `{Error}` - on a transport or protocol error

ldap.searchPaged(conn as Conn, baseDn as string, scope as int, filter as asn1.Value, attributes as list of string, pageSize as int)

searchPaged runs a search in pages of pageSize via the simple-paged-results control.

Parameters

- `conn {Conn}` - an open, bound connection
- `baseDn {string}` - the search base DN
- `scope {int}` - ldap.SCOPE_BASE, SCOPE_ONE, or SCOPE_SUB
- `filter {asn1.Value}` - the filter, from parseFilter or a filter constructor
- `attributes {list of string}` - the attributes to return (empty = all user attributes)
- `pageSize {int}` - the maximum entries per page

Returns `{list of Entry}` - all matching entries across every page

Throws

- `{Error}` - on a transport or protocol error

ldap.serve(dir as Directory, address as string)

serve binds address and serves dir until the process ends (blocks).

Parameters

- `dir {Directory}` - the directory to answer queries from
- `address {string}` - the listen address "host:port" (port defaults to 389)

Throws

- `{Error}` - if the socket cannot be opened

ldap.serveOn(dir as Directory, listener as net.Listener)

serveOn serves dir on an already-open listener, one spawn per connection, until the listener is closed (which ends the accept loop). Close the listener from another task for a graceful stop.

Parameters

- `dir {Directory}` - the directory to answer queries from
- `listener {net.Listener}` - an open listener from listen

ldap.setAttribute(dir as Directory, dn as string, name as string, vals as list of string)

setAttribute creates or replaces one attribute's values on an existing entry.

Parameters

- `dir {Directory}` - the directory to modify
- `dn {string}` - the DN of the entry to update
- `name {string}` - the attribute to create or replace
- `vals {list of string}` - the new values for the attribute

Throws

- `{Error}` - if no entry with dn exists

ldap.startTls(conn as Conn)

startTls upgrades a plaintext connection to TLS via the StartTLS extended operation, then hands the same handle back (now encrypted).

Parameters

- `conn {Conn}` - an open plaintext connection from connect

Returns `{Conn}` - the same connection, now encrypted

Throws

- `{Error}` - if the server refuses StartTLS

ldap.substrings(attr as string, initial as string, anyParts as list of string, final as string)

substrings builds a substring filter, e.g. (cn=a *b* c): initial "a", any ["b"], final "c". Pass "" for an absent initial or final and an empty list for no interior parts.

Parameters

- `attr {string}` - the attribute name
- `initial {string}` - the required leading substring, or "" for none
- `anyParts {list of string}` - interior substrings in order, or empty for none

- `final {string}` - the required trailing substring, or "" for none

Returns `{asn1.Value}` - the encoded filter, for search

ldap.unbind(conn as Conn)

unbind sends an unbind request and closes the connection.

Parameters

- `conn {Conn}` - the connection to unbind and close

ldap.values(entry as Entry, name as string)

values returns an entry's values for an attribute (case-insensitive name), or an empty list.

Parameters

- `entry {Entry}` - the entry to read
- `name {string}` - the attribute name, matched case-insensitively

Returns `{list of string}` - the attribute's values, or an empty list if absent

Structs

ldap.Attribute

One attribute of a directory entry.

Field	Type	Description
name	string	the attribute type (e.g. "mail", "objectClass")
values	list of string	its values (LDAP attributes are multi-valued)

ldap.Change

One change in a modify request.

Field	Type	Description
operation	int	ldap.MOD_ADD / MOD_DELETE / MOD_REPLACE
name	string	the attribute to change
values	list of string	the values (empty deletes the whole attribute for MOD_DELETE)

ldap.Conn

An open LDAP connection.

Field	Type	Description
handle	net.Conn	the underlying TCP (or TLS) connection
timeoutMs	int	the per-read timeout in ms (0 blocks with no deadline)

ldap.Directory

A mutable in-memory directory the server answers for. It is backed by a shared kv store, so edits made with addEntry / modifyEntry / deleteEntry / setAttribute are visible to a running server across its connection tasks.

Field	Type	Description
store	kv.Store	the shared backing store holding every entry

ldap.Entry

A directory entry.

Field	Type	Description
dn	string	the distinguished name
attributes	list of ldap.Attribute	the entry's attributes

ldap.Result

The outcome of a bind (or any LDAPResult).

Field	Type	Description
code	int	the result code (ldap.SUCCESS, or ldap.INVALID_CREDENTIALS on a bad bind)
matchedDn	string	the matched DN the server reports, if any
message	string	the server's diagnostic message

Constants

Constant	Type	Description
ldap.INVALID_CREDENTIALS	int	LDAP result code: invalid credentials (a failed bind).
ldap.MOD_ADD	int	modify() change type: add the given values to the attribute.
ldap.MOD_DELETE	int	modify() change type: delete the given values (or the whole attribute if none).
ldap.MOD_REPLACE	int	modify() change type: replace the attribute with the given values.
ldap.SCOPE_BASE	int	Search scope: the base entry only.
ldap.SCOPE_ONE	int	Search scope: the base's immediate children.
ldap.SCOPE_SUB	int	Search scope: the base and its whole subtree.
ldap.SUCCESS	int	LDAP result code: success.

log API reference

Leveled, structured logging. A `log.Logger` carries a minimum level (`debug` < `info` < `warn` < `error`), an output format (`text` / `logfmt` / `json`), and a sink; `log.info(logger, message, fields)` (and the sibling levels) render one record - a timestamp, the level, the message, and the caller's key/value `fields` - and write it, dropping records below the logger's level.

Sinks: `stdout` / `stderr` (both binaries, via `io`), a `file` (append, via `fs`), and an RFC 5424 **syslog** sink over UDP (`net` , so the syslog sink needs the default `jennifer` binary; the console / file sinks work on both). Over `io` / `fs` + `json` + `strings` + `time` + `os` (+ `net` for syslog).

Import with `import "log.j" as log;` . See the `log` guide for prose and examples.

Functions

log.at(logger as Logger, level as string, message as string, fields as map of string to string)

Emit a record at an explicit level (skipped if below the logger's level).

Parameters

- `logger {Logger}` - the logger
- `level {string}` - "debug", "info", "warn", or "error"
- `message {string}` - the log message
- `fields {map of string to string}` - structured key/value fields ({} for none)

Throws

- `{Error}` - on a sink failure (a positioned `fs` / `net` error)

log.debug(logger as Logger, message as string, fields as map of string to string)

Emit a debug record.

Parameters

- `logger {Logger}` - the logger
- `message {string}` - the log message
- `fields {map of string to string}` - structured fields ({} for none)

log.error(logger as Logger, message as string, fields as map of string to string)

Emit an error record.

Parameters

- `logger {Logger}` - the logger
- `message {string}` - the log message
- `fields {map of string to string}` - structured fields ({} for none)

log.fatal(logger as Logger, message as string, fields as map of string to string)

Emit a fatal record and then terminate the program with exit code 1. The "fatal" level ranks above every other, so the record is always emitted regardless of the logger's minimum level.

Parameters

- `logger` `{Logger}` - the logger
- `message` `{string}` - the log message
- `fields` `{map of string to string}` - structured fields ({} for none)

log.info(logger as Logger, message as string, fields as map of string to string)

Emit an info record.

Parameters

- `logger` `{Logger}` - the logger
- `message` `{string}` - the log message
- `fields` `{map of string to string}` - structured fields ({} for none)

log.new(level as string, format as string)

A logger writing to standard output.

Parameters

- `level` `{string}` - the minimum level ("debug"/"info"/"warn"/"error")
- `format` `{string}` - "text", "logfmt", or "json"

Returns `{Logger}` - the logger

log.toFile(level as string, format as string, path as string)

A logger appending to a file (created if missing).

Parameters

- `level` `{string}` - the minimum level
- `format` `{string}` - "text", "logfmt", or "json"
- `path` `{string}` - the file path

Returns `{Logger}` - the logger

log.toStderr(level as string, format as string)

A logger writing to standard error.

Parameters

- `level` `{string}` - the minimum level
- `format` `{string}` - "text", "logfmt", or "json"

Returns `{Logger}` - the logger

log.toSyslog(level as string, address as string, app as string)

A logger sending RFC 5424 records to a syslog server over UDP. Needs the default `jennifer` binary (`net`).

Parameters

- `level` `{string}` - the minimum level
- `address` `{string}` - the syslog server "host:port" (e.g. "localhost:514")
- `app` `{string}` - the APP-NAME tag

Returns `{Logger}` - the logger

log.warn(logger as Logger, message as string, fields as map of string to string)

Emit a warn record.

Parameters

- `logger {Logger}` - the logger
- `message {string}` - the log message
- `fields {map of string to string}` - structured fields ({} for none)

log.with(logger as Logger, fields as map of string to string)

A child logger carrying persistent context `fields` . Every record emitted through the returned logger includes these fields, merged with any per-call fields (a per-call key of the same name wins). Composes: calling `with` again on the result merges the new fields on top of the carried ones. The parent logger is unchanged (value semantics).

Parameters

- `logger {Logger}` - the parent logger
- `fields {map of string to string}` - the persistent fields to attach

Returns `{Logger}` - a new logger with the merged persistent fields

Structs

log.Logger

A value-semantic logger configuration. Build one with `new` / `toStderr` / `toFile` / `toSyslog` .

Field	Type	Description
level	string	the minimum level to emit: "debug", "info", "warn", or "error"
format	string	the record format: "text", "logfmt", or "json" (ignored by the syslog sink)
sink	string	where records go: "stdout", "stderr", "file", or "syslog"
target	string	the file path (file sink) or "host:port" (syslog sink); "" for a console sink
app	string	an application / tag name (the syslog APP-NAME; "" means none)
fields	map of string to string	persistent context fields attached to every record (see with); {} for none

markdown API reference

A lightweight Markdown renderer for a small CommonMark subset: ATX headings, bold / italic emphasis, inline code, links, images, fenced code blocks, unordered / ordered lists (**nested** by indentation), blockquotes (nesting recursively), and GFM tables. Renders to HTML (through the `html` module, so escaping is handled for you) and to styled terminal text (through the `ansi` module). It also authors Markdown text (header / style / link / list / codeBlock / table). Pure Jennifer; line-oriented block parsing with a small inline scanner. Not full CommonMark: inline spans do not nest (the content of `**...**` , `...` , a link, or an image alt is plain text), and there is no thematic break, setext heading, or reference-link support. A link / image URL cannot contain an unescaped `)` (the scanner closes on the first one).

Import with `import "markdown.j" as markdown;` . See the `markdown` guide for prose and examples.

Functions

markdown.attr(node as Node, name as string)

A named string attribute of a node: `"href"` / `"title"` (a `link` / `image`), `"lang"` (a `code` block), `"align"` (a table `cell`), `"ordered"` ("true" / "false", a `list`), or `"level"` (a heading, as a string). Returns "" for an absent attribute.

Parameters

- `node` `{Node}` - the node
- `name` `{string}` - the attribute name

Returns `{string}` - the attribute value, or ""

markdown.bullets(items as list of string)

Render an unordered list, one `- item` per line.

Parameters

- `items` `{list of string}` - the list items

Returns `{string}` - the Markdown bullet list

markdown.children(node as Node)

A node's direct children.

Parameters

- `node` `{Node}` - the node

Returns `{list of Node}` - the child nodes

markdown.codeBlock(text as string)

Render a fenced code block around verbatim text.

Parameters

- `text` `{string}` - the verbatim code

Returns `{string}` - the fenced Markdown code block

markdown.findAll(node as Node, selector as string)

Every node matching a `/` -separated `selector` relative to `node` : each step is a kind name, `*` (any kind), or `name[k]` (the k-th such child, 1-based). Steps match direct children.

Parameters

- `node` `{Node}` - the node to search under
- `selector` `{string}` - the selector path (e.g. "list/item")

Returns `{list of Node}` - the matching nodes (empty if none)

markdown.get(node as Node, selector as string)

The first node matching `selector` (see `findAll`), or an empty node (`typeOf` "") if there is no match.

Parameters

- `node` `{Node}` - the node to search under
- `selector` `{string}` - the selector path

Returns `{Node}` - the first match, or an empty node

markdown.gray(level as int)

A grey fill of the given 0-255 level (0 black, 255 white).

Parameters

- `level` `{int}` - the grey level, 0-255

Returns `{Fill}` - the fill

markdown.has(node as Node, selector as string)

Whether any node matches `selector` (see `findAll`).

Parameters

- `node` `{Node}` - the node to search under
- `selector` `{string}` - the selector path

Returns `{bool}` - true if at least one node matches

markdown.header(level as string, text as string)

Render an ATX heading.

Parameters

- `level` `{string}` - the heading depth, "h1".."h6"
- `text` `{string}` - the heading text

Returns `{string}` - the Markdown heading line

Throws

- `{Error}` - when level is not "h1".."h6"

markdown.headingStyle(background as Fill)

A heading style with the given background fill.

Parameters

- `background` `{Fill}` - the background fill (off for none)

Returns `{HeadingStyle}` - the style

markdown.level(node as Node)

A heading's level (1-6); 0 for any other node.

Parameters

- `node` `{Node}` - the node

Returns `{int}` - the heading level

markdown.link(text as string, url as string)

Render an inline link `text (url)` .

Parameters

- `text` `{string}` - the link text
- `url` `{string}` - the link target

Returns `{string}` - the Markdown link

markdown.noFill()

No fill (a transparent background).

Returns `{Fill}` - an off fill

markdown.numbered(items as list of string)

Render an ordered list, `1. item` upward.

Parameters

- `items` `{list of string}` - the list items

Returns `{string}` - the Markdown numbered list

markdown.pageBreak()

A page-break node. Placed in a document tree (typically between the blocks of a hand-assembled book), it makes `renderPdf` / `toPdf` start the following content on a fresh page - a lever independent of the level-one-heading page break. The HTML and ANSI renderers ignore it. In Markdown source, a lone `<!--pagebreak-->` comment parses to the same node.

Returns `{Node}` - a page_break node

markdown.parse(md as string)

Parse Markdown into a document tree, walked by the reader accessors (`typeOf` / `children` / `text` / `level` / `attr` / `get` / `findAll` / `has`) - so a caller can inspect or transform a document (pull the headings for a table of contents, rewrite links, convert to another format) before `render` ing it. The returned node is the document root (its `typeOf` is `"document"`); its children are the top-level blocks. A document that nests too deep or holds too many nodes is a catchable `"markdown"` error.

Parameters

- `md` `{string}` - the Markdown source

Returns `{Node}` - the document root node

markdown.pdfDefaults()

The default options: US Letter, 54-point margins, the Helvetica family for body / bold / italic / headings, Courier for code, 11-point body. Copy and tweak fields (value semantics) to customise.

Returns `{PdfOptions}` - the default options

markdown.render(doc as Node, format as string)

Render a document tree (a `parse` result, or a hand-built one) to a string. `format` is `"html"` (block elements concatenated, no indentation) or `"ansi"` (styled terminal text, blocks separated by a blank line). A `"document"` node renders its children; any other node renders as a single block. An unknown format is a catchable `"markdown"` error.

Parameters

- `doc {Node}` - the document (or block) node to render
- `format {string}` - "html" or "ansi"

Returns `{string}` - the rendered document

markdown.renderPdf(doc as Node, opts as PdfOptions)

Render a parsed (or hand-built / transformed) `markdown` document tree to PDF bytes with the given options. A one-liner over `renderPdfDoc` + `pdf.render` ; call those two directly when you need to touch the `pdf.Document` in between.

Parameters

- `doc {Node}` - the document root node from `parse`
- `opts {PdfOptions}` - the page geometry and fonts

Returns `{bytes}` - the PDF document

markdown.renderPdfDoc(doc as Node, opts as PdfOptions)

Lay a parsed (or hand-built / transformed) `markdown` document tree out to a `pdf.Document` , ready for any document-level work - a running header or footer (`pdf.setHeader` / `pdf.setFooter` , with `%page%` / `%pages%`), extra `pdf.info` metadata, more bookmarks - before it is serialised with `pdf.render` . This is the seam `renderPdf` renders through; use it directly when you need the document itself (a page number in a book's footer needs the total page count, which only exists once the whole document is laid out).

Parameters

- `doc {Node}` - the document root node from `parse`
- `opts {PdfOptions}` - the page geometry and fonts

Returns `{pdf.Document}` - the laid-out document, not yet serialised

markdown.rgb(r as int, g as int, b as int)

An RGB fill (each component 0-255).

Parameters

- `r {int}` - red 0-255
- `g {int}` - green 0-255
- `b {int}` - blue 0-255

Returns `{Fill}` - the fill

markdown.style(kind as string, text as string)

Wrap text in an inline emphasis span.

Parameters

- `kind {string}` - the emphasis, "bold" / "italic" / "code"
- `text {string}` - the text to wrap

Returns `{string}` - the emphasised Markdown span

Throws

- `{Error}` - when kind is not "bold" / "italic" / "code"

markdown.table(headings as list of string, aligns as list of string, rows as list of list of string)

Render a GFM table. Columns follow `headings` : a short row is padded with empty cells, extra cells are

dropped. Pipes and newlines in a cell are made safe.

Parameters

- `headings` `{list of string}` - the column headings
- `aligns` `{list of string}` - the per-column alignment ("left" / "right" / "center" / "none"; `[]` for all-default)
- `rows` `{list of list of string}` - the data rows, each a list of cell strings

Returns `{string}` - the GFM table source

Throws

- `{Error}` - when an align value is not "left" / "right" / "center" / "none"

markdown.tablePretty(md as string)

Reformat every GFM table in Markdown text so its source columns line up (padded cells, aligned delimiters), leaving all other lines exactly as written. The handcraft-then-prettify workflow, in one call.

Parameters

- `md` `{string}` - the Markdown source

Returns `{string}` - the source with its tables aligned

markdown.text(node as Node)

The text content of a node: a leaf's own literal text, else the concatenation of its descendants' text - so `text` of a heading (or any container) is its full flattened text, handy for a table of contents.

Parameters

- `node` `{Node}` - the node

Returns `{string}` - the flattened text

markdown.toAnsi(md as string)

Render Markdown to styled terminal text, blocks separated by a blank line.

Parameters

- `md` `{string}` - the Markdown source

Returns `{string}` - the rendered terminal text

markdown.toHtml(md as string)

Render Markdown to an HTML string (block elements concatenated, no indentation). **Safe by default:** a raw HTML block in the source (a line opening with `<tag>`) is escaped and rendered as literal text, so passing untrusted Markdown (a README, a user comment) cannot inject a `<script>` or an event-handler attribute. Everything the renderer builds is already escaped (text, link URLs via an allow-list, image attributes). To pass raw HTML through for **trusted** input, use `toHtmlWith` with `allowRawHtml: true` .

Parameters

- `md` `{string}` - the Markdown source

Returns `{string}` - the rendered HTML

markdown.toHtmlWith(md as string, opts as HtmlOptions)

Render Markdown to HTML with explicit options. The only option today is `allowRawHtml` : set it true to emit raw HTML blocks verbatim (for trusted Markdown), or false (the default `toHtml` behavior) to escape them.

Parameters

- `md` `{string}` - the Markdown source

- `opts` `{HtmlOptions}` - the render options

Returns `{string}` - the rendered HTML

markdown.toPdf(md as string)

Render a Markdown string to PDF bytes with the default options.

Parameters

- `md` `{string}` - the Markdown source

Returns `{bytes}` - the PDF document

markdown.toPdfWith(md as string, opts as PdfOptions)

Render a Markdown string to PDF bytes with the given options.

Parameters

- `md` `{string}` - the Markdown source
- `opts` `{PdfOptions}` - the page geometry and fonts

Returns `{bytes}` - the PDF document

markdown.typeOf(node as Node)

The kind of a node (`"document"` , `"heading"` , `"link"` , `"text"` , ...).

Parameters

- `node` `{Node}` - the node

Returns `{string}` - the node kind

Structs

markdown.Fill

A fill colour (0-255 RGB) that may be off. `on: false` means "no fill" (draw nothing); an `on: true` fill paints a background behind a heading or a table header row. Build one with `gray` / `rgb` , or `noFill` for none.

Field	Type	Description
on	bool	whether the fill is applied at all
r	int	red 0-255
g	int	green 0-255
b	int	blue 0-255

markdown.HeadingStyle

The style for a heading level - currently just a `background` fill painted behind the heading text (a shaded bar). A struct so more style knobs can be added later.

Field	Type	Description
background	Fill	the background fill (off for none)

markdown.HtmlOptions

Options for HTML rendering.

Field	Type	Description

allowRawHtml	bool	pass raw HTML blocks through verbatim (trusted input only). Default (zero value) is false: raw HTML is escaped, so untrusted Markdown cannot inject scripts or event handlers.

markdown.Node

A node in a parsed Markdown document tree. The tree is walked with the reader accessors (`typeOf` / `children` / `text` / `level` / `attr` / `get` / `findAll` / `has`), the same family vocabulary as `xml` / `html` . A node's `kind` is one of the block kinds `"document"` / `"heading"` / `"paragraph"` / `"code"` / `"list"` / `"item"` / `"table"` / `"row"` / `"cell"` / `"quote"` , or the inline kinds `"text"` / `"codespan"` / `"strong"` / `"emphasis"` / `"link"` / `"image"` .

Field	Type	Description
kind	string	the node kind
level	int	a heading's level (1-6); 0 otherwise
text	string	literal text (a text / codespan / code node, or an image's alt); "" for a container (read its content with the text accessor)
lang	string	a fenced code block's language ("" if none)
ordered	bool	whether a list is ordered
url	string	a link / image target
title	string	a link / image title ("" if none)
align	string	a table cell's alignment ("left" / "right" / "center" / "")
children	list of Node	the child nodes

markdown.PdfOptions

Page geometry and fonts for a render. `margin` is applied to all four sides; `bodyFont` / `boldFont` / `italicFont` / `monoFont` are standard-14 font names for body text and its inline styles, `headingFont` for headings. `bodySize` is the body point size; heading sizes derive from the heading level.

Field	Type	Description
pageWidth	int	page width in points (Letter 612, A4 595)
pageHeight	int	page height in points (Letter 792, A4 842)
margin	int	margin on every side, in points
bodyFont	string	standard-14 font for body text
boldFont	string	standard-14 font for **strong**
italicFont	string	standard-14 font for *emphasis*
monoFont	string	standard-14 font for inline / block code
headingFont	string	standard-14 font for headings
bodySize	int	body text point size
tablePad	int	padding inside a table cell, in points (table density)
tableHeaderFill	Fill	background fill behind a table's header row (off for none)
headingStyles	list of HeadingStyle	per-level heading style; index 0 is h1, 1 is h2, and so on (missing / off entries render with no background)
title	string	PDF document Title metadata ("" = unset)
author	string	PDF document Author metadata ("" = unset)
subject	string	PDF document Subject metadata ("" = unset)
keywords	string	PDF document Keywords metadata ("" = unset)

bookmarkLevel	int	bookmark headings up to this level (0 = none; 2 = h1 + h2)
unencodable	string	substitute for a character the standard-14 fonts cannot encode ("?" default; "" drops it)
codeFill	Fill	background fill behind a code block (off for none)
codeBorder	Fill	border stroke around a code block (off for none)
quoteFill	Fill	background fill behind a blockquote (off for none)
quoteRule	Fill	colour of the vertical bar down a blockquote's left edge (off for none)
creator	string	PDF document Creator metadata ("" = unset)
producer	string	PDF document Producer metadata ("" = keep the pdf default)

mcp API reference

Model Context Protocol (https://modelcontextprotocol.io) - the stateless JSON-RPC 2.0 surface an LLM host and an MCP peer speak. This module is both a **server** (expose tools / resources / prompts to a host) and a **client** (call a remote MCP server) over either HTTP or a launched stdio subprocess. It is JSON-RPC 2.0 on the wire, so the HTTP client is a thin layer over `jsonrpc` (which is over `http` + `json`) and the server is a purpose-built router (`handle`) that dispatches only to *registered* items - unlike `jsonrpc.handle` 's open name dispatch, MCP is an allow-list.

A `Server` is built value-semantically: `server(name, version)` then `addTool` / `addResource` / `addPrompt` , each returning a fresh `Server` . Each registered item is handled by a top-level `func NAME(arg as json.Value)` in the entry program, passed as a `func` value and called through its home interpreter (the same mechanism the `web` module uses), so it resolves its own imports and builds its `json.Value` result in the entry program's context. `handle(server, requestBody) -> replyBody` is the transport- agnostic dispatcher; `serveStdio(server)` runs the primary stdio transport (newline-delimited JSON-RPC on the program's own stdin/stdout).

Scope: the stateless protocol only. No SSE, no sessions. The stdio server (`serveStdio`), the HTTP client (`connect` , over `jsonrpc`), and the stdio subprocess client (`connectStdio` , over `os.run`) are all implemented; the stdio client's exchange is one-shot (launch, handshake + op, read replies) since there is no persistent child pipe. Because it uses `jsonrpc` / `http` / `os.run` , this module needs the default `jennifer` binary.

Security. A tool call is dispatched only when the tool name is in the registry; an unknown tool is a tool error, not an arbitrary-name dispatch. A handler that throws yields a generic error to the peer - the thrown message is never put on the wire, matching `jsonrpc` 's posture.

Import with `import "mcp.j" as mcp;` . See the `mcp` guide for prose and examples.

Functions

mcp.addPrompt(server as Server, name as string, description as string, arguments as list of PromptArg, handler as func)

Register a prompt, returning a fresh server (value-semantic). `arguments` declares the arguments a host should collect (build each with `promptArg` ; pass `[]` for a prompt that takes none).

Parameters

- `server` `{Server}` - the server to extend
- `name` `{string}` - the prompt's unique name
- `description` `{string}` - a description of the prompt
- `arguments` `{list of PromptArg}` - the declared arguments (may be empty)
- `handler` `{func}` - the handler func value that builds the messages

Returns `{Server}` - a server with the prompt added

mcp.addResource(server as Server, uri as string, name as string, description as string, mimeType as string, handler as func)

Register a resource, returning a fresh server (value-semantic).

Parameters

- `server` `{Server}` - the server to extend
- `uri` `{string}` - the resource's unique URI
- `name` `{string}` - a short display name
- `description` `{string}` - a description of the resource

- `mimeType` `{string}` - the content type of the resource's text
- `handler` `{func}` - the handler func value that produces the content

Returns `{Server}` - a server with the resource added

mcp.addTool(server as Server, name as string, description as string, inputSchema as json.Value, handler as func)

Register a tool, returning a fresh server (value-semantic).

Parameters

- `server` `{Server}` - the server to extend
- `name` `{string}` - the tool's unique name
- `description` `{string}` - a description of the tool
- `inputSchema` `{json.Value}` - the JSON Schema for the tool's arguments
- `handler` `{func}` - the handler func value the call dispatches to

Returns `{Server}` - a server with the tool added

mcp.callTool(client as Client, name as string, arguments as json.Value)

Call a remote tool by name. Returns the whole tool result (a `content` array plus an `isError` flag), so the caller can check `isError` .

Parameters

- `client` `{Client}` - the MCP client
- `name` `{string}` - the tool name
- `arguments` `{json.Value}` - the tool arguments object

Returns `{json.Value}` - the `tools/call` result

mcp.connect(endpoint as string)

Connect to a remote MCP server over HTTP.

Parameters

- `endpoint` `{string}` - the MCP endpoint URL (`http://` or `https://`)

Returns `{Client}` - a configured client

mcp.connectStdio(argv as list of string)

Connect to an MCP server launched as a subprocess over **stdio** - the primary MCP transport (a host runs the server and speaks newline-delimited JSON-RPC on its stdin/stdout). `argv` is the server command line (program first). Because the exchange is one-shot (`os.run` , no persistent pipe), each call launches the server, sends the `initialize` handshake plus the operation as newline-delimited JSON on stdin, closes stdin, and reads the replies from stdout - so a stateless server sees a complete short session per call. Needs the default binary (`os.run`).

Parameters

- `argv` `{list of string}` - the server command line (`["python", "srv.py"]` , ...)

Returns `{Client}` - a configured stdio client

mcp.connectWith(endpoint as string, headers as map of string to string)

Connect to a remote MCP server with extra request headers (auth, ...).

Parameters

- `endpoint` `{string}` - the MCP endpoint URL

- `headers {map of string to string}` - headers sent with every request

Returns `{Client}` - a configured client

mcp.getPrompt(client as Client, name as string, arguments as json.Value)

Get a remote prompt by name. Returns the whole `prompts/get` result (a description plus the messages array).

Parameters

- `client {Client}` - the MCP client
- `name {string}` - the prompt name
- `arguments {json.Value}` - the prompt arguments object

Returns `{json.Value}` - the `prompts/get` result

mcp.handle(server as Server, requestBody as string)

Dispatch a single JSON-RPC request body against the server and return the reply body (transport-agnostic: wire it to `httpd` / `net` / stdio). A notification (no `id` , or any `notifications/` method) yields `""` . A parse failure is a `-32700` reply; an unknown protocol method is `-32601` . A `tools/call` to an unknown tool, or a tool handler that throws, is a successful reply carrying a tool result with `isError: true` (the thrown message is never put on the wire).

Parameters

- `server {Server}` - the server whose registries route the request
- `requestBody {string}` - the raw request JSON

Returns `{string}` - the reply JSON, or `""` when no reply is owed

mcp.initialize(client as Client)

Perform the `initialize` handshake and return the server's result (its protocol version, capabilities, and serverInfo). Per the MCP lifecycle, this also sends the required `notifications/initialized` once the handshake succeeds, so the session is ready for further requests. Throws `Error{kind: "jsonrpc"}` on any transport or protocol failure.

Parameters

- `client {Client}` - the MCP client

Returns `{json.Value}` - the server's `initialize` result

mcp.listPrompts(client as Client)

List the remote server's prompts. Returns the `prompts` array from the `prompts/list` result.

Parameters

- `client {Client}` - the MCP client

Returns `{json.Value}` - the array of prompt descriptors

mcp.listResources(client as Client)

List the remote server's resources. Returns the `resources` array from the `resources/list` result.

Parameters

- `client {Client}` - the MCP client

Returns `{json.Value}` - the array of resource descriptors

mcp.listTools(client as Client)

List the remote server's tools. Returns the `tools` array from the `tools/list` result.

Parameters

- `client` `{Client}` - the MCP client

Returns `{json.Value}` - the array of tool descriptors

mcp.promptArg(name as string, description as string, required as bool)

Build one prompt argument declaration for `addPrompt` .

Parameters

- `name` `{string}` - the argument's name
- `description` `{string}` - a description of the argument
- `required` `{bool}` - whether the host must supply it

Returns `{PromptArg}` - the argument declaration

mcp.property(schema as json.Value, name as string, jsonType as string, description as string, required as bool)

Add a property to a schema, returning a fresh schema. Sets `properties/<name>` = `{type, description}` and appends `name` to `required` when `required` is true.

Parameters

- `schema` `{json.Value}` - the schema to extend
- `name` `{string}` - the property name
- `jsonType` `{string}` - the JSON Schema type (`"string"` , `"integer"` , ...)
- `description` `{string}` - a description of the property
- `required` `{bool}` - whether the property is required

Returns `{json.Value}` - the extended schema

mcp.readResource(client as Client, uri as string)

Read a remote resource by URI. Returns the whole `resources/read` result (a `contents` array).

Parameters

- `client` `{Client}` - the MCP client
- `uri` `{string}` - the resource URI

Returns `{json.Value}` - the `resources/read` result

mcp.schema()

Build an empty JSON Schema object skeleton (`type: object`) for a tool's `inputSchema` . Add fields with `property` .

Returns `{json.Value}` - a `{"type":"object","properties":{},"required":[]}` skeleton

mcp.serveStdio(server as Server)

Run the stdio MCP server: read newline-delimited JSON-RPC requests from the program's own stdin, dispatch each through `handle` , and write each non-empty reply to stdout. Blank lines are skipped; the loop ends at EOF. This is the primary MCP transport (an LLM host launches the program and speaks to it over stdin/stdout).

Parameters

- `server` `{Server}` - the server whose registries route each request

mcp.server(name as string, version as string)

Build an empty server with the given identity.

Parameters

- `name` `{string}` - the server name
- `version` `{string}` - the server version

Returns `{Server}` - a server with no registered items

Structs

mcp.Client

An MCP client. Over HTTP (`connect` / `connectWith`) it wraps a `jsonrpc.Client` ; over stdio (`connectStdio`) it holds the server's `argv` and drives it as a subprocess. The same call surface (`initialize` / `listTools` / `callTool` / ...) works for both transports.

Field	Type	Description
rpc	jsonrpc.Client	the underlying JSON-RPC HTTP client (HTTP transport)
argv	list of string	the stdio server command line (stdio transport)
isStdio	bool	true for the stdio transport, false for HTTP

mcp.Prompt

A prompt template the server exposes. Its `handler` is a `func` value `func NAME(arguments as json.Value)` returning the prompt messages as a json.Value; `arguments` declares what a host should collect (surfaced by `prompts/list`).

Field	Type	Description
name	string	the prompt's unique name (the prompts/get key)
description	string	a human / model readable description
arguments	list of PromptArg	the declared arguments (may be empty)
handler	func	the handler func value that builds the messages

mcp.PromptArg

One argument a prompt declares, so a host can collect it before `prompts/get` . Build with `promptArg` .

Field	Type	Description
name	string	the argument's name
description	string	a human / model readable description
required	bool	whether the host must supply the argument

mcp.Resource

A resource the server exposes. Its `handler` is a `func` value `func NAME(uri as json.Value)` returning the resource's text.

Field	Type	Description
uri	string	the resource's unique URI (the resources/read key)
name	string	a short display name
description	string	a human / model readable description
mimeType	string	the content type of the resource's text
handler	func	the handler func value that produces the content

mcp.Server

An MCP server: an identity plus registries of tools, resources, and prompts. Value-semantic; build with `server` and the `add*` functions.

Field	Type	Description
name	string	the server name reported in initialize
version	string	the server version reported in initialize
tools	list of Tool	the registered tools
resources	list of Resource	the registered resources
prompts	list of Prompt	the registered prompts

mcp.Tool

A tool the server exposes. Its `handler` is a `func` value `func NAME(args as json.Value)` in the entry program that runs when the tool is called.

Field	Type	Description
name	string	the tool's unique name (the tools/call key)
description	string	a human / model readable description
handler	func	the handler func value to dispatch the call to
inputSchema	json.Value	the JSON Schema for the tool's arguments

memcache API reference

A client for a memcached server, speaking its classic text protocol over the `net` system library. Store with an expiration (`set` / `add`), read (`get`), remove (`delete`), count atomically (`incr` / `decr`), and re-arm a key's expiry (`touch`). memcached is a volatile cache - keys expire on their `exptime` and the server evicts under memory pressure - so it suits sessions, rate limits, and derived data, not a system of record. `exptime` is seconds (0 = never expire, until evicted). A protocol error (`ERROR` / `CLIENT_ERROR` / `SERVER_ERROR`) throws a catchable `Error` (kind "memcache"). The value block is framed over bytes by the server's byte count, so a value whose byte length differs from its rune length (any non-ASCII UTF-8 text) round-trips byte-exact. Needs the default `jennifer` binary (uses `net`).

Import with `import "memcache.j" as memcache;` . See the `memcache` guide for prose and examples.

Functions

memcache.add(session as Session, key as string, value as string, exptime as int)

Store `value` at `key` only if the key is absent. The atomic build block for locks and "create if new".

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to write
- `value` `{string}` - the value to store
- `exptime` `{int}` - the TTL in seconds (0 = never expire, until evicted)

Returns `{bool}` - whether it was stored (false when the key already exists)

Throws

- `{Error}` - kind "memcache" on an unexpected reply

memcache.cas(session as Session, key as string, value as string, exptime as int, casId as int)

Check-and-set: store `value` at `key` only if its CAS token still matches `casId` (from a prior `gets`) - i.e. only if no one else changed it meanwhile. The optimistic-concurrency primitive: `gets` , compute a new value, `cas` , and retry on `"exists"` . Returns `"stored"` (success), `"exists"` (someone else changed it - retry), or `"not_found"` (the key is gone / expired).

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to write
- `value` `{string}` - the new value
- `exptime` `{int}` - the TTL in seconds (0 = never expire, until evicted)
- `casId` `{int}` - the CAS token from a prior `gets`

Returns `{string}` - "stored", "exists", or "not_found"

Throws

- `{Error}` - kind "memcache" on an unexpected reply

memcache.connect(opts as Options)

Open a session to the memcached server.

Parameters

- `opts` `{Options}` - the connection settings

Returns `{Session}` - the open session

memcache.decr(session as Session, key as string, delta as int)

Atomically subtract `delta` from the counter at `key` (not below 0).

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the counter key
- `delta` `{int}` - the amount to subtract

Returns `{int}` - the new value, or -1 when the key is absent

memcache.delete(session as Session, key as string)

Remove `key` .

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to remove

Returns `{bool}` - whether the key existed

Throws

- `{Error}` - kind "memcache" on an unexpected reply

memcache.get(session as Session, key as string)

Return the string value of `key` , or "" when the key is absent / expired.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to read

Returns `{string}` - the value, or "" when absent / expired

memcache.getBytes(session as Session, key as string)

Return the raw `bytes` value of `key` , or empty `bytes` when absent / expired (the binary counterpart to `get`). It never UTF-8-decodes, so a value stored with `setBytes` round-trips exactly.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to read

Returns `{bytes}` - the value, or empty bytes when absent / expired

memcache.getMulti(session as Session, keys as list of string)

Fetch several keys in one round-trip (multi-key `get`) as a `map of string to string` ; a missing / expired key is simply absent from the map, so `maps.has` distinguishes it. Cheaper than N separate `get` s.

Parameters

- `session` `{Session}` - the open session
- `keys` `{list of string}` - the keys to read

Returns `{map of string to string}` - the found key -> value pairs

memcache.gets(session as Session, key as string)

Read `key` with its CAS token (`gets`), for a check-and-set update. The returned `Item` carries the `value` , the `cas` token, and whether the key was `found` ; pass its `cas` to `cas` so the store only succeeds if no one else changed the value meanwhile.

Parameters

- `session {Session}` - the open session
- `key {string}` - the key to read

Returns `{Item}` - the value, CAS token, and found flag

memcache.incr(session as Session, key as string, delta as int)

Atomically add `delta` to the counter at `key` .

Parameters

- `session {Session}` - the open session
- `key {string}` - the counter key
- `delta {int}` - the amount to add

Returns `{int}` - the new value, or -1 when the key is absent

memcache.quit(session as Session)

End the session and close the connection.

Parameters

- `session {Session}` - the open session

memcache.set(session as Session, key as string, value as string, exptime as int)

Store `value` at `key` with an `exptime` -second TTL, replacing any existing value.

Parameters

- `session {Session}` - the open session
- `key {string}` - the key to write
- `value {string}` - the value to store
- `exptime {int}` - the TTL in seconds (0 = never expire, until evicted)

Throws

- `{Error}` - kind "memcache" on a non-STORED reply

memcache.setBytes(session as Session, key as string, value as bytes, exptime as int)

Store a raw `bytes` value at `key` , byte-for-byte (the binary counterpart to `set`). A serialized blob, a compressed payload, or an image round-trips exactly; read it back with `getBytes` , not `get` (the string reader throws on a non-UTF-8 value).

Parameters

- `session {Session}` - the open session
- `key {string}` - the key to write
- `value {bytes}` - the raw value to store
- `exptime {int}` - the TTL in seconds (0 = never expire, until evicted)

Throws

- `{Error}` - kind "memcache" on a non-STORED reply

memcache.touch(session as Session, key as string, exptime as int)

Re-arm `key` 's expiry to `exptime` seconds.

Parameters

- `session {Session}` - the open session
- `key {string}` - the key to re-arm
- `exptime {int}` - the new TTL in seconds (0 = never expire, until evicted)

Returns `{bool}` - whether the key existed

Throws

- `{Error}` - kind "memcache" on an unexpected reply

Structs

memcache.Item

One value read by `gets` : its string value, the server's CAS token (an opaque version counter), and whether the key was found. Pass `cas` to `cas` for a check-and-set that only stores if no one else changed the value meanwhile.

Field	Type	Description
value	string	the value ("" when not found)
cas	int	the CAS token to hand to cas (0 when not found)
found	bool	whether the key was present

memcache.Options

Connection settings (plaintext; memcached's text protocol has no auth / TLS).

Field	Type	Description
host	string	the server host
port	int	the server port

memcache.Session

An open memcached connection.

Field	Type	Description
conn	net.Conn	the underlying socket
timeout	int	per-read idle timeout in milliseconds (0 disables it)

mikrotik API reference

A MikroTik RouterOS API client over `net` (not SSH). Connect to a router's binary API (plain TCP 8728, or api-ssl 8729 over TLS), log in, and run commands. The wire protocol is sentence-based: a sentence is a run of length-prefixed words ending in a zero-length word; `talk` sends a command sentence (`/interface/print`) with `=key=value` attribute words and folds each `!re` reply sentence into a row map. `talkQuery` / `printWhere` add `?...` query words to filter rows on the router. `print` is read sugar, `run` is for mutating commands (returning the `!done` 's `=ret=` , e.g. a new item id).

Login is plaintext (`=name=` / `=password=` , RouterOS 6.43+ and all v7), with an automatic MD5 challenge-response fallback for pre-6.43 routers. A `!trap` or `!fatal` reply throws `Error{kind: "mikrotik"}` . Needs the default `jennifer` binary (`net`). Over `net` + `hash` (MD5 fallback) + `encoding` + the bitwise operators.

SECURITY: `mikrotik.options` uses **cleartext** TCP (port 8728), so the credentials and every command are readable on the wire. Use `mikrotik.optionsTLS` (api-ssl, port 8729) on any untrusted network. A `CONNECT_TIMEOUT_MS` deadline bounds the dial and every read, and a server-declared word length is capped at `MAX_WORD_BYTES` (64 MiB), so a wedged or hostile router cannot hang or OOM the program.

Import with `import "mikrotik.j" as mikrotik;` . See the `mikrotik` guide for prose and examples.

Functions

mikrotik.cancel(session as Session, tag as string)

Stop a streaming command started with `listen` by issuing `/cancel` naming its tag. Fire-and-forget: the resulting `!trap` (interrupted) and `!done` for the stream, plus the `/cancel` 's own reply, are drained by the `receiveReply` loop (which then returns {} and lets the loop exit).

Parameters

- `session {Session}` - the session
- `tag {string}` - the stream tag returned by `listen`

mikrotik.close(session as Session)

Close the session's connection.

Parameters

- `session {Session}` - the session

mikrotik.connect(opts as Options)

Connect to a router and log in.

Parameters

- `opts {Options}` - the connection options

Returns `{Session}` - the logged-in session

Throws

- `{Error}` - kind "mikrotik" on a login failure

mikrotik.listen(session as Session, command as string, params as map of string to string)

Start a **streaming** command (one that pushes `!re` sentences over time, e.g. "/interface/monitor" or "/ping") and return its auto-generated tag. Does NOT read a reply - the stream has no `!done` until it is stopped. Pull each pushed reply with `receiveReply($s, tag)` (typically from a `spawn` ed loop), and stop it with `cancel($s,`

tag) .

Parameters

- `session {Session}` - the session
- `command {string}` - the streaming API command
- `params {map of string to string}` - attribute words for the command ({} for none)

Returns `{string}` - the tag naming this stream (feed it to receiveReply / cancel)

mikrotik.options(host as string, user as string, password as string)

Plain-TCP options (port 8728). SECURITY: this transport is **cleartext** - the login exchange and every command travel unencrypted, so an on-path attacker reads the router's admin credentials. Prefer `optionsTLS` (api-ssl, port 8729) on any network you do not fully trust; use `options` only on a loopback or an otherwise-secured link.

Parameters

- `host {string}` - the router host
- `user {string}` - the API username
- `password {string}` - the API password

Returns `{Options}` - the options

mikrotik.optionsTLS(host as string, user as string, password as string)

api-ssl (TLS) options (port 8729).

Parameters

- `host {string}` - the router host
- `user {string}` - the API username
- `password {string}` - the API password

Returns `{Options}` - the options

mikrotik.print(session as Session, path as string)

Read sugar: run `path` + `"/print"` with no attributes.

Parameters

- `session {Session}` - the session
- `path {string}` - the menu path (e.g. "/interface")

Returns `{list of map of string to string}` - the reply rows

Throws

- `{Error}` - kind "mikrotik" on a !trap / !fatal reply

mikrotik.printWhere(session as Session, path as string, queries as list of string)

Read sugar: run `path` + `"/print"` filtered by query words (no attributes) - `printWhere($s, "/interface", ["?type=ether"])` returns only the ether ports.

Parameters

- `session {Session}` - the session
- `path {string}` - the menu path (e.g. "/interface")
- `queries {list of string}` - raw query words, each starting with "?"

Returns `{list of map of string to string}` - the matching reply rows

Throws

- `{Error}` - kind "mikrotik" on a bad query word or a !trap / !fatal reply

mikrotik.receiveReply(session as Session, tag as string)

Block until the next pushed reply for `tag` arrives and return it as a `=key=value` field map. Returns an **empty map** when the stream has ended (a `!done` for the tag - e.g. after `cancel`), which is the loop's stop signal. Replies carrying a different tag are skipped (they belong to another stream on the same session). A `cancel` -induced `!trap` (category "interrupted") is absorbed; any other `!trap` / `!fatal` throws.

Parameters

- `session` `{Session}` - the session
- `tag` `{string}` - the stream tag returned by `listen`

Returns `{map of string to string}` - the next reply's fields, or {} at stream end

Throws

- `{Error}` - kind "mikrotik" on a genuine !trap / !fatal reply

mikrotik.run(session as Session, command as string, attrs as map of string to string)

Run a mutating command (add / set / remove) and return the `!done` `=ret=` value (e.g. the new item's id for `add` ; "" when there is none).

Parameters

- `session` `{Session}` - the session
- `command` `{string}` - the API command (e.g. "/ip/address/add")
- `attrs` `{map of string to string}` - attribute words

Returns `{string}` - the `=ret=` value, or ""

Throws

- `{Error}` - kind "mikrotik" on a !trap / !fatal reply

mikrotik.talk(session as Session, command as string, attrs as map of string to string)

Run a command with attribute words and return the `!re` reply rows (each a `=key=value` map). The general call - reads, adds, sets, and removes all go through it.

Parameters

- `session` `{Session}` - the session
- `command` `{string}` - the API command (e.g. "/interface/print")
- `attrs` `{map of string to string}` - attribute words ({} for none)

Returns `{list of map of string to string}` - the reply rows

Throws

- `{Error}` - kind "mikrotik" on a !trap / !fatal reply

mikrotik.talkQuery(session as Session, command as string, attrs as map of string to string, queries as list of string)

Like `talk` , but also sends RouterOS **query words** to filter the reply on the router. Each query word is raw and starts with `?` : `?type=ether` (property equals), `?disabled` (property present), `?>mtu=1000` (greater), plus the stack operators `?#!` / `?#&` / `?#|` for compound queries. Attribute words still apply (e.g. `=.proplist=name,type` to trim the columns). The general filtered read the higher-level helpers build on.

Parameters

- `session` `{Session}` - the session
- `command` `{string}` - the API command (e.g. "/interface/print")
- `attrs` `{map of string to string}` - attribute words ({} for none)
- `queries` `{list of string}` - raw query words, each starting with "?"

Returns `{list of map of string to string}` - the matching reply rows

Throws

- `{Error}` - kind "mikrotik" on a bad query word or a !trap / !fatal reply

mikrotik.talkTagged(session as Session, command as string, attrs as map of string to string, tag as string)

Like `talk` , but attaches a `.tag=<id>` word so the router echoes the tag on every reply, letting a caller correlate this command's replies. Pass `tag: ""` to auto-generate a unique tag. The reply is still collected to `!done` (a one-shot command); for a command that pushes replies over time use `listen` + `receiveReply` .

Parameters

- `session` `{Session}` - the session
- `command` `{string}` - the API command (e.g. "/interface/print")
- `attrs` `{map of string to string}` - attribute words ({} for none)
- `tag` `{string}` - the correlation tag, or "" to auto-generate one

Returns `{list of map of string to string}` - the reply rows

Throws

- `{Error}` - kind "mikrotik" on a !trap / !fatal reply

mikrotik.withPort(o as Options, port as int)

Copy options with a different port.

Parameters

- o `{Options}` - the options
- `port` `{int}` - the port

Returns `{Options}` - a fresh options

Structs

mikrotik.Options

Connection options.

Field	Type	Description
host	string	the router host
port	int	the API port (8728 plain, 8729 api-ssl)
user	string	the API username
password	string	the API password
tls	bool	whether to use api-ssl (TLS)

mikrotik.Session

An open, logged-in API session.

Field	Type	Description

socket	net.Conn	the underlying (plain or TLS) connection

mime API reference

Build and parse MIME messages (RFC 5322 headers + RFC 2045/2046 bodies): a header-and-boundary structure that is exactly the text orchestration a `.j` module does well. The transfer codecs (base64, quoted-printable) are delegated to the `encoding` system library. This is the message-structure foundation the mail clients (SMTP / POP3 / IMAP) build on; it does no networking itself. A `Part` is a leaf (headers + a decoded `body` / `data` with a transfer `encoding`) or a multipart container (headers + child `parts` + a `boundary`); bodies are held decoded as text, encode applies the transfer encoding and parse removes it. RFC 2047 encoded-words are applied automatically on encode and decoded on parse, with `encodeWord` / `decodeWord` exposed for manual use. Every leaf carries its raw decoded `data` (`bytes`), so binary attachments round-trip; `body` is the text view, populated for text parts and decoded per the Content-Type `charset` (UTF-8 by default; ISO-8859 / Windows single-byte codecs are honoured, with a UTF-8 fallback for an unknown label). Attachment filenames are read and written in RFC 2231 extended form (`filename*=`) when they carry non-ASCII characters. Use `walk` / `attachments` / `textBodies` to pull parts out of a received message.

Import with `import "mime.j" as mime;` . See the `mime` guide for prose and examples.

Functions

mime.address(name as string, email as string)

Format an RFC 5322 mailbox: `email` alone, or `Name <email>` with the name quoted when it carries a special character.

Parameters

- `name` `{string}` - the display name (empty for a bare address)
- `email` `{string}` - the email address

Returns `{string}` - the formatted mailbox

mime.attachment(filename as string, contentType as string, body as string)

Build a base64 leaf part with a filename. `body` is text content (UTF-8); binary bodies are not yet supported.

Parameters

- `filename` `{string}` - the attachment filename
- `contentType` `{string}` - the media type
- `body` `{string}` - the text content to attach

Returns `{Part}` - the attachment part

mime.attachmentBytes(filename as string, contentType as string, data as bytes)

Build a base64 attachment part from raw `bytes` (images, PDFs, any binary). The binary counterpart to `attachment` ; `encode` base64-wraps the data and `parse` restores it, so it round-trips intact.

Parameters

- `filename` `{string}` - the attachment filename
- `contentType` `{string}` - the media type (e.g. "image/png")
- `data` `{bytes}` - the raw file content

Returns `{Part}` - the attachment part

mime.attachments(part as Part)

Return every attachment leaf in the message (see `isAttachment`). Each part's bytes are `data($part)` and its name `filename($part)`.

Parameters

- `part {Part}` - the parsed message (or any subtree)

Returns `{list of Part}` - the attachment parts

mime.body(part as Part)

Return a leaf part's decoded text body.

Parameters

- `part {Part}` - the leaf part

Returns `{string}` - the decoded body text

mime.contentType(part as Part)

Return a part's media type without parameters (e.g. "text/plain").

Parameters

- `part {Part}` - the part to read

Returns `{string}` - the media type

mime.data(part as Part)

Return a leaf part's raw decoded content bytes (any type, including binary attachments). This is the byte-accurate counterpart to `body`.

Parameters

- `part {Part}` - the leaf part

Returns `{bytes}` - the raw decoded content

mime.decodeWord(value as string)

Decode every RFC 2047 encoded-word in `value`, dropping the linear whitespace that separates two adjacent encoded-words (as a reader should). A word that fails to decode is left verbatim so parse never crashes.

Parameters

- `value {string}` - the header value possibly carrying encoded-words

Returns `{string}` - the decoded text

mime.disposition(part as Part)

Return a part's Content-Disposition (lowercased, without parameters): "attachment", "inline", or "" when the header is absent.

Parameters

- `part {Part}` - the part to read

Returns `{string}` - the disposition

mime.encode(part as Part)

Serialize a part (and its subtree) to a MIME message string with CRLF line endings and the declared transfer encodings applied.

Parameters

- `part {Part}` - the part to serialize

Returns `{string}` - the encoded MIME message

mime.encodeWord(text as string)

Render `text` as one or more RFC 2047 UTF-8 base64 encoded-words, each kept under the 75-character limit (split on rune boundaries, never mid-character) and folded with CRLF + space when more than one is needed.

Parameters

- `text {string}` - the text to encode

Returns `{string}` - the encoded-word sequence

mime.filename(part as Part)

Return a part's filename: the Content-Disposition `filename` parameter, else the Content-Type `name` parameter. RFC 2231 extended / continued forms (`filename*=` , `filename*0=` ...) and RFC 2047 encoded-words are both decoded; "" when neither is present.

Parameters

- `part {Part}` - the part to read

Returns `{string}` - the filename

mime.findParts(part as Part, mediaType as string)

Return every leaf whose media type equals `mediaType` (e.g. "text/html").

Parameters

- `part {Part}` - the parsed message (or any subtree)
- `mediaType {string}` - the exact media type to match (case-insensitive)

Returns `{list of Part}` - the matching leaf parts

mime.headerValue(part as Part, name as string)

Return a part's header value (case-insensitive) or "".

Parameters

- `part {Part}` - the part to read
- `name {string}` - the header name

Returns `{string}` - the header value, or "" when absent

mime.isAttachment(part as Part)

Report whether a part is an attachment: Content-Disposition is "attachment", or a filename is set.

Parameters

- `part {Part}` - the part to read

Returns `{bool}` - true when the part is an attachment

mime.multipart(subtype as string, boundary as string, parts as list of Part)

Build a container part; every child ships under one `boundary` .

Parameters

- `subtype {string}` - the multipart subtype (e.g. "mixed", "alternative")
- `boundary {string}` - the boundary delimiter separating children
- `parts {list of Part}` - the child parts

Returns {Part} - the multipart container part

mime.parse(text as string)

Read a MIME message string into a Part tree: headers are unfolded, a multipart body is split on its boundary (recursively), and a leaf body is transfer-decoded.

Parameters

- text {string} - the MIME message text

Returns {Part} - the parsed part tree

mime.parts(part as Part)

Return a multipart container's child parts.

Parameters

- part {Part} - the multipart container

Returns {list of Part} - the child parts

mime.text(contentType as string, body as string)

Build a leaf text part; the body is sent 7bit when ASCII, else quoted-printable. charset=utf-8 is appended to the content type.

Parameters

- contentType {string} - the media type (e.g. "text/plain")
- body {string} - the decoded text body

Returns {Part} - the leaf text part

mime.textBodies(part as Part)

Return the readable text bodies: text parts (any text/ subtype) that are not attachments (so both the text/plain and text/html alternatives of a message).

Parameters

- part {Part} - the parsed message (or any subtree)

Returns {list of Part} - the text body parts

mime.walk(part as Part)

Flatten a part tree to its leaf parts (depth-first). A leaf is a part with no child parts ; the building block for attachments / textBodies .

Parameters

- part {Part} - the part (leaf or container) to walk

Returns {list of Part} - every leaf part, in document order

mime.withHeader(part as Part, name as string, value as string)

Return a copy of the part with name set (replaced or appended).

Parameters

- part {Part} - the part to copy
- name {string} - the header name to set
- value {string} - the header value

Returns {Part} - the updated copy

Structs

mime.Header

A single header field.

Field	Type	Description
name	string	the field name (e.g. "Subject")
value	string	the field value

mime.Part

A MIME part: a leaf (`data` / `body` + `encoding` , `parts` empty) or a multipart container (`parts` + `boundary` , `body` / `data` empty).

Field	Type	Description
headers	list of Header	the part's header fields
body	string	the decoded text body of a text part ("" for a binary part)
encoding	string	the transfer encoding ("7bit", "base64", "quoted-printable")
parts	list of Part	the child parts of a multipart container
boundary	string	the multipart boundary delimiter
data	bytes	the raw decoded content of a leaf (any type, incl. binary)

mqtt API reference

An MQTT 3.1.1 publish/subscribe client over the `net` system library - the same "protocol clients are modules, `net` is the transport" line the other network clients follow. MQTT packets are a 1-byte fixed header, a variable remaining-length integer, then a length-prefixed payload, all built and parsed here with Jennifer's bitwise operators and `bytes` . QoS 0 and QoS 1 publish / subscribe (a synchronous PUBACK handshake), retained messages, a Last-Will set in the CONNECT, and `reconnect` for session resumption, on top of a single-threaded `poll` with timeout (via `net.setDeadline`) so one flow can wait for a packet and send keepalives without a spawned reader, plus blocking `receive` , `ping` , and `disconnect` . QoS 2 and MQTT 5 properties are out of scope. Needs the default `jennifer` binary (uses `net`).

Import with `import "mqtt.j" as mqtt;` . See the `mqtt` guide for prose and examples.

Functions

mqtt.connect(opts as Options)

Open a connection, send CONNECT, and check the CONNACK return code. Uses a clean session and no Last-Will; for a will or session resumption use `connectWith` .

Parameters

- `opts {Options}` - the connection settings

Returns `{Client}` - the open client

Throws

- `{Error}` - kind "mqtt" when the broker refuses the connection

mqtt.connectWith(opts as Options, will as Will, cleanSession as bool)

Open a connection with an explicit Last-Will and clean-session flag. A `will` whose topic is "" registers no will; `cleanSession` false asks the broker to resume a persistent session keyed by `opts.clientId` (retained subscriptions and queued QoS-1 messages).

Parameters

- `opts {Options}` - the connection settings
- `will {Will}` - the Last-Will to register ("" topic disables it)
- `cleanSession {bool}` - false to resume a persistent session

Returns `{Client}` - the open client

Throws

- `{Error}` - kind "mqtt" when the broker refuses the connection

mqtt.disconnect(client as Client)

Send DISCONNECT and close the connection.

Parameters

- `client {Client}` - the open client

mqtt.ping(client as Client)

Send a PINGREQ keepalive (fire and forget). The matching PINGRESP is consumed by the next `poll` / `receive` .

Parameters

- `client {Client}` - the open client

mqtt.poll(client as Client, timeoutMs as int)

Poll for a message, waiting up to timeoutMs milliseconds. Returns a list of zero or one Message: empty when nothing arrived in the window (the caller can then `ping` and loop), one Message when a PUBLISH was received. A QoS-1 PUBLISH is acknowledged with a PUBACK before it is returned. Non-PUBLISH control packets are consumed and reported as an empty poll.

Parameters

- `client {Client}` - the open client
- `timeoutMs {int}` - how long to wait for the next packet, in milliseconds

Returns `{list of Message}` - zero or one received message

mqtt.publish(client as Client, topic as string, message as string)

Publish a text message to a topic at QoS 0 (UTF-8 encoded).

Parameters

- `client {Client}` - the open client
- `topic {string}` - the topic to publish to
- `message {string}` - the message text

mqtt.publishBytes(client as Client, topic as string, payload as bytes)

Publish a raw byte payload to a topic at QoS 0 (fire and forget).

Parameters

- `client {Client}` - the open client
- `topic {string}` - the topic to publish to
- `payload {bytes}` - the message bytes

mqtt.publishBytesRetain(client as Client, topic as string, payload as bytes, retain as bool)

Publish a raw byte payload at QoS 0 with an explicit retain flag. A retained message is stored by the broker and delivered to every future subscriber of the topic; publishing an empty payload with retain true clears it.

Parameters

- `client {Client}` - the open client
- `topic {string}` - the topic to publish to
- `payload {bytes}` - the message bytes
- `retain {bool}` - whether the broker retains the message

mqtt.publishQos1(client as Client, topic as string, payload as bytes, retain as bool)

Publish a raw byte payload at QoS 1 (at-least-once) and block until the matching PUBACK. The PUBLISH carries a packet identifier; an unacknowledged send is retried as a DUP (the duplicate-delivery flag set) up to a fixed number of attempts, so at-least-once delivery is honored across a slow or flapping link. Because the call blocks for the PUBACK, only one message is ever in flight, so a fixed non-zero packet identifier is safe. A duplicate the broker already delivered is de-duplicated by the packet id on its side.

Parameters

- `client {Client}` - the open client
- `topic {string}` - the topic to publish to
- `payload {bytes}` - the message bytes
- `retain {bool}` - whether the broker retains the message

Throws

- `{Error}` - kind "mqtt" when no PUBACK arrives within the retry budget

mqtt.publishRetain(client as Client, topic as string, message as string, retain as bool)

Publish a text message at QoS 0 with an explicit retain flag (UTF-8 encoded).

Parameters

- `client {Client}` - the open client
- `topic {string}` - the topic to publish to
- `message {string}` - the message text
- `retain {bool}` - whether the broker retains the message

mqtt.receive(client as Client)

Block until the next application message arrives, returning it. A QoS-1 PUBLISH is acknowledged with a PUBACK before it is returned. Non-PUBLISH control packets (e.g. a PINGRESP) are consumed and skipped.

Parameters

- `client {Client}` - the open client

Returns `{Message}` - the next received message

mqtt.reconnect(client as Client)

Re-dial the broker and re-CONNECT with this client's stored settings, then re-subscribe every tracked subscription. Returns a fresh Client the caller reassigns (value semantics - the old handle is best-effort closed).

Session resumption: when the client connected with `cleanSession` false, the broker keeps the session (subscriptions and queued QoS-1 messages) across the drop and reports it in the CONNACK; the re-subscribe here is idempotent and also covers the clean-session case, where the broker discarded the session.

Parameters

- `client {Client}` - the (typically disconnected) client to re-establish

Returns `{Client}` - a freshly connected client with the tracked routes restored

Throws

- `{Error}` - kind "mqtt" when the re-connect or a re-subscribe fails

mqtt.subscribe(client as Client, topic as string)

Subscribe to a topic filter at QoS 0, wait for the SUBACK, and track the subscription on the returned Client (so `reconnect` can restore it). Reassign the result: `$c = mqtt.subscribe($c, "topic");` .

Parameters

- `client {Client}` - the open client
- `topic {string}` - the topic filter (may contain + / # wildcards)

Returns `{Client}` - the client with the subscription tracked

Throws

- `{Error}` - kind "mqtt" when the broker rejects the subscription

mqtt.subscribeQos1(client as Client, topic as string)

Subscribe to a topic filter requesting QoS 1, wait for the SUBACK, and track the subscription (with the QoS the broker granted) on the returned Client. With QoS 1 the broker delivers each matching message as a QoS-1

PUBLISH, which `receive` / `poll` acknowledge with a PUBACK. Reassign the result: `$c = mqtt.subscribeQos1($c, "topic");` .

Parameters

- `client` `{Client}` - the open client
- `topic` `{string}` - the topic filter (may contain + / # wildcards)

Returns `{Client}` - the client with the subscription tracked

Throws

- `{Error}` - kind "mqtt" when the broker rejects the subscription

Structs

mqtt.Client

An open MQTT connection. Beyond the socket it carries the settings needed to re-establish the session (`opts` , `will` , `cleanSession`) and the list of subscriptions `subscribe` / `subscribeQos1` have tracked, so `reconnect` can re-dial and re-subscribe. Value-semantic: copies share the socket handle but each carries its own settings and subscription list.

Field	Type	Description
conn	net.Conn	the underlying socket
opts	Options	the settings this client connected with
will	Will	the Last-Will registered in the CONNECT ("" topic = none)
cleanSession	bool	the clean-session flag (false resumes the session)
subs	list of Subscription	the tracked subscriptions for reconnect

mqtt.Message

One received application message.

Field	Type	Description
topic	string	the topic it was published to
payload	bytes	the raw message bytes (convert to text as needed)

mqtt.Options

Connection settings for mqtt.connect.

Field	Type	Description
host	string	the broker host
port	int	the broker port (1883 plaintext, 8883 TLS by convention)
clientId	string	the client identifier the broker sees
keepalive	int	the keepalive interval in seconds (0 disables)
security	transport.Security	transport.Security.None (plaintext) or .Tls (mqtts); .Starttls is rejected (MQTT has no in-band upgrade)
username	string	the CONNECT username ("" to omit)
password	string	the CONNECT password ("" to omit)

mqtt.Subscription

One tracked subscription, remembered on the Client so `reconnect` can restore the session's routes.

Field	Type	Description
topic	string	the subscribed topic filter
qos	int	the QoS the broker granted (0x80 means it was rejected)

mqtt.Will

A Last-Will-and-Testament message the broker publishes on this client's behalf if the connection drops without a clean DISCONNECT. An empty `topic` means "no will" (the CONNECT sets no will flags).

Field	Type	Description
topic	string	the will topic ("" disables the will)
payload	bytes	the will message bytes
qos	int	the will QoS (0 or 1)
retain	bool	whether the broker retains the will message

multipart API reference

Build and parse `multipart/form-data` bodies (RFC 7578) - the file-upload counterpart to `mime` 's email multipart. `build` assembles a list of `Part` s (form fields and files) into a `(contentType, body)` pair ready to POST; `parse` takes a `Content-Type` header and a body and returns the parts. Bodies are `bytes` , so binary file content round-trips intact.

Pairs with `web` / `httpd` for handling uploads and with `http` for sending them. Pure `.j` over `strings` + `bytes` ; runs on both binaries.

Import with `import "multipart.j" as multipart;` . See the `multipart` guide for prose and examples.

Functions

multipart.build(parts as list of Part)

Build a form body with a fresh random boundary.

Parameters

- `parts` `{list of Part}` - the parts

Returns `{Built}` - the Content-Type and encoded body

multipart.buildWith(parts as list of Part, boundary as string)

Build a form body with an explicit boundary (deterministic).

Parameters

- `parts` `{list of Part}` - the parts
- `boundary` `{string}` - the boundary token (must not occur in any part body)

Returns `{Built}` - the Content-Type and encoded body

multipart.field(name as string, value as string)

A plain text form field.

Parameters

- `name` `{string}` - the field name
- `value` `{string}` - the field value

Returns `{Part}` - the part

multipart.file(name as string, filename as string, contentType as string, data as bytes)

A file part.

Parameters

- `name` `{string}` - the field name
- `filename` `{string}` - the file name
- `contentType` `{string}` - the file's Content-Type (e.g. "text/plain")
- `data` `{bytes}` - the file content

Returns `{Part}` - the part

multipart.isFile(p as Part)

Report whether a part is a file (has a filename).

Parameters

- `p` `{Part}` - the part

Returns `{bool}` - true if the part carries a filename

multipart.parse(contentType as string, body as bytes)

Parse a `multipart/form-data` body into its parts.

Parameters

- `contentType` `{string}` - the Content-Type header (carrying the boundary)
- `body` `{bytes}` - the encoded body

Returns `{list of Part}` - the parts (fields and files)

Throws

- `{Error}` - kind "multipart" if the boundary is missing or a part is malformed

multipart.text(p as Part)

Decode a part's body as a UTF-8 string (for reading text field values).

Parameters

- `p` `{Part}` - the part

Returns `{string}` - the decoded body

Structs

multipart.Built

A built form body plus the matching Content-Type header (with the boundary).

Field	Type	Description
contentType	string	the multipart/form-data; boundary=... value
body	bytes	the encoded body

multipart.Part

One form part: a field or a file.

Field	Type	Description
name	string	the field name
filename	string	the file name ("" for a plain field)
contentType	string	the part's Content-Type ("" for a plain field)
data	bytes	the part body

ntp API reference

A simple SNTP client (RFC 4330 / 5905): query a time server over UDP and get back its time plus the local clock offset and round-trip delay. This is the one-shot query half of NTP - it speaks the standard NTP wire protocol (the 48-byte packet on port 123) but does not discipline the clock or run as a daemon; it reports the measurement and leaves acting on it to the caller.

The packet is packed / unpacked with `bytes` and the bitwise operators, and NTP timestamps (seconds since 1900) are converted to `time.Time` through `time` . Needs the default `jennifer` binary (`net`); a query throws `Error{kind: "ntp"}` on timeout or a short reply.

Import with `import "ntp.j" as ntp;` . See the `ntp` guide for prose and examples.

Functions

ntp.query(host as string)

Query an NTP server by host name (port 123, a 5-second timeout).

Parameters

- `host` `{string}` - the server host (e.g. "pool.ntp.org")

Returns `{Result}` - the server time, clock offset, and round-trip delay

Throws

- `{Error}` - kind "ntp" on timeout or a malformed reply

ntp.queryWith(address as string, timeoutMs as int)

Query an NTP server at a full `host:port` address with a timeout in milliseconds.

Parameters

- `address` `{string}` - the server "host:port"
- `timeoutMs` `{int}` - the receive timeout in milliseconds

Returns `{Result}` - the server time, clock offset, and round-trip delay

Throws

- `{Error}` - kind "ntp" on timeout or a malformed reply

Structs

ntp.Result

The result of a query.

Field	Type	Description
serverTime	time.Time	the server's transmit time
offset	time.Duration	the local clock offset (server minus local)
delay	time.Duration	the measured round-trip delay

oauth API reference

A generic OAuth2 client: the get-a-token half of OAuth2 (the use-a-token half is `sasl` XOAUTH2). Acquires and refreshes access tokens against any OAuth2 token endpoint, over `http` + `json` . Ships the flows that need no extra dependencies: Client Credentials (a service authenticating as itself), Refresh Token (trade a refresh token for a fresh access token), and the Device Authorization Grant (the CLI-friendly flow: show the user a URL + code, poll the token endpoint until they approve). Authorization Code + PKCE and the service-account JWT assertion need a local redirect server and crypto-grade signing, so they land later. Provider presets for Google and Microsoft 365 fill in the endpoints. Because it builds on `http` (which uses `net`), this module needs the default `jennifer` binary. A token-endpoint error surfaces as a catchable `Error` (kind "oauth").

Import with `import "oauth.j" as oauth;` . See the `oauth` guide for prose and examples.

Functions

oauth.clientCredentials(config as Config)

Acquire a token for the client itself (no user).

Parameters

- `config` `{Config}` - the client settings

Returns `{Token}` - the issued access token

Throws

- `{Error}` - kind "oauth" on a token-endpoint error

oauth.deviceStart(config as Config)

Begin the device flow: return the code + URL to show the user.

Parameters

- `config` `{Config}` - the client settings

Returns `{DeviceAuth}` - the device-authorization handle

Throws

- `{Error}` - kind "oauth" on a device-endpoint error

oauth.deviceWait(config as Config, deviceAuth as DeviceAuth)

Poll the token endpoint until the user approves (or a terminal error), returning the token. Sleeps `interval` seconds between polls, backing off on `slow_down` .

Parameters

- `config` `{Config}` - the client settings
- `deviceAuth` `{DeviceAuth}` - the handle from `deviceStart`

Returns `{Token}` - the issued access token

Throws

- `{Error}` - kind "oauth" if device authorization fails

oauth.google(clientId as string, clientSecret as string, scope as string)

Return a Config for Google's OAuth2 endpoints.

Parameters

- `clientId` `{string}` - the OAuth2 client identifier

- `clientSecret` `{string}` - the OAuth2 client secret
- `scope` `{string}` - the space-separated requested scopes

Returns `{Config}` - a Config wired to Google's endpoints

oauth.isExpired(token as Token)

Report whether a token has expired (30s skew buffer).

Parameters

- `token` `{Token}` - the token to check

Returns `{bool}` - true if the token is at or past its expiry

oauth.load(path as string)

Read a token previously written by save.

Parameters

- `path` `{string}` - the file path to read

Returns `{Token}` - the loaded token

oauth.microsoft(tenant as string, clientId as string, clientSecret as string, scope as string)

Return a Config for a Microsoft 365 / Entra tenant's endpoints.

Parameters

- `tenant` `{string}` - the tenant identifier (or "common")
- `clientId` `{string}` - the OAuth2 client identifier
- `clientSecret` `{string}` - the OAuth2 client secret
- `scope` `{string}` - the space-separated requested scopes

Returns `{Config}` - a Config wired to the tenant's endpoints

oauth.refresh(config as Config, refreshToken as string)

Trade a refresh token for a new access token, preserving the refresh token when the server omits it from the reply.

Parameters

- `config` `{Config}` - the client settings
- `refreshToken` `{string}` - the refresh token to redeem

Returns `{Token}` - the new access token

Throws

- `{Error}` - kind "oauth" on a token-endpoint error

oauth.save(path as string, token as Token)

Write a token to a file as JSON (its own field shape, round-trips with load; absolute `expiresAt` is preserved). The file is tightened to owner-only (`0600`) after writing, since an OAuth token is a bearer credential - the same file-permission model `gh` / `aws` use for their token stores, so it is not left world-readable at the write verbs' default `0644` .

Parameters

- `path` `{string}` - the file path to write
- `token` `{Token}` - the token to persist

Structs

oauth.Config

The OAuth2 client settings for one provider / application.

Field	Type	Description
tokenUrl	string	the token endpoint URL
deviceUrl	string	the device-authorization endpoint URL
clientId	string	the OAuth2 client identifier
clientSecret	string	the OAuth2 client secret
scope	string	the space-separated requested scopes

oauth.DeviceAuth

A device-authorization handle: show the user `verificationUri` + `userCode` , then `deviceWait` polls with `deviceCode` every `interval` seconds.

Field	Type	Description
deviceCode	string	the code polled at the token endpoint
userCode	string	the code the user types at the verification URL
verificationUri	string	the URL to show the user
interval	int	seconds to wait between polls
expiresAt	int	Unix timestamp when the device code expires

oauth.Token

An issued token. `expiresAt` is a Unix timestamp (seconds; 0 = no known expiry).

Field	Type	Description
accessToken	string	the bearer access token
tokenType	string	the token type (e.g. "Bearer")
refreshToken	string	the refresh token ("" if none)
scope	string	the scopes granted with this token
expiresAt	int	Unix expiry timestamp in seconds (0 = unknown)

orm API reference

A minimal relational mapper over the `sql` library. It is **Data Mapper, not Active Record** - Jennifer structs are value-semantic and carry no methods, and a module holds no state, so a row cannot `save()` itself. Instead you pass a record and a `Schema` to repository functions: `orm.insert` / `find` / `update` / `delete` , and `orm.all` over a query.

There is no reflection, so you declare the table mapping once as an `orm.Schema` (built with `orm.schema` + `orm.column`), which also carries the SQL **dialect** (`orm.Dialect.Mysql` or `orm.Dialect.Postgres`) - a backend selector on one module, not parallel modules. The query builder is functional (like the `json` write surface): each step returns a fresh `orm.Query` , rendered to **parameterized** SQL by `orm.toSql` . The surface covers ordinary queries: column projection (`select`) and aggregates (`count` / `aggregate`), `where` / `orWhere` / `whereIn` (AND / OR / IN conditions), `join` / `leftJoin` / `rightJoin` , `groupBy` + `having` , `orderBy` / `limit` / `offset` .

Values bind only through placeholders (injection safety inherited from `sql`); the tokens that cannot be parameterized - table / column identifiers, comparison operators, aggregate functions, join kinds, sort directions - are validated against fixed allowlists (identifiers must be bare SQL names). They are checked both at **build** time (for an early, friendly error) and again at **render** time inside `toSql` / `createTable` / the CRUD builders, so even a hand-built `orm.Query` / `orm.Schema` struct literal that skipped the builder cannot inject SQL.

A record - both the input to `insert` / `update` and the result of `find` / `all` - is a `map of string to string` keyed by column name (the row form that needs no map-to-struct conversion; a typed-struct form waits on that language feature). The database coerces the string values to the column types.

Needs `sql` , so the default `jennifer` binary.

Import with `import "orm.j" as orm;` . See the `orm` guide for prose and examples.

Functions

orm.addColumn(table as string, name as string, kind as ColumnKind, dialect as Dialect)

The `ALTER TABLE ... ADD COLUMN` DDL for a new column (name + type, no attributes; express attribute-rich columns in a `createTable` schema).

Parameters

- `table` `{string}` - the table name
- `name` `{string}` - the new column name
- `kind` `{ColumnKind}` - the column value kind
- `dialect` `{Dialect}` - the SQL dialect (for the type spelling)

Returns `{string}` - the ALTER TABLE ADD COLUMN statement

orm.addForeignKey(table as string, name as string, column as string, refTable as string, refColumn as string)

The `ALTER TABLE ... ADD CONSTRAINT ... FOREIGN KEY` DDL.

Parameters

- `table` `{string}` - the table carrying the foreign key
- `name` `{string}` - the constraint name
- `column` `{string}` - the local foreign-key column
- `refTable` `{string}` - the referenced table
- `refColumn` `{string}` - the referenced column

Returns `{string}` - the ALTER TABLE ADD CONSTRAINT statement

orm.aggregate(q as Query, fn as string, col as string, alias as string)

A copy of `q` adding `func(col) AS alias` to the projection. `func` is one of `COUNT` / `SUM` / `AVG` / `MIN` / `MAX` ; `col` may be `"*"` (for COUNT).

Parameters

- `q` `{Query}` - the source query
- `fn` `{string}` - the aggregate function
- `col` `{string}` - the column (or `"*"`)
- `alias` `{string}` - the result-column alias

Returns `{Query}` - the extended query

orm.all(session as Session, q as Query)

Run a query and return every matching row.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `q` `{Query}` - the query (from the builder)

Returns `{list of map of string to string}` - the rows

orm.autoIncrement(s as Schema)

A copy of `s` marking its most-recently-added column auto-incrementing (`SERIAL` on Postgres, `AUTO_INCREMENT` on MySQL).

Parameters

- `s` `{Schema}` - the source schema

Returns `{Schema}` - the schema with the last column auto-incrementing

orm.belongsTo(s as Schema, name as string, target as string, foreignKey as string)

Declare a **belongs-to** relation: **this** table holds `foreignKey` , which references the target's primary key (`id` by convention). E.g. a post belongs to an author via `posts.authorId -> authors.id` .

Parameters

- `s` `{Schema}` - the source schema
- `name` `{string}` - the relation name
- `target` `{string}` - the target table
- `foreignKey` `{string}` - the foreign-key column on this table

Returns `{Schema}` - the schema with the relation added

orm.column(s as Schema, name as string, kind as ColumnKind)

A copy of `s` with a column appended.

Parameters

- `s` `{Schema}` - the source schema
- `name` `{string}` - the column name
- `kind` `{ColumnKind}` - the value kind (`orm.ColumnKind.Int` / `String` / `Float` / `Bool` / `Bytes`)

Returns `{Schema}` - the extended schema

orm.count(q as Query, alias as string)

A copy of q adding `COUNT(*) AS alias` to the projection.

Parameters

- `q` `{Query}` - the source query
- `alias` `{string}` - the result-column alias

Returns `{Query}` - the extended query

orm.createIndex(name as string, table as string, columns as list of string, isUnique as bool)

The `CREATE [UNIQUE] INDEX` DDL over one or more columns.

Parameters

- `name` `{string}` - the index name
- `table` `{string}` - the table name
- `columns` `{list of string}` - the indexed columns (non-empty)
- `isUnique` `{bool}` - whether the index is UNIQUE

Returns `{string}` - the CREATE INDEX statement

orm.createTable(s as Schema)

The CREATE TABLE DDL for a schema, in its dialect - including each column's `NOT NULL` / `DEFAULT` / `UNIQUE` / auto-increment attributes and the primary key. Pair it with the migration runner (`orm.migrate`) or run it directly.

Parameters

- `s` `{Schema}` - the schema

Returns `{string}` - the CREATE TABLE statement

orm.delete(session as Session, s as Schema, id as string)

Delete a row by primary-key value.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `id` `{string}` - the primary-key value

Returns `{sql.Result}` - the affected-rows result

orm.deleteWhere(session as Session, s as Schema, q as Query)

Bulk-delete every row matching a query's `WHERE` . Refuses a query with no `WHERE` (use a raw `sql.exec` to truncate deliberately).

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `q` `{Query}` - the query whose `WHERE` selects the rows

Returns `{sql.Result}` - the affected-rows result

orm.distinct(q as Query)

A copy of q rendered as `SELECT DISTINCT ...` .

Parameters

- `q` `{Query}` - the source query

Returns `{Query}` - the query with DISTINCT set

orm.dropColumn(table as string, name as string)

The `ALTER TABLE ... DROP COLUMN` DDL.

Parameters

- `table` `{string}` - the table name
- `name` `{string}` - the column to drop

Returns `{string}` - the ALTER TABLE DROP COLUMN statement

orm.dropForeignKey(table as string, name as string, dialect as Dialect)

The `ALTER TABLE ... DROP` foreign-key DDL (Postgres `DROP CONSTRAINT` ; MySQL `DROP FOREIGN KEY`).

Parameters

- `table` `{string}` - the table name
- `name` `{string}` - the constraint name
- `dialect` `{Dialect}` - the SQL dialect

Returns `{string}` - the ALTER TABLE DROP statement

orm.dropIndex(name as string, table as string, dialect as Dialect)

The `DROP INDEX` DDL (Postgres drops by index name; MySQL needs the table).

Parameters

- `name` `{string}` - the index name
- `table` `{string}` - the table the index is on
- `dialect` `{Dialect}` - the SQL dialect

Returns `{string}` - the DROP INDEX statement

orm.dropTable(table as string)

The `DROP TABLE` DDL for a table.

Parameters

- `table` `{string}` - the table name

Returns `{string}` - the DROP TABLE statement

orm.exists(session as Session, q as Query)

Whether a query matches any row (adds `LIMIT 1`).

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `q` `{Query}` - the query

Returns `{bool}` - true if at least one row matches

orm.find(session as Session, s as Schema, id as string)

Find a single row by primary-key value.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `id` `{string}` - the primary-key value

Returns `{map of string to string}` - the row (column name -> value)

Throws

- `{Error}` - when no row has that key

orm.findBy(session as Session, s as Schema, col as string, value as string)

The first row of the schema's table where `col` = `value` , or `{}` if none.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `col` `{string}` - the column to match
- `value` `{string}` - the value to match

Returns `{map of string to string}` - the matching row, or `{}` if none

orm.first(session as Session, q as Query)

The first row a query matches (adds `LIMIT 1`), or an **empty map** when there is none.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `q` `{Query}` - the query

Returns `{map of string to string}` - the first row, or `{}` if none

orm.from(s as Schema)

A base query selecting all rows of the schema's table.

Parameters

- `s` `{Schema}` - the schema

Returns `{Query}` - the base query

orm.groupBy(q as Query, cols as list of string)

A copy of q grouping by the named columns.

Parameters

- `q` `{Query}` - the source query
- `cols` `{list of string}` - the GROUP BY columns

Returns `{Query}` - the extended query

orm.hasMany(s as Schema, name as string, target as string, foreignKey as string)

Declare a **has-many** relation: the **target** table holds `foreignKey` , which references this table's primary key. E.g. an author has many posts via `posts.authorId` -> `authors.id` .

Parameters

- `s` `{Schema}` - the source schema
- `name` `{string}` - the relation name

- `target` `{string}` - the target table
- `foreignKey` `{string}` - the foreign-key column on the target table

Returns `{Schema}` - the schema with the relation added

orm.hasOne(s as Schema, name as string, target as string, foreignKey as string)

Declare a **has-one** relation: the **target** table holds `foreignKey` , which references this table's primary key. E.g. a user has one profile via `profiles.userId -> users.id` .

Parameters

- `s` `{Schema}` - the source schema
- `name` `{string}` - the relation name
- `target` `{string}` - the target table
- `foreignKey` `{string}` - the foreign-key column on the target table

Returns `{Schema}` - the schema with the relation added

orm.having(q as Query, fn as string, col as string, op as string, value as string)

A copy of q with a HAVING condition over an aggregate (`func(col) op value`), AND-joined. HAVING filters groups, so it is used with `groupBy` / `aggregate` .

Parameters

- `q` `{Query}` - the source query
- `fn` `{string}` - the aggregate function (COUNT / SUM / AVG / MIN / MAX)
- `col` `{string}` - the aggregated column (or `"*"`)
- `op` `{string}` - the operator
- `value` `{string}` - the value to bind

Returns `{Query}` - the extended query

orm.insert(session as Session, s as Schema, record as map of string to string)

Insert a record. Only the columns present in `record` are written (so an auto-generated primary key is simply omitted).

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `record` `{map of string to string}` - column name -> value

Returns `{sql.Result}` - the affected-rows / last-insert-id result

orm.insertMany(session as Session, s as Schema, records as list of map of string to string)

Insert many records in one multi-row `INSERT` . Every record must set the same columns (those present in the first). A batch whose placeholder count would exceed the per-statement limit is split across several `INSERT` s (never silently capped); wrap the call in `orm.transaction` for all-or-nothing.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema

- `records` {`list of map of string to string`} - the rows to insert

Returns {`int`} - the number of records inserted

orm.insertReturning(session as Session, s as Schema, record as map of string to string)

Insert `record` and return the generated primary-key value (Postgres `RETURNING pk` ; MySQL `LAST_INSERT_ID` via the result's `lastId`).

Parameters

- `session` {`Session`} - the session (`orm.session` or `orm.transaction`)
- `s` {`Schema`} - the schema
- `record` {`map of string to string`} - the row to insert

Returns {`string`} - the generated primary-key value

orm.join(q as Query, table as string, leftCol as string, rightCol as string)

A copy of q with an INNER JOIN.

Parameters

- `q` {`Query`} - the source query
- `table` {`string`} - the table to join
- `leftCol` {`string`} - the left join column (`table.col`)
- `rightCol` {`string`} - the right join column

Returns {`Query`} - the extended query

orm.joinRelation(q as Query, s as Schema, relationName as string)

A copy of q with the `JOIN` (s) that a declared relation implies - an `INNER JOIN` on the correct key columns (two joins for a many-to-many, through its join table). Built over the `join` primitive.

Parameters

- `q` {`Query`} - the source query
- `s` {`Schema`} - the schema declaring the relation
- `relationName` {`string`} - the relation to join

Returns {`Query`} - the extended query

orm.leftJoin(q as Query, table as string, leftCol as string, rightCol as string)

A copy of q with a LEFT JOIN.

Parameters

- `q` {`Query`} - the source query
- `table` {`string`} - the table to join
- `leftCol` {`string`} - the left join column (`table.col`)
- `rightCol` {`string`} - the right join column

Returns {`Query`} - the extended query

orm.limit(q as Query, n as int)

A copy of q with a LIMIT.

Parameters

- `q` {`Query`} - the source query

- `n` `{int}` - the row limit

Returns `{Query}` - the extended query

orm.load(session as Session, schema as Schema, q as Query)

Run a query and eager-load its `orm.with` -marked relations in a **fixed 1 + R** queries (R = relations requested): the base query once, then **one** batched `WHERE fk IN (...)` per relation - never one query per row. Returns a `Result` walked by `orm.rows` / `related` / `relatedOne` .

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `schema` `{Schema}` - the base schema (declares the relations)
- `q` `{Query}` - the query, with relations marked by `orm.with`

Returns `{Result}` - the base rows plus the eager-loaded relations

orm.manyToMany(s as Schema, name as string, target as string, joinTable as string, localFk as string, targetFk as string)

Declare a **many-to-many** relation through a join table. E.g. a post has many tags via `post_tags(postId, tagId)` : `manyToMany(s, "tags", "tags", "post_tags", "postId", "tagId")` . The target's primary key is `id` by convention, and this table's key is its primary key.

Parameters

- `s` `{Schema}` - the source schema
- `name` `{string}` - the relation name
- `target` `{string}` - the target table
- `joinTable` `{string}` - the join (through) table
- `localFk` `{string}` - the join-table column referencing this table
- `targetFk` `{string}` - the join-table column referencing the target

Returns `{Schema}` - the schema with the relation added

orm.notNull(s as Schema)

A copy of `s` marking its most-recently-added column `NOT NULL` .

Parameters

- `s` `{Schema}` - the source schema

Returns `{Schema}` - the schema with the last column made non-nullable

orm.offset(q as Query, n as int)

A copy of `q` with an OFFSET.

Parameters

- `q` `{Query}` - the source query
- `n` `{int}` - the row offset

Returns `{Query}` - the extended query

orm.orHaving(q as Query, fn as string, col as string, op as string, value as string)

Like `having` , but OR-joined to the previous HAVING condition.

Parameters

- `q` `{Query}` - the source query

- `fn` `{string}` - the aggregate function
- `col` `{string}` - the aggregated column (or `"*"`)
- `op` `{string}` - the operator
- `value` `{string}` - the value to bind

Returns `{Query}` - the extended query

orm.orWhere(q as Query, col as string, op as string, value as string)

Like `where` , but OR-joined to the previous condition. SQL binds `AND` tighter than `OR` , so `where(...).orWhere(...)` reads `a AND b OR c = (a AND b) OR c` .

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column
- `op` `{string}` - the operator
- `value` `{string}` - the value to bind

Returns `{Query}` - the extended query

orm.orWhereIn(q as Query, col as string, values as list of string)

Like `whereIn` , but OR-joined to the previous condition.

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column
- `values` `{list of string}` - the values (must be non-empty)

Returns `{Query}` - the extended query

orm.orderBy(q as Query, col as string, dir as string)

A copy of q with an ORDER BY term added.

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column
- `dir` `{string}` - `"asc"` or `"desc"`

Returns `{Query}` - the extended query

orm.page(q as Query, pageNum as int, pageSize as int)

A copy of q with `LIMIT` / `OFFSET` set for a 1-based page: page `pageNum` of `pageSize` rows (page 1 starts at offset 0).

Parameters

- `q` `{Query}` - the source query
- `pageNum` `{int}` - the 1-based page number
- `pageSize` `{int}` - the rows per page

Returns `{Query}` - the paginated query

orm.pluck(session as Session, q as Query, col as string)

The values of one column across every row a query matches (the column must be in the projection; the default `SELECT *` includes it).

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `q` `{Query}` - the query
- `col` `{string}` - the column to pluck

Returns `{list of string}` - the column's value for each matching row

orm.related(result as Result, row as map of string to string, name as string)

The eager-loaded child rows of `row` for a has-many / many-to-many relation (an empty list if the row has none). No query - a lookup into the `Result` .

Parameters

- `result` `{Result}` - the load result
- `row` `{map of string to string}` - one base row (from `orm.rows`)
- `name` `{string}` - the relation name

Returns `{list of map of string to string}` - the related child rows

orm.relatedOne(result as Result, row as map of string to string, name as string)

The single eager-loaded row of `row` for a belongs-to / has-one relation, or an **empty map** when there is none (a null / unmatched foreign key). No query.

Parameters

- `result` `{Result}` - the load result
- `row` `{map of string to string}` - one base row (from `orm.rows`)
- `name` `{string}` - the relation name

Returns `{map of string to string}` - the related row, or `{}` if none

orm.renameColumn(table as string, fromName as string, toName as string)

The `ALTER TABLE ... RENAME COLUMN` DDL (modern MySQL 8+ / MariaDB / Postgres syntax).

Parameters

- `table` `{string}` - the table name
- `fromName` `{string}` - the current column name
- `toName` `{string}` - the new column name

Returns `{string}` - the ALTER TABLE RENAME COLUMN statement

orm.rightJoin(q as Query, table as string, leftCol as string, rightCol as string)

A copy of q with a RIGHT JOIN.

Parameters

- `q` `{Query}` - the source query
- `table` `{string}` - the table to join
- `leftCol` `{string}` - the left join column (`table.col`)
- `rightCol` `{string}` - the right join column

Returns `{Query}` - the extended query

orm.rows(result as Result)

The base rows of a `Result` (the same rows `orm.all` would return).

Parameters

- `result` `{Result}` - the load result

Returns `{list of map of string to string}` - the base rows

orm.save(session as Session, s as Schema, record as map of string to string)

Insert `record` when it carries **no** primary key, else update it (matched by the primary key). A Data-Mapper convenience - still no per-row method. (A new row with an explicitly-assigned primary key should use `orm.insert` .)

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `record` `{map of string to string}` - the row

Returns `{sql.Result}` - the affected-rows result

orm.schema(table as string, primaryKey as string, dialect as Dialect)

Start a schema for a table with its primary-key column and dialect. Add columns with `orm.column` .

Parameters

- `table` `{string}` - the table name
- `primaryKey` `{string}` - the primary-key column
- `dialect` `{Dialect}` - `orm.Dialect.Mysql` or `orm.Dialect.Postgres`

Returns `{Schema}` - the schema (no columns yet)

orm.select(q as Query, cols as list of string)

A copy of q projecting only the named columns (instead of `SELECT *`). Additive: call again, or combine with `count` / `aggregate` .

Parameters

- `q` `{Query}` - the source query
- `cols` `{list of string}` - the columns to project

Returns `{Query}` - the extended query

orm.session(conn as sql.Connection)

A session that runs each statement directly on a connection (auto-commit).

Parameters

- `conn` `{sql.Connection}` - the open connection

Returns `{Session}` - an auto-committing session

orm.toSql(q as Query)

Render a query to parameterized SQL for its dialect. Pure - the whole query-builder surface is testable without a database. Every identifier and operator is re-validated here (`validateQuery`), so a hand-built `Query` literal that skipped the builder guards still cannot inject.

Parameters

- `q` `{Query}` - the query

Returns `{Rendered}` - the SQL text and ordered bind values

orm.transaction(tx as sql.Tx)

A session that runs its statements inside a caller-managed transaction. The caller owns `sql.begin` / `commit` / `rollback` ; orm just executes through the `Tx` . The idiom is `sql.begin` + `errdefer sql.rollback` + `orm.transaction` + `sql.commit` .

Parameters

- `tx` `{sql.Tx}` - the open transaction

Returns `{Session}` - a transaction-bound session

orm.unique(s as Schema)

A copy of `s` adding a `UNIQUE` constraint to its most-recently-added column.

Parameters

- `s` `{Schema}` - the source schema

Returns `{Schema}` - the schema with the last column made unique

orm.update(session as Session, s as Schema, record as map of string to string)

Update a record, matched by its primary-key value (which the record must carry). Every other present column is written.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `record` `{map of string to string}` - the row, including the primary key

Returns `{sql.Result}` - the affected-rows result

Throws

- `{Error}` - when the record has no primary-key value

orm.updateWhere(session as Session, s as Schema, assignments as map of string to string, q as Query)

Bulk-update every row matching a query's `WHERE` , setting `assignments` (column -> value, bound as parameters). Refuses a query with no `WHERE` .

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `assignments` `{map of string to string}` - the columns to set
- `q` `{Query}` - the query whose `WHERE` selects the rows

Returns `{sql.Result}` - the affected-rows result

orm.upsert(session as Session, s as Schema, record as map of string to string, conflictCols as list of string)

Insert `record` , or update the conflicting row if a unique / primary-key conflict on `conflictCols` occurs (Postgres `ON CONFLICT ... DO UPDATE` , MySQL `ON DUPLICATE KEY UPDATE`). The non-conflict present columns are updated.

Parameters

- `session` `{Session}` - the session (`orm.session` or `orm.transaction`)
- `s` `{Schema}` - the schema
- `record` `{map of string to string}` - the row to insert / update
- `conflictCols` `{list of string}` - the conflict-target columns (unique / PK; non-empty)

Returns `{sql.Result}` - the affected-rows result

orm.where(q as Query, col as string, op as string, value as string)

A copy of `q` with a `column op value` condition AND-joined to the rest. The value binds as a parameter.

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column
- `op` `{string}` - the operator (`=` , `>` , `<` , `>=` , `<=` , `!=` , `<>` , `LIKE` , `NOT LIKE`)
- `value` `{string}` - the value to bind

Returns `{Query}` - the extended query

orm.whereBetween(q as Query, col as string, lo as string, hi as string)

A copy of `q` with a `column BETWEEN lo AND hi` condition, AND-joined. Both bounds bind as parameters.

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column
- `lo` `{string}` - the lower bound (inclusive)
- `hi` `{string}` - the upper bound (inclusive)

Returns `{Query}` - the extended query

orm.whereIn(q as Query, col as string, values as list of string)

A copy of `q` with a `column IN (...)` condition AND-joined, one placeholder bound per value.

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column
- `values` `{list of string}` - the values (must be non-empty)

Returns `{Query}` - the extended query

orm.whereNotIn(q as Query, col as string, values as list of string)

A copy of `q` with a `column NOT IN (...)` condition AND-joined.

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column
- `values` `{list of string}` - the values (must be non-empty)

Returns `{Query}` - the extended query

orm.whereNotNull(q as Query, col as string)

A copy of `q` with a `column IS NOT NULL` condition, AND-joined.

Parameters

- `q` `{Query}` - the source query
- `col` `{string}` - the column

Returns {Query} - the extended query

orm.whereNull(q as Query, col as string)

A copy of q with a column IS NULL condition, AND-joined.

Parameters

- q {Query} - the source query
- col {string} - the column

Returns {Query} - the extended query

orm.with(q as Query, relationName as string)

A copy of q marking a declared relation to **eager-load** with orm.load . Additive: call again for more relations. Does not change the base SQL (toSql ignores it); load runs one extra batched query per marked relation.

Parameters

- q {Query} - the source query
- relationName {string} - the relation to eager-load

Returns {Query} - the extended query

orm.withDefault(s as Schema, value as string)

A copy of s giving its most-recently-added column a DEFAULT . The value is rendered by the column kind: an Int / Float default is validated numeric and rendered bare; a Bool default accepts true / false / 1 / 0 ; a String / Bytes default is rendered as a quoted, escaped SQL string literal (backslashes and control characters are rejected). So no default value can inject DDL.

Parameters

- s {Schema} - the source schema
- value {string} - the default value (interpreted per the column kind)

Returns {Schema} - the schema with the last column defaulted

Structs

orm.Column

One column in a schema: its name, value kind, and DDL attributes. The attributes drive createTable / addColumn DDL; set them with the fluent setters (notNull / unique / withDefault / autoIncrement), which decorate the most-recently-added column. A column is **nullable by default** (SQL's own default); notNull opts into NOT NULL .

Field	Type	Description
name	string	the column name
kind	ColumnKind	the value kind
nullable	bool	whether the column allows NULL (true = no NOT NULL clause)
unique	bool	whether the column carries a UNIQUE constraint
hasDefault	bool	whether a DEFAULT is set (guards default)
default	string	the DEFAULT value (rendered by kind: numeric/bool bare, string quoted+escaped)
autoIncrement	bool	whether the column auto-increments (SERIAL / AUTO_INCREMENT)

orm.Condition

A single WHERE condition within a `Query` ; its bound value(s) live in the query's `params` list (positionally). Built by `orm.where` / `orWhere` / `whereIn` , not directly.

Field	Type	Description
column	string	the column
op	string	the comparison operator (or IN / NOT IN)
connector	string	"AND" or "OR" - how this joins the previous condition
valueCount	int	the number of placeholders this condition consumes (1, or N for IN)

orm.Having

A single HAVING condition over an aggregate, its value in `havingParams` . Built by `orm.having` , not directly.

Field	Type	Description
func	string	the aggregate function (COUNT / SUM / AVG / MIN / MAX)
column	string	the aggregated column (or "" for COUNT())
op	string	the comparison operator
connector	string	"AND" or "OR" - how this joins the previous HAVING condition

orm.Join

One JOIN clause: its kind and the `left` = `right` equality. Built by `orm.join` / `orm.leftJoin` / `orm.rightJoin` , not directly.

Field	Type	Description
kind	string	"INNER", "LEFT", or "RIGHT"
table	string	the joined table
leftCol	string	the left join column (table.col)
rightCol	string	the right join column

orm.Order

One ORDER BY term. Built by `orm.orderBy` , not directly.

Field	Type	Description
column	string	the column
dir	string	"ASC" or "DESC"

orm.Query

A composable, non-mutating SELECT query. Build it with `from` / `select` / `count` / `aggregate` / `where` / `orWhere` / `whereIn` / `join` / `leftJoin` / `groupBy` / `having` / `orderBy` / `limit` / `offset` , then render with `toSql` . Every identifier is re-validated at render time, so a hand-built `Query` literal cannot bypass the injection guards.

Field	Type	Description
table	string	the base table
dialect	Dialect	the SQL dialect
selects	list of SelectItem	the projection (empty = SELECT *)
wheres	list of Condition	the WHERE conditions
params	list of string	the bind values for the WHERE conditions
joins	list of Join	the JOIN clauses

groups	list of string	the GROUP BY columns
havings	list of Having	the HAVING conditions
havingParams	list of string	the bind values for the HAVING conditions
orders	list of Order	the ORDER BY terms
hasLimit	bool	whether a LIMIT is set
limitN	int	the LIMIT value
hasOffset	bool	whether an OFFSET is set
offsetN	int	the OFFSET value
withRelations	list of string	relation names to eager-load (consumed by load, ignored by toSql)
distinctSelect	bool	whether to render SELECT DISTINCT

orm.Relation

A declared association from a schema to another table. Metadata only - built by `orm.belongsTo` / `hasOne` / `hasMany` / `manyToMany` , read by `orm.joinRelation` (and, later, eager loading). For a `BelongsTo` , `foreignKey` is on **this** table and `localKey` is the target's key; for `HasOne` / `HasMany` , `foreignKey` is on the **target** table and `localKey` is this table's key. `ManyToMany` links through `through` (a join table) via its two keys.

Field	Type	Description
name	string	the relation's name (the lookup key for joinRelation / eager loading)
kind	RelationKind	the association kind
target	string	the target table
foreignKey	string	the foreign-key column (side depends on kind)
localKey	string	the referenced key column (the "one" side's key)
through	string	the join table for ManyToMany (else "")
throughLocalKey	string	the join-table column referencing this table (else "")
throughTargetKey	string	the join-table column referencing the target (else "")

orm.RelationData

The loaded child rows for one eager-loaded relation: a parent-key -> child-rows lookup, built once from a single batched query. Internal to a `Result` ; read it through `orm.related` / `orm.relatedOne` , not directly.

Field	Type	Description
name	string	the relation name
parentKeyColumn	string	the base-row column whose value indexes the lookup
byParentKey	map of string to list of map of string to string	parent-key value -> its child rows

orm.Rendered

A rendered, parameterized statement: the SQL text and the ordered bind values.

Field	Type	Description
sql	string	the SQL with dialect placeholders
params	list of string	the bind values, in placeholder order

orm.Result

The result of `orm.load` : the base query's rows plus the eager-loaded relations. Read the base rows with `orm.rows` and a row's related rows with `orm.related` (has-many / many-to-many) or `orm.relatedOne` (belongs-to / has-one).

Field	Type	Description
rows	list of map of string to string	the base query's rows
relations	list of RelationData	one entry per eager-loaded relation

orm.Schema

A table mapping: the table name, its columns, the primary-key column, the SQL dialect, and any declared relations. Value-semantic; `column` returns a fresh schema.

Field	Type	Description
table	string	the table name
columns	list of Column	the columns
primaryKey	string	the primary-key column name
dialect	Dialect	the SQL dialect (placeholder + DDL spelling)
relations	list of Relation	the declared associations

orm.SelectItem

One item in a SELECT projection: a plain column (`func` "") or an aggregate (`func` one of COUNT / SUM / AVG / MIN / MAX, rendered `func(column)` AS `alias`). Built by `orm.select` / `orm.count` / `orm.aggregate` , not directly.

Field	Type	Description
func	string	the aggregate function, or "" for a plain column
column	string	the column (or "" for COUNT())
alias	string	the AS alias for an aggregate, or "" for a plain column

orm.Session

A unit of work wrapping either a `sql.Connection` (auto-committing) or a `sql.Tx` (inside a caller's transaction). Every persistence / query-executing function takes a `Session` as its first argument, so the same call runs standalone or inside a transaction. Value-semantic; build it with `orm.session` or `orm.transaction` , not directly (only the handle selected by `inTx` is ever touched).

Field	Type	Description
conn	sql.Connection	the connection (used when inTx is false)
tx	sql.Tx	the transaction (used when inTx is true)
inTx	bool	which handle is live

Enums

orm.ColumnKind

A column's value kind: the SQL type family `createTable` renders. One of `orm.ColumnKind.Int` / `String` / `Float` / `Bool` / `Bytes` .

orm.Dialect

The SQL dialect: the backend selector that governs placeholder syntax and DDL spelling. `orm.Dialect.Mysql` or `orm.Dialect.Postgres` .

orm.RelationKind

The kind of an association between two tables: `orm.RelationKind.BelongsTo` (this table holds the foreign key), `HasOne` / `HasMany` (the target table holds it), or `ManyToMany` (a join table links the two).

password API reference

Password generation, validation, and complexity scoring against a policy schema. A `Schema` names the allowed character classes (lowercase, uppercase, digits, symbols), a length range, and per-class minimum counts; `generate(schema)` produces a conforming password, `validate(schema, pw)` reports whether a password meets the policy (and why not), and `complexity(pw)` estimates a password's strength in bits of entropy.

Randomness comes from the `crypto` library's crypto-grade source (character choice and the final shuffle), so a generated password is unpredictable and safe to mint as a real credential. Pure `.j` over `crypto` / `strings` / `convert` ; runs on both binaries.

Import with `import "password.j" as password;` . See the `password` guide for prose and examples.

Functions

password.complexity(pw as string)

Estimate a password's strength. The alphabet size is the sum of the class sizes present (lower/upper 26 each, digits 10, symbols the default-set size), and entropy is `length * log2(alphabet)` bits.

Parameters

- `pw` `{string}` - the password

Returns `{Strength}` - length, class count, pool size, entropy bits, and a label

password.generate(s as Schema)

Generate a password conforming to the schema: a length picked in `[minLength, maxLength]` , at least the per-class minimums, the remainder drawn from the enabled alphabet, then shuffled.

Parameters

- `s` `{Schema}` - the policy

Returns `{string}` - a fresh password

Throws

- `{Error}` - kind "password" if the schema is infeasible (no classes, an empty required pool, or minimums that exceed the length)

password.schema()

A strong default policy: exactly 16 characters, all four classes, at least one of each, the default symbol set, ambiguous glyphs allowed.

Returns `{Schema}` - the default policy

password.validate(s as Schema, pw as string)

Validate a password against a schema. Checks the length bounds and each per-class minimum; it does not reject characters outside the alphabet (a policy sets minimums, not a whitelist).

Parameters

- `s` `{Schema}` - the policy
- `pw` `{string}` - the password to check

Returns `{Report}` - `valid` plus the list of failed-rule `reasons`

password.withClasses(s as Schema, lo as bool, up as bool, dig as bool, sym as bool)

Copy a schema with new class enables (which classes appear in generation).

Parameters

- `s` `{Schema}` - the base schema
- `lo` `{bool}` - lowercase
- `up` `{bool}` - uppercase
- `dig` `{bool}` - digits
- `sym` `{bool}` - symbols

Returns `{Schema}` - a fresh schema

password.withLength(s as Schema, lo as int, hi as int)

Copy a schema with a new length range (set `lo` == `hi` for a fixed length).

Parameters

- `s` `{Schema}` - the base schema
- `lo` `{int}` - minimum length
- `hi` `{int}` - maximum length

Returns `{Schema}` - a fresh schema

password.withMinimums(s as Schema, lo as int, up as int, dig as int, sym as int)

Copy a schema with new per-class minimum counts.

Parameters

- `s` `{Schema}` - the base schema
- `lo` `{int}` - minimum lowercase
- `up` `{int}` - minimum uppercase
- `dig` `{int}` - minimum digits
- `sym` `{int}` - minimum symbols

Returns `{Schema}` - a fresh schema

password.withSymbolSet(s as Schema, setChars as string)

Copy a schema with a replacement symbol set.

Parameters

- `s` `{Schema}` - the base schema
- `setChars` `{string}` - the symbol characters to draw from

Returns `{Schema}` - a fresh schema

password.withoutAmbiguous(s as Schema)

Copy a schema that excludes ambiguous glyphs (0 O o 1 l I |) from generation.

Parameters

- `s` `{Schema}` - the base schema

Returns `{Schema}` - a fresh schema

Structs

password.Report

A validation result: whether the password conforms, and a list of the failed rules (empty when valid).

Field	Type	Description
valid	bool	true when every rule passed
reasons	list of string	human-readable failed rules

password.Schema

A password policy: allowed classes, a length range, per-class minimum counts, the symbol set, and whether to drop ambiguous glyphs.

Field	Type	Description
minLength	int	shortest allowed length (also the generated length floor)
maxLength	int	longest allowed length (generation picks in the range)
lower	bool	include lowercase in the generation alphabet
upper	bool	include uppercase
digits	bool	include digits
symbols	bool	include symbols (from symbolSet)
symbolSet	string	the symbol characters to draw from
minLower	int	minimum lowercase (generation guarantees, validation requires)
minUpper	int	minimum uppercase
minDigits	int	minimum digits
minSymbols	int	minimum symbols
excludeAmbiguous	bool	drop ambiguous glyphs (0 O o 1 l I |) from generation

password.Strength

An estimated-strength summary of a password.

Field	Type	Description
length	int	character count
classes	int	how many of the four classes are present (0-4)
poolSize	int	the estimated alphabet size (guessing space per char)
entropy	float	estimated bits of entropy (length * log2(poolSize))
label	string	a qualitative band: very weak / weak / reasonable / strong / very strong

pdf API reference

Generate PDF documents - text, lines, rectangles - the way `html` / `label` generate their formats. Build a `Document` of `Page` s with value-semantic builders, then `render()` writes the PDF object / xref structure by hand (no stdlib PDF) as `bytes` . Content streams are FlateDecode-compressed via `compress` . Two font paths: the standard-14 Type1 fonts (Helvetica / Times / Courier families + Symbol / ZapfDingbats) via `text` , and **embedded TrueType fonts** via `loadFont` / `addFont` / `textUnicode` - a Type0 / CIDFontType2 composite (Identity-H) with an embedded `FontFile2` and a ToUnicode map, so any script the font covers (accented Latin, Greek, Cyrillic, CJK, ...) renders and stays selectable. Raster images embed as image XObjects via `loadImage` / `addImage` / `drawImage` - PNG (greyscale / RGB / palette, plus 8-bit alpha as a soft mask) and JPEG (DCTDecode). Text layout - `measureText` width measurement (standard-14 AFM metrics + embedded-font advances), `wrapText` word-wrap, and `textBlock` wrapped / aligned paragraphs (left / right / center / justify) - flows text into a column. Object numbers are assigned dynamically. Pure Jennifer (over the `font` module); both binaries.

Coordinates are in PDF points (1/72 inch), origin bottom-left, y upward. Common page sizes: US Letter 612 x 792, A4 595 x 842. Colours are 0-255 RGB.

Import with `import "pdf.j" as pdf;` . See the `pdf` guide for prose and examples.

Functions

pdf.addFont(doc as Document, lf as LoadedFont)

A copy of the document with an embedded font registered, so `render` writes its font program and dictionaries.

Parameters

- `doc` `{Document}` - the document
- `lf` `{LoadedFont}` - the loaded font (from `loadFont`)

Returns `{Document}` - a fresh document with the font registered

pdf.addImage(doc as Document, img as Image)

A copy of the document with an embedded image registered, so `render` writes its image XObject. Draw it on a page with `drawImage` .

Parameters

- `doc` `{Document}` - the document
- `img` `{Image}` - the loaded image (from `loadImage`)

Returns `{Document}` - a fresh document with the image registered

pdf.addPage(doc as Document, pg as Page)

A copy of the document with a page appended.

Parameters

- `doc` `{Document}` - the document
- `pg` `{Page}` - the page to add

Returns `{Document}` - a fresh document with the page appended

pdf.bookmark(doc as Document, page as int, y as int, title as string, level as int)

A copy of the document with one outline (bookmark) entry appended (value-semantic). Entries are nested by `level` in the order added: a level-2 entry becomes a child of the most recent level-1, and so on. The destination

scrolls page `page` (0-based) so that `y` (PDF points, origin bottom-left) is at the top of the view. With any outline present, the viewer opens showing the bookmark panel.

Parameters

- `doc` `{Document}` - the document
- `page` `{int}` - the 0-based index of the destination page
- `y` `{int}` - the destination y in PDF points
- `title` `{string}` - the bookmark label
- `level` `{int}` - the nesting level (>= 1)

Returns `{Document}` - a copy with the entry appended

pdf.color(pg as Page, red as int, green as int, blue as int)

Set the fill and stroke colour (0-255 RGB) for subsequent drawing on the page.

Parameters

- `pg` `{Page}` - the page
- `red` `{int}` - the red component 0-255
- `green` `{int}` - the green component 0-255
- `blue` `{int}` - the blue component 0-255

Returns `{Page}` - a fresh page with the colour set

pdf.document()

An empty document. The `Producer` metadata defaults to "Jennifer pdf"; everything else is unset (no `CreationDate` is stamped, so output stays deterministic - set one explicitly with `info` + `pdfDate` if you want it).

Returns `{Document}` - the document

pdf.drawImage(pg as Page, img as Image, x as int, y as int, width as int, height as int)

Draw a registered image scaled into the rectangle at (x, y) of the given width and height (points). The image keeps no aspect ratio of its own - it fills the box - so pass a width / height in the image's own proportion to avoid stretching. `addImage` the image into the document first so the resource name resolves.

Parameters

- `pg` `{Page}` - the page
- `img` `{Image}` - the image (from `loadImage`)
- `x` `{int}` - the lower-left x in points
- `y` `{int}` - the lower-left y in points
- `width` `{int}` - the drawn width in points
- `height` `{int}` - the drawn height in points

Returns `{Page}` - a fresh page with the image drawn

pdf.foldLine(font as string, size as int, text as string, maxWidth as int)

Hard-fold one line so every piece fits `maxWidth` (points). Breaks at the last space or seam punctuation before the overflow, or mid-token when there is none (a bare URL, a slash-joined identifier). Unlike `wrapText` , it does not reflow on every space - it only breaks where a line would otherwise run past the box - so it is the right tool for a code line or an unbreakable token that must stay on the page. A single character wider than `maxWidth` is emitted alone.

Parameters

- `font {string}` - the standard-14 font
- `size {int}` - the point size
- `text {string}` - the line to fold (no embedded newlines)
- `maxWidth {int}` - the target width in points

Returns `{list of string}` - the fitted pieces (at least one)

pdf.getCurrentPageNr(doc as Document)

The 1-based number of the page currently being built - the next page to be added, i.e. `getTotalPages(doc) + 1` . Use it while filling a page (before `addPage`) to label it.

Parameters

- `doc {Document}` - the document

Returns `{int}` - the current page number

pdf.getTotalPages(doc as Document)

The total number of pages added to the document (the value `{pages}` expands to).

Parameters

- `doc {Document}` - the document

Returns `{int}` - the page count

pdf.info(doc as Document, key as string, value as string)

A copy of the document with one metadata field set (value-semantic). `key` is a PDF Info key - "Title", "Author", "Subject", "Keywords", "Creator", "Producer", "CreationDate", "ModDate" (or any custom key).

Parameters

- `doc {Document}` - the document
- `key {string}` - the Info-dictionary key
- `value {string}` - the value

Returns `{Document}` - a fresh document with the metadata set

pdf.line(pg as Page, fromX as int, fromY as int, toX as int, toY as int)

Draw a straight line from (fromX, fromY) to (toX, toY).

Parameters

- `pg {Page}` - the page
- `fromX {int}` - the start x
- `fromY {int}` - the start y
- `toX {int}` - the end x
- `toY {int}` - the end y

Returns `{Page}` - a fresh page with the line added

pdf.link(pg as Page, x as int, y as int, width as int, height as int, uri as string)

Add a link annotation: a clickable rectangle (lower-left `x` , `y` , plus `width` / `height` , in PDF points) that opens `uri` . The border is invisible, so place it over text drawn with `text` (measure the text width with `measureText`) to make that text a hyperlink - a footer backlink, a citation, a URL in prose.

Parameters

- `pg {Page}` - the page

- `x` `{int}` - lower-left x of the clickable rectangle
- `y` `{int}` - lower-left y of the clickable rectangle
- `width` `{int}` - rectangle width in points
- `height` `{int}` - rectangle height in points
- `uri` `{string}` - the URI the link opens

Returns `{Page}` - the page with the link recorded

pdf.loadFont(name as string, data as bytes)

Load an embeddable font from its bytes, under a resource name used to select it in `textUnicode` . The font is embedded (and its Unicode text made selectable) when the document renders.

Parameters

- `name` `{string}` - the resource name (e.g. "Body"; letters / digits)
- `data` `{bytes}` - the TrueType (.ttf) font file

Returns `{LoadedFont}` - the loaded font

pdf.loadImage(name as string, data as bytes)

Load a raster image (PNG or JPEG) from its bytes, under a resource name used to select it in `drawImage` . The format is detected from the file signature. PNG: non-interlaced greyscale / RGB / palette (embedded directly with a FlateDecode predictor) and 8-bit greyscale+alpha / RGBA (decoded to a colour stream plus an alpha soft mask). JPEG: baseline / progressive greyscale / RGB / CMYK, embedded as-is via DCTDecode.

Parameters

- `name` `{string}` - the resource name (e.g. "Logo"; letters / digits)
- `data` `{bytes}` - the PNG or JPEG file

Returns `{Image}` - the loaded image

Throws

- `{Error}` - kind "pdf" if the data is not a supported PNG / JPEG

pdf.measureText(font as string, size as int, str as string)

Measure the rendered width (in points) of a string in a standard-14 font at a point size, using the Adobe Core-14 AFM metrics. A character outside WinAnsi, or a Symbol / ZapfDingbats font, throws.

Parameters

- `font` `{string}` - a standard-14 base font name
- `size` `{int}` - the font size in points
- `str` `{string}` - the text to measure

Returns `{float}` - the width in points

Throws

- `{Error}` - kind "pdf" for an unknown / metric-less font

pdf.measureTextUnicode(lf as LoadedFont, size as int, str as string)

Measure the rendered width (in points) of a string in an embedded font at a point size, using the font's own glyph advances.

Parameters

- `lf` `{LoadedFont}` - the embedded font
- `size` `{int}` - the font size in points
- `str` `{string}` - the text to measure

Returns `{float}` - the width in points

pdf.page(width as int, height as int)

A blank page of the given size in points (e.g. `page(612, 792)` for Letter, `page(595, 842)` for A4).

Parameters

- `width` `{int}` - the page width in points
- `height` `{int}` - the page height in points

Returns `{Page}` - the page

pdf.pageLabel()

A blank page label (empty slots, Helvetica 9pt, 36-point margin, no border, black). Copy it and set slots / options, then attach with `setHeader` / `setFooter` .

Returns `{PageLabel}` - the default label

pdf.pdfDate(t as time.Time)

Format an instant as a PDF date string (`D:YYYYMMDDHHmmSS+HH'mm'`), for use as a `CreationDate` / `ModDate` value. Passing a time is explicit, so it does not make `render` non-deterministic on its own.

Parameters

- `t` `{time.Time}` - the instant

Returns `{string}` - the PDF date string

pdf.rect(pg as Page, x as int, y as int, width as int, height as int, filled as bool)

Draw a rectangle at (x, y) of the given size. `filled` fills it; otherwise it is stroked (outline only).

Parameters

- `pg` `{Page}` - the page
- `x` `{int}` - the lower-left x
- `y` `{int}` - the lower-left y
- `width` `{int}` - the width in points
- `height` `{int}` - the height in points
- `filled` `{bool}` - true to fill, false to stroke

Returns `{Page}` - a fresh page with the rectangle added

pdf.render(doc as Document)

Render the document to PDF bytes (PDF 1.7). Any running header / footer is drawn onto every page first. Content streams are FlateDecode-compressed; standard-14 fonts become shared Type1 objects and each embedded font a Type0 / CIDFontType2 with an embedded FontFile2 and a ToUnicode map. Registered images become image XObjects (with a soft-mask XObject when they carry alpha). An outline (bookmarks) is emitted when the document has any, with the catalog set to open the bookmark panel. Objects are numbered dynamically, so any mix of pages, fonts, images, and resources references correctly.

Parameters

- `doc` `{Document}` - the document to render

Returns `{bytes}` - the PDF file contents

pdf.setFooter(doc as Document, label as PageLabel)

Attach a running footer, drawn on every page when the document renders.

Parameters

- `doc` `{Document}` - the document
- `label` `{PageLabel}` - the footer spec

Returns `{Document}` - the document with the footer set

pdf.setHeader(doc as Document, label as PageLabel)

Attach a running header, drawn on every page when the document renders.

Parameters

- `doc` `{Document}` - the document
- `label` `{PageLabel}` - the header spec

Returns `{Document}` - the document with the header set

pdf.text(pg as Page, x as int, y as int, font as string, size as int, str as string)

Draw a line of text at (x, y) in the given standard-14 font and point size.

Parameters

- `pg` `{Page}` - the page
- `x` `{int}` - the x position (points from the left)
- `y` `{int}` - the y position (points from the bottom)
- `font` `{string}` - a standard-14 base font name (e.g. "Helvetica")
- `size` `{int}` - the font size in points
- `str` `{string}` - the text to draw

Returns `{Page}` - a fresh page with the text added

Throws

- `{Error}` - kind "pdf" if the font is not a standard-14 name

pdf.textBlock(pg as Page, x as int, y as int, width as int, font as string, size as int, leading as int, str as string, align as string)

Flow standard-14 text into a column: word-wrap `str` to `width` points and draw each line, the first line's baseline at (x, y) and each subsequent line `leading` points lower. `align` is "left" / "right" / "center" / "justify" (justify pads inter-word gaps on every line but the last of each paragraph). The drawn block is `len(wrapText(...)) * leading` points tall.

Parameters

- `pg` `{Page}` - the page
- `x` `{int}` - the column's left x in points
- `y` `{int}` - the first line's baseline y in points
- `width` `{int}` - the column width in points
- `font` `{string}` - a standard-14 base font name
- `size` `{int}` - the font size in points
- `leading` `{int}` - the line-to-line spacing in points
- `str` `{string}` - the text (newlines are hard breaks)
- `align` `{string}` - "left" / "right" / "center" / "justify"

Returns `{Page}` - a fresh page with the text block drawn

Throws

- `{Error}` - kind "pdf" for an unknown align or font

pdf.textBlockUnicode(pg as Page, x as int, y as int, width as int, lf as LoadedFont, size as int, leading as int, str as string, align as string)

Flow embedded-font (Unicode) text into a column - `textBlock` for a font from `loadFont` / `addFont` . Same wrapping / alignment / leading behaviour.

Parameters

- `pg` `{Page}` - the page
- `x` `{int}` - the column's left x in points
- `y` `{int}` - the first line's baseline y in points
- `width` `{int}` - the column width in points
- `lf` `{LoadedFont}` - the embedded font
- `size` `{int}` - the font size in points
- `leading` `{int}` - the line-to-line spacing in points
- `str` `{string}` - the text (newlines are hard breaks)
- `align` `{string}` - "left" / "right" / "center" / "justify"

Returns `{Page}` - a fresh page with the text block drawn

Throws

- `{Error}` - kind "pdf" for an unknown align

pdf.textUnicode(pg as Page, x as int, y as int, lf as LoadedFont, size as int, str as string)

Draw Unicode text at (x, y) in an embedded font (from `loadFont` + `addFont`) at the given point size. Each character maps through the font's cmap to a glyph, so any script the font covers - including CJK - renders and stays selectable / copyable in a viewer.

Parameters

- `pg` `{Page}` - the page
- `x` `{int}` - the x position (points from the left)
- `y` `{int}` - the y position (points from the bottom)
- `lf` `{LoadedFont}` - the embedded font
- `size` `{int}` - the font size in points
- `str` `{string}` - the text to draw

Returns `{Page}` - a fresh page with the text added

pdf.toWinAnsi(s as string, replacement as string)

Return `s` with every character the standard-14 fonts cannot encode (outside WinAnsi / windows-1252) replaced by `replacement` ("" drops it). A clean string (the common case) is returned unchanged after a single whole-string check, so the per-character path runs only when there is something to replace. This lets a caller keep one out-of-range glyph from aborting a whole render.

Parameters

- `s` `{string}` - the text
- `replacement` `{string}` - the substitute for an un-encodable character

Returns `{string}` - the WinAnsi-safe text

pdf.wrapText(font as string, size as int, str as string, maxWidth as int)

Word-wrap a standard-14 string to a maximum line width (points), returning the lines. Existing newlines are honoured as hard breaks; runs of spaces collapse. A word-wrapped line still wider than the box (an unbreakable token) is hard-folded with `foldLine` , so no output line runs past `maxWidth` .

Parameters

- `font` `{string}` - a standard-14 base font name
- `size` `{int}` - the font size in points
- `str` `{string}` - the text to wrap
- `maxWidth` `{int}` - the maximum line width in points

Returns `{list of string}` - the wrapped lines

pdf.wrapTextUnicode(lf as LoadedFont, size as int, str as string, maxWidth as int)

Word-wrap an embedded-font string to a maximum line width (points).

Parameters

- `lf` `{LoadedFont}` - the embedded font
- `size` `{int}` - the font size in points
- `str` `{string}` - the text to wrap
- `maxWidth` `{int}` - the maximum line width in points

Returns `{list of string}` - the wrapped lines

Structs

pdf.Document

A PDF document: an ordered list of pages, the document metadata (the PDF Info dictionary, keyed by PDF key name - "Title", "Author", ...), any embedded fonts registered with `addFont` , any outline (bookmark) entries, and an optional running header / footer drawn on every page at render time.

Field	Type	Description
pages	list of Page	the document's pages
info	map of string to string	the Info-dictionary metadata
embedded	list of LoadedFont	the embedded fonts
images	list of Image	the embedded raster images
outline	list of OutlineEntry	the outline / bookmark entries, in document order
header	PageLabel	the running header (drawn when headerOn)
footer	PageLabel	the running footer (drawn when footerOn)
headerOn	bool	whether a header has been set
footerOn	bool	whether a footer has been set

pdf.GlyphUse

A record of one glyph used by embedded text: which font, its glyph id, and the Unicode codepoint it came from - collected so `render` can build the font's width array and ToUnicode map.

Field	Type	Description
font	string	the font resource name
gid	int	the glyph id
cp	int	the source Unicode codepoint

pdf.Image

A loaded raster image (from `loadImage`), ready to embed as a PDF image XObject and draw with `drawImage` . Register it in a document with `addImage` .

Field	Type	Description
name	string	the resource name (referenced in drawImage)
width	int	the pixel width
height	int	the pixel height
bits	int	bits per colour component
colorSpace	string	the PDF colour-space token (e.g. "/DeviceRGB")
filter	string	the stream filter ("DCTDecode" for JPEG, "FlateDecode" for PNG)
predictor	int	15 when the FlateDecode stream carries PNG predictors, else 0
colors	int	colour components per pixel (for the predictor DecodeParms)
decode	string	an optional /Decode array token (Adobe CMYK JPEG), else ""
data	bytes	the image stream bytes
smask	bytes	the soft-mask (alpha) stream, empty when opaque
hasSmask	bool	whether the image carries an alpha soft mask

pdf.LinkAnnot

A link annotation: a clickable rectangle on a page that opens a URI. The rect is given in PDF points by its lower-left corner and its size; the border is drawn invisibly, so only the underlying text (or graphic) shows.

Field	Type	Description
x	int	lower-left x of the clickable rectangle
y	int	lower-left y of the clickable rectangle
width	int	rectangle width in points
height	int	rectangle height in points
uri	string	the URI the link opens

pdf.LoadedFont

A loaded, embeddable font: a resource name (referenced in `textUnicode`) and the parsed `font.Font` . Register it in a document with `addFont` .

Field	Type	Description
name	string	the resource name (e.g. "Body")
f	font.Font	the parsed font

pdf.OutlineEntry

One outline (bookmark) entry: a title that jumps to a position on a page, plus the heading level that nests it under the outline tree. Built with `bookmark` .

Field	Type	Description
title	string	the bookmark label
page	int	the 0-based page index the bookmark points at
y	int	the destination y coordinate in PDF points (origin bottom-left)
level	int	the nesting level (1 = top; a level-2 entry nests under the last level-1)

pdf.Page

A single page: its size (points), its accumulated content-stream operators, and its link annotations.

Field	Type	Description
width	int	the page width in points
height	int	the page height in points
content	string	the content-stream operators built so far
fonts	list of string	the distinct standard-14 fonts referenced on this page
glyphUses	list of GlyphUse	embedded-font glyphs drawn on this page
annots	list of LinkAnnot	the page's link annotations

pdf.PageLabel

A running header or footer, drawn on every page at render time. Its three text slots are left- / centre- / right-aligned (alignment measured with the font's own metrics), and each may contain the placeholders `%page%` (the 1-based page number) and `%pages%` (the total page count), filled in per page - so a footer `"sample.pdf"` / `""` / `"%page%/%pages%"` reads `sample.pdf ... 13/108` . The placeholders are percent-delimited (not braces) so they do not collide with Jennifer's own `{expr}` string interpolation. `margin` is the distance from the page edge to the text on every side. With `border` on, a thin rule is drawn under a header / over a footer.

Field	Type	Description
left	string	left-aligned slot (may use %page% / %pages%)
center	string	centre-aligned slot
right	string	right-aligned slot
font	string	a standard-14 font name
size	int	the point size
margin	int	distance from the page edge, in points
border	bool	draw a separating rule
red	int	text / rule colour red 0-255
green	int	text / rule colour green 0-255
blue	int	text / rule colour blue 0-255
leftUri	string	if set, makes the left slot a link to this URI ("" = none)
centerUri	string	if set, makes the centre slot a link to this URI ("" = none)
rightUri	string	if set, makes the right slot a link to this URI ("" = none)

plot API reference

Data plotting to SVG: turn numbers into a self-contained <svg> chart string you can write to a .svg file or drop into HTML. A unified chart renders one or more Series (line / points / both / area, solid or dashed, with optional error bars and marker shapes) with a legend; line / scatter / bar / bars / histogram are focused wrappers. bars groups or stacks multiple series and supports negative (diverging) values. Charts support log scales, a date axis, fonts, configurable margins, positioned legends, reference lines, data labels, and native <title> hover tooltips. save writes the SVG to a file. Pure Jennifer over math and time / fs / strings / lists / convert ; both binaries; the visual companion to the stats / ml numeric stack.

Import with import "plot.j" as plot; . See the plot guide for prose and examples.

Functions

plot.bar(labels as list of string, values as list of float, opts as Options)

A single-series vertical bar chart: one bar per labelled category, from zero (negative values diverge below). A convenience over bars .

Parameters

- labels {list of string} - the category labels
- values {list of float} - the bar heights
- opts {Options} - chart options

Returns {string} - a complete SVG document

Throws

- {Error} - kind "plot" when labels / values are empty or unequal length

plot.bars(labels as list of string, data as list of Series, opts as Options)

A multi-series bar chart: one group of bars per labelled category. Each Series contributes its ys (one value per category) as a colour; series are drawn side by side ("grouped", opts.barMode) or on top of each other ("stacked"). Values may be negative (bars diverge from a zero baseline). A named series is listed in the legend.

Parameters

- labels {list of string} - the category labels, drawn under each group
- data {list of Series} - the series (each ys the same length as labels)
- opts {Options} - chart options

Returns {string} - a complete SVG document

Throws

- {Error} - kind "plot" when labels / data are empty or a series length differs

plot.chart(data as list of Series, opts as Options)

Render one or more Series on shared axes, with a legend for named series. Each series draws per its mark (line / points / both / area), with optional dashing, marker shapes, and error bars. The domain spans every series, its error bars, and any reference-line value on a linear axis.

Parameters

- data {list of Series} - the series to plot (each non-empty, xs / ys equal length)
- opts {Options} - chart options

Returns {string} - a complete SVG document

Throws

- `{Error}` - kind "plot" when data is empty, a series is malformed, or a log axis sees a non-positive value

plot.defaults()

Sensible defaults: a 640x400 canvas, no title / labels, blue data on white, a sans-serif 11px base font, gridlines and a top-right legend on, linear scales, grouped bars, tooltips on, no data labels, no reference lines.

Returns `{Options}` - the default options

plot.floats(xs as list of int)

Convert a `list of int` to a `list of float` , since the chart functions take float data and an int list does not match a float parameter.

Parameters

- `xs` `{list of int}` - the integers

Returns `{list of float}` - the same values as floats

plot.histogram(data as list of float, bins as int, opts as Options)

A histogram of `data` into `bins` equal-width buckets over the data range, drawing the bucket counts as adjacent bars.

Parameters

- `data` `{list of float}` - the samples
- `bins` `{int}` - the number of buckets (>= 1)
- `opts` `{Options}` - chart options

Returns `{string}` - a complete SVG document

Throws

- `{Error}` - kind "plot" when data is empty or bins < 1

plot.hline(value as float, label as string)

A horizontal reference line at y = value (a threshold / target).

Parameters

- `value` `{float}` - the y value
- `label` `{string}` - an optional caption ("" = none)

Returns `{RefLine}` - the reference line

plot.line(xs as list of float, ys as list of float, opts as Options)

A line chart of a single series. A convenience over `chart` .

Parameters

- `xs` `{list of float}` - the x coordinates
- `ys` `{list of float}` - the y coordinates
- `opts` `{Options}` - chart options

Returns `{string}` - a complete SVG document

Throws

- `{Error}` - kind "plot" when xs / ys are empty or unequal length

plot.points(name as string, xs as list of float, ys as list of float)

A points (scatter) series (mark "points").

Parameters

- `name` `{string}` - the legend label
- `xs` `{list of float}` - the x coordinates
- `ys` `{list of float}` - the y coordinates

Returns `{Series}` - the series

plot.save(svg as string, path as string)

Write an SVG chart string to a file, returning the path. A convenience over `fs.writeString` so `plot.save(plot.line(...), "chart.svg")` reads in one line. (Jennifer values have no methods, so it is `plot.save($svg, path)` , not `$svg.save(path)` .)

Parameters

- `svg` `{string}` - the SVG document (from any chart function)
- `path` `{string}` - the destination file path

Returns `{string}` - the path written

plot.scatter(xs as list of float, ys as list of float, opts as Options)

A scatter plot of a single series. A convenience over `chart` .

Parameters

- `xs` `{list of float}` - the x coordinates
- `ys` `{list of float}` - the y coordinates
- `opts` `{Options}` - chart options

Returns `{string}` - a complete SVG document

Throws

- `{Error}` - kind "plot" when xs / ys are empty or unequal length

plot.series(name as string, xs as list of float, ys as list of float)

A line series (mark "line"). Set `.mark` / `.color` / `.dash` / `.yErr` / `.shape` on the result.

Parameters

- `name` `{string}` - the legend label
- `xs` `{list of float}` - the x coordinates
- `ys` `{list of float}` - the y coordinates

Returns `{Series}` - the series

plot.vline(value as float, label as string)

A vertical reference line at x = value (a marker in time / along x).

Parameters

- `value` `{float}` - the x value
- `label` `{string}` - an optional caption ("" = none)

Returns `{RefLine}` - the reference line

Structs

plot.Options

Chart options: canvas size, captions, colours, fonts, margins, axis modes, legend placement, data labels, tooltips, and reference lines. Copy `defaults()` and set the fields you want.

Field	Type	Description
width	int	canvas width in pixels
height	int	canvas height in pixels
title	string	chart title, centred at the top ("" = none)
xLabel	string	x-axis caption ("" = none)
yLabel	string	y-axis caption, rotated ("" = none)
color	string	the default data colour (single-series line / scatter / bar)
background	string	the canvas background colour
fontFamily	string	the CSS font-family for all text
fontSize	int	the base tick-label size; the title and captions scale off it
grid	bool	draw gridlines (false = short tick marks instead)
legend	bool	draw a legend for named / multiple series
legendPos	string	"top-right" / "top-left" / "bottom-right" / "bottom-left"
xLog	bool	log10 x scale (values must be positive)
yLog	bool	log10 y scale (values must be positive)
xDate	bool	treat x values as Unix seconds and label them as dates
dateFormat	string	strftime pattern for date labels ("" = chosen by tick spacing)
marginLeft	int	left margin (room for y labels)
marginRight	int	right margin
marginTop	int	top margin (room for the title)
marginBottom	int	bottom margin (room for x labels)
barMode	string	"grouped" or "stacked" for a multi-series bars chart
barLabels	bool	draw the value above each grouped bar
hover	bool	attach <title> tooltips to marks (native browser hover)
refLines	list of RefLine	reference lines drawn over the data

plot.RefLine

A reference line drawn across the plot at a constant value on one axis - a threshold, target, or marker.

Field	Type	Description
axis	string	"y" for a horizontal line, "x" for a vertical one
value	float	the data value the line sits at
label	string	an optional caption drawn by the line ("" = none)
color	string	the line colour ("" = a default red)
dash	bool	dashed (true) or solid

plot.Series

One plotted data series. For line / scatter / area `xs` and `ys` are the coordinates; for `bars` only `ys` (the per-category values) is used. `yErr` , when the same length as `ys` , draws symmetric error bars (linear y only); `shape` picks the scatter marker.

Field	Type	Description
name	string	the legend label ("" = not listed)
xs	list of float	the x coordinates (unused by bars)

ys	list of float	the y coordinates / bar values
color	string	the series colour ("" = auto from the palette)
mark	string	"line" / "points" / "both" / "area"
dash	bool	draw the line dashed
yErr	list of float	symmetric +/- error per point ([] = none)
shape	string	marker for points: "circle" / "square" / "triangle" / "diamond"

pop API reference

A POP3 receive client (RFC 1939): the line-oriented status dialogue ("+OK" / "-ERR") over the `net` system library, with plaintext, implicit TLS, or STLS, and auth by USER / PASS, XOAUTH2, CRAM-MD5, or SCRAM-SHA-1 / SCRAM-SHA-256. Retrieved messages come back as strings, ready for the `mime` module to parse. Because it uses `net` , this module needs the default `jennifer` binary (`jennifer-tiny` has no network stack). A session is stateful: `connect` , then `stat` / `sizes` / `retrieve` / `deleteMessage` , then `quit` . `fetchAll` wraps the common "get every message" case. A server "-ERR" throws a catchable `Error` (kind "pop3").

Import with `import "pop.j" as pop;` . See the `pop` guide for prose and examples.

Functions

pop.connect(opts as Options)

Open a session: greet, optional STLS upgrade, then USER / PASS auth.

Parameters

- `opts` `{Options}` - the connection settings

Returns `{Session}` - a live authenticated session

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.count(session as Session)

Return just the number of messages waiting.

Parameters

- `session` `{Session}` - the live session

Returns `{int}` - the message count

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.deleteMessage(session as Session, n as int)

Mark message n for deletion (DELE); it is removed at QUIT.

Parameters

- `session` `{Session}` - the live session
- n `{int}` - the 1-based message number

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.fetchAll(opts as Options)

Connect, retrieve every message (without deleting), and quit.

Parameters

- `opts` `{Options}` - the connection settings

Returns `{list of string}` - every message as a raw string, in message order

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.noop(session as Session)

Do nothing but keep the connection alive (NOOP) - the server resets its idle timer and replies "+OK". Useful between long pauses in a session.

Parameters

- `session {Session}` - the live session

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.quit(session as Session)

End the session (committing any deletions) and close the connection.

Parameters

- `session {Session}` - the live session

pop.reset(session as Session)

Unmark every message marked for deletion this session (RSET), so a QUIT after it deletes nothing. Use it to abort a batch of `deleteMessage` calls before committing.

Parameters

- `session {Session}` - the live session

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.retrieve(session as Session, n as int)

Fetch message `n` as a raw message string (RETR), ready for `mime.parse` .

Parameters

- `session {Session}` - the live session
- `n {int}` - the 1-based message number

Returns `{string}` - the raw message text

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.sizes(session as Session)

Return the octet size of each message, in message order (LIST).

Parameters

- `session {Session}` - the live session

Returns `{list of int}` - the size in octets of each message

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.stat(session as Session)

Return the mailbox message count and total size.

Parameters

- `session` `{Session}` - the live session

Returns `{Stat}` - the mailbox totals

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

pop.top(session as Session, n as int, lines as int)

Preview message n : its full headers plus the first `lines` body lines (TOP). `lines` = 0 fetches headers only - a cheap way to read Subject / From / Date (parse the result with `mime.parse`) and decide whether to `retrieve` the whole body. TOP is widely but not universally supported (a server lacking it answers "-ERR").

Parameters

- `session` `{Session}` - the live session
- n `{int}` - the 1-based message number
- `lines` `{int}` - how many leading body lines to include (0 = headers only)

Returns `{string}` - the header block plus the requested body lines

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply (or no TOP support)

pop.uidl(session as Session)

Return the stable unique id of every message (UIDL), as `MessageId` `{number, id}` pairs in message order. Unlike the per-session message number, a UIDL `id` persists across sessions, so this is the key to leave-on-server / skip-seen: keep the ids you have processed, and next time retrieve only the `number` s whose `id` is new.

Parameters

- `session` `{Session}` - the live session

Returns `{list of MessageId}` - the unique id of each message

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply (or no UIDL support)

pop.uidlOne(session as Session, n as int)

Return the stable unique id of one message (UIDL n).

Parameters

- `session` `{Session}` - the live session
- n `{int}` - the 1-based message number

Returns `{string}` - the message's persistent unique id

Throws

- `{Error}` - kind "pop3" on a server "-ERR" reply

Structs

pop.MessageId

One message's UIDL entry: its message number and its server-assigned unique id. The `id` is stable across sessions (unlike the `number` , which is per-session), so it is the key for "download only what is new": remember the ids you have seen, and on the next connection retrieve only the numbers whose id is new.

Field	Type	Description
number	int	the 1-based message number this session
id	string	the persistent unique id (RFC 1939 UIDL)

pop.Options

Connection settings. `security` is "none", "tls" (implicit TLS on connect, port 995), or "starttls" (STLS upgrade on port 110). `auth` is "" (USER / PASS) or "xoauth2" (SASL bearer token, where `pass` holds the access token).

Field	Type	Description
host	string	the server hostname
port	int	the server port (110 plaintext / STLS, 995 implicit TLS)
security	transport.Security	transport.Security.None, .Tls, or .Starttls
user	string	the account username
pass	string	the password (or access token for xoauth2)
auth	string	"" (default - USER / PASS), "auto" (pick the strongest mechanism the server offers, falling back to USER / PASS), "apop" (RFC 1939 APOP), "xoauth2", "cram" (CRAM-MD5), "scram-sha-1", or "scram-sha-256"

pop.Session

A live POP3 session over one connection.

Field	Type	Description
conn	net.Conn	the underlying network connection

pop.Stat

Mailbox totals from STAT.

Field	Type	Description
count	int	the number of messages in the mailbox
size	int	the total mailbox size in octets

prometheus API reference

A Prometheus metrics module in two halves. **Exposition** builds a metric set and renders the Prometheus text format (`# HELP` / `# TYPE` / sample lines) - pure text over `strings` / `maps` / `lists` / `convert` , transport-agnostic (write the string to a `*.prom` file for the node_exporter textfile collector, POST it to a Pushgateway, or serve it from a `/metrics` handler). **Retrieval** is a read client for Prometheus's HTTP query API (`query` / `queryRange`) over the `http` module + `json` . The exposition half runs on both binaries; the query half needs the default `jennifer` binary (it uses `net` through `http`).

Import with `import "prometheus.j" as prometheus;` . See the `prometheus` guide for prose and examples.

Functions

prometheus.counter(name as string, help as string)

Build an empty counter metric. Throws on an invalid metric name.

Parameters

- `name {string}` - the metric name (`[a-zA-Z_:][a-zA-Z0-9_:]*`)
- `help {string}` - the HELP text (empty to omit the HELP line)

Returns `{Metric}` - the new counter metric

Throws

- `{Error}` - kind "prometheus" when the name is invalid

prometheus.gauge(name as string, help as string)

Build an empty gauge metric. Throws on an invalid metric name.

Parameters

- `name {string}` - the metric name (`[a-zA-Z_:][a-zA-Z0-9_:]*`)
- `help {string}` - the HELP text (empty to omit the HELP line)

Returns `{Metric}` - the new gauge metric

Throws

- `{Error}` - kind "prometheus" when the name is invalid

prometheus.histogram(name as string, help as string, buckets as list of float)

Build an empty histogram metric. Buckets are the cumulative upper bounds (`le`); they are sorted ascending, and an `+Inf` bucket is always rendered in addition to the given bounds. Throws on an invalid metric name.

Parameters

- `name {string}` - the metric name (`[a-zA-Z_:][a-zA-Z0-9_:]*`)
- `help {string}` - the HELP text (empty to omit the HELP line)
- `buckets {list of float}` - the cumulative upper bounds (`le`); sorted ascending internally

Returns `{Metric}` - the new histogram metric

Throws

- `{Error}` - kind "prometheus" when the name is invalid

prometheus.observe(metric as Metric, labels as map of string to string, value as float)

Record an observation for a label set, returning a new Metric (value-semantic). For a `Counter` / `Gauge` , a sample with an equal label set is replaced (last write wins). For a `Histogram` / `Summary` , the observation accumulates: the count and sum grow, histogram buckets increment, and summary observations are retained for quantile computation. Throws on an invalid label name.

Parameters

- `metric` `{Metric}` - the metric to extend
- `labels` `{map of string to string}` - the sample's label set ({} for none)
- `value` `{float}` - the observed value

Returns `{Metric}` - a new Metric with the observation recorded

Throws

- `{Error}` - kind "prometheus" when a label name is invalid

prometheus.observeAt(metric as Metric, labels as map of string to string, value as float, timestampMs as int)

Record an observation carrying an explicit millisecond timestamp, returning a new Metric (value-semantic). Same accumulation rules as `observe` ; the timestamp is appended after the value in the rendered exposition (`metric value timestamp`). Throws on an invalid label name.

Parameters

- `metric` `{Metric}` - the metric to extend
- `labels` `{map of string to string}` - the sample's label set ({} for none)
- `value` `{float}` - the observed value
- `timestampMs` `{int}` - the sample timestamp in milliseconds since the Unix epoch

Returns `{Metric}` - a new Metric with the timestamped observation recorded

Throws

- `{Error}` - kind "prometheus" when a label name is invalid

prometheus.pushgatewayPath(job as string, grouping as map of string to string)

Build the Pushgateway URL path for a job and a set of grouping labels: `/metrics/job/<job>/<k1>/<v1>/...` with the job name and label values percent-encoded and the grouping keys sorted for a deterministic path. The caller POSTs the `render` output to `base` + `pushgatewayPath(job, grouping)` . A pure string helper - no network. Throws on an invalid grouping label name.

Parameters

- `job` `{string}` - the Pushgateway job name
- `grouping` `{map of string to string}` - additional grouping labels ({} for none)

Returns `{string}` - the Pushgateway path segment (leading slash, no trailing slash)

Throws

- `{Error}` - kind "prometheus" when a grouping label name is invalid

prometheus.query(base as string, promql as string)

Run an instant query against `base` (a Prometheus server URL) via `/api/v1/query` , returning the parsed

result set.

Parameters

- `base` `{string}` - the Prometheus base URL (e.g. "http://localhost:9090")
- `promql` `{string}` - the PromQL expression

Returns `{Result}` - the parsed result set

Throws

- `{Error}` - kind "prometheus" when the server reports a query error

prometheus.queryRange(base as string, promql as string, start as string, end as string, step as string)

Run a range query against `base` via `/api/v1/query_range` , returning the parsed result matrix. `start` / `end` are RFC 3339 or Unix-timestamp strings; `step` is a duration ("15s") or a seconds string.

Parameters

- `base` `{string}` - the Prometheus base URL
- `promql` `{string}` - the PromQL expression
- `start` `{string}` - the range start (RFC 3339 or Unix timestamp)
- `end` `{string}` - the range end (RFC 3339 or Unix timestamp)
- `step` `{string}` - the resolution step (duration or seconds)

Returns `{Result}` - the parsed result matrix

Throws

- `{Error}` - kind "prometheus" when the server reports a query error

prometheus.render(metrics as list of Metric)

Render a list of metrics as the Prometheus text exposition format. A `Histogram` emits `_bucket{le="..."}` (cumulative, ascending, `+Inf` last), `_sum` , and `_count` series; a `Summary` emits `{quantile="..."}` , `_sum` , and `_count` series. A sample carrying a timestamp appends it after the value.

Parameters

- `metrics` `{list of Metric}` - the metrics to render

Returns `{string}` - the exposition text (one trailing newline per line)

prometheus.summary(name as string, help as string, quantiles as list of float)

Build an empty summary metric. Quantiles are values in `[0, 1]` ; they are sorted ascending and reported as `{quantile="..."}` child series, computed from the observed values at render time. Throws on an invalid metric name or an out-of-range quantile.

Parameters

- `name` `{string}` - the metric name (`[a-zA-Z_:][a-zA-Z0-9_:]*`)
- `help` `{string}` - the HELP text (empty to omit the HELP line)
- `quantiles` `{list of float}` - the reported quantiles, each in `[0, 1]` ; sorted ascending internally

Returns `{Metric}` - the new summary metric

Throws

- `{Error}` - kind "prometheus" when the name is invalid or a quantile is out of range

Structs

prometheus.Metric

A metric family: a name, help text, a type, its samples, and (for a `Histogram` / `Summary`) its bucket bounds / quantiles.

Field	Type	Description
name	string	the metric name ([a-zA-Z_:][a-zA-Z0-9_:]*)
help	string	the HELP text (empty to omit the HELP line)
type	MetricType	the metric type
samples	list of Sample	the recorded series
buckets	list of float	the histogram upper bounds (le), ascending; empty for other types
quantiles	list of float	the summary quantiles in [0, 1], ascending; empty for other types

prometheus.Point

One (timestamp, value) point of a query result series.

Field	Type	Description
timestamp	float	the sample time as a Unix timestamp (seconds)
value	float	the sample value

prometheus.Result

A parsed query result set.

Field	Type	Description
resultType	string	"vector", "matrix", "scalar", or "string"
series	list of Series	the result series

prometheus.Sample

One recorded series of a metric, keyed by its label set. For a `Counter` or `Gauge` only `value` is used. For a `Histogram`, `count` / `sum` accumulate and `buckets` holds the cumulative per-bucket counts (parallel to the owning `Metric.buckets`). For a `Summary`, `count` / `sum` accumulate and `observations` holds the raw observed values (quantiles are computed at render time). An optional millisecond `timestamp` is appended after the value when `hasTimestamp` is set.

Field	Type	Description
labels	map of string to string	the label name/value pairs ({} for none)
value	float	the sample value (Counter / Gauge)
count	float	the observation count (Histogram / Summary)
sum	float	the observation sum (Histogram / Summary)
buckets	list of float	cumulative per-bucket counts (Histogram; parallel to Metric.buckets)
observations	list of float	the raw observed values (Summary; source for quantiles)
timestamp	int	an explicit millisecond timestamp, appended after the value when set
hasTimestamp	bool	whether timestamp is present and should be rendered

prometheus.Series

One result series: its label set and its points (one point for an instant vector, many for a range matrix).

Field	Type	Description
metric	map of string to string	the series label set
values	list of Point	the series points

Enums

prometheus.MetricType

A Prometheus metric type: `Counter` (a monotonically increasing total), `Gauge` (a value that can go up and down), `Histogram` (bucketed observations with `_bucket` / `_sum` / `_count` child series), or `Summary` (quantile observations with `{quantile="..."}` / `_sum` / `_count` series).

ratelimit API reference

A rate limiter over a selectable key/value backend (`kvstore.Store` - `memcache` , `redis` , or the in-process `kv`). Each key (an IP, a user, an API token) is counted against a limit per time window; a `check` records a hit and returns a `Result` with the decision plus the metadata for a compliant `429` (remaining budget, reset time, and Retry-After). Two algorithms, both driven by the wall clock and window-aligned keys (so they work identically on every backend, needing only atomic increment): - **fixed window** - a counter per aligned window; simplest, but a burst can straddle a window boundary (up to 2x the limit across the seam). - **sliding window** - a weighted blend of the current and previous window counts, which smooths that boundary burst.

Import with `import "ratelimit.j" as ratelimit;` . See the `ratelimit` guide for prose and examples.

Functions

ratelimit.check(limiter as Limiter, key as string)

Record one hit against `key` and return the `Result` (decision + remaining + reset + retry-after). The counter is created and armed with the window TTL on the first hit, so it resets on its own - nothing to reap.

Parameters

- `limiter` `{Limiter}` - the configured limiter
- `key` `{string}` - the counter key (an IP, user, or API token)

Returns `{Result}` - the decision and rate-limit metadata

ratelimit.fixedWindow(store as kvstore.Store, limit as int, window as int)

A fixed-window limiter: `limit` hits per aligned `window` seconds.

Parameters

- `store` `{kvstore.Store}` - the backend store
- `limit` `{int}` - the maximum hits per window
- `window` `{int}` - the window length in seconds

Returns `{Limiter}` - the configured limiter

ratelimit.peek(limiter as Limiter, key as string)

Report the current state for `key` **without** recording a hit: `allowed` is whether the next hit would be within the limit, plus the same remaining / reset / retry-after metadata.

Parameters

- `limiter` `{Limiter}` - the configured limiter
- `key` `{string}` - the counter key

Returns `{Result}` - the current rate-limit state

ratelimit.slidingWindow(store as kvstore.Store, limit as int, window as int)

A sliding-window limiter: like `fixedWindow` , but blends the current and previous window counts so a burst cannot straddle a window boundary.

Parameters

- `store` `{kvstore.Store}` - the backend store
- `limit` `{int}` - the maximum hits per window
- `window` `{int}` - the window length in seconds

Returns `{Limiter}` - the configured limiter

Structs

ratelimit.Limiter

A configured limiter: a backend store, a per-window `limit` , a `window` in seconds, and the algorithm. Value-semantic; build with `fixedWindow` / `slidingWindow` .

Field	Type	Description
store	kvstore.Store	the backend store
limit	int	the maximum hits allowed per window
window	int	the window length in seconds
algorithm	string	"fixed" or "sliding"

ratelimit.Result

The outcome of a `check` / `peek` .

Field	Type	Description
allowed	bool	whether this hit is within the limit
remaining	int	hits left in the window before the limit (0 once exhausted)
retryAfter	int	seconds to wait before retrying (0 when allowed) - the Retry-After value
resetSeconds	int	seconds until the current window rolls over

redis API reference

A Redis client speaking RESP2 (the REdis Serialization Protocol) over the `net` system library. Commands go out as RESP arrays of bulk strings; replies (`+OK` , `-ERR` , `:int` , `$bulk` , `*array`) parse back into a `Reply` . Typed per-command helpers (`get` / `set` / `incr` / `keys` / ...) keep the common path fully typed; `command` is the generic escape hatch for anything else. A `-ERR` reply throws a catchable `Error` (kind "redis"). The reply parser frames over bytes and counts bulk-string lengths in bytes, so a value whose byte length differs from its rune length (any non-ASCII UTF-8 text) is read byte-exact. Needs the default `jennifer` binary (uses `net`).

Import with `import "redis.j" as redis;` . See the `redis` guide for prose and examples.

Functions

redis.command(session as Session, args as list of string)

Send one command (its arguments) and return the reply.

Parameters

- `session` `{Session}` - the open session
- `args` `{list of string}` - the command name and its arguments

Returns `{Reply}` - the parsed reply

Throws

- `{Error}` - kind "redis" on a `-ERR` reply

redis.connect(opts as Options)

Open a session, authenticating and selecting a database when set.

Parameters

- `opts` `{Options}` - the connection settings

Returns `{Session}` - the open session

redis.decr(session as Session, key as string)

Atomically decrement `key` and return the new value.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the counter key

Returns `{int}` - the new value

redis.del(session as Session, key as string)

Delete `key` and return the number of keys removed (0 or 1).

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to delete

Returns `{int}` - the number of keys removed (0 or 1)

redis.discard(session as Session)

Discard the queued transaction (DISCARD).

Parameters

- `session {Session}` - the open session

redis.exec(session as Session)

Execute the queued transaction (EXEC) and return one reply per queued command, in order. An aborted transaction (a `nil` EXEC reply) yields an empty list.

Parameters

- `session {Session}` - the open session

Returns `{list of Reply}` - the queued commands' replies

redis.exists(session as Session, key as string)

Report whether `key` is present.

Parameters

- `session {Session}` - the open session
- `key {string}` - the key to test

Returns `{bool}` - whether the key exists

redis.get(session as Session, key as string)

Return the string value of `key` , or "" when the key is missing.

Parameters

- `session {Session}` - the open session
- `key {string}` - the key to read

Returns `{string}` - the value, or "" when missing

redis.getBytes(session as Session, key as string)

Return the raw `bytes` value of `key` , or empty `bytes` when the key is missing (use `exists` to tell an empty value from a missing key). The byte-exact counterpart to `get` : it never UTF-8-decodes, so a binary value stored with `setBytes` round-trips exactly.

Parameters

- `session {Session}` - the open session
- `key {string}` - the key to read

Returns `{bytes}` - the value, or empty bytes when missing

Throws

- `{Error}` - kind "redis" on a `-ERR` reply

redis.hdel(session as Session, key as string, field as string)

Delete `field` from hash `key` (HDEL); returns the number removed (0 or 1).

Parameters

- `session {Session}` - the open session
- `key {string}` - the hash key
- `field {string}` - the field to delete

Returns `{int}` - the number of fields removed

redis.hget(session as Session, key as string, field as string)

Return hash `key` 's `field` (HGET), or "" when the field or key is missing.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the hash key
- `field` `{string}` - the field name

Returns `{string}` - the value, or "" when missing

redis.hgetAll(session as Session, key as string)

Return every field and value of hash `key` (HGETALL) as a `map of string to string` (empty when the key is missing).

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the hash key

Returns `{map of string to string}` - the field -> value pairs

redis.hset(session as Session, key as string, field as string, value as string)

Set hash `key` 's `field` to `value` (HSET); returns the number of fields newly created (0 when it already existed and was updated).

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the hash key
- `field` `{string}` - the field name
- `value` `{string}` - the field value

Returns `{int}` - the number of new fields (0 or 1)

redis.incr(session as Session, key as string)

Atomically increment `key` and return the new value.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the counter key

Returns `{int}` - the new value

redis.keys(session as Session, pattern as string)

Return the keys matching a glob `pattern` (e.g. " *", "user:* ").

Parameters

- `session` `{Session}` - the open session
- `pattern` `{string}` - the glob pattern

Returns `{list of string}` - the matching keys

redis.llen(session as Session, key as string)

Return the length of list `key` (LLEN); 0 when the key is missing.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the list key

Returns `{int}` - the list length

redis.lpop(session as Session, key as string)

Remove and return the first element of list `key` (LPOP), or "" when empty.

Parameters

- `session {Session}` - the open session
- `key {string}` - the list key

Returns `{string}` - the popped element, or "" when the list is empty

redis.lpush(session as Session, key as string, value as string)

Prepend `value` to list `key` (LPUSH); returns the list's new length.

Parameters

- `session {Session}` - the open session
- `key {string}` - the list key
- `value {string}` - the value to prepend

Returns `{int}` - the new list length

redis.lrange(session as Session, key as string, start as int, stop as int)

Return the elements of list `key` from `start` to `stop` inclusive (LRANGE; negative indices count from the end, so `0, -1` is the whole list).

Parameters

- `session {Session}` - the open session
- `key {string}` - the list key
- `start {int}` - the start index
- `stop {int}` - the stop index (inclusive)

Returns `{list of string}` - the elements in range

redis.multi(session as Session)

Begin a transaction (MULTI). Commands issued after this are queued (each replies "+QUEUED") until `exec` runs them atomically or `discard` drops them.

Parameters

- `session {Session}` - the open session

redis.ping(session as Session)

Return the server's PONG (a liveness check).

Parameters

- `session {Session}` - the open session

Returns `{string}` - the server's reply ("PONG")

redis.pipeline(session as Session, commands as list of list of string)

Send several commands in one write and read exactly one reply per command (a single network round trip). Unlike `command` , a `-ERR` reply does not throw - each command's outcome is returned in order, so inspect each `Reply` .

Parameters

- `session {Session}` - the open session

- `commands` `{list of list of string}` - each command as an argument list

Returns `{list of Reply}` - one reply per command, in order

redis.psubscribe(session as Session, patterns as list of string)

Subscribe to one or more glob patterns (e.g. "news.*").

Parameters

- `session` `{Session}` - the open session
- `patterns` `{list of string}` - the patterns to subscribe to

redis.publish(session as Session, channel as string, message as string)

Publish `message` to `channel` and return the number of subscribers that received it. Runs on an ordinary (non-subscribed) connection.

Parameters

- `session` `{Session}` - the open session
- `channel` `{string}` - the channel to publish to
- `message` `{string}` - the message body

Returns `{int}` - the number of subscribers that received the message

redis.punsubscribe(session as Session, patterns as list of string)

Unsubscribe from the given patterns, or from all patterns when the list is empty.

Parameters

- `session` `{Session}` - the open session
- `patterns` `{list of string}` - the patterns to leave ([] means all)

redis.quit(session as Session)

End the session and close the connection.

Parameters

- `session` `{Session}` - the open session

redis.receiveMessage(session as Session)

Block until the next `message` / `pmessage` push arrives, skipping the (un)subscribe confirmation frames. Wrap the call in a `spawn` to receive concurrently with the rest of the program; the per-read idle timeout (`Session.timeout` , milliseconds) bounds the wait and raises a catchable error on a stall.

Parameters

- `session` `{Session}` - the open, subscribed session

Returns `{Message}` - the next channel / pattern push

Throws

- `{Error}` - kind "redis" on a server error reply or a closed connection

redis.rpush(session as Session, key as string, value as string)

Append `value` to list `key` (RPUSH); returns the list's new length.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the list key

- `value` `{string}` - the value to append

Returns `{int}` - the new list length

redis.sadd(session as Session, key as string, member as string)

Add `member` to set `key` (SADD); returns the number newly added (0 or 1).

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the set key
- `member` `{string}` - the member to add

Returns `{int}` - the number of members added

redis.scan(session as Session, cursor as int, pattern as string, count as int)

Walk the keyspace one page at a time (SCAN), the production-safe iterator: start with `cursor` 0 and repeat until the returned cursor is 0 again. Pass `pattern` "" to skip the MATCH glob and `count` 0 to skip the COUNT hint. Prefer this over `keys` , which blocks the server on a large keyspace.

Parameters

- `session` `{Session}` - the open session
- `cursor` `{int}` - the cursor (0 to begin)
- `pattern` `{string}` - a glob to filter keys ("" for no filter)
- `count` `{int}` - a per-page size hint (0 for the server default)

Returns `{ScanResult}` - the next cursor and this page's keys

redis.scard(session as Session, key as string)

Return the number of members in set `key` (SCARD); 0 when missing.

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the set key

Returns `{int}` - the member count

redis.set(session as Session, key as string, value as string)

Store `value` at `key` .

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to write
- `value` `{string}` - the value to store

redis.setBytes(session as Session, key as string, value as bytes)

Store a raw `bytes` value at `key` , byte-for-byte. Unlike `set` (whose string value is UTF-8-encoded onto the wire), this stores arbitrary binary - a serialized blob, a compressed payload, an image - which `getBytes` reads back exactly. Read it with `getBytes` , not `get` (the string reader throws on a non-UTF-8 value).

Parameters

- `session` `{Session}` - the open session
- `key` `{string}` - the key to write
- `value` `{bytes}` - the raw value to store

Throws

- `{Error}` - kind "redis" on a `-ERR` reply

redis.sismember(session as Session, key as string, member as string)

Report whether `member` is in set `key` (SISMEMBER).

Parameters

- `session {Session}` - the open session
- `key {string}` - the set key
- `member {string}` - the member to test

Returns `{bool}` - whether the member is present

redis.smembers(session as Session, key as string)

Return every member of set `key` (SMEMBERS) as a list (empty when missing; order is unspecified).

Parameters

- `session {Session}` - the open session
- `key {string}` - the set key

Returns `{list of string}` - the members

redis.srem(session as Session, key as string, member as string)

Remove `member` from set `key` (SREM); returns the number removed (0 or 1).

Parameters

- `session {Session}` - the open session
- `key {string}` - the set key
- `member {string}` - the member to remove

Returns `{int}` - the number of members removed

redis.subscribe(session as Session, channels as list of string)

Subscribe to one or more channels. After subscribing, the connection is in subscribed mode and may only run (P)SUBSCRIBE / (P)UNSUBSCRIBE / PING / QUIT until fully unsubscribed; read pushes with `receiveMessage` .

Parameters

- `session {Session}` - the open session
- `channels {list of string}` - the channels to subscribe to

redis.unsubscribe(session as Session, channels as list of string)

Unsubscribe from the given channels, or from all channels when the list is empty.

Parameters

- `session {Session}` - the open session
- `channels {list of string}` - the channels to leave ([] means all)

Structs

redis.Message

A pushed pub/sub message (from `receiveMessage`).

Field	Type	Description

kind	string	"message" (a plain channel push) or "pmessage" (a pattern push)
channel	string	the channel the message was published to
pattern	string	the subscribed glob pattern ("" for a plain message)
payload	string	the message body

redis.Options

Connection settings.

Field	Type	Description
host	string	the server host
port	int	the server port
security	transport.Security	transport.Security.None (plaintext) or .Tls (rediss); .Starttls is rejected (Redis has no in-band upgrade)
user	string	the AUTH username ("" for password-only or no auth)
password	string	the AUTH password; "" skips AUTH
db	int	the database to SELECT (0 is the default)

redis.Reply

A parsed RESP reply.

Field	Type	Description
kind	string	"string" (simple or bulk), "error", "int", "nil", or "array"
str	string	the string / error text
num	int	the integer value
items	list of Reply	an array reply's elements

redis.ScanResult

One step of a `scan` cursor walk.

Field	Type	Description
cursor	int	the cursor to pass to the next scan (0 ends the walk)
keys	list of string	the keys returned by this step

redis.Session

An open Redis connection.

Field	Type	Description
conn	net.Conn	the underlying socket
timeout	int	per-read idle timeout in milliseconds (0 disables it)

resque API reference

Background jobs on Redis, wire-compatible with Resque. Schedule a job onto a named queue now (`enqueue`), reserve and run it from a worker later (`reserve`). The Redis layout is the de-facto Resque standard - queues are lists at `resque:queue:NAME` , the queue registry is a set at `resque:queues` , and a job is the JSON envelope `{"class":"WorkerName","args":[...]}` - so a job Jennifer enqueues can be processed by a Ruby-resque / php-resque worker and vice versa. `args` is a `list of string` mapping to Ruby-resque's positional arguments; the class string is resolved on the worker's side. Built on the `redis` module, so it needs the default `jennifer` binary.

Import with `import "resque.j" as resque;` . See the `resque` guide for prose and examples.

Functions

resque.enqueue(session as redis.Session, queue as string, class as string, args as list of string)

Schedule a job: register the queue, then push the envelope onto it.

Parameters

- `session` `{redis.Session}` - the open Redis session
- `queue` `{string}` - the queue name to push onto
- `class` `{string}` - the worker class the job names
- `args` `{list of string}` - the job's positional arguments

resque.fail(session as redis.Session, job as Job, message as string)

Record a failed job on the failed list (a simplified failure entry).

Parameters

- `session` `{redis.Session}` - the open Redis session
- `job` `{Job}` - the job that failed
- `message` `{string}` - the failure message

resque.queueLength(session as redis.Session, queue as string)

Return the number of pending jobs on one queue.

Parameters

- `session` `{redis.Session}` - the open Redis session
- `queue` `{string}` - the queue name

Returns `{int}` - the number of pending jobs

resque.queues(session as redis.Session)

Return the registered queue names.

Parameters

- `session` `{redis.Session}` - the open Redis session

Returns `{list of string}` - the registered queue names

resque.reserve(session as redis.Session, queues as list of string)

Pop the next job from the first non-empty queue, checking `queues` in the given priority order.

Parameters

- `session` `{redis.Session}` - the open Redis session
- `queues` `{list of string}` - the queue names to check, in priority order

Returns `{Job}` - the reserved job, or an empty `Job` (`class` "") when all drained

resque.reserveBlocking(session as redis.Session, queues as list of string, timeoutSec as int)

Reserve the next job, BLOCKING on the given queues until one has work or `timeoutSec` elapses (Redis `BLPOP`) instead of polling with `reserve` . A worker loop parks on the socket rather than spinning. Queues are honoured in priority order (BLPOP returns from the first non-empty). `timeoutSec` 0 blocks indefinitely (Redis semantics).

Parameters

- `session` `{redis.Session}` - the open Redis session
- `queues` `{list of string}` - the queue names to watch, in priority order
- `timeoutSec` `{int}` - seconds to block; 0 blocks until a job arrives

Returns `{Job}` - the reserved job, or an empty `Job` (`class` "") on timeout

resque.size(session as redis.Session)

Return the total number of pending jobs across every registered queue.

Parameters

- `session` `{redis.Session}` - the open Redis session

Returns `{int}` - the total number of pending jobs

Structs

resque.Job

A reserved job. A drained `reserve` returns an empty `Job` (`class` is "").

Field	Type	Description
queue	string	the origin queue the job came from
class	string	the worker class to run
args	list of string	the job's positional string arguments

rest API reference

An ergonomic REST layer over the `http` client and `json` . Hold a value-semantic Client (base URL + default headers + an `http.Options` request policy) and call JSON-aware verbs; the module handles base-URL joining, query strings, `Content-Type` , and Bearer / Basic auth headers. It is pure composition - no sockets, no parsing of its own - so all the transport lives in `http` (which uses `net`), and this module needs the default `jennifer` binary. A 4xx / 5xx is a normal Response (inspect `.status`), not a crash. Build a client with `rest.client` ; layer on TLS (`rest.withCA` / `rest.insecure`), a per-request timeout (`rest.withTimeout`), redirect-following (`rest.withRedirects`), and retries (`rest.withRetries`) - each inherited from `http.send` . For paginated collections, `rest.paginate` (Link header) and `rest.paginateCursor` (cursor field) walk every page.

Import with `import "rest.j" as rest;` . See the `rest` guide for prose and examples.

Functions

rest.basic(user as string, pass as string)

Build an `Authorization` value for HTTP Basic auth (base64 of "user:password").

Parameters

- `user` `{string}` - the username
- `pass` `{string}` - the password

Returns `{string}` - the "Basic <base64>" header value

rest.bearer(token as string)

Build an `Authorization` value for a Bearer token.

Parameters

- `token` `{string}` - the bearer token

Returns `{string}` - the "Bearer <token>" header value

rest.client(baseUrl as string)

Build a Client for a base URL, with no default headers and full TLS verification. Layer auth / headers on with `withHeader` and TLS relaxation with `withCA` / `insecure` .

Parameters

- `baseUrl` `{string}` - the base URL every path joins onto

Returns `{Client}` - a new client

rest.delete(c as Client, path as string, query as map of string to string)

Issue a DELETE with an optional query map.

Parameters

- `c` `{Client}` - the client
- `path` `{string}` - the request path, joined onto the base URL
- `query` `{map of string to string}` - query parameters ({} for none)

Returns `{Response}` - the response

rest.get(c as Client, path as string, query as map of string to string)

Issue a GET with an optional query map ({} for none).

Parameters

- `c {Client}` - the client
- `path {string}` - the request path, joined onto the base URL
- `query {map of string to string}` - query parameters ({} for none)

Returns `{Response}` - the response

rest.getJson(c as Client, path as string, query as map of string to string)

Issue a GET and decode the response body as JSON.

Parameters

- `c {Client}` - the client
- `path {string}` - the request path, joined onto the base URL
- `query {map of string to string}` - query parameters ({} for none)

Returns `{json.Value}` - the decoded response body

rest.insecure(c as Client)

Return a copy of the client that skips TLS certificate verification for every `https://` request. This disables server authentication and exposes the connection to a man-in-the-middle - use only for a trusted LAN endpoint you cannot give a proper CA; prefer `withCA` .

Parameters

- `c {Client}` - the client to copy

Returns `{Client}` - a new client that accepts any certificate

rest.paginate(c as Client, path as string, query as map of string to string, maxPages as int)

Walk a **Link-header** paginated collection (GitHub / GitLab style), following each response's `Link: <url>; rel="next"` until it is absent or `maxPages` is reached, and return the list of decoded page bodies. The first request uses `path` + `query` ; each `next` URL is followed verbatim (it carries its own query). `maxPages` bounds the walk so a mis-behaving server cannot loop forever.

Parameters

- `c {Client}` - the client
- `path {string}` - the first page's path, joined onto the base URL
- `query {map of string to string}` - query parameters for the first page ({} for none)
- `maxPages {int}` - the maximum number of pages to fetch

Returns `{list of json.Value}` - the decoded body of each page, in order

Throws

- `{Error}` - kind "rest" on a non-2xx page

rest.paginateCursor(c as Client, path as string, query as map of string to string, cursorPointer as string, cursorParam as string, maxPages as int)

Walk a **cursor** paginated collection: fetch a page, read the next cursor from the response body at `cursorPointer` (a JSON Pointer, e.g. "/meta/next_cursor"), and re-request with that cursor set as the `cursorParam` query parameter, until the cursor is absent / null / empty or `maxPages` is reached. Returns the list of decoded page bodies. The cursor may be a JSON string or integer.

Parameters

- `c {Client}` - the client

- `path {string}` - the collection path, joined onto the base URL
- `query {map of string to string}` - the initial query parameters ({} for none)
- `cursorPointer {string}` - a JSON Pointer to the next cursor in each page body
- `cursorParam {string}` - the query-parameter name to send the cursor as
- `maxPages {int}` - the maximum number of pages to fetch

Returns `{list of json.Value}` - the decoded body of each page, in order

Throws

- `{Error}` - kind "rest" on a non-2xx page

rest.patch(c as Client, path as string, contentType as string, body as string)

Issue a PATCH with a content type and body.

Parameters

- `c {Client}` - the client
- `path {string}` - the request path, joined onto the base URL
- `contentType {string}` - the `Content-Type` header value
- `body {string}` - the request body

Returns `{Response}` - the response

rest.patchJson(c as Client, path as string, body as json.Value)

Issue a PATCH with a JSON body.

Parameters

- `c {Client}` - the client
- `path {string}` - the request path, joined onto the base URL
- `body {json.Value}` - the JSON request body

Returns `{Response}` - the response

rest.post(c as Client, path as string, contentType as string, body as string)

Issue a POST with a content type and body.

Parameters

- `c {Client}` - the client
- `path {string}` - the request path, joined onto the base URL
- `contentType {string}` - the `Content-Type` header value
- `body {string}` - the request body

Returns `{Response}` - the response

rest.postJson(c as Client, path as string, body as json.Value)

Issue a POST with a JSON body; returns the Response (inspect status).

Parameters

- `c {Client}` - the client
- `path {string}` - the request path, joined onto the base URL
- `body {json.Value}` - the JSON request body

Returns `{Response}` - the response

rest.put(c as Client, path as string, contentType as string, body as string)

Issue a PUT with a content type and body.

Parameters

- `c` `{Client}` - the client
- `path` `{string}` - the request path, joined onto the base URL
- `contentType` `{string}` - the `Content-Type` header value
- `body` `{string}` - the request body

Returns `{Response}` - the response

rest.putJson(c as Client, path as string, body as json.Value)

Issue a PUT with a JSON body.

Parameters

- `c` `{Client}` - the client
- `path` `{string}` - the request path, joined onto the base URL
- `body` `{json.Value}` - the JSON request body

Returns `{Response}` - the response

rest.withBackoff(c as Client, backoffMs as int)

Return a copy of the client with a base retry backoff in milliseconds (doubled each attempt; 0 = the 250ms default). Pairs with `rest.withRetries` .

Parameters

- `c` `{Client}` - the client to copy
- `backoffMs` `{int}` - the base backoff in milliseconds

Returns `{Client}` - a new client with the backoff set

rest.withCA(c as Client, pem as bytes)

Return a copy of the client that trusts a private-CA / self-signed PEM certificate (in addition to the system roots) for every `https://` request. The safer alternative to `insecure` : the server is still authenticated.

Parameters

- `c` `{Client}` - the client to copy
- `pem` `{bytes}` - a PEM certificate to trust

Returns `{Client}` - a new client pinned to the CA

rest.withHeader(c as Client, name as string, value as string)

Return a copy of the client with one default header set.

Parameters

- `c` `{Client}` - the client to copy
- `name` `{string}` - the header name
- `value` `{string}` - the header value

Returns `{Client}` - a new client with the header set

rest.withRedirects(c as Client, maxRedirects as int)

Return a copy of the client that follows up to `maxRedirects` 3xx redirects (0 = none; the 3xx is returned as-is). Inherits `http.send` 's redirect rules.

Parameters

- `c` `{Client}` - the client to copy
- `maxRedirects` `{int}` - the redirect hop limit

Returns `{Client}` - a new client that follows redirects

rest.withRetries(c as Client, maxRetries as int)

Return a copy of the client that retries a 429 / 5xx up to `maxRetries` times with exponential backoff (honouring a numeric `Retry-After`). Use `rest.withBackoff` for a non-default base backoff.

Parameters

- `c` `{Client}` - the client to copy
- `maxRetries` `{int}` - the retry limit

Returns `{Client}` - a new client that retries

rest.withTimeout(c as Client, timeoutMs as int)

Return a copy of the client with a per-request idle timeout (milliseconds; 0 = the 30s default). Bounds each read, so a hung server fails rather than blocking.

Parameters

- `c` `{Client}` - the client to copy
- `timeoutMs` `{int}` - the per-read idle timeout in milliseconds

Returns `{Client}` - a new client with the timeout set

Structs

rest.Client

A REST client: a base URL every path joins onto, default headers sent with every request (auth lives here), and the `http.Options` request policy (per-request timeout, body cap, redirect-following, retry / backoff, and TLS) applied to every call. Value-semantic; thread it per call.

Field	Type	Description
baseUrl	string	the base URL every path joins onto
headers	map of string to string	default headers sent with every request
options	http.Options	the request policy (timeout, cap, redirects, retries, tls); zero value = defaults, full TLS verification, no redirects / retries

rest.Response

A REST response: the status code, response headers, and the body text.

Field	Type	Description
status	int	the HTTP status code
headers	map of string to string	the response headers (lowercased keys)
body	string	the response body text

ringbuffer API reference

A fixed-capacity ring buffer of strings: a bounded FIFO that overwrites the oldest entry when full. `push` appends (dropping the oldest once capacity is exceeded); `pop` removes the oldest; `first` / `last` peek without removing. Elements are ordered oldest-to-newest. Useful for a sliding window of recent events, log lines, or samples.

Value-semantic: `push` / `pop` return a fresh buffer (chain them, `$rb = ringbuffer.push($rb, x)`). Since a value-semantic `pop` cannot return both the item and the new buffer, read the oldest with `first` before you `pop` it. Over `lists` ; runs on both binaries. Stores strings - serialize other values with `convert.toString` or `json` .

Import with `import "ringbuffer.j" as ringbuffer;` . See the `ringbuffer` guide for prose and examples.

Functions

ringbuffer.capacity(rb as RingBuffer)

The buffer's capacity.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{int}` - the capacity

ringbuffer.first(rb as RingBuffer)

The oldest item, without removing it.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{string}` - the oldest item

Throws

- `{Error}` - kind "ringbuffer" if the buffer is empty

ringbuffer.isEmpty(rb as RingBuffer)

Whether the buffer holds no entries.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{bool}` - true if empty

ringbuffer.isFull(rb as RingBuffer)

Whether the buffer is at capacity.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{bool}` - true if full

ringbuffer.last(rb as RingBuffer)

The newest item, without removing it.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{string}` - the newest item

Throws

- `{Error}` - kind "ringbuffer" if the buffer is empty

ringbuffer.new(capacity as int)

Create an empty ring buffer of the given capacity.

Parameters

- `capacity` `{int}` - the maximum number of entries (must be >= 1)

Returns `{RingBuffer}` - the empty buffer

Throws

- `{Error}` - kind "ringbuffer" if capacity is < 1

ringbuffer.pop(rb as RingBuffer)

Remove the oldest item. Returns a fresh buffer. Read the item with `first` before popping it.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{RingBuffer}` - the buffer without its oldest item

Throws

- `{Error}` - kind "ringbuffer" if the buffer is empty

ringbuffer.push(rb as RingBuffer, item as string)

Append an item, dropping the oldest if the buffer is already full. Returns a fresh buffer.

Parameters

- `rb` `{RingBuffer}` - the buffer
- `item` `{string}` - the item to append

Returns `{RingBuffer}` - the updated buffer

ringbuffer.size(rb as RingBuffer)

The number of entries currently held.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{int}` - the entry count

ringbuffer.toList(rb as RingBuffer)

A copy of the entries, oldest to newest.

Parameters

- `rb` `{RingBuffer}` - the buffer

Returns `{list of string}` - the entries

Structs

ringbuffer.RingBuffer

A ring buffer.

Field	Type	Description
items	list of string	the entries, oldest first
capacity	int	the maximum number of entries

s3 API reference

An S3-compatible object-storage client: get / put / delete objects and list a bucket, signing every request with **AWS Signature Version 4** . The endpoint is configurable, so one module serves AWS S3 and every S3-compatible store (MinIO, Cloudflare R2, Backblaze B2) - a selectable backend, not a module per vendor. Path-style addressing (`{endpoint}/{bucket}/{key}`). SigV4 is HMAC-SHA256 key-chaining, so this builds on `hash.hmac` + `hash.compute` + `encoding` (hex) + `time` (the request timestamp) + `http` . Needs the default `jennifer` binary (`net` via `http`).

Import with `import "s3.j" as s3;` . See the `s3` guide for prose and examples.

Functions

s3.abortMultipartUpload(client as Client, bucketName as string, key as string, uploadId as string)

Abort a multipart upload, discarding its uploaded parts (call this on failure so the parts do not linger and accrue storage cost).

Parameters

- `client` `{Client}` - the client
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key
- `uploadId` `{string}` - the id from `createMultipartUpload`

Returns `{http.Response}` - the response (204 on success)

s3.completeMultipartUpload(client as Client, bucketName as string, key as string, uploadId as string, etags as list of string)

Complete a multipart upload, assembling the uploaded parts into the final object. `etags` are the ETags from `uploadPart` in part-number order (part 1 first).

Parameters

- `client` `{Client}` - the client
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key
- `uploadId` `{string}` - the id from `createMultipartUpload`
- `etags` `{list of string}` - the part ETags, in part-number order

Returns `{http.Response}` - the response (200 with a CompleteMultipartUploadResult body)

s3.connect(endpoint as string, region as string, accessKey as string, secretKey as string)

Build a client for an S3 endpoint. The endpoint is any S3-compatible base URL (`scheme://host[:port]` , no trailing slash).

Parameters

- `endpoint` `{string}` - the base URL
- `region` `{string}` - the signing region
- `accessKey` `{string}` - the access key id
- `secretKey` `{string}` - the secret access key

Returns `{Client}` - a configured client (30 s timeout; set `.timeout` to change it)

s3.copy(client as Client, srcBucket as string, srcKey as string, dstBucket as string, dstKey as string)

Copy an object server-side (no download / re-upload): PUT the destination with an `x-amz-copy-source` of `/{srcBucket}/{srcKey}` (signed). Source and destination may share a bucket (a rename when paired with `delete`).

Parameters

- `client` `{Client}` - the client
- `srcBucket` `{string}` - the source bucket
- `srcKey` `{string}` - the source object key
- `dstBucket` `{string}` - the destination bucket
- `dstKey` `{string}` - the destination object key

Returns `{http.Response}` - the response (200 with a CopyObjectResult body on success)

s3.createMultipartUpload(client as Client, bucketName as string, key as string, contentType as string)

Begin a multipart upload (for objects beyond the ~5 GB single-PUT limit, or to stream parts). Returns the upload id to pass to `uploadPart` / `completeMultipartUpload` / `abortMultipartUpload`.

Parameters

- `client` `{Client}` - the client
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key
- `contentType` `{string}` - the object content type ("" = default)

Returns `{string}` - the upload id

Throws

- `{Error}` - kind "s3" when the response carries no UploadId

s3.delete(client as Client, bucketName as string, key as string)

DELETE an object.

Parameters

- `client` `{Client}` - the client
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key

Returns `{http.Response}` - the response (204 on success)

s3.get(client as Client, bucketName as string, key as string)

GET an object. The response body is the object's contents; a missing object comes back as a 404 `http.Response`, not an error.

Parameters

- `client` `{Client}` - the client
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key

Returns `{http.Response}` - the response (body = object contents on 200)

s3.getBytes(client as Client, bucketName as string, key as string)

GET an object as raw `bytes` (the byte-safe download). Use this for binary objects (images, archives) that a UTF-8 string body cannot hold; `get` stays the text convenience.

Parameters

- `client {Client}` - the client
- `bucketName {string}` - the bucket
- `key {string}` - the object key

Returns `{http.BytesResponse}` - the response (body = raw object bytes on 200)

s3.head(client as Client, bucketName as string, key as string)

HEAD an object: its metadata (status + headers) without the body. A present object returns 200 with `Content-Length` / `Content-Type` / `x-amz-meta-*` headers; a missing one returns 404.

Parameters

- `client {Client}` - the client
- `bucketName {string}` - the bucket
- `key {string}` - the object key

Returns `{http.Response}` - the response (headers only, empty body)

s3.isTruncated(xml as string)

Report whether a ListObjectsV2 XML body was truncated (more pages remain).

Parameters

- `xml {string}` - the body from a list call

Returns `{bool}` - true when another page is available

s3.listObjects(client as Client, bucketName as string)

List a bucket's objects (S3 ListObjectsV2). The response body is the S3 XML listing; pass it to `s3.objectKeys` to pull out the keys. S3 returns at most 1000 keys per page: check `s3.isTruncated` on the body and, if true, call `s3.listObjectsFrom` with `s3.nextContinuationToken` to fetch the next page (loop until not truncated).

Parameters

- `client {Client}` - the client
- `bucketName {string}` - the bucket

Returns `{http.Response}` - the response (body = ListBucketResult XML on 200)

s3.listObjectsFrom(client as Client, bucketName as string, token as string)

Fetch one further page of a ListObjectsV2 listing, starting after `token` (the value from `s3.nextContinuationToken` on the previous page's body).

Parameters

- `client {Client}` - the client
- `bucketName {string}` - the bucket
- `token {string}` - the continuation token from the previous page

Returns `{http.Response}` - the response (body = ListBucketResult XML on 200)

s3.nextContinuationToken(xml as string)

Extract the continuation token to fetch the next page, or "" when none.

Parameters

- `xml` `{string}` - the body from a list call

Returns `{string}` - the next continuation token, or "" if the listing is complete

s3.objectKeys(xml as string)

Extract the object keys from a ListObjectsV2 XML body (the <Key> elements).

Parameters

- `xml` `{string}` - the body from `s3.listObjects`

Returns `{list of string}` - the object keys, in listing order

s3.presign(client as Client, method as string, bucketName as string, key as string, expiresSeconds as int)

Build a presigned URL that grants time-limited access to an object without exposing the secret key, using SigV4 query-signing (`X-Amz-*` query parameters). The URL works from any HTTP client until it expires. Pure (no request is sent), so it runs on both binaries.

Parameters

- `client` `{Client}` - the client
- `method` `{string}` - the HTTP method the URL authorizes (e.g. "GET", "PUT")
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key
- `expiresSeconds` `{int}` - the validity window in seconds (max 604800 = 7 days)

Returns `{string}` - the presigned URL

s3.put(client as Client, bucketName as string, key as string, body as string)

PUT (upload / overwrite) an object with the given body.

Parameters

- `client` `{Client}` - the client
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key
- `body` `{string}` - the object contents

Returns `{http.Response}` - the response (200 on success)

s3.putBytes(client as Client, bucketName as string, key as string, data as bytes)

PUT (upload / overwrite) an object from a raw `bytes` body, written byte-for-byte so a binary payload round-trips intact.

Parameters

- `client` `{Client}` - the client
- `bucketName` `{string}` - the bucket
- `key` `{string}` - the object key
- `data` `{bytes}` - the object contents

Returns `{http.Response}` - the response (200 on success)

s3.putBytesWith(client as Client, bucketName as string, key as string, data as bytes, contentType as string, metadata as map of string to string)

PUT an object (raw `bytes` body) with an explicit content type and user metadata - the binary counterpart to `putWith` .

Parameters

- `client {Client}` - the client
- `bucketName {string}` - the bucket
- `key {string}` - the object key
- `data {bytes}` - the object contents
- `contentType {string}` - the object content type ("" = default)
- `metadata {map of string to string}` - user metadata (stored as x-amz-meta-<key>)

Returns `{http.Response}` - the response (200 on success)

s3.putWith(client as Client, bucketName as string, key as string, body as string, contentType as string, metadata as map of string to string)

PUT an object (string body) with an explicit content type and user metadata. The Content-Type and each `x-amz-meta-<key>` header are part of the signature, so they reach S3 intact. Pass `""` for the default content type and `{}` for no metadata.

Parameters

- `client {Client}` - the client
- `bucketName {string}` - the bucket
- `key {string}` - the object key
- `body {string}` - the object contents
- `contentType {string}` - the object content type (e.g. "text/html"; "" = default)
- `metadata {map of string to string}` - user metadata (stored as x-amz-meta-<key>)

Returns `{http.Response}` - the response (200 on success)

s3.uploadPart(client as Client, bucketName as string, key as string, uploadId as string, partNumber as int, data as bytes)

Upload one part of a multipart upload (each part >= 5 MiB except the last). Returns the part's ETag, which `completeMultipartUpload` needs (collect them in part-number order).

Parameters

- `client {Client}` - the client
- `bucketName {string}` - the bucket
- `key {string}` - the object key
- `uploadId {string}` - the id from `createMultipartUpload`
- `partNumber {int}` - the 1-based part number
- `data {bytes}` - the part contents

Returns `{string}` - the part's ETag

Structs

s3.Client

A configured S3 client: the endpoint (scheme + host, no trailing slash), the signing region, and the access-key pair. Value-semantic; build with `connect` .

Field	Type	Description
endpoint	string	e.g. "https://s3.us-east-1.amazonaws.com" or "http://localhost:9000"
region	string	the signing region, e.g. "us-east-1"

accessKey	string	the access key id
secretKey	string	the secret access key
timeout	int	per-request idle timeout in milliseconds (0 disables it); connect defaults it to 30000

sasl API reference

The SASL authentication mechanisms shared by the mail clients (`smtp` / `pop` / `imap`). These format the client tokens; the protocol clients run the mechanism-specific wire dialogue (SMTP `AUTH` , IMAP `AUTHENTICATE` , POP3 `AUTH`) around them. No networking; TinyGo-clean, so it runs on both binaries.

- **Simple encoders** (`plain` , `loginUser` / `loginPass` , `bearer`) - one base64 token per step. `bearer` builds the SASL XOAUTH2 response from an OAuth2 bearer token (named `bearer` , not `xoauth2` , because a Jennifer method name is letters-only; "XOAUTH2" is the string the client sends).
- **Challenge-response** (`cram` , and the `SCRAM` family) - keyed by the `hash` / `crypto` libraries. `cram` (CRAM-MD5, RFC 2195) answers a server challenge in one step. SCRAM is a multi-step exchange over a `Scram` handle: `scramStart` -> `scramClientFirst` (send), receive server-first -> `scramClientFinal` + `scramFinalToken` (send), receive server-final -> `scramVerify` (checks the server signature, so a MITM without the password is caught). Mechanism "sha1" is SCRAM-SHA-1, "sha256" is SCRAM-SHA-256.

Import with `import "sasl.j" as sasl;` . See the `sasl` guide for prose and examples.

Functions

sasl.bearer(user as string, token as string)

Build the SASL XOAUTH2 response for an OAuth2 bearer token: base64("user=" user 0x01 "auth=Bearer " token 0x01 0x01). This is how Google and Microsoft 365 authenticate mail (both have retired password auth).

Parameters

- `user` `{string}` - the account username
- `token` `{string}` - the OAuth2 bearer access token

Returns `{string}` - the base64-encoded XOAUTH2 response

sasl.cram(user as string, pass as string, challenge as string)

Build the CRAM-MD5 response (RFC 2195): base64 of "user <hex>", where <hex> is the lowercase HMAC-MD5 of the server's (base64-encoded) challenge keyed by the password. A one-step challenge-response mechanism. Named `cram` (not `cramMd5`) because a Jennifer method name is letters-only; the wire mechanism is "CRAM-MD5".

Parameters

- `user` `{string}` - the login username
- `pass` `{string}` - the login password
- `challenge` `{string}` - the server's base64-encoded challenge

Returns `{string}` - the base64-encoded CRAM-MD5 response

sasl.loginPass(pass as string)

Build the password step of SASL LOGIN (a server sends the "Password:" prompt).

Parameters

- `pass` `{string}` - the login password

Returns `{string}` - the base64-encoded password

sasl.loginUser(user as string)

Build the username step of SASL LOGIN (a server sends the "Username:" prompt).

Parameters

- `user` `{string}` - the login username

Returns `{string}` - the base64-encoded username

sasl.negotiate(advertised as list of string)

Choose the strongest password mechanism this module implements from the list a server advertises, returned as the `auth` token a mail client uses: "scram-sha-256" > "scram-sha-1" > "cram", or "" when the server offers none (the caller then falls back to its protocol default - PLAIN / LOGIN / USER-PASS). XOAUTH2 is never auto-selected: it needs a bearer token, not a password.

Parameters

- `advertised {list of string}` - the mechanism names the server advertises (any case)

Returns `{string}` - the chosen auth token, or "" for the protocol default

sasl.plain(user as string, pass as string)

Build the SASL PLAIN response: base64 of "\0user\0pass".

Parameters

- `user {string}` - the login username
- `pass {string}` - the login password

Returns `{string}` - the base64-encoded PLAIN token

sasl.scramClientFinal(s as Scram, serverFirst as string, password as string)

Process the server-first message and compute the client-final message. Derives the salted password (PBKDF2 in the mechanism's hash), the client proof, and the expected server signature; the returned state carries both. Throws `Error{kind: "sasl"}` if the server's nonce does not extend the client nonce.

Parameters

- `s {Scram}` - the exchange state
- `serverFirst {string}` - the base64 server-first message
- `password {string}` - the login password

Returns `{Scram}` - the updated state; pass it to scramFinalToken and scramVerify

sasl.scramClientFirst(s as Scram)

The base64 client-first message to send to the server (SASL initial response).

Parameters

- `s {Scram}` - the exchange state from scramStart

Returns `{string}` - the base64-encoded client-first message

sasl.scramFinalToken(s as Scram)

The base64 client-final message to send (after scramClientFinal).

Parameters

- `s {Scram}` - the state returned by scramClientFinal

Returns `{string}` - the base64-encoded client-final message

sasl.scramStart(user as string, algo as string)

Begin a SCRAM exchange: generate a crypto-grade client nonce and build the initial state. `algo` selects the mechanism ("sha1" or "sha256").

Parameters

- `user` `{string}` - the login username
- `algo` `{string}` - "sha1" (SCRAM-SHA-1) or "sha256" (SCRAM-SHA-256)

Returns `{Scram}` - the exchange state; pass it to scramClientFirst

sasl.scramVerify(s as Scram, serverFinal as string)

Verify the server-final message: the server's signature must match the one derived in scramClientFinal (constant-time), proving the server also knows the password. Reject the session on false.

Parameters

- `s` `{Scram}` - the state returned by scramClientFinal
- `serverFinal` `{string}` - the base64 server-final message

Returns `{bool}` - true if the server signature verifies

Structs

sasl.Scram

The state of an in-progress SCRAM exchange, threaded through the four calls. Value-semantic: each step returns an updated copy.

Field	Type	Description
algo	string	the hash mechanism: "sha1" (SCRAM-SHA-1) or "sha256" (SCRAM-SHA-256)
clientNonce	string	the client-generated nonce
clientFirstBare	string	the client-first message without the gs2 header
clientFinal	string	the client-final message (set by scramClientFinal)
serverSig	string	the base64 ServerSignature to expect (set by scramClientFinal)

screen API reference

Terminal user interfaces: an explicit `screen` , not a GUI framework. Two layers. The output-only layer is pure strings over ANSI control sequences - a cell `Buffer` you draw into (`text` / `box` / `fill`) and paint with `render` , plus `diff` for a flicker-free update loop (dashboards, progress, self-updating tables); it needs no host capability and runs on both binaries. The interactive layer decodes a raw byte stream into `Key` events (`decodeKey` , pure and testable) and drives an event loop over the `term` library's raw mode (`begin` / `nextKey` / `end`) for menus, forms, and key navigation; that layer needs `term` , so it works on the default `jennifer` (a friendly error on `jennifer-tiny` , which stubs `term`).

Coordinates are 0-based with the origin at the top-left `(0, 0)` ; `x` is the column, `y` the row. Drawing that runs past an edge is clipped, not an error.

Import with `import "screen.j" as screen;` . See the `screen` guide for prose and examples.

Functions

screen.begin()

Enter full-screen interactive mode: put the terminal in raw mode and switch to the alternate screen with the cursor hidden and the screen cleared. Pair with `end` , ideally via `defer screen.end($state);` . Requires the `term` library (default binary).

Returns `{term.State}` - the raw-mode handle to pass to `end`

screen.box(buf as Buffer, x as int, y as int, w as int, h as int)

A copy of `buf` with a single-line box border drawn at `(x, y)` of outer size `w` by `h` , using Unicode box-drawing glyphs. The interior is untouched.

Parameters

- `buf` `{Buffer}` - the source buffer
- `x` `{int}` - the left column (0-based)
- `y` `{int}` - the top row (0-based)
- `w` `{int}` - the outer width in columns (>= 2)
- `h` `{int}` - the outer height in rows (>= 2)

Returns `{Buffer}` - the updated buffer

screen.clear()

The sequence that clears the whole screen and homes the cursor.

Returns `{string}` - the clear-screen escape sequence

screen.clearBuffer(buf as Buffer)

A copy of `buf` with every cell reset to a space.

Parameters

- `buf` `{Buffer}` - the source buffer

Returns `{Buffer}` - the cleared buffer

screen.clearLine()

The sequence that clears the current line.

Returns `{string}` - the clear-line escape sequence

screen.decodeKey(seq as list of int)

Decode one raw key byte sequence into a `Key` . Pure and total: `seq` is the bytes of a single key press (`[65]` for `A` , `[27, 91, 65]` for Up, `[27, 91, 51, 126]` for Delete). Unrecognized input decodes to `"unknown"` ; an empty sequence to `"eof"` . This is what `nextKey` calls after reading the bytes, exposed separately so key handling is testable without a terminal.

Parameters

- `seq {list of int}` - the raw byte values of one key event

Returns `{Key}` - the decoded key

screen.diff(old as Buffer, new as Buffer)

The minimal escape string that turns the terminal showing `old` into `new` - only the changed cells are repositioned and rewritten, in row runs. When the two buffers differ in size it falls back to a full `render(new)` . This is the flicker-free update path: keep the previous buffer, `diff` against the next.

Parameters

- `old {Buffer}` - the buffer currently on screen
- `new {Buffer}` - the buffer to display

Returns `{string}` - the minimal-update escape string

screen.down(n as int)

The sequence that moves the cursor down `n` rows.

Parameters

- `n {int}` - the number of rows

Returns `{string}` - the cursor-down escape sequence

screen.end(state as term.State)

Leave interactive mode: show the cursor, leave the alternate screen, and restore the terminal from the handle `begin` returned.

Parameters

- `state {term.State}` - the handle returned by `begin`

Returns `{null}` - nothing

screen.enterAlt()

The sequence that switches to the alternate screen buffer (so the app's output does not scroll the user's shell history).

Returns `{string}` - the enter-alternate-screen escape sequence

screen.exitAlt()

The sequence that leaves the alternate screen buffer, restoring the prior terminal contents.

Returns `{string}` - the leave-alternate-screen escape sequence

screen.fill(buf as Buffer, x as int, y as int, w as int, h as int, ch as string)

A copy of `buf` with the rectangle at `(x, y)` of size `w` by `h` filled with the glyph `ch` . Clipped to the buffer.

Parameters

- `buf {Buffer}` - the source buffer
- `x {int}` - the left column (0-based)
- `y {int}` - the top row (0-based)

- `w` `{int}` - the width in columns
- `h` `{int}` - the height in rows
- `ch` `{string}` - the fill glyph (one column)

Returns `{Buffer}` - the updated buffer

screen.get(buf as Buffer, x as int, y as int)

The contents of the cell at `(x, y)`, or an empty string if out of range.

Parameters

- `buf` `{Buffer}` - the buffer
- `x` `{int}` - the column (0-based)
- `y` `{int}` - the row (0-based)

Returns `{string}` - the cell contents

screen.hideCursor()

The sequence that hides the text cursor.

Returns `{string}` - the hide-cursor escape sequence

screen.hline(buf as Buffer, x as int, y as int, n as int, ch as string)

A copy of `buf` with a horizontal line of n cells of `ch` from `(x, y)`.

Parameters

- `buf` `{Buffer}` - the source buffer
- `x` `{int}` - the starting column (0-based)
- `y` `{int}` - the row (0-based)
- `n` `{int}` - the length in columns
- `ch` `{string}` - the line glyph (one column)

Returns `{Buffer}` - the updated buffer

screen.home()

The sequence that homes the cursor to the top-left.

Returns `{string}` - the cursor-home escape sequence

screen.left(n as int)

The sequence that moves the cursor left n columns.

Parameters

- `n` `{int}` - the number of columns

Returns `{string}` - the cursor-back escape sequence

screen.moveTo(x as int, y as int)

The sequence that moves the cursor to `(x, y)` (0-based; origin top-left).

Parameters

- `x` `{int}` - the target column (0-based)
- `y` `{int}` - the target row (0-based)

Returns `{string}` - the cursor-position escape sequence

screen.newScreen(rows as int, cols as int)

A new blank `Buffer` of `rows` by `cols` cells, every cell a space.

Parameters

- `rows` `{int}` - the number of rows (must be positive)
- `cols` `{int}` - the number of columns (must be positive)

Returns `{Buffer}` - the blank buffer

Throws

- `{Error}` - when `rows` or `cols` is not positive

screen.nextKey()

Read and decode the next key from the terminal (blocking). Reads raw bytes through the `term` library, assembling a full escape sequence before decoding, and returns the `Key` . At end of input the name is `"eof"` . Requires raw mode (see `begin`) and the `term` library (default binary).

A lone Escape press is only reported once the next byte arrives, because `term.readByte` has no timeout to distinguish it from the start of an escape sequence; prefer a named key (or Ctrl-C) to quit an event loop.

Returns `{Key}` - the next key event

screen.render(buf as Buffer)

The escape string that paints the whole `buf` to the terminal (positions the cursor at the start of each row and writes its cells). Print it with `io.printf` ; usually preceded once by `clear` .

Parameters

- `buf` `{Buffer}` - the buffer to paint

Returns `{string}` - the full-repaint escape string

screen.right(n as int)

The sequence that moves the cursor right n columns.

Parameters

- `n` `{int}` - the number of columns

Returns `{string}` - the cursor-forward escape sequence

screen.set(buf as Buffer, x as int, y as int, cell as string)

A copy of `buf` with the single cell at `(x, y)` set to `cell` (one column). Out-of-range coordinates are clipped (returns `buf` unchanged).

Parameters

- `buf` `{Buffer}` - the source buffer
- `x` `{int}` - the column (0-based)
- `y` `{int}` - the row (0-based)
- `cell` `{string}` - the cell contents (one visible column)

Returns `{Buffer}` - the updated buffer

screen.showCursor()

The sequence that shows the text cursor.

Returns `{string}` - the show-cursor escape sequence

screen.size()

The terminal size as a `term.Size` (`{rows, cols}`), a convenience passthrough to `term.size` so an app stays in one namespace. Requires the `term` library.

Returns `{term.Size}` - the terminal dimensions

screen.text(buf as Buffer, x as int, y as int, s as string)

A copy of `buf` with `s` written left-to-right starting at `(x, y)`, one rune per cell, clipped at the row's right edge (no wrapping).

Parameters

- `buf` `{Buffer}` - the source buffer
- `x` `{int}` - the starting column (0-based)
- `y` `{int}` - the row (0-based)
- `s` `{string}` - the text to write

Returns `{Buffer}` - the updated buffer

screen.textColor(buf as Buffer, x as int, y as int, s as string, color as string)

A copy of `buf` with `s` written from `(x, y)` in the named foreground colour, one rune per cell, clipped at the row's right edge.

Parameters

- `buf` `{Buffer}` - the source buffer
- `x` `{int}` - the starting column (0-based)
- `y` `{int}` - the row (0-based)
- `s` `{string}` - the text to write
- `color` `{string}` - the colour name (e.g. `"red"`, `"cyan"`)

Returns `{Buffer}` - the updated buffer

Throws

- `{Error}` - when `color` is not a known colour

screen.up(n as int)

The sequence that moves the cursor up n rows.

Parameters

- `n` `{int}` - the number of rows

Returns `{string}` - the cursor-up escape sequence

screen.vline(buf as Buffer, x as int, y as int, n as int, ch as string)

A copy of `buf` with a vertical line of n cells of `ch` from `(x, y)`.

Parameters

- `buf` `{Buffer}` - the source buffer
- `x` `{int}` - the column (0-based)
- `y` `{int}` - the starting row (0-based)
- `n` `{int}` - the length in rows
- `ch` `{string}` - the line glyph (one column)

Returns `{Buffer}` - the updated buffer

Structs

screen.Buffer

A rectangular grid of character cells, drawn into then painted to the terminal. `cells` is row-major (`y * cols + x`); each cell is a one-column string, optionally wrapped in an SGR colour sequence. Value-semantic like any struct: the drawing functions return a fresh `Buffer` .

Field	Type	Description
rows	int	the number of rows (height)
cols	int	the number of columns (width)
cells	list of string	row-major cell contents, length rows * cols

screen.Key

A decoded key event from `nextKey` / `decodeKey` . `name` is a symbolic key name: a printable key is `"char"` (with the character in `char`); the rest are named - `"up"` / `"down"` / `"left"` / `"right"` , `"enter"` / `"tab"` / `"escape"` / `"backspace"` / `"delete"` / `"insert"` , `"home"` / `"end"` / `"pageup"` / `"pagedown"` , `"f1"` .. `"f12"` , `"ctrl-a"` .. `"ctrl-z"` , `"alt-<c>"` , `"eof"` at end of input, and `"unknown"` .

Field	Type	Description
name	string	the symbolic key name
char	string	the character for a "char" / "alt-" key, else empty

semver API reference

Strict Semantic Versioning 2.0.0 (https://semver.org): parse, compare, increment, and range-match version numbers - the full surface a package registry or dependency resolver needs. A pure-Jennifer reference module (no Go, no system library): parsing uses the canonical SemVer regex (via the `regex` library); precedence comparison, sorting, and range matching are hand-written Jennifer.

Ranges follow the npm / Composer grammar: caret `^1.2.0` , tilde `~1.2` , comparators `>=1.0.0 <2.0.0` (space or comma = AND), OR sets `^1 || ^2` , hyphen ranges `1.2.3 - 2.3.4` , x-ranges `1.x` / `1.2.*` , and `*` for any. A prerelease version satisfies a range only when a comparator in the same clause pins a prerelease at the same `major.minor.patch` (the npm rule).

Import with `import "semver.j" as semver;` . See the `semver` guide for prose and examples.

Functions

semver.clean(s as string)

Normalise a version string: trim whitespace and a leading = / v , then return the canonical form if it is a valid full version, else "". Strict (unlike `coerce`): `clean("v1.2.3")` -> "1.2.3", but `clean("1.2")` -> "".

Parameters

- `s {string}` - the version text

Returns `{string}` - the canonical version, or "" if not valid

semver.coerce(s as string)

Extract a version from a loose string - a git tag, a partial, or text with a version-like run - and return canonical `major.minor.patch` (missing parts are 0). Handles a leading v and surrounding noise. Returns "" when no numeric core is found. `coerce("v1.2.3")` -> "1.2.3", `coerce("1.2")` -> "1.2.0".

Parameters

- `s {string}` - the loose text

Returns `{string}` - a canonical version, or "" if none was found

semver.compare(a as Version, b as Version)

Compare two versions by SemVer precedence: numeric core, then a prerelease ranks below its release, then prerelease fields. Build metadata is ignored.

Parameters

- `a {Version}` - the left version
- `b {Version}` - the right version

Returns `{int}` - -1 if a < b, 0 if equal, 1 if a > b

semver.diff(a as Version, b as Version)

The kind of change from a to b: "major", "minor", "patch", "prerelease" (only the prerelease tag differs), or "" when the two are equal. The highest-order difference wins.

Parameters

- `a {Version}` - the left version
- `b {Version}` - the right version

Returns `{string}` - the release-type difference

semver.eq(a as Version, b as Version)

Report whether a and b have equal precedence (build metadata ignored).

Parameters

- a `{Version}` - the left version
- b `{Version}` - the right version

Returns `{bool}` - true when a and b compare equal

semver.gt(a as Version, b as Version)

Report whether a orders after b.

Parameters

- a `{Version}` - the left version
- b `{Version}` - the right version

Returns `{bool}` - true when a > b by SemVer precedence

semver.gte(a as Version, b as Version)

Report whether a orders at or after b.

Parameters

- a `{Version}` - the left version
- b `{Version}` - the right version

Returns `{bool}` - true when a >= b

semver.gtr(version as string, range as string)

Report whether a version is greater than every version a range allows (beyond its upper extreme). False for an unbounded range or a version in an interior gap.

Parameters

- `version` `{string}` - the version to test
- `range` `{string}` - the range expression

Returns `{bool}` - true when version is above the whole range

semver.incMajor(v as Version)

Bump the major version, resetting minor / patch and clearing the tags.

Parameters

- v `{Version}` - the starting version

Returns `{Version}` - a new version with major + 1 and minor = patch = 0

semver.incMinor(v as Version)

Bump the minor version, resetting patch and clearing the tags.

Parameters

- v `{Version}` - the starting version

Returns `{Version}` - a new version with minor + 1 and patch = 0

semver.incPatch(v as Version)

Bump the patch version, clearing the tags.

Parameters

- v {Version} - the starting version

Returns {Version} - a new version with patch + 1

semver.intersects(rangeA as string, rangeB as string)

Report whether two ranges share at least one satisfying version, prereleases included (a prerelease overlap needs both ranges to pin the same major.minor.patch). ^1.2.0 intersects >=1.5.0 (true) but not ^2.0.0 . Invalid ranges never intersect.

Parameters

- rangeA {string} - the first range
- rangeB {string} - the second range

Returns {bool} - true when the ranges overlap

semver.isPrerelease(v as Version)

Report whether the version carries a prerelease tag.

Parameters

- v {Version} - the version to classify

Returns {bool} - true when a prerelease tag is present

semver.isStable(v as Version)

Report whether the version is stable: a released (major >= 1) version with no prerelease tag. A 0.y.z version is unstable by SemVer convention.

Parameters

- v {Version} - the version to classify

Returns {bool} - true when major >= 1 and there is no prerelease tag

semver.isValid(s as string)

Report whether a string is a valid SemVer 2.0.0 version.

Parameters

- s {string} - the candidate version text

Returns {bool} - true when s parses as a valid version

semver.lt(a as Version, b as Version)

Report whether a orders before b.

Parameters

- a {Version} - the left version
- b {Version} - the right version

Returns {bool} - true when a < b by SemVer precedence

semver.lte(a as Version, b as Version)

Report whether a orders at or before b.

Parameters

- a {Version} - the left version
- b {Version} - the right version

Returns `{bool}` - true when a <= b

semver.ltr(version as string, range as string)

Report whether a version is less than every version a range allows (below its lower extreme).

Parameters

- `version` `{string}` - the version to test
- `range` `{string}` - the range expression

Returns `{bool}` - true when version is below the whole range

semver.maxSatisfying(versions as list of string, range as string)

Pick the highest version from a list that satisfies the range. Versions that are not valid SemVer are skipped. Returns "" when none match.

Parameters

- `versions` `{list of string}` - the candidate versions
- `range` `{string}` - the range expression

Returns `{string}` - the highest satisfying version, or "" if none match

semver.minSatisfying(versions as list of string, range as string)

Pick the lowest version from a list that satisfies the range. Versions that are not valid SemVer are skipped. Returns "" when none match.

Parameters

- `versions` `{list of string}` - the candidate versions
- `range` `{string}` - the range expression

Returns `{string}` - the lowest satisfying version, or "" if none match

semver.minVersion(range as string)

The lowest version that could satisfy a range (its floor), or "" when the range is empty or invalid. Prerelease-precise: `minVersion(">=1.2.3-rc.1")` is "1.2.3-rc.1", `minVersion("^1.2.0")` is "1.2.0", `minVersion(">1.2.3")` is "1.2.4".

Parameters

- `range` `{string}` - the range expression

Returns `{string}` - the lowest satisfying version, or ""

semver.neq(a as Version, b as Version)

Report whether a and b differ in precedence.

Parameters

- a `{Version}` - the left version
- b `{Version}` - the right version

Returns `{bool}` - true when a and b are not equal

semver.outside(version as string, range as string)

Report whether a version is beyond a range's extremes (above it or below it). A version in an interior gap of a multi-clause range is not "outside".

Parameters

- `version` `{string}` - the version to test
- `range` `{string}` - the range expression

Returns `{bool}` - true when version is above or below the whole range

semver.parse(s as string)

Parse a version string into a Version.

Parameters

- `s` `{string}` - the version text (e.g. "1.2.3-rc.1+build.5")

Returns `{Version}` - the parsed version

Throws

- `{Error}` - when s is not a valid SemVer 2.0.0 string

semver.rsort(vs as list of Version)

Return a new list ordered descending by SemVer precedence (highest first).

Parameters

- `vs` `{list of Version}` - the versions to order

Returns `{list of Version}` - a new list sorted descending

semver.satisfies(version as string, range as string)

Report whether a concrete version satisfies a range. The range grammar is the npm / Composer set: caret `^1.2.0` , tilde `~1.2` , comparators `>=1.0.0 <2.0.0` (space or comma = AND), OR sets `^1 || ^2` , hyphen ranges `1.2.3 - 2.3.4` , x-ranges `1.x` / `1.2.*` , and `*` / `""` / `"any"` for any release. A prerelease version matches only when a comparator in the same clause pins a prerelease at the same major.minor.patch. An invalid version never satisfies anything.

Parameters

- `version` `{string}` - the concrete version to test (e.g. "1.4.0")
- `range` `{string}` - the range expression

Returns `{bool}` - true when version satisfies range

semver.simplifyRange(versions as list of string, range as string)

Simplify a range against a known list of versions: return the shortest range that matches exactly the same subset of `versions` . Runs of consecutive matching versions collapse to `>=lo <=hi` clauses joined by `||` ; the original range is kept when it is already at least as short. "*" when every listed version matches, `<0.0.0-0` (matches nothing) when none do.

Parameters

- `versions` `{list of string}` - the known versions
- `range` `{string}` - the range to simplify

Returns `{string}` - the simplified range

semver.sort(vs as list of Version)

Return a new list ordered ascending by SemVer precedence. lists.sort is scalar-only, so this is a merge sort over compare() - O(n log n), where an insertion sort was O(n^2).

Parameters

- `vs` `{list of Version}` - the versions to order

Returns `{list of Version}` - a new list sorted ascending

semver.subset(inner as string, outer as string)

Report whether every version allowed by `inner` is also allowed by `outer` (inner is the tighter / implied constraint), prereleases included. `subset("^1.5.0", "^1.0.0")` is true; `subset("^1.0.0", "^1.5.0")` is false.

Parameters

- `inner` `{string}` - the candidate subset range
- `outer` `{string}` - the superset range

Returns `{bool}` - true when inner is a subset of outer

semver.toString(v as Version)

Render a Version back to its canonical string form.

Parameters

- `v` `{Version}` - the version to format

Returns `{string}` - the "major.minor.patch[-prerelease][+build]" text

semver.validRange(range as string)

Report whether a range expression is well-formed (parseable). Does not evaluate it against any version.

Parameters

- `range` `{string}` - the range expression

Returns `{bool}` - true when the range is valid

Structs

semver.Version

A parsed SemVer version: numeric core plus optional prerelease / build tags.

Field	Type	Description
major	int	the major version
minor	int	the minor version
patch	int	the patch version
prerelease	string	the prerelease tag, or "" if none
build	string	the build metadata, or "" if none

session API reference

Server-side sessions over a selectable key/value backend. A session is a `json.Value` held under a `sess:ID` key with a sliding TTL, so it expires on its own when idle. The store is a `kvstore.Store` - back it with `memcache` or `redis` (distributed, shared across processes) or the in-process `kv` library (single process, no server), all behind one API. Session values are a `json.Value` , so a session holds structured data (nested objects, arrays, numbers), not just flat strings. The JSON is stored base64-wrapped, so any value is binary-safe. Session IDs are UUID v4 from the `crypto` library's crypto-grade random source (122 unguessable bits), safe to hand a client as the session's bearer token. Sessions are volatile (a cache of soft state, not a store of record - memcache / kv evict, so treat expiry as expected).

Import with `import "session.j" as session;` . See the `session` guide for prose and examples.

Functions

session.create(store as kvstore.Store, ttl as int)

Mint a new session ID and store an empty session with a ttl-second expiry.

Parameters

- `store` `{kvstore.Store}` - the backend store
- `ttl` `{int}` - the session lifetime in seconds

Returns `{string}` - the new session ID

session.destroy(store as kvstore.Store, id as string)

Remove the session.

Parameters

- `store` `{kvstore.Store}` - the backend store
- `id` `{string}` - the session ID

Returns `{bool}` - true when the session existed

session.load(store as kvstore.Store, id as string)

Return the session's data as a `json.Value` , or an empty object when the session is absent or expired.

Parameters

- `store` `{kvstore.Store}` - the backend store
- `id` `{string}` - the session ID

Returns `{json.Value}` - the session data (an empty object if absent or expired)

session.save(store as kvstore.Store, id as string, data as json.Value, ttl as int)

Write the session's data and re-arm its expiry to ttl seconds.

Parameters

- `store` `{kvstore.Store}` - the backend store
- `id` `{string}` - the session ID
- `data` `{json.Value}` - the session data to store
- `ttl` `{int}` - the session lifetime in seconds

session.touch(store as kvstore.Store, id as string, ttl as int)

Re-arm the session's expiry without rewriting its data.

Parameters

- `store` `{kvstore.Store}` - the backend store
- `id` `{string}` - the session ID
- `ttl` `{int}` - the new lifetime in seconds

Returns `{bool}` - true when the session still existed

slack API reference

Post messages to a Slack channel through an Incoming Webhook, on top of the `http` client - a sibling of `gotify` / `discord` . `send(webhookUrl, text)` posts a plain message; the `message` builder assembles a richer payload from Slack Block Kit blocks (`section` / `header` / `divider`) and `sendMessage` posts it. Needs the default `jennifer` binary (uses `net` via `http`). The webhook URL is a secret belonging to the caller - read it from the environment or a config file; never commit it.

Import with `import "slack.j" as slack;` . See the `slack` guide for prose and examples.

Functions

slack.actionsBlock(m as Message, buttons as list of string)

Add an actions block from one or more buttons (built with `button`). Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `buttons` `{list of string}` - pre-rendered button fragments (see `button`)

Returns `{Message}` - a message with the actions block appended

slack.button(text as string, url as string)

Build a single URL button element for an actions block. Returns the pre-rendered button JSON fragment; pass a list of these to `actionsBlock` .

Parameters

- `text` `{string}` - the button label (plain text)
- `url` `{string}` - the URL the button opens

Returns `{string}` - the button JSON fragment

slack.contextBlock(m as Message, text as string)

Add a context block (small, muted helper text in mrkdwn). Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `text` `{string}` - the context text (mrkdwn)

Returns `{Message}` - a message with the context block appended

slack.divider(m as Message)

Add a divider block (a horizontal rule). Returns a fresh message.

Parameters

- `m` `{Message}` - the message

Returns `{Message}` - a message with the divider appended

slack.fieldsSection(m as Message, fields as list of string)

Add a section block laid out as two-column fields (each entry rendered as mrkdwn). Slack packs the fields into a two-column grid. Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `fields` `{list of string}` - the field texts (mrkdwn), in order

Returns `{Message}` - a message with the fields section appended

slack.header(m as Message, heading as string)

Add a header block (plain text, bold and large). Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `heading` `{string}` - the header text (plain text)

Returns `{Message}` - a message with the header appended

slack.message()

Start an empty rich message (no fallback text, no blocks).

Returns `{Message}` - a fresh message

slack.render(m as Message)

Render a message to its JSON payload string.

Parameters

- `m` `{Message}` - the message

Returns `{string}` - the JSON payload Slack expects

slack.section(m as Message, markdown as string)

Add a section block (Slack mrkdwn). Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `markdown` `{string}` - the section body (mrkdwn)

Returns `{Message}` - a message with the section appended

slack.send(webhookUrl as string, text as string)

Post a plain-text message to a Slack Incoming Webhook.

Parameters

- `webhookUrl` `{string}` - the Incoming Webhook URL
- `text` `{string}` - the message text (Slack mrkdwn)

Returns `{http.Response}` - Slack answers 200 with body "ok" on success

slack.sendMessage(webhookUrl as string, m as Message)

Post a built rich message to a Slack Incoming Webhook.

Parameters

- `webhookUrl` `{string}` - the Incoming Webhook URL
- `m` `{Message}` - the message to send

Returns `{http.Response}` - Slack answers 200 with body "ok" on success

slack.text(m as Message, text as string)

Set the top-level fallback text (shown in notifications and by clients that do not render blocks). Returns a fresh message.

Parameters

- `m` `{Message}` - the message
- `text` `{string}` - the fallback text

Returns `{Message}` - a message with the fallback text set

Structs

slack.Message

A rich message under construction: a fallback text (shown in notifications) plus a list of pre-rendered Block Kit block JSON fragments.

Field	Type	Description
text	string	the top-level fallback / notification text ("" to omit)
blocks	list of string	pre-rendered block JSON fragments

smtp API reference

An SMTP send client: the line-oriented command/response dialogue of RFC 5321, over the `net` system library (plaintext, implicit TLS, or STARTTLS) with SASL AUTH: PLAIN, LOGIN, XOAUTH2, CRAM-MD5, and SCRAM-SHA-1 / SCRAM-SHA-256. The message body is any string, typically built by the `mime` module. Because it uses `net` , this module runs on the default `jennifer` binary only (`jennifer-tiny` stubs the network stack). `send` (the one-shot convenience) throws a catchable `Error` (kind "smtp") when the server rejects a command. To deliver a queue of messages over a single TLS + auth handshake, `open` a persistent `Session` , `sendOn` each message, and `close` . TLS certificate verification is the `net` default. An IDN host or envelope domain is IDNA-encoded to its `xn--` form (via the `idna` module); a non-ASCII address local part still throws (it needs SMTPUTF8).

Import with `import "smtp.j" as smtp;` . See the `smtp` guide for prose and examples.

Functions

smtp.close(session as Session)

Close a session: send QUIT (best-effort) and close the socket. Idempotent-safe to call once; do not use the session afterwards.

Parameters

- `session {Session}` - the session to close

smtp.open(opts as Options)

Open an authenticated SMTP session: connect per `opts.security` , read the greeting, EHLO, upgrade with STARTTLS when asked (refusing a server that did not advertise it - anti-downgrade), and authenticate when credentials are present. The returned `Session` is ready for one or more `sendOn` calls, so a queue of N messages pays this TLS + auth handshake once instead of N times.

Parameters

- `opts {Options}` - the connection and auth settings

Returns `{Session}` - an authenticated, ready session

Throws

- `{Error}` - kind "smtp" if the connection, greeting, STARTTLS, or auth fails

smtp.send(opts as Options, from as string, recipients as list of string, message as string)

Deliver `message` (a full RFC 5322 message, e.g. from mime.encode) to every recipient, with `from` as the envelope sender - the one-shot convenience: `open` a session, `sendOn` this one message, and `close` . To send several messages over one handshake, call `open` / `sendOn` / `close` yourself.

Parameters

- `opts {Options}` - the connection and auth settings
- `from {string}` - the envelope sender address
- `recipients {list of string}` - the envelope recipient addresses
- `message {string}` - the full RFC 5322 message body

Throws

- `{Error}` - kind "smtp" when the server rejects a command or an address is not ASCII-safe

smtp.sendOn(session as Session, from as string, recipients as list of string, message as string)

Deliver one `message` (a full RFC 5322 message, e.g. from mime.encode) over an open session: RSET, MAIL FROM / RCPT TO / DATA. The session is reusable afterwards (call `sendOn` again for the next message; `close` when done).

Parameters

- `session {Session}` - an open session (from `open`)
- `from {string}` - the envelope sender address
- `recipients {list of string}` - the envelope recipient addresses
- `message {string}` - the full RFC 5322 message body

Throws

- `{Error}` - kind "smtp" when the session is closed, the server rejects a command, or an address is not ASCII-safe

Structs

smtp.Options

Connection settings for an SMTP session.

Field	Type	Description
host	string	the mail server hostname (IDNA-encoded when non-ASCII)
port	int	the server port (e.g. 25, 465, 587)
security	transport.Security	transport.Security.None (plaintext), .Tls (implicit TLS on connect), or .Starttls (upgrade after EHLO)
clientName	string	the EHLO identity (defaults to "localhost" when empty)
user	string	the SASL username (empty means no auth)
pass	string	the SASL password, or the OAuth2 access token for xoauth2
auth	string	the SASL mechanism: "" (default - no auth when user is empty, else PLAIN), "auto" (negotiate the strongest mechanism the server's EHLO advertises, falling back to PLAIN), "plain", "login", "xoauth2", "cram" (CRAM-MD5), "scram-sha-1", or "scram-sha-256"
allowInsecureAuth	bool	force SASL auth over an unencrypted ("none") connection (default false - credentials are refused over plaintext)

smtp.Session

A live, authenticated SMTP connection: the handshake (greeting, EHLO, STARTTLS, and AUTH) has run once, so `sendOn` can deliver many messages over it. Value-semantic, but the `conn` handle is shared across copies (a `net.Conn`), so the socket survives being passed to `sendOn` . Build with `open` ; retire with `close` .

Field	Type	Description
conn	net.Conn	the underlying (possibly TLS-upgraded) socket, shared across copies
open	bool	whether the session is usable (false after close)

snmp API reference

An SNMP v1 / v2c client and agent (RFC 1157 / RFC 3416), over UDP with community-string authentication. The client queries an agent with GET / GETNEXT / SET / walk; the agent (server) answers those for a MIB you supply - a hardware simulator, or a way to expose an app's metrics over SNMP. The wire messages are ASN.1 BER, built and parsed with the `asn1` library; the transport is `net` UDP. No SNMPv3 / USM (that is the security model a later tier would add).

Import with `import "snmp.j" as snmp;` . See the `snmp` guide for prose and examples.

Functions

snmp.agent(community as string, version as int, bindings as list of Varbind)

Build an agent (server) that answers GET / GETNEXT / SET for a MIB. Build the bindings with `varbind` / `intVar` / `stringVar` / `oidVar` ; each is one OID the agent serves.

Parameters

- `community {string}` - the community string the agent accepts
- `version {int}` - snmp.VERSION1 or snmp.VERSION2C
- `bindings {list of Varbind}` - the MIB

Returns `{Agent}` - the configured agent

snmp.client(host as string, community as string)

Construct a client for an agent, using UDP port 161, SNMP v2c, a 2s timeout, and one retry.

Parameters

- `host {string}` - the agent host or IP
- `community {string}` - the community string

Returns `{Client}` - the configured client

snmp.clientWith(address as string, community as string, version as int, timeoutMs as int, retries as int)

Construct a client with full control over the endpoint and timing.

Parameters

- `address {string}` - the agent as "host:port"
- `community {string}` - the community string
- `version {int}` - snmp.VERSION1 or snmp.VERSION2C
- `timeoutMs {int}` - the per-attempt receive timeout, in milliseconds
- `retries {int}` - the number of extra attempts after the first

Returns `{Client}` - the configured client

snmp.get(c as Client, oids as list of string)

GET the values of one or more OIDs.

Parameters

- `c {Client}` - the agent client
- `oids {list of string}` - the OIDs to fetch

Returns `{list of Varbind}` - the returned bindings

Throws

- `{}` - snmp if the agent reports an error, or does not answer

snmp.getNext(c as Client, oids as list of string)

GETNEXT: fetch the binding lexically following each OID (the walk primitive).

Parameters

- `c` `{Client}` - the agent client
- `oids` `{list of string}` - the starting OIDs

Returns `{list of Varbind}` - the returned bindings

Throws

- `{}` - snmp if the agent reports an error, or does not answer

snmp.intVar(oid as string, n as int)

Build an integer-valued binding for snmp.set.

Parameters

- `oid` `{string}` - the object identifier, dotted
- `n` `{int}` - the integer value

Returns `{Varbind}` - the binding

snmp.oidVar(oid as string, target as string)

Build an OID-valued binding for snmp.set.

Parameters

- `oid` `{string}` - the object identifier, dotted
- `target` `{string}` - the OID value, dotted

Returns `{Varbind}` - the binding

snmp.serve(a as Agent, address as string)

Bind `address` ("host:port", e.g. ":161") and serve requests forever. Blocks; run it as the program's main loop, or in a `spawn` . A request with the wrong community, or a malformed datagram, is dropped silently (as a real agent does). To serve with a shutdown control, bind the socket yourself and use serveOn.

Parameters

- `a` `{Agent}` - the agent to serve
- `address` `{string}` - the UDP bind address

Throws

- `{}` - snmp if the address cannot be bound

snmp.serveOn(a as Agent, socket as net.UDPSocket, stop as channel of bool)

Serve requests on an already-bound UDP socket until a shutdown is signalled. Useful for embedding an agent beside a client in one program (bind first, then `spawn` this, so there is no bind race). To stop it, `channel.send($stop, true)` (a capacity >= 1 channel); the loop returns within SHUTDOWN_POLL_MS. `task.wait` the spawned handle afterwards for a graceful join. The caller owns the socket's lifetime; a request with the wrong community or a malformed datagram is dropped silently.

Parameters

- a `{Agent}` - the agent to serve
- `socket` `{net.UDPSocket}` - a bound UDP socket
- `stop` `{channel of bool}` - a value on this channel stops the loop

snmp.set(c as Client, varbinds as list of Varbind)

SET one or more bindings (build them with intVar / stringVar / oidVar).

Parameters

- `c` `{Client}` - the agent client
- `varbinds` `{list of Varbind}` - the bindings to write

Returns `{list of Varbind}` - the agent's echoed bindings

Throws

- `{}` - snmp if the agent reports an error, or does not answer

snmp.stringVar(oid as string, s as string)

Build an octet-string-valued binding for snmp.set.

Parameters

- `oid` `{string}` - the object identifier, dotted
- `s` `{string}` - the string value

Returns `{Varbind}` - the binding

snmp.varbind(oid as string, type as string, value as string, number as int)

Build a binding of any SNMP type (for a SET, or a MIB entry served by an agent). `type` is a value type name (see Varbind); `number` is used for the numeric types, `value` for the string / oid / ipAddress types.

Parameters

- `oid` `{string}` - the object identifier, dotted
- `type` `{string}` - the SNMP value type name
- `value` `{string}` - the string / oid / ipAddress rendering
- `number` `{int}` - the integer for a numeric type

Returns `{Varbind}` - the binding

snmp.walk(c as Client, rootOid as string)

Walk a subtree: repeated GETNEXT from rootOid until the returned OID leaves the subtree or the agent signals endOfMibView.

Parameters

- `c` `{Client}` - the agent client
- `rootOid` `{string}` - the subtree root OID, dotted

Returns `{list of Varbind}` - every binding under the subtree, in order

Throws

- `{}` - snmp if the agent reports an error, does not answer, or does not advance

Structs

snmp.Agent

An SNMP agent (server): a community, a protocol version, and a MIB - the set of bindings it answers for. Build one with snmp.agent and serve it with snmp.serve.

Field	Type	Description
community	string	the community string it accepts (others are dropped)
version	int	snmp.VERSION1 or snmp.VERSION2C
bindings	list of Varbind	the MIB, one binding per OID it serves

snmp.Client

A configured SNMP agent endpoint.

Field	Type	Description
address	string	the agent as "host:port"
community	string	the community string (the v1 / v2c credential)
version	int	the protocol version (snmp.VERSION1 or snmp.VERSION2C)
timeoutMs	int	the per-attempt receive timeout, in milliseconds
retries	int	the number of extra attempts after the first

snmp.Varbind

One variable binding: an OID and its value. On a returned binding, `type` is the SNMP value type ("integer", "octetString", "oid", "null", "counter32", "gauge32", "timeTicks", "ipAddress", "counter64", "opaque", or the exception "noSuchObject" / "noSuchInstance" / "endOfMibView"); `value` is a string rendering, and `number` holds the integer for a numeric type (0 otherwise).

Field	Type	Description
oid	string	the object identifier, dotted
type	string	the value's SNMP type name
value	string	the value rendered as a string
number	int	the integer value for a numeric type (0 otherwise)

Constants

Constant	Type	Description
snmp.VERSION1	int	SNMP version 1.
snmp.VERSION2C	int	SNMP version 2c.

sqlmigrate API reference

A version-tracked schema-migration runner over the `sql` library. It applies ordered, reversible DDL migrations and records what ran in a `schema_migrations` tracking table, so each migration runs **exactly once** and can be rolled back.

It is deliberately **decoupled from the orm mapper** : a migration's `up` / `down` are plain `list of string` DDL statements, so you build them with `orm` 's DDL helpers (`orm.createTable` / `orm.dropTable` / `orm.addColumn` / ...) **or** hand-written SQL - the runner never touches a `Schema` or a `Query` . That keeps it usable for any `sql` program, ORM or not.

The runner owns transaction control: each migration's `up` (or `down`) statements plus its tracking-table write run inside **one** `sql` transaction, so a failed statement rolls the whole step back and leaves `schema_migrations` consistent. It takes a raw `sql.Connection` (not an `orm.Session`) for that reason.

Migrations are ordered by `version` **lexically** , so zero-pad numeric versions (`"001"` , `"002"` , ...) or use a sortable timestamp (`"20260101120000"`). The tracking table (`version VARCHAR(255) PRIMARY KEY, description TEXT`) is accepted verbatim by both MySQL and PostgreSQL, so the runner needs no dialect.

Needs `sql` , so the default `jennifer` binary.

Import with `import "sqlmigrate.j" as sqlmigrate;` . See the `sqlmigrate` guide for prose and examples.

Functions

sqlmigrate.migrate(conn as sql.Connection, migrations as list of Migration)

Apply every pending migration in `version` order, each inside its own transaction, recording it in the `schema_migrations` tracking table (created if absent). Idempotent: already-applied versions are skipped.

Parameters

- `conn` `{sql.Connection}` - the open connection
- `migrations` `{list of Migration}` - the full migration set (any order)

Returns `{int}` - the number of migrations applied this run

sqlmigrate.migrationStatus(conn as sql.Connection, migrations as list of Migration)

The applied / pending state of every migration, in `version` order.

Parameters

- `conn` `{sql.Connection}` - the open connection
- `migrations` `{list of Migration}` - the full migration set

Returns `{list of MigrationStatus}` - one entry per migration, in version order

sqlmigrate.rollbackMigrations(conn as sql.Connection, migrations as list of Migration, steps as int)

Roll back the `steps` most-recently-applied migrations (by version order), running each one's `down` statements newest-first inside its own transaction and removing its tracking-table row.

Parameters

- `conn` `{sql.Connection}` - the open connection
- `migrations` `{list of Migration}` - the full migration set

- `steps {int}` - how many applied migrations to reverse

Returns `{int}` - the number of migrations rolled back

Structs

sqlmigrate.Migration

One schema migration: an ordered `up` and reverse `down` list of DDL statements (build them with `orm` 's DDL helpers or hand-written SQL), identified by `version` . Runs are ordered by `version` **lexically** , so zero-pad numeric versions (`"001"` , `"002"` , ...) or use a sortable timestamp.

Field	Type	Description
version	string	the ordering key ([A-Za-z0-9._-])
description	string	a human-readable label (recorded in the tracking table)
up	list of string	the forward DDL statements
down	list of string	the reverse DDL statements

sqlmigrate.MigrationStatus

The applied / pending state of one migration, from `sqlmigrate.migrationStatus` .

Field	Type	Description
version	string	the migration version
description	string	its description
applied	bool	whether it has been applied

statsd API reference

A StatsD client (over UDP): emit `metric:value|type` lines for a StatsD / Datadog / Telegraf agent to aggregate. This is the push counterpart to a pull-based scrape - it is fire-and-forget (UDP, no reply, no error on a missing agent), so a metric costs one datagram and never blocks the program.

A `Client` holds the sending socket, the agent address, and an optional metric-name prefix (namespace). The verbs map to the StatsD types: `count` / `increment` / `decrement` are counters (`c`), `gauge` a gauge (`g`), `timing` a timer in milliseconds (`ms`), and `set` a unique-member set (`s`). Needs the default `jennifer` binary (`net`).

Optional StatsD / DogStatsD extensions are layered on top: a `|@rate` sample-rate suffix (the `*Rate` verbs), `|#key:value` Datadog tags (the `*Tagged` verbs, carrying a `map of string to string`), float-valued counters / gauges (`countFloat` / `gaugeFloat`), and a `Batch` accumulator that packs several metrics into one datagram (`batch` / `add*` / `flush`). Every line - metric name, prefix, value, and tag keys / values - is control-character validated before it reaches the wire, so untrusted request data cannot forge extra metrics.

Import with `import "statsd.j" as statsd;` . See the `statsd` guide for prose and examples.

Functions

statsd.addCount(b as Batch, name as string, value as int)

Append a counter delta to the batch. Returns a new `Batch` .

Parameters

- `b` `{Batch}` - the accumulator
- `name` `{string}` - the metric name
- `value` `{int}` - the counter delta

Returns `{Batch}` - a new accumulator with "name:value|c" appended

statsd.addDecrement(b as Batch, name as string)

Append a counter -1 to the batch. Returns a new `Batch` .

Parameters

- `b` `{Batch}` - the accumulator
- `name` `{string}` - the metric name

Returns `{Batch}` - a new accumulator with "name:-1|c" appended

statsd.addGauge(b as Batch, name as string, value as int)

Append an absolute gauge to the batch. A negative value appends a 0-then-decrement pair (see `gauge`). Returns a new `Batch` .

Parameters

- `b` `{Batch}` - the accumulator
- `name` `{string}` - the metric name
- `value` `{int}` - the gauge value

Returns `{Batch}` - a new accumulator with "name:value|g" appended

statsd.addIncrement(b as Batch, name as string)

Append a counter +1 to the batch. Returns a new `Batch` .

Parameters

- `b` `{Batch}` - the accumulator
- `name` `{string}` - the metric name

Returns `{Batch}` - a new accumulator with "name:1|c" appended

statsd.addSet(b as Batch, name as string, value as string)

Append a unique set member to the batch. Returns a new `Batch` .

Parameters

- `b` `{Batch}` - the accumulator
- `name` `{string}` - the metric name
- `value` `{string}` - the set member

Returns `{Batch}` - a new accumulator with "name:value|s" appended

statsd.addTiming(b as Batch, name as string, ms as int)

Append a timing (milliseconds) to the batch. Returns a new `Batch` .

Parameters

- `b` `{Batch}` - the accumulator
- `name` `{string}` - the metric name
- `ms` `{int}` - the duration in milliseconds

Returns `{Batch}` - a new accumulator with "name:ms|ms" appended

statsd.batch(c as Client)

Start an empty `Batch` , capturing the client's metric-name prefix.

Parameters

- `c` `{Client}` - the client whose prefix the batched lines inherit

Returns `{Batch}` - an empty accumulator

statsd.client(host as string)

Open a client to a StatsD agent on `host` at the default port (8125), with no metric prefix.

Parameters

- `host` `{string}` - the agent host (e.g. "127.0.0.1")

Returns `{Client}` - a ready-to-use client

statsd.clientWith(address as string, prefix as string)

Open a client to a StatsD agent at a full `host:port` address, with a metric-name prefix ("" for none). The prefix is joined to every metric name with a "." (so prefix "web" and metric "hits" send "web.hits").

Parameters

- `address` `{string}` - the agent "host:port"
- `prefix` `{string}` - a metric-name namespace ("" for none)

Returns `{Client}` - a ready-to-use client

statsd.close(c as Client)

Close the client's sending socket.

Parameters

- `c` `{Client}` - the client

statsd.count(c as Client, name as string, value as int)

Adjust a counter by `value` (may be negative). Sends "name:value|c".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {int}` - the counter delta

statsd.countFloat(c as Client, name as string, value as float)

Adjust a counter by a fractional `value` . Sends "name:value|c" with the float formatted compactly (e.g. "3.5", no trailing zeros beyond the canonical form).

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {float}` - the counter delta

statsd.countRate(c as Client, name as string, value as int, rate as float)

Adjust a counter by `value` at a sample `rate` . Sends "name:value|c|@rate" (the "|@rate" suffix is omitted when `rate` >= 1). The agent scales the received count back up by 1/rate, so a `rate` of 0.1 emitted from a tenth of the events reconstructs the full total.

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {int}` - the counter delta
- `rate {float}` - the sample rate in (0, 1]; >= 1 sends no suffix

statsd.countTagged(c as Client, name as string, value as int, tags as map of string to string)

Adjust a counter by `value` , carrying DogStatsD tags. Sends "name:value|c|#k:v,...". Tag keys / values are control-character validated.

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {int}` - the counter delta
- `tags {map of string to string}` - tags rendered as "|#k:v,..." (insertion order)

statsd.decrement(c as Client, name as string)

Decrement a counter by 1. Sends "name:-1|c".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name

statsd.decrementTagged(c as Client, name as string, tags as map of string to string)

Decrement a counter by 1, carrying DogStatsD tags. Sends "name:-1|c|#k:v,...".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `tags {map of string to string}` - tags rendered as "|#k:v,..." (insertion order)

statsd.flush(c as Client, b as Batch)

Send every accumulated line in one UDP datagram (lines joined by "\n"). An empty batch sends nothing. Fire-and-forget, like the single-metric verbs.

Parameters

- `c {Client}` - the client
- `b {Batch}` - the accumulator to flush

statsd.gauge(c as Client, name as string, value as int)

Set a gauge to an absolute value. Sends "name:value|g".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {int}` - the gauge value

statsd.gaugeFloat(c as Client, name as string, value as float)

Set a gauge to a fractional absolute value (e.g. a load average). Sends "name:value|g", the float formatted compactly. A negative value is set as a 0-then-decrement pair (see `gauge`).

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {float}` - the gauge value

statsd.gaugeTagged(c as Client, name as string, value as int, tags as map of string to string)

Set a gauge to an absolute value, carrying DogStatsD tags. Sends "name:value|g|#k:v,...". A negative value is set as a 0-then-decrement pair (see `gauge`), each line carrying the tags.

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {int}` - the gauge value
- `tags {map of string to string}` - tags rendered as "|#k:v,..." (insertion order)

statsd.increment(c as Client, name as string)

Increment a counter by 1. Sends "name:1|c".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name

statsd.incrementTagged(c as Client, name as string, tags as map of string to string)

Increment a counter by 1, carrying DogStatsD tags. Sends "name:1|c|#k:v,...".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `tags {map of string to string}` - tags rendered as "|#k:v,..." (insertion order)

statsd.set(c as Client, name as string, value as string)

Record a unique member in a set (the agent counts distinct values). Sends "name:value|s".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {string}` - the set member

statsd.setTagged(c as Client, name as string, value as string, tags as map of string to string)

Record a unique set member, carrying DogStatsD tags. Sends "name:value|s|#k:v,...".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `value {string}` - the set member
- `tags {map of string to string}` - tags rendered as "|#k:v,..." (insertion order)

statsd.timing(c as Client, name as string, ms as int)

Record a timing in milliseconds. Sends "name:ms|ms".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `ms {int}` - the duration in milliseconds

statsd.timingRate(c as Client, name as string, ms as int, rate as float)

Record a timing in milliseconds at a sample `rate` . Sends "name:ms|ms|@rate" (the "|@rate" suffix is omitted when `rate` >= 1).

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `ms {int}` - the duration in milliseconds
- `rate {float}` - the sample rate in (0, 1]; >= 1 sends no suffix

statsd.timingTagged(c as Client, name as string, ms as int, tags as map of string to string)

Record a timing in milliseconds, carrying DogStatsD tags. Sends "name:ms|ms|#k:v,...".

Parameters

- `c {Client}` - the client
- `name {string}` - the metric name
- `ms {int}` - the duration in milliseconds
- `tags {map of string to string}` - tags rendered as "|#k:v,..." (insertion order)

Structs

statsd.Batch

An accumulator of formatted metric lines to pack into one UDP datagram (the StatsD multi-metric packet, lines separated by "\n"). Value-semantic: each `add*` verb returns a fresh `Batch` with the line appended, leaving the input unchanged. `flush` sends the whole packet through a client. The prefix is captured from the client at `batch` time and applied to every line.

Field	Type	Description
prefix	string	the metric-name namespace copied from the client
lines	list of string	the accumulated wire lines

statsd.Client

A StatsD client: a bound sending socket plus the agent address and an optional metric-name prefix. Value-copies share the underlying socket (the usual handle carve-out), so copying a `Client` is safe and cheap.

Field	Type	Description
socket	net.UDPSocket	the sending socket
address	string	the agent "host:port"
prefix	string	a metric-name namespace ("" for none)

telegram API reference

A Telegram Bot API client over the `http` client + `json` . Hold a `Bot` (token + API base URL), send messages with `sendMessage` / `sendPhoto` / `sendChatAction` , identify the bot with `getMe` , and poll for incoming updates with `getUpdates` (long-poll). Larger than the one-shot notifiers (`slack` / `discord` / `gotify`): `getUpdates` drives a stateful receive loop where the caller advances the `offset` past each processed update.

Needs the default `jennifer` binary (uses `net` via `http`). An API error (`{"ok": false, ...}`) throws `Error{kind: "telegram"}` . The bot token is a secret belonging to the caller - read it from the environment; never commit it.

Import with `import "telegram.j" as telegram;` . See the `telegram` guide for prose and examples.

Functions

telegram.answerCallbackQuery(b as Bot, callbackId as string, text as string)

Acknowledge a button press (`answerCallbackQuery`). Stops the client's loading spinner; the optional `text` shows a brief notification to the user.

Parameters

- b `{Bot}` - the bot
- `callbackId` `{string}` - the `CallbackQuery.id` to answer
- `text` `{string}` - a short notification ("" for none)

Returns `{bool}` - true on success

Throws

- `{Error}` - kind "telegram" on an API error

telegram.bot(token as string)

Create a bot against the public Telegram API.

Parameters

- `token` `{string}` - the bot token

Returns `{Bot}` - a ready bot

telegram.botWith(token as string, baseUrl as string)

Create a bot against a specific API base URL (a self-hosted Bot API server, or a test endpoint).

Parameters

- `token` `{string}` - the bot token
- `baseUrl` `{string}` - the API base URL (no trailing slash)

Returns `{Bot}` - a ready bot

telegram.buildUpload(field as string, chatId as int, filename as string, contentType as string, data as bytes)

Assemble the `multipart/form-data` parts for a file upload: a `chat_id` field plus the file itself under `field` (`"photo"` for `sendPhoto` , `"document"` for `sendDocument`). Pure - takes the file bytes, so it is testable with no disk or network. Feed the result's `contentType` / `body` to the API POST.

Parameters

- `field {string}` - the file's form-field name ("photo" or "document")
- `chatId {int}` - the target chat id
- `filename {string}` - the file name reported to Telegram
- `contentType {string}` - the file's MIME type ("" lets Telegram sniff)
- `data {bytes}` - the file content

Returns `{multipart.Built}` - the built form (Content-Type header + body)

telegram.getMe(b as Bot)

Fetch the bot's own identity (a good connectivity / token check).

Parameters

- b `{Bot}` - the bot

Returns `{User}` - the bot user

Throws

- `{Error}` - kind "telegram" on an API error

telegram.getUpdates(b as Bot, offset as int, timeout as int)

Long-poll for updates. Pass `offset` as the last processed `updateId` + `1` (0 for the first call) and `timeout` as the long-poll wait in seconds; the HTTP read is bounded a few seconds beyond that.

Parameters

- b `{Bot}` - the bot
- `offset {int}` - the first update id to fetch (last processed + 1)
- `timeout {int}` - the long-poll timeout in seconds

Returns `{list of Update}` - the pending updates (empty when none arrived)

Throws

- `{Error}` - kind "telegram" on an API error

telegram.inlineButton(text as string, callbackData as string)

A callback button: pressing it sends a `callback_query` carrying `callbackData` back to the bot (observe it via `getUpdates` + `parseCallbackQuery` , ack it with `answerCallbackQuery`).

Parameters

- `text {string}` - the button label
- `callbackData {string}` - the callback payload (<= 64 bytes)

Returns `{Button}` - the button

telegram.parseCallbackQuery(update as json.Value)

Parse an incoming update's `callback_query` into a `CallbackQuery` . The `update` is the raw decoded JSON of one update object (a `json.Value`). Pure: no network. Reads `/callback_query/{id,from,data,message/message_id}` ; a field that is absent lands as its zero value, and a field present but of the wrong JSON type raises a `telegram` error (rather than a raw `json` one), so callers catch malformed updates by the module's own error kind.

Parameters

- `update {json.Value}` - the decoded update object

Returns `{CallbackQuery}` - the parsed callback query (zero-valued fields when absent)

Throws

- `{Error}` - kind "telegram" when a present sub-field has the wrong type

telegram.redactToken(s as string, token as string)

Redact a bot token from a string (an error message, a log line). The token is a secret that rides in the request URL path (`/bot<TOKEN>/<method>`), so an error carrying the URL would otherwise leak it. Replaces the `bot<TOKEN>` path segment with `bot<redacted>` and any bare token occurrence with `<redacted>` . Applied to every error message this module rethrows from the HTTP layer.

Parameters

- `s {string}` - the string to scrub
- `token {string}` - the bot token to remove

Returns `{string}` - the string with the token removed

telegram.renderInlineKeyboard(rows as list of list of Button)

Render an inline-keyboard `reply_markup` to its JSON string. `rows` is a list of rows, each row a list of `Button` s. Pure - the exact JSON `sendMessageWith` variants attach as the `reply_markup` form parameter. A URL button emits `{"text":...,"url":...}` ; a callback button `{"text":...,"callback_data":...}` .

Parameters

- `rows {list of list of Button}` - the keyboard rows

Returns `{string}` - the `reply_markup` JSON (a `{"inline_keyboard":[...]}` object)

telegram.sendChatAction(b as Bot, chatId as int, action as string)

Send a chat action (e.g. "typing", "upload_photo") to show activity.

Parameters

- `b {Bot}` - the bot
- `chatId {int}` - the target chat id
- `action {string}` - the action

Returns `{bool}` - true on success

Throws

- `{Error}` - kind "telegram" on an API error

telegram.sendDocumentFile(b as Bot, chatId as int, filePath as string, contentType as string)

Upload a document from a local file path via `multipart/form-data` . Like `sendPhotoFile` but under the `document` field (any file type).

Parameters

- `b {Bot}` - the bot
- `chatId {int}` - the target chat id
- `filePath {string}` - the local file path
- `contentType {string}` - the file's MIME type ("" to let Telegram sniff)

Returns `{Message}` - the sent message

Throws

- `{Error}` - kind "telegram" on an API error (token-redacted)

telegram.sendMessage(b as Bot, chatId as int, text as string)

Send a text message to a chat.

Parameters

- b {Bot} - the bot
- chatId {int} - the target chat id
- text {string} - the message text

Returns {Message} - the sent message

Throws

- {Error} - kind "telegram" on an API error

telegram.sendMessageWith(b as Bot, chatId as int, text as string, parseMode as string)

Send a text message with a parse mode ("Markdown", "MarkdownV2", "HTML", or "" for plain).

Parameters

- b {Bot} - the bot
- chatId {int} - the target chat id
- text {string} - the message text
- parseMode {string} - the parse mode

Returns {Message} - the sent message

Throws

- {Error} - kind "telegram" on an API error

telegram.sendMessageWithKeyboard(b as Bot, chatId as int, text as string, rows as list of list of Button)

Send a text message with an inline keyboard attached. rows is a list of button rows (see urlButton / inlineButton); it is rendered to the reply_markup form parameter via renderInlineKeyboard .

Parameters

- b {Bot} - the bot
- chatId {int} - the target chat id
- text {string} - the message text
- rows {list of list of Button} - the inline-keyboard rows

Returns {Message} - the sent message

Throws

- {Error} - kind "telegram" on an API error

telegram.sendPhoto(b as Bot, chatId as int, photo as string, caption as string)

Send a photo by URL (or file id) with an optional caption.

Parameters

- b {Bot} - the bot
- chatId {int} - the target chat id
- photo {string} - a photo URL or a Telegram file id
- caption {string} - the caption ("" for none)

Returns {Message} - the sent message

Throws

- `{Error}` - kind "telegram" on an API error

telegram.sendPhotoFile(b as Bot, chatId as int, filePath as string, contentType as string)

Upload a photo from a local file path via `multipart/form-data` . Reads the file with `fs.readBytes` , names the part from the path's basename, and POSTs it as `sendPhoto` . The complement to `sendPhoto` , which takes a URL or file id. `contentType` may be "" to let Telegram sniff the format.

Parameters

- `b {Bot}` - the bot
- `chatId {int}` - the target chat id
- `filePath {string}` - the local file path
- `contentType {string}` - the file's MIME type ("" to let Telegram sniff)

Returns `{Message}` - the sent message

Throws

- `{Error}` - kind "telegram" on an API error (token-redacted)

telegram.urlButton(text as string, url as string)

A URL button: pressing it opens `url` in the client.

Parameters

- `text {string}` - the button label
- `url {string}` - the URL to open

Returns `{Button}` - the button

Structs

telegram.Bot

A bot: an API token and the API base URL.

Field	Type	Description
token	string	the bot token from BotFather
baseUrl	string	the API base URL (no trailing slash)

telegram.Button

One inline-keyboard button. Exactly one action is set: a `url` (opens a link) or a `callbackData` (fires a `callback_query` back to the bot). Build with `urlButton` / `inlineButton` rather than the raw literal.

Field	Type	Description
text	string	the button label
url	string	the URL to open ("" when this is a callback button)
callbackData	string	the callback payload ("" when this is a URL button)

telegram.CallbackQuery

An incoming button press (a Telegram `callback_query`). Acknowledge it with `answerCallbackQuery` so the client stops its loading spinner.

Field	Type	Description
id	string	the callback query id (pass to answerCallbackQuery)

from	User	the user who pressed the button
data	string	the button's callbackData ("" if none)
messageId	int	the id of the message the button is attached to (0 if absent)

telegram.Message

A message (the text-relevant fields).

Field	Type	Description
messageId	int	the message id
chatId	int	the chat id the message belongs to
text	string	the message text ("" for non-text messages)
date	int	the send date as a Unix timestamp

telegram.Update

One polled update.

Field	Type	Description
updateId	int	the update id (advance the poll offset to this + 1)
hasMessage	bool	whether this update carries a message
message	Message	the message (zero-valued when hasMessage is false)

telegram.User

A Telegram user (or bot).

Field	Type	Description
id	int	the user id
isBot	bool	whether the user is a bot
firstName	string	the first name
username	string	the @username ("" if none)

tengine API reference

A small text template engine for lightweight CMS-style rendering - a subset of Go's `text/template` (the Go / Hugo style), evaluated directly over a `json.Value` data tree. There is no compile step and no AST: the engine re-scans block bodies as it renders, which is fine at page / CMS scale. Dotted paths resolve as JSON Pointers against the current node (`.a.b` -> `/a/b` ; `.` is the current node); `$` is the root node, `$var` a variable; a missing key renders as empty.

SECURITY: output is **not** auto-escaped (this is `text/template` , not `html/template`). Rendering untrusted data into an HTML page is an XSS hole unless you escape it yourself: pipe every value that reaches HTML through the `html` pipe (`{{ .userInput | html }}`), or escape upstream. The engine will emit whatever the data contains, `<script>` included.

Actions: value output; `if` / `else if` / `else` with the comparison / boolean functions `eq ne lt le gt ge and or not` (parenthesised nesting allowed); `range` (with an optional `$i, $e :=` index / element bind) and `with` , each with an `else` ; variable assignment `{{ $x := PIPE }}` ; `{{ define }}` / `{{ template }}` / `{{ block }}` layout inheritance; `{{/* comments */}}` ; `{{-` / `-}}` whitespace-trim markers; and output pipes `upper lower title trim html urlize default truncate join len` . Over the compiled-in `json` / `strings` / `lists` / `maps` / `convert` libraries, so it runs on **both** binaries.

Import with `import "tengine.j" as tengine;` . See the `tengine` guide for prose and examples.

Functions

tengine.add(set as Set, name as string, src as string)

Add a template source under `name` , returning a new Set. Any top-level `{{ define "x" }}...{{ end }}` blocks are pulled out and registered as their own entries `x` , so a page can define the sections its base layout pulls in. Whitespace-trim markers (`{{-` / `-}}`) are applied as the source is stored.

Parameters

- `set` `{Set}` - the set to extend
- `name` `{string}` - the template name
- `src` `{string}` - the template source text

Returns `{Set}` - a new Set with the template (and any `define` s) added

Throws

- `{Error}` - kind "tengine" on an unterminated action

tengine.newSet()

Create an empty template set.

Returns `{Set}` - a fresh, empty set

tengine.render(set as Set, entry as string, data as json.Value)

Render the named template against a `json.Value` data tree.

Parameters

- `set` `{Set}` - the template set
- `entry` `{string}` - the name of the template to render
- `data` `{json.Value}` - the root data node

Returns `{string}` - the rendered output

Throws

- `{Error}` - kind "tengine" if `entry` (or a referenced template) is absent

Structs

tengine.Set

A value-semantic set of named template sources (kept as two parallel lists). Build it with `newSet` , populate it with `add` , and render an entry with `render` .

Field	Type	Description
names	list of string	the template names
sources	list of string	the template sources, positionally paired with names

totp API reference

Time-based one-time passwords (RFC 6238 TOTP over RFC 4226 HOTP) - the six-digit two-factor codes authenticator apps show. A shared **secret** (a base32 string, the same one an app stores) plus the current time yields a short numeric code that both sides compute independently. Built on `hash.hmac` (HMAC-SHA1 by default; SHA-256 / SHA-512 optional), `encoding` (base32 secrets), and `time` (the 30-second step); the dynamic-truncation step uses `bytes` + bitwise operators. Pure `.j` , runs on both binaries.

`generate` / `verify` read the clock; `generateAt` / `verifyAt` take an explicit Unix time (deterministic - use them in tests). `verify` accepts a +/-1-step clock-skew window. `uri` builds the `otpauth://` provisioning string an authenticator app scans as a QR code.

Import with `import "totp.j" as totp;` . See the `totp` guide for prose and examples.

Functions

totp.generate(secret as string, opts as Options)

Compute the TOTP code for the current time.

Parameters

- `secret {string}` - the base32 shared secret
- `opts {Options}` - digits / period / algorithm (zero-value = defaults)

Returns `{string}` - the zero-padded numeric code

totp.generateAt(secret as string, unixSeconds as int, opts as Options)

Compute the TOTP code for an explicit Unix time (seconds). Deterministic.

Parameters

- `secret {string}` - the base32 shared secret
- `unixSeconds {int}` - the Unix time in seconds
- `opts {Options}` - digits / period / algorithm (zero-value = defaults)

Returns `{string}` - the zero-padded numeric code

totp.generateSecret()

Generate a fresh random shared secret with the RFC 6238-recommended length for SHA-1 TOTP (20 bytes / 160 bits), as an unpadded base32 string.

Returns `{string}` - the base32-encoded secret

totp.generateSecretN(nbytes as int)

Generate a fresh random shared secret of an explicit byte length, as a base32 string (unpadded, the authenticator format `decodeSecret` accepts). Randomness is crypto-grade (`crypto.randBytes`), so the secret is unguessable. Round-trips: the result works with `generate` / `verify` .

Parameters

- `nbytes {int}` - the number of random bytes (RFC 6238 recommends >= 20 for SHA-1)

Returns `{string}` - the base32-encoded secret

totp.hotp(secret as string, counter as int)

The raw RFC 4226 HOTP: the six-digit HMAC-SHA1 code for an explicit counter, the counter-based building block TOTP layers a clock over. Use it directly for HOTP (event-based OTP) or to pin against the RFC 4226 test vectors; TOTP itself is just this over `unixSeconds // period` .

Parameters

- `secret` `{string}` - the base32 shared secret
- `counter` `{int}` - the RFC 4226 moving-factor counter

Returns `{string}` - the six-digit zero-padded numeric code

totp.uri(issuer as string, account as string, secret as string, opts as Options)

Build the `otpauth://totp/...` provisioning URI an authenticator app scans (as a QR code) to enrol an account. The label is `issuer:account` .

Parameters

- `issuer` `{string}` - the service name (e.g. "ACME")
- `account` `{string}` - the account name (e.g. "jane@acme.example")
- `secret` `{string}` - the base32 shared secret
- `opts` `{Options}` - digits / period / algorithm (zero-value = defaults)

Returns `{string}` - the otpauth provisioning URI

totp.verify(secret as string, code as string, opts as Options)

Verify a code against the current time, allowing a +/-1-step skew.

Parameters

- `secret` `{string}` - the base32 shared secret
- `code` `{string}` - the code to check
- `opts` `{Options}` - digits / period / algorithm (zero-value = defaults)

Returns `{bool}` - true if the code matches

totp.verifyAt(secret as string, code as string, unixSeconds as int, opts as Options)

Verify a code against an explicit Unix time, allowing a +/-1-step skew (so a code from the previous or next window still passes). Deterministic.

Parameters

- `secret` `{string}` - the base32 shared secret
- `code` `{string}` - the code to check
- `unixSeconds` `{int}` - the Unix time in seconds
- `opts` `{Options}` - digits / period / algorithm (zero-value = defaults)

Returns `{bool}` - true if the code matches this step or an adjacent one

totp.verifyWindow(secret as string, code as string, window as int, opts as Options)

Verify a code against the current time, allowing a `window` -step skew on each side of now.

Parameters

- `secret` `{string}` - the base32 shared secret
- `code` `{string}` - the code to check
- `window` `{int}` - the number of period-steps to accept on each side of now
- `opts` `{Options}` - digits / period / algorithm (zero-value = defaults)

Returns `{bool}` - true if the code matches any step within the window

totp.verifyWindowAt(secret as string, code as string, unixSeconds as int, window as int, opts as Options)

Verify a code against an explicit Unix time, allowing a `window` -step skew on each side of now (so `window` steps of clock drift in either direction still pass). `window` 0 checks only the current step; `window` 1 is the +/-1-step default `verifyAt` uses. Deterministic.

Parameters

- `secret` `{string}` - the base32 shared secret
- `code` `{string}` - the code to check
- `unixSeconds` `{int}` - the Unix time in seconds
- `window` `{int}` - the number of period-steps to accept on each side of now
- `opts` `{Options}` - digits / period / algorithm (zero-value = defaults)

Returns `{bool}` - true if the code matches any step within the window

Structs

totp.Options

TOTP parameters. A zero-value struct (`def o as totp.Options;`) means the common defaults: 6 digits, a 30-second step, and HMAC-SHA1.

Field	Type	Description
digits	int	the code length; 0 means 6
period	int	the time step in seconds; 0 means 30
algorithm	Algorithm	the HMAC digest: totp.Algorithm.Sha1 (the zero-value default), .Sha256, or .Sha512

Enums

totp.Algorithm

The HMAC digest algorithm behind a TOTP code: `Sha1` (the default), `Sha256` , or `Sha512` . Selected through `Options.algorithm` ; the zero value is `Sha1` .

transport API reference

The shared connection-transport security mode, used by every network-client module that opens a socket (`smtp` , `pop` , `imap` , `redis` , `amqp` , `mqtt`). One closed enum instead of a stringly-typed `security` field per module, so a value is typed and a typo is a compile-time error, not a silent plaintext downgrade.

`Security.None` is plaintext, `Security.Tls` is implicit TLS on connect, and `Security.Starttls` is an in-band upgrade after connecting (SMTP STARTTLS, POP3 STLS, IMAP STARTTLS). A module that has no in-band upgrade (`redis` , `amqp` , `mqtt`) rejects `Security.Starttls` with a clear error rather than treating it as anything else.

Build a connection option with a variant directly: `smtp.Options{security: transport.Security.Starttls, ...}` - which needs `import "transport.j" as transport;` alongside the client module.

Import with `import "transport.j" as transport;` . See the `transport` guide for prose and examples.

Functions

transport.checkReceiveSize(n as int, what as string)

Reject a received / declared size that is negative or exceeds `MAX_RECEIVED_BYTES` , raising a catchable `Error{kind: "transport"}` that names `what` . Call it on a peer-declared body / literal size before reading it, and on an accumulating buffer's length inside a read loop, so an endless or oversized stream fails typed instead of exhausting memory.

Parameters

- `n` `{int}` - the size to check (a declared size, or a running buffer length)
- `what` `{string}` - a short label for the message (e.g. "amqp message body")

transport.encrypted(s as Security)

Whether a security mode encrypts the connection (`Tls` or `Starttls`). `None` is the only unencrypted mode. Handy for a "refuse to send credentials over plaintext" guard.

Parameters

- `s` `{Security}` - the security mode

Returns `{bool}` - true if the connection is (or becomes) encrypted

Enums

transport.Security

A connection's transport security mode. - `None` - plaintext (no encryption). - `Tls` - implicit TLS negotiated on connect (SMTPS / POP3S / IMAPS / rediss / ...). - `Starttls` - connect in plaintext, then upgrade in-band (SMTP STARTTLS, POP3 STLS, IMAP STARTTLS). Only the protocols with an upgrade command accept it.

Constants

Constant	Type	Description
transport.MAX_RECEIVED_BYTES	int	The tree-wide 64 MiB ceiling on a single received payload. A wire client that reads a size (or grows a buffer) from an untrusted peer checks it against this with checkReceiveSize, so a hostile server cannot drive the recover-less interpreter to an unbounded allocation. One shared constant instead of a per-module copy, so the whole client suite caps identically.

uri API reference

URL parsing, building, and query-string handling (RFC 3986). The shared URL layer the network modules (`http` , `rest` , `s3` , `oauth` , `influxdb` , ...) build on instead of re-splitting strings and hand-rolling percent-encoding.

`parse` splits a URL into a `Uri` (scheme / user / host / port / path / query / fragment); `build` reassembles one. `encode` / `decode` are RFC 3986 percent-encoding (over the `encoding` library's `uri-percent` codec), and `encodeForm` / `decodeForm` are the `application/x-www-form-urlencoded` variant (space as "+", the `uri-form` codec). `buildQuery` / `parseQuery` convert between a `map of string to string` and a `key=value&...` string using form encoding (the query-string convention). `resolve` applies a relative reference to a base URL (RFC 3986 section 5), the operation a feed reader or crawler needs to turn a relative link absolute.

Pure Jennifer over `strings` + `encoding` + `convert` - no Go, no network - so it runs on both binaries. An IPv6 literal host keeps its brackets in `host` (`[::1]`) with the port split out; everything else follows RFC 3986.

Import with `import "uri.j" as uri;` . See the `uri` guide for prose and examples.

Functions

uri.build(u as Uri)

Reassemble a URL string from its parts (the inverse of parse). Components are emitted verbatim (already-encoded); build does not re-encode.

Parameters

- `u {Uri}` - the parts to assemble

Returns `{string}` - the URL string

uri.buildQuery(params as map of string to string)

Build a `key=value&...` query string from a map, form-encoding every key and value (the query-string convention: a space becomes "+"). Pairs keep the map's insertion order.

Parameters

- `params {map of string to string}` - the query parameters

Returns `{string}` - the encoded query string (without a leading "?")

uri.decode(s as string)

Reverse percent-encoding: %XX triples decode to their byte, and a literal "+" is left as "+" (RFC 3986, not form encoding - use parseQuery for query strings, which treats "+" as a space).

Parameters

- `s {string}` - the percent-encoded text

Returns `{string}` - the decoded text

Throws

- `{Error}` - on a malformed "%" escape

uri.decodeForm(s as string)

Reverse form-encoding: %XX triples decode to their byte and a "+" decodes to a space (the form-urlencoded convention). Use this for query-string values; use `decode` for RFC 3986 path segments where "+" is literal.

Parameters

- `s` `{string}` - the form-encoded text

Returns `{string}` - the decoded text

Throws

- `{Error}` - on a malformed "%" escape

uri.encode(s as string)

Percent-encode a string per RFC 3986 (component encoding): every byte outside the unreserved set `A-Za-z0-9-._~` becomes `%XX` , space becomes `%20` . Safe in any URL position (a path segment, a query value).

Parameters

- `s` `{string}` - the text to encode

Returns `{string}` - the percent-encoded text

uri.encodeForm(s as string)

Form-encode a string per `application/x-www-form-urlencoded` : like `encode` , but a space becomes "+" instead of "%20". This is the encoding for query-string values and HTML form bodies (`buildQuery` uses it).

Parameters

- `s` `{string}` - the text to encode

Returns `{string}` - the form-encoded text

uri.parse(raw as string)

Parse a URL into its parts. Absent components are "". The authority (user/host/port) is only populated when the URL has a "//" authority, so a scheme-only URL like "mailto:x@y" keeps "x@y" in `path` .

Parameters

- `raw` `{string}` - the URL text

Returns `{Uri}` - the parsed parts

uri.parseQuery(q as string)

Parse a `key=value&...` query string into a map, decoding each component. A "+" decodes to a space (the form-urlencoded convention), a repeated key keeps its last value, and a bare "key" (no "=") maps to "".

Parameters

- `q` `{string}` - the query string (without a leading "?")

Returns `{map of string to string}` - the decoded parameters

Throws

- `{Error}` - on a malformed "%" escape

uri.resolve(base as string, ref as string)

Resolve a (possibly relative) reference against a base URL, per RFC 3986 section 5: turn "../img.png" plus "https://h/a/b/page" into "https://h/a/img.png". An absolute reference (with its own scheme) is returned as-is.

Parameters

- `base` `{string}` - the absolute base URL
- `ref` `{string}` - the reference to resolve (absolute or relative)

Returns `{string}` - the resolved absolute URL

Structs

uri.Uri

The parts of a parsed URL. Missing components are the empty string (so a relative URL has an empty scheme and host). `port` is a string ("" when absent) rather than an int, to distinguish "no port" from port 0.

Field	Type	Description
scheme	string	the scheme without the trailing ":" (e.g. "https")
user	string	the userinfo before "@" ("" when absent)
host	string	the host name or IP literal ("" when absent)
port	string	the port after ":" ("" when absent)
path	string	the path, including its leading "/" when present
query	string	the raw query string without the leading "?"
fragment	string	the fragment without the leading "#"

validate API reference

Declarative validation of a `map of string to string` (a form body, a query, a config) against a rule set, returning a structured error list instead of ad-hoc per-field `if` checks. Rules compose as value-semantic descriptors built by the `validate.required` / `isInt` / `min` / `pattern` / `email` / ... family; group them per field in a `map of string to list of Rule` and call `validate.check` . Pure Jennifer over `regex` + `uri` + `time` + `password` + `convert` + `lists` + `strings` + `maps` , so it runs on **both binaries** .

Values are strings (the shape `web.bodyForm` / `dotenv` / a query string produce): a type rule (`isInt` / `isFloat` / `isBool`) checks the string parses; `min` / `max` parse then compare; `minLen` / `maxLen` measure the string. A field that is absent or blank passes every rule **except** `required` - so an optional field left empty is valid. Only the fields named in the rule set are checked; extra data keys are ignored.

Import with `import "validate.j" as validate;` . See the `validate` guide for prose and examples.

Functions

validate.byField(errs as list of Failure)

Group error messages by field (for re-rendering a form with per-field errors).

Parameters

- `errs {list of Failure}` - the failures

Returns `{map of string to list of string}` - field -> its messages

validate.check(data as map of string to string, rules as map of string to list of Rule)

Validate `data` against a rule set, returning every failure. Empty means valid.

Parameters

- `data {map of string to string}` - the field values
- `rules {map of string to list of Rule}` - rules per field

Returns `{list of Failure}` - the failures (empty when valid)

validate.custom(fn as func, message as string)

A custom rule: `fn` is a `func(value as string)` returning true when valid.

Parameters

- `fn {func}` - the predicate
- `message {string}` - the message when the predicate fails

Returns `{Rule}` - the rule

validate.datetime(format as string)

Require the value to be a valid date/time in the given `strftime` format (`time` 's format codes): `"%d.%m.%Y"` for `dd.mm.yyyy` , `"%m/%d/%Y"` for `mm/dd/yyyy` , `"%Y-%m-%d"` for ISO, `"%Y-%m-%d %H:%M"` for a date-time. The value must both match the format *and* be a real calendar date (month 13 or day 32 fail), since it is checked with `time.parse` . Pair with `withMessage` for a user-facing "use DD.MM.YYYY" hint.

Parameters

- `format {string}` - the strftime format the value must match

Returns `{Rule}` - the rule

validate.email()

Require the value to look like an email address.

Returns `{Rule}` - the rule

validate.isBool()

Require the value to be exactly "true" or "false".

Returns `{Rule}` - the rule

validate.isFloat()

Require the value to parse as a number (int or float).

Returns `{Rule}` - the rule

validate.isInt()

Require the value to parse as an integer.

Returns `{Rule}` - the rule

validate.localize(failures as list of Failure, templates as map of string to string)

Re-render failures with caller-supplied per-rule message templates - for non-English messages, or a house style. `templates` maps a rule id (the `Failure.rule` , e.g. `"min"` / `"email"`) to a template string; `%param%` is replaced with the failure's `param` (a threshold, joined choices) and `%field%` with its field name. A rule **not** in `templates` keeps its default `message` , so a partial map overrides only the rules it names. The templates can come from anywhere - a literal map, a config, or `intl.tr` per rule id. Returns the failures with `message` replaced, so it composes with `messages` / `byField` . The markers are `%name%` -style (brace-free) on purpose: they never collide with the language's `{expr}` string interpolation, so a template reads the same in a cooked or raw string. Substitution is a single pass (a substituted value is never re-scanned), and a literal `%` is written `%%` .

Parameters

- `failures` `{list of Failure}` - the failures from `check`
- `templates` `{map of string to string}` - rule id -> message template

Returns `{list of Failure}` - the failures with localized messages

validate.max(n as float)

Require the numeric value to be at most `n` .

Parameters

- `n` `{float}` - the maximum

Returns `{Rule}` - the rule

validate.maxLen(n as int)

Require the string to be at most `n` characters (runes).

Parameters

- `n` `{int}` - the maximum length

Returns `{Rule}` - the rule

validate.messages(errs as list of Failure)

Render an error list as "field: message" strings (for logging or a flash).

Parameters

- `errs` `{list of Failure}` - the failures

Returns `{list of string}` - one line per error

validate.min(n as float)

Require the numeric value to be at least `n` .

Parameters

- `n` `{float}` - the minimum

Returns `{Rule}` - the rule

validate.minLen(n as int)

Require the string to be at least `n` characters (runes).

Parameters

- `n` `{int}` - the minimum length

Returns `{Rule}` - the rule

validate.noneOf(blocked as list of string)

Reject the value if it is one of `blocked` (a blacklist - reserved usernames, banned words). Exact, case-sensitive matching, like `oneOf` ; for a case-insensitive blacklist use a `pattern` with the `(?i)` flag, or normalise the value first.

Parameters

- `blocked` `{list of string}` - the forbidden values

Returns `{Rule}` - the rule

validate.ok(data as map of string to string, rules as map of string to list of Rule)

Whether `data` satisfies every rule (a short-circuit over `check`).

Parameters

- `data` `{map of string to string}` - the field values
- `rules` `{map of string to list of Rule}` - rules per field

Returns `{bool}` - true when valid

validate.oneOf(allowed as list of string)

Require the value to be one of `allowed` .

Parameters

- `allowed` `{list of string}` - the permitted values

Returns `{Rule}` - the rule

validate.password(policy as pw.Schema)

Require the value to satisfy a `password` policy - length bounds and per-class minimums - by delegating to the `password` module's `validate` . Build `policy` with `password.schema()` and its `with*` builders. On failure the message is the policy's own failed-rule reasons, joined; override with `withMessage` for a single user-facing hint. Like every rule but `required` , an absent or blank value is skipped - pair with `required` to also demand a value.

Parameters

- `policy` `{pw.Schema}` - the policy to enforce (a `Schema` from the `password` module)

Returns `{Rule}` - the rule

validate.pattern(re as string)

Require the value to match an RE2 regular expression (anchored yourself with ^ / $ if you want a full match).

Parameters

- `re` `{string}` - the regex source

Returns `{Rule}` - the rule

validate.required()

Require the field to be present and non-empty. Every other rule is skipped for an absent or blank field, so pair `required` with them to also validate a value.

Returns `{Rule}` - the rule

validate.url()

Require the value to be an absolute URL (a scheme and a host).

Returns `{Rule}` - the rule

validate.withMessage(r as Rule, message as string)

Override the message of an already-built rule (fluent).

Parameters

- `r` `{Rule}` - the rule
- `message` `{string}` - the message to use on failure

Returns `{Rule}` - the updated rule

Structs

validate.Failure

One validation failure. `rule` is a stable id (the rule kind) and `param` is the rule's argument in string form (a threshold, a joined choices list) - the two a caller needs to render a custom or localized message via `localize` . The built-in `message` is the default (English).

Field	Type	Description
field	string	the field that failed
rule	string	the rule kind that failed (the stable message id)
param	string	the rule's argument for message interpolation ("" if none)
message	string	the default (English) human-readable message

validate.Rule

One validation rule (descriptor). Built by the rule family (`required` / `isInt` / `min` / ...); not usually constructed directly.

Field	Type	Description
kind	string	the rule kind ("required", "isInt", "min", "pattern", ...)
num	float	the numeric threshold for min / max

intVal	int	the length threshold for minLen / maxLen
str	string	the regex source for pattern
choices	list of string	the allowed values for oneOf / the blocked values for noneOf
fn	func	the predicate for custom (a func(value as string) -> bool)
schema	pw.Schema	the policy for password (a Schema from the password module)
message	string	an override message ("" = use the rule's default)

vcard API reference

Build and parse vCard (RFC 6350, vCard 4.0): a `Card` of contact fields encoded to a `VCARD` and parsed back. The contacts counterpart to `ical` - it shares the same content-line codec (TEXT escaping, 75-char line folding) through the included `ical_vcard_shared.inc.j` . Pure Jennifer over `strings` / `lists` ; both binaries.

A `Card` carries a formatted name (`FN`), a full structured name (`N` : family / given / additional / prefixes / suffixes), a nickname, organisation / title, any number of emails / phones / addresses (each with an optional `TYPE` like `work` / `home`), a birthday, a photo URI, categories, a URL, and a note. `encode` writes `FN` and `VERSION:4.0` and omits empty fields; `parse` reads one or many `VCARD` s, including the `TYPE` parameter on emails / phones / addresses. Text values are escaped / unescaped and long lines folded, so `parse(encode(card))` round-trips.

Import with `import "vcard.j" as vcard;` . See the `vcard` guide for prose and examples.

Functions

vcard.addAddress(c as Card, a as Address)

A copy of the card with a postal address appended.

Parameters

- `c {Card}` - the card
- `a {Address}` - the address

Returns `{Card}` - a fresh card with the address added

vcard.addCategory(c as Card, category as string)

A copy of the card with a category (`CATEGORIES` tag) appended.

Parameters

- `c {Card}` - the card
- `category {string}` - the category tag

Returns `{Card}` - a fresh card with the category added

vcard.addEmail(c as Card, email as string)

A copy of the card with an email appended.

Parameters

- `c {Card}` - the card
- `email {string}` - the email address

Returns `{Card}` - a fresh card with the email added

vcard.addEmailTyped(c as Card, email as string, type as string)

A copy of the card with a `TYPE` d email appended (e.g. `type` "work" / "home").

Parameters

- `c {Card}` - the card
- `email {string}` - the email address
- `type {string}` - the `TYPE` (e.g. "work" / "home"; "" for none)

Returns `{Card}` - a fresh card with the email added

vcard.addPhone(c as Card, phone as string)

A copy of the card with a phone number appended.

Parameters

- `c {Card}` - the card
- `phone {string}` - the phone number

Returns `{Card}` - a fresh card with the phone added

vcard.addPhoneTyped(c as Card, phone as string, type as string)

A copy of the card with a `TYPE` d phone number appended (e.g. `type` "cell").

Parameters

- `c {Card}` - the card
- `phone {string}` - the phone number
- `type {string}` - the `TYPE` (e.g. "work" / "home" / "cell"; "" for none)

Returns `{Card}` - a fresh card with the phone added

vcard.address(street as string, locality as string, region as string, postalCode as string, country as string)

A postal address.

Parameters

- `street {string}` - the street address
- `locality {string}` - the city / locality
- `region {string}` - the state / province / region
- `postalCode {string}` - the postal / ZIP code
- `country {string}` - the country name

Returns `{Address}` - the address

vcard.addressTyped(street as string, locality as string, region as string, postalCode as string, country as string, type as string)

A postal address with a `TYPE` (e.g. "work" / "home").

Parameters

- `street {string}` - the street address
- `locality {string}` - the city / locality
- `region {string}` - the state / province / region
- `postalCode {string}` - the postal / ZIP code
- `country {string}` - the country name
- `type {string}` - the `TYPE` ("work" / "home"; "" for none)

Returns `{Address}` - the address

vcard.card(formattedName as string)

A card with just its formatted name (`FN`). Other fields are empty until set.

Parameters

- `formattedName {string}` - the display name

Returns `{Card}` - the card

vcard.encode(c as Card)

Render a single card to vCard 4.0 text (a `VCARD` , CRLF-terminated). Empty optional fields are omitted.

Parameters

- `c {Card}` - the card to encode

Returns `{string}` - the vCard text

vcard.encodeAll(cards as list of Card)

Render many cards to one vCard text (concatenated `VCARD` s).

Parameters

- `cards {list of Card}` - the cards to encode

Returns `{string}` - the vCard text

vcard.parse(text as string)

Parse vCard text into a list of `Card` s (one entry per `VCARD`). Unfolds folded lines, ignores property parameters (the `;KEY=VALUE` after a name), reads the structured `N` / `ADR` values, and unescapes text values.

Parameters

- `text {string}` - the vCard text

Returns `{list of Card}` - the parsed cards (empty when the text has none)

vcard.withBday(c as Card, bday as string)

A copy of the card with its birthday (`BDAY`) set. The value is stored verbatim (a `YYYYMMDD` date, or a partial date like `--0315`).

Parameters

- `c {Card}` - the card
- `bday {string}` - the birthday value

Returns `{Card}` - a fresh card with the birthday set

vcard.withFullName(c as Card, family as string, given as string, additional as string, prefixes as string, suffixes as string)

A copy of the card with its full structured name (`N`) set: family, given, additional (middle) name(s), and honorific prefixes / suffixes.

Parameters

- `c {Card}` - the card
- `family {string}` - the family (last) name
- `given {string}` - the given (first) name
- `additional {string}` - the additional (middle) name(s)
- `prefixes {string}` - the honorific prefixes (e.g. "Dr.")
- `suffixes {string}` - the honorific suffixes (e.g. "Jr.", "PhD")

Returns `{Card}` - a fresh card with the full name set

vcard.withName(c as Card, family as string, given as string)

A copy of the card with its structured name (`N` family / given) set. The additional (middle) name and honorific prefixes / suffixes are left empty; use `withFullName` to set all five components.

Parameters

- `c {Card}` - the card
- `family {string}` - the family (last) name
- `given {string}` - the given (first) name

Returns `{Card}` - a fresh card with the name set

vcard.withNickname(c as Card, nickname as string)

A copy of the card with its nickname (`NICKNAME`) set.

Parameters

- `c {Card}` - the card
- `nickname {string}` - the nickname

Returns `{Card}` - a fresh card with the nickname set

vcard.withNote(c as Card, note as string)

A copy of the card with its note set.

Parameters

- `c {Card}` - the card
- `note {string}` - the note text

Returns `{Card}` - a fresh card with the note set

vcard.withOrg(c as Card, organization as string, title as string)

A copy of the card with its organisation and title set.

Parameters

- `c {Card}` - the card
- `organization {string}` - the organisation name
- `title {string}` - the job title

Returns `{Card}` - a fresh card with the org / title set

vcard.withPhoto(c as Card, photo as string)

A copy of the card with its photo (`PHOTO`) URI set.

Parameters

- `c {Card}` - the card
- `photo {string}` - the photo URI

Returns `{Card}` - a fresh card with the photo set

vcard.withUrl(c as Card, url as string)

A copy of the card with its URL set.

Parameters

- `c {Card}` - the card
- `url {string}` - the URL

Returns `{Card}` - a fresh card with the URL set

Structs

vcard.Address

A postal address (`ADR`). The RFC's PO-box and extended components are not modelled; the five common fields are, plus an optional `TYPE` .

Field	Type	Description

street	string	the street address
locality	string	the city / locality
region	string	the state / province / region
postalCode	string	the postal / ZIP code
country	string	the country name
type	string	the TYPE parameter ("work" / "home" / ...; "" if none)

vcard.Card

A contact card.

Field	Type	Description
formattedName	string	the FN display name (required by vCard 4.0)
family	string	the N family (last) name
given	string	the N given (first) name
additional	string	the N additional (middle) name(s)
prefixes	string	the N honorific prefixes (e.g. "Dr.")
suffixes	string	the N honorific suffixes (e.g. "Jr.", "PhD")
nickname	string	the NICKNAME ("" when unset)
organization	string	the ORG organisation ("" when unset)
title	string	the TITLE job title ("" when unset)
emails	list of Typed	the EMAIL addresses (each with an optional TYPE)
phones	list of Typed	the TEL phone numbers (each with an optional TYPE)
addresses	list of Address	the ADR postal addresses
url	string	the URL ("" when unset)
bday	string	the BDAY birthday ("" when unset; a YYYYMMDD or partial date)
photo	string	the PHOTO URI ("" when unset)
categories	list of string	the CATEGORIES tags
note	string	the NOTE free-text note ("" when unset)

vcard.Typed

A value with an optional TYPE parameter (e.g. an email or phone marked `work` / `home`). `type` is "" when unspecified.

Field	Type	Description
value	string	the property value (the email / phone number)
type	string	the TYPE parameter ("work" / "home" / "cell" / ...; "" if none)

web API reference

An ergonomic HTTP framework over the `httpd` server engine. Register routes against handler **func values** , then `web.run` owns the accept loop, matches each request, and dispatches to the handler. A handler is a `func` value - `func name(ctx as web.Context) { ... }` , passed by its bare name (`web.get($app, "/x", showUser)`) - and it is called through its home interpreter, so an entry-program handler runs in the entry program's context across the module boundary. Routing supports `:param` segments, a middleware chain, and a custom not-found handler. Response and request helpers hang off the `web.Context` so a handler rarely reaches for `httpd` directly. Needs the default `jennifer` binary (the `httpd` engine is net-backed; the constrained `jennifer-tiny` has no network stack).

CONCURRENCY: the serve loop handles requests **concurrently** - it accepts a request, hands it to its own `spawn` ed worker, and immediately accepts the next, so a handler that blocks (a `sql` query, an outbound `http` call, a slow filesystem read) no longer stalls every other request. In-flight concurrency is bounded by the `httpd` engine's accept rate. Dispatch calls each handler func value, which runs in its home (entry-program) context, and that cross-boundary call path is race-safe: each worker carries its own call-depth counter, and reads of shared program state (globals, constants, maps) are safe. The one thing that is *not* safe by construction is a handler **writing** a shared top-level variable of the entry program: those workers run against the same host globals (unlike a top-level `spawn` , which snapshots), so two handlers mutating one global race. Keep per-request mutable state in the request/response or a synchronized store (the `session` module), not in a top-level `def` a handler assigns.

Import with `import "web.j" as web;` . See the web guide for prose and examples.

Functions

web.basicAuth(ctx as Context)

Parse the request's HTTP Basic credentials. Checking them (against a user store) and sending a `401` challenge are the app's - web only decodes the header.

Parameters

- `ctx` `{Context}` - the request context

Returns `{BasicCredentials}` - the decoded credentials (`present` false if absent / invalid)

web.bearerToken(ctx as Context)

Return the request's bearer token (the `<token>` in `Authorization: Bearer <token>`), or "" when absent. Validate it yourself (an opaque lookup, or `jwt.verify` once the `jwt` module lands).

Parameters

- `ctx` `{Context}` - the request context

Returns `{string}` - the bearer token, or "" when absent

web.before(app as App, handler as func)

Register a middleware handler, run before each route handler. A middleware is `func name(ctx as web.Context) { ...; return true; }` : return true to continue to the route handler, or respond and return false to halt.

Parameters

- `app` `{App}` - the router to extend
- `handler` `{func}` - the middleware func value

Returns `{App}` - a new App with the middleware appended to the chain

web.body(ctx as Context)

Return the raw request body as bytes (binary-safe). Use `web.form` / `web.bodyJson` for the common structured bodies, or `convert.stringFromBytes` for text.

Parameters

- `ctx` `{Context}` - the request context

Returns `{bytes}` - the request body

web.bodyJson(ctx as Context)

Decode the request body as JSON. Errors (invalid JSON) propagate - catch them in the handler or let the framework's 500 net answer.

Parameters

- `ctx` `{Context}` - the request context

Returns `{json.Value}` - the decoded request body

web.cookie(ctx as Context, name as string)

Return the value of the request cookie named `name` , or "" if absent.

Parameters

- `ctx` `{Context}` - the request context
- `name` `{string}` - the cookie name

Returns `{string}` - the cookie value, or "" when absent

web.cors(app as App, opts as CorsOptions)

Set the CORS policy for the whole app, returning a new App. When set (a non-empty `opts.allowOrigin`), the serve loop adds the `Access-Control-*` headers to every response and answers a preflight `OPTIONS` request with a `204` before routing.

Parameters

- `app` `{App}` - the router to configure
- `opts` `{CorsOptions}` - the CORS policy

Returns `{App}` - a new App with the policy set

web.csrfCheck(ctx as Context, secret as string)

Validate the request's CSRF token: the submitted token (the `X-CSRF-Token` header, else the `csrf` form field) must equal the `csrf` cookie and its signature must verify. Guard unsafe methods (POST / PUT / PATCH / DELETE) with a `web.before` middleware that calls this and rejects on false.

Parameters

- `ctx` `{Context}` - the request context
- `secret` `{string}` - the app's CSRF secret

Returns `{bool}` - true when the request carries a valid token

web.csrfToken(ctx as Context, secret as string)

Mint a CSRF token, set it in the `csrf` cookie, and return it for embedding in a form (a hidden `csrf` field) or handing to the client for an `X-CSRF-Token` header. Call from the GET handler that renders the form.

Parameters

- `ctx` `{Context}` - the request context

- `secret` `{string}` - the app's CSRF secret (stable per deployment)

Returns `{string}` - the token to embed in the form / send as a header

web.delete(app as App, pattern as string, handler as func)

Register a DELETE handler for a pattern, returning a new App.

Parameters

- `app` `{App}` - the router to extend
- `pattern` `{string}` - the route pattern, with optional `:param` segments
- `handler` `{func}` - the handler func value

Returns `{App}` - a new App with the route added

web.etag(ctx as Context, tag as string)

Set an `ETag` and honour a conditional GET. Sets the `ETag` response header to `tag` (quoted) and, if the request's `If-None-Match` matches, answers `304 Not Modified` and returns true - the handler should then stop. Returns false when the client has no matching cached copy, so the handler sends the full body. `tag` is the app's choice of validator: a content hash (via `hash`), a row version, an mtime - so `web` needs no hashing of its own.

Parameters

- `ctx` `{Context}` - the request context
- `tag` `{string}` - the entity tag identifying this version of the response

Returns `{bool}` - true if a 304 was sent (stop), false to send the full response

web.form(ctx as Context)

Parse an `application/x-www-form-urlencoded` request body into a map of decoded field names to values.

Parameters

- `ctx` `{Context}` - the request context

Returns `{map of string to string}` - the form fields

web.formValue(ctx as Context, name as string)

Return one form field's value, or "" when absent.

Parameters

- `ctx` `{Context}` - the request context
- `name` `{string}` - the field name

Returns `{string}` - the field value, or "" when absent

web.get(app as App, pattern as string, handler as func)

Register a GET handler for a pattern, returning a new App.

Parameters

- `app` `{App}` - the router to extend
- `pattern` `{string}` - the route pattern, with optional `:param` segments
- `handler` `{func}` - the handler func value

Returns `{App}` - a new App with the route added

web.header(ctx as Context, name as string)

Return a request header value.

Parameters

- `ctx` `{Context}` - the request context
- `name` `{string}` - the header name

Returns `{string}` - the header value, or "" if absent

web.html(ctx as Context, status as int, body as string)

Answer with a text/html body.

Parameters

- `ctx` `{Context}` - the request context
- `status` `{int}` - the HTTP status code
- `body` `{string}` - the HTML body

web.joinRoute(prefix as string, pattern as string)

Concatenate a mount prefix and a route pattern into one clean pattern: drops empty segments (so stray or doubled slashes never matter) and always yields a single leading slash. `:param` and trailing `*wildcard` segments carry no slash, so they pass through intact. `""` + `"/deck"` -> `"/deck"` ; `"/v1"` + `"/"` -> `"/v1"` ; `""` + `""` -> `"/"` . Exposed so a layer above (e.g. `webapi`) can reconstruct the full pattern a mounted route is served under.

Parameters

- `prefix` `{string}` - the mount prefix
- `pattern` `{string}` - the route pattern

Returns `{string}` - the joined pattern

web.method(ctx as Context)

Return the request's HTTP method.

Parameters

- `ctx` `{Context}` - the request context

Returns `{string}` - the HTTP method (e.g. "GET")

web.mount(app as App, prefix as string, sub as App)

Mount a sub-router's routes under a path prefix, returning a new App. Every route in `sub` is re-registered on `app` with `prefix` prepended to its pattern (`web.mount($app, "/v1", $apiV1)` serves `apiV1` 's `/deck` at `/v1/deck`). A `prefix` of `""` (or `"/"`) mounts at the root, so the same sub-router can be mounted under several prefixes to serve one route set at many base paths.

Only the sub-router's **routes** are composed; app-level middleware, the not-found handler, CORS, and the error handler stay `app` 's. `:param` and trailing `*wildcard` segments are preserved.

Parameters

- `app` `{App}` - the router to extend
- `prefix` `{string}` - the base path to mount under ("" / "/" = root)
- `sub` `{App}` - the sub-router whose routes are mounted

Returns `{App}` - a new App with the sub-router's routes added under the prefix

web.multipartForm(ctx as Context)

Parse a `multipart/form-data` request body (a file-upload form) into its parts, using the request's

Content-Type header (with the boundary) and the raw body. The completion of the body-parser family alongside `web.form` (urlencoded) and `web.bodyJson` (JSON). Each `multipart.Part` is a field or a file (`multipart.isFile` / `multipart.text` distinguish them); the handler imports `multipart.j` to name the type. A missing boundary or malformed part throws `Error{kind: "multipart"}` .

Parameters

- `ctx` `{Context}` - the request context

Returns `{list of multipart.Part}` - the form parts (fields and files)

web.new()

Return an empty App with no routes or middleware.

Returns `{App}` - a fresh, empty router

web.notFound(app as App, handler as func)

Set a custom handler for unmatched requests (default: a plain 404).

Parameters

- `app` `{App}` - the router to extend
- `handler` `{func}` - the not-found handler func value

Returns `{App}` - a new App with the not-found handler set

web.onError(app as App, handler as func)

Register an error handler, returning a new App. When a handler or middleware throws, `web` catches it (so one failing request never stops the server), writes the failure to stderr, and *then also* invokes this handler with the thrown value - the `Error` struct `{kind, message, file, line, col}` for a runtime failure or a thrown `Error` (which crosses the module boundary intact, so the handler may declare its parameter `as Error` ; a handler that throws some other value delivers that value instead). The hook is **additive** : stderr always gets the error, so registering a handler adds alerting / struct- ured logging without ever losing the default diagnostic. The handler runs after the 500 response is already committed, so it cannot change the response; it is for observability. An error *from* the error handler is itself caught and logged, never re-raised.

Parameters

- `app` `{App}` - the router to extend
- `handler` `{func}` - the error-handler func value

Returns `{App}` - a new App with the error handler set

web.param(ctx as Context, name as string)

Return a captured path parameter, or "" if the route had none by that name.

Parameters

- `ctx` `{Context}` - the request context
- `name` `{string}` - the path parameter name

Returns `{string}` - the captured value, or "" if absent

web.patch(app as App, pattern as string, handler as func)

Register a PATCH handler for a pattern, returning a new App.

Parameters

- `app` `{App}` - the router to extend
- `pattern` `{string}` - the route pattern, with optional `:param` segments

- `handler` `{func}` - the handler func value

Returns `{App}` - a new App with the route added

web.path(ctx as Context)

Return the request's URL path.

Parameters

- `ctx` `{Context}` - the request context

Returns `{string}` - the request path

web.post(app as App, pattern as string, handler as func)

Register a POST handler for a pattern, returning a new App.

Parameters

- `app` `{App}` - the router to extend
- `pattern` `{string}` - the route pattern, with optional `:param` segments
- `handler` `{func}` - the handler func value

Returns `{App}` - a new App with the route added

web.put(app as App, pattern as string, handler as func)

Register a PUT handler for a pattern, returning a new App.

Parameters

- `app` `{App}` - the router to extend
- `pattern` `{string}` - the route pattern, with optional `:param` segments
- `handler` `{func}` - the handler func value

Returns `{App}` - a new App with the route added

web.query(ctx as Context, name as string)

Return a query-string parameter value.

Parameters

- `ctx` `{Context}` - the request context
- `name` `{string}` - the query parameter name

Returns `{string}` - the parameter value, or "" if absent

web.redirect(ctx as Context, status as int, location as string)

Redirect to `location` with the given status (301 / 302 / 303 / 307 / 308).

Parameters

- `ctx` `{Context}` - the request context
- `status` `{int}` - the redirect status code
- `location` `{string}` - the target URL for the Location header

web.remoteAddr(ctx as Context)

Return the client's network address (host:port).

Parameters

- `ctx` `{Context}` - the request context

Returns `{string}` - the remote address

web.renewSession(ctx as Context, cookieName as string)

Rotate the session id: mint a fresh UUID, set it as the session cookie, and return the new id. Call this right after any privilege change - a successful login, a logout, a role change - so a session id an attacker may have fixed before authentication is thrown away. This is the standard session-fixation defence; the application must also move its own session data to the new id.

Parameters

- `ctx` `{Context}` - the request context
- `cookieName` `{string}` - the session-id cookie name (e.g. "sid")

Returns `{string}` - the freshly minted session id

web.respond(ctx as Context, status as int, body as string)

Answer the request with a status code and body.

Parameters

- `ctx` `{Context}` - the request context
- `status` `{int}` - the HTTP status code
- `body` `{string}` - the response body

web.route(app as App, method as string, pattern as string, handler as func)

Register a handler for a method + pattern, returning a new App.

Parameters

- `app` `{App}` - the router to extend
- `method` `{string}` - the HTTP method to match
- `pattern` `{string}` - the route pattern, with optional `:param` segments
- `handler` `{func}` - the handler func value

Returns `{App}` - a new App with the route added

web.run(app as App, addr as string)

Listen on addr and serve forever, dispatching each request to its matched handler. Blocks; interrupt to stop.

Parameters

- `app` `{App}` - the router
- `addr` `{string}` - the listen address (e.g. ":8080")

web.sendGzip(ctx as Context, status as int, body as string)

Answer with a body, gzip-compressed when the client accepts it. Sets `Vary: Accept-Encoding` always; if the request's `Accept-Encoding` names gzip, the body is compressed and sent with `Content-Encoding: gzip` , otherwise it is sent as-is. Set the `Content-Type` yourself (via `web.setHeader`) before calling. Worth it for large text / JSON / HTML; skip it for already-compressed payloads (images, archives).

Parameters

- `ctx` `{Context}` - the request context
- `status` `{int}` - the HTTP status code
- `body` `{string}` - the response body

web.sendJson(ctx as Context, status as int, doc as json.Value)

Answer with an application/json body encoded from a json.Value. (Named sendJson, not json, because a

method may not shadow the `json` namespace this module imports.)

Parameters

- `ctx` `{Context}` - the request context
- `status` `{int}` - the HTTP status code
- `doc` `{json.Value}` - the JSON document to encode

web.serveDir(ctx as Context, root as string)

Serve static files from a directory root (path-safe; 404 for a missing file).

Parameters

- `ctx` `{Context}` - the request context
- `root` `{string}` - the directory to serve from

web.serveFile(ctx as Context, path as string)

Answer with a file from disk.

Parameters

- `ctx` `{Context}` - the request context
- `path` `{string}` - the filesystem path to serve

web.serveOn(app as App, srv as httpd.Server)

Serve on an already-listening httpd.Server, dispatching each request to its matched handler. Blocks until the server is shut down (httpd.accept then errors and the loop exits). Use this when you want to hold the server handle yourself - e.g. to shut it down from another task, or to serve from a `spawn` . `web.run` is the listen-and-serve convenience over it.

Parameters

- `app` `{App}` - the router
- `srv` `{httpd.Server}` - the already-listening server handle

web.sessionId(ctx as Context, cookieName as string)

Return the request's session id, minting a new one on first use. If the `cookieName` cookie is present its value is returned; otherwise a fresh UUID v4 (crypto-grade random, so unguessable as a session token) is generated and set as a `Secure` , `HttpOnly` , `SameSite=Lax` , path- `/` cookie (`Secure` is on by default; set `JENNIFER_WEB_INSECURE_COOKIES=1` for local plaintext-HTTP dev). `web` manages only the id cookie - the session data itself lives in a store the app owns (e.g. the `session` module over `memcache`), so `web` forces no store or network dependency. Call once per request and capture the returned id.

Parameters

- `ctx` `{Context}` - the request context
- `cookieName` `{string}` - the session-id cookie name (e.g. "sid")

Returns `{string}` - the session id (existing or newly minted)

web.setCookie(ctx as Context, name as string, value as string, opts as CookieOptions)

Set a response cookie with the given attributes (a `Set-Cookie` header).

Parameters

- `ctx` `{Context}` - the request context
- `name` `{string}` - the cookie name
- `value` `{string}` - the cookie value

- `opts` `{CookieOptions}` - the cookie attributes (zero-value = a session cookie)

web.setHeader(ctx as Context, name as string, value as string)

Set a response header.

Parameters

- `ctx` `{Context}` - the request context
- `name` `{string}` - the header name
- `value` `{string}` - the header value

web.text(ctx as Context, status as int, body as string)

Answer with a text/plain body.

Parameters

- `ctx` `{Context}` - the request context
- `status` `{int}` - the HTTP status code
- `body` `{string}` - the response body

Structs

web.App

The value-semantic router state: the routes, the middleware chain (handler func values run before each route), and an optional not-found handler (`hasNotFound` false = the built-in 404). Every registrar returns a fresh App.

Field	Type	Description
routes	list of Route	the registered routes
middleware	list of func	handler func values run before each route
notFound	func	the not-found handler (only when hasNotFound)
hasNotFound	bool	whether a custom not-found handler is set (else built-in 404)
cors	CorsOptions	the CORS policy applied by the serve loop (zero-value = off)
onError	func	the error-handler (only when hasOnError); see web.onError
hasOnError	bool	whether an error handler is set (stderr always logs regardless)

web.BasicCredentials

Parsed HTTP Basic credentials from the request. `present` is false when the request carried no valid `Authorization: Basic` header.

Field	Type	Description
user	string	the username
password	string	the password
present	bool	true when valid Basic credentials were supplied

web.Context

What a handler receives: the underlying request handle plus the path parameters captured from the matched route (`:id` -> params["id"]).

Field	Type	Description
req	httpd.Request	the underlying request handle
params	map of string to string	the captured path parameters

web.CookieOptions

Attributes for a Set-Cookie header. A zero-value struct is a session cookie (no attributes); set the fields you need.

Field	Type	Description
path	string	the Path attribute ("" omits it)
domain	string	the Domain attribute ("" omits it)
maxAge	int	the Max-Age in seconds; 0 omits it, a negative value expires the cookie now
httpOnly	bool	add HttpOnly (hide the cookie from JavaScript)
secure	bool	add Secure (send only over HTTPS)
sameSite	string	the SameSite attribute: "Lax", "Strict", "None", or "" to omit

web.CorsOptions

A cross-origin (CORS) policy. When set on an App via `web.cors` , the serve loop adds the `Access-Control-*` headers to every response and answers a preflight `OPTIONS` request with `204` . A zero-value struct (empty `allowOrigin`) leaves CORS off.

Field	Type	Description
allowOrigin	string	the Access-Control-Allow-Origin value ("*" or an origin; "" = off)
allowMethods	string	the Access-Control-Allow-Methods value ("" omits it)
allowHeaders	string	the Access-Control-Allow-Headers value ("" omits it)
allowCredentials	bool	add Access-Control-Allow-Credentials: true
maxAge	int	the Access-Control-Max-Age in seconds (0 omits it)

web.Route

One registered (method, pattern, handler) triple. Exported only to satisfy the referential-closure rule (App exposes a list of Route); programs never build one directly - they call web.get / web.route.

Field	Type	Description
method	string	the HTTP method (e.g. "GET")
pattern	string	the route pattern, with optional :param segments
handler	func	the handler func value to dispatch to

webapi API reference

A JSON-API conventions layer over the bundled `web` module. `web` is the HTTP framework - routing, `:param` captures, middleware, cookies, sessions, CORS, ETag; `webapi` adds the pieces every JSON API re-implements by hand on top of it: a uniform error envelope, request validation (reusing `validate`), versioned route mounting, pluggable bearer auth and rate limiting (the app supplies a verifier and a counter as `func` values, so this module depends on no `jwt` / store), content negotiation, and pagination.

It is a value-semantic builder finished once, before the server runs: `webapi.new()` -> add mounts, an authenticator, routes with a `Spec` -> `webapi.install($api, $app, apiGuard)`. Routes and the authenticator / limiter are `func` values; the one small guard shim in the entry program exists only to bind the built `Api` into `web`'s middleware chain (Jennifer has no closures yet, so the shim captures `$api` through a top-level binding):

```
func apiGuard(ctx as web.Context) { return webapi.guard($api, $ctx); }
```

The `Spec`-evaluation core (`webapi.evaluate`) is a pure function of `(Spec, Identity, data)`, testable without a server. Needs the default `jennifer` binary (`web` needs `net`); unavailable under `jennifer-tiny`.

Import with `import "webapi.j" as webapi;`. See the `webapi` guide for prose and examples.

Functions

webapi.alias(a as Api, version as int)

Additionally serve `version` at the bare root, so `/v1/deck` is also reachable as `/deck`.

Parameters

- a `{Api}` - the builder to extend
- `version` `{int}` - the version to also mount at the root

Returns `{Api}` - a new Api with the root alias added

webapi.authenticator(a as Api, handler as func)

Set the authenticator: an entry-program `func(token as string) -> webapi.Identity` value. The module calls it for a `Bearer` route; it never learns how a token is verified.

Parameters

- a `{Api}` - the builder to extend
- `handler` `{func}` - the verifier func value

Returns `{Api}` - a new Api with the authenticator set

webapi.delete(a as Api, pattern as string, handler as func, spec as Spec)

Register a DELETE route with a `Spec`.

Parameters

- a `{Api}` - the builder to extend
- `pattern` `{string}` - the route pattern
- `handler` `{func}` - the entry-program handler func value
- `spec` `{Spec}` - the route's specification

Returns `{Api}` - a new Api with the route added

webapi.denied(ctx as web.Context, message as string)

Send a `403` error envelope.

Parameters

- `ctx` `{web.Context}` - the request context
- `message` `{string}` - the error message

webapi.deprecate(a as Api, version as int, sunset as string)

Mark every mount of `version` deprecated with a sunset date, surfaced in the discovery document.

Parameters

- a `{Api}` - the builder to extend
- `version` `{int}` - the version to deprecate
- `sunset` `{string}` - the sunset date (e.g. an ISO date)

Returns `{Api}` - a new Api with the version marked deprecated

webapi.discovery(a as Api, registry as string, specVersion as string)

Build the discovery document from the route table, so the advertised versions and features cannot drift from what is actually served: `{"registry", "spec", "apis": [{version, basePath, deprecated, sunset}], "features": [name, ...]}` . `features` is the set of route feature labels.

Parameters

- a `{Api}` - the built API
- `registry` `{string}` - the registry / service name to advertise
- `specVersion` `{string}` - the spec version string to advertise

Returns `{json.Value}` - the discovery document

webapi.evaluate(spec as Spec, identity as Identity, data as map of string to string)

The pure core: decide whether a request bearing `identity` and `data` may proceed under `spec` , without touching the engine. Returns proceed=true, or proceed=false with the status / message (and validation failures) to answer. Order: auth (401) -> scopes (403) -> validation (422). Rate limiting is applied by the guard (it is stateful), not here.

Parameters

- `spec` `{Spec}` - the route specification
- `identity` `{Identity}` - the authenticated caller (ok=false when none)
- `data` `{map of string to string}` - the request data to validate

Returns `{Decision}` - proceed, or a status + message to answer

webapi.fail(ctx as web.Context, status as int, message as string)

Send the uniform error envelope `{"error": message}` at `status` .

Parameters

- `ctx` `{web.Context}` - the request context
- `status` `{int}` - the HTTP status
- `message` `{string}` - the error message

webapi.failWith(ctx as web.Context, status as int, message as string, failures as list of validate.Failure)

Send the error envelope with field-level `failures` detail: `{"error": message, "failures": [{field, rule, message}, ...]}` .

Parameters

- `ctx` `{web.Context}` - the request context
- `status` `{int}` - the HTTP status
- `message` `{string}` - the error message
- `failures` `{list of validate.Failure}` - the per-field failures

webapi.feature(a as Api, feature as string)

Set an explicit discovery feature label for the most recently added route (defaults to "METHOD /pattern"), so the discovery document reads meaningfully.

Parameters

- a `{Api}` - the builder whose last route to label
- `feature` `{string}` - the feature name

Returns `{Api}` - a new Api with the last route's feature set

Throws

- `{Error}` - kind "webapi" when there is no route to label

webapi.formData(ctx as web.Context)

The form body (`application/x-www-form-urlencoded`) as a string map.

Parameters

- `ctx` `{web.Context}` - the request context

Returns `{map of string to string}` - the form fields

webapi.get(a as Api, pattern as string, handler as func, spec as Spec)

Register a GET route with a `Spec` .

Parameters

- a `{Api}` - the builder to extend
- `pattern` `{string}` - the route pattern (`:param` captures allowed)
- `handler` `{func}` - the entry-program handler func value
- `spec` `{Spec}` - the route's specification

Returns `{Api}` - a new Api with the route added

webapi.guard(a as Api, ctx as web.Context)

The `Spec` -enforcing middleware body. Wire it from an entry-program shim named in `install` : `func apiGuard(ctx) { return webapi.guard($api, $ctx); }` . It matches the request to its route, authenticates, checks scopes, validates, and rate-limits, answering the request and returning false on any failure. A request that matches no API route passes through (returns true).

Parameters

- a `{Api}` - the built API
- `ctx` `{web.Context}` - the request context

Returns `{bool}` - true to proceed to the handler, false when it has answered

webapi.identity(a as Api, ctx as web.Context)

Inside a handler: the authenticated identity for this request (re-derived via the authenticator). Returns a zero `Identity` (`ok: false`) for a public route or an absent credential.

Parameters

- a `{Api}` - the built API
- `ctx` `{web.Context}` - the request context

Returns `{Identity}` - the caller's identity

webapi.install(a as Api, app as web.App, guard as func)

Register every route on the `web.App` , once per mount path (via `web.mount`), and wire the `Spec` -enforcing guard as a `before` middleware. `guard` must be an entry-program `func` value that calls `webapi.guard($api, $ctx)` ; the shim is only needed to bind the `Api` (Jennifer has no closures yet for the guard to capture it directly).

Parameters

- a `{Api}` - the built API
- `app` `{web.App}` - the web router to register onto
- `guard` `{func}` - the entry-program guard shim func value

Returns `{web.App}` - a new App with the routes and guard installed

webapi.jsonData(ctx as web.Context, fields as list of string)

The named top-level JSON body fields, flattened to strings (a nested or array-valued field is omitted; read those with `web.bodyJson`).

Parameters

- `ctx` `{web.Context}` - the request context
- `fields` `{list of string}` - the JSON keys to read

Returns `{map of string to string}` - the present scalar fields

webapi.limiter(a as Api, handler as func)

Set the rate limiter: an entry-program `func(key as string, limit as int) -> bool` value (true = allowed). Keyed on the identity's subject when authenticated, else the remote address.

Parameters

- a `{Api}` - the builder to extend
- `handler` `{func}` - the limiter func value

Returns `{Api}` - a new Api with the limiter set

webapi.mount(a as Api, version as int, path as string)

Serve this version's routes under `path` , returning a new Api. A version may be mounted at several paths (`mount` then `alias`); each is served by `install` .

Parameters

- a `{Api}` - the builder to extend
- `version` `{int}` - the version label (surfaced in discovery)
- `path` `{string}` - the base path to mount under

Returns `{Api}` - a new Api with the mount added

webapi.new()

A fresh, empty API builder.

Returns `{Api}` - an API with no routes, mounts, or auth

webapi.notFound(ctx as web.Context, message as string)

Send a `404` error envelope.

Parameters

- `ctx` `{web.Context}` - the request context
- `message` `{string}` - the error message

webapi.onError(app as web.App, handler as func)

Register a `web.onError` envelope handler on `app` , so an uncaught throw becomes a `500` in the same shape rather than a bare engine error. Pair with `install` ; the original error is still logged by `web` .

Parameters

- `app` `{web.App}` - the router to extend
- `handler` `{func}` - an entry-program `func(e as Error)` value that calls `webapi`

Returns `{web.App}` - a new App with the error handler set

webapi.page(ctx as web.Context, defaultLimit as int, maxLimit as int)

Parse and clamp the `offset` / `limit` query parameters into a `Page` . `limit` defaults to `defaultLimit` and is clamped to `[1, maxLimit]` ; `offset` is clamped to `>= 0` , so a client cannot ask for the whole table or a bad window.

Parameters

- `ctx` `{web.Context}` - the request context
- `defaultLimit` `{int}` - the limit when none is supplied
- `maxLimit` `{int}` - the largest allowed limit

Returns `{Page}` - the clamped window

webapi.patch(a as Api, pattern as string, handler as func, spec as Spec)

Register a PATCH route with a `Spec` .

Parameters

- `a` `{Api}` - the builder to extend
- `pattern` `{string}` - the route pattern
- `handler` `{func}` - the entry-program handler func value
- `spec` `{Spec}` - the route's specification

Returns `{Api}` - a new Api with the route added

webapi.post(a as Api, pattern as string, handler as func, spec as Spec)

Register a POST route with a `Spec` .

Parameters

- `a` `{Api}` - the builder to extend
- `pattern` `{string}` - the route pattern
- `handler` `{func}` - the entry-program handler func value
- `spec` `{Spec}` - the route's specification

Returns `{Api}` - a new Api with the route added

webapi.public()

A zero `Spec` : a public JSON route with no scopes, rules, or rate limit. Keeps an unauthenticated route a one-liner: `webapi.get($a, "/x", h, webapi.public())` .

Returns `{Spec}` - the zero specification

webapi.put(a as Api, pattern as string, handler as func, spec as Spec)

Register a PUT route with a `Spec` .

Parameters

- a `{Api}` - the builder to extend
- `pattern` `{string}` - the route pattern
- `handler` `{func}` - the entry-program handler func value
- `spec` `{Spec}` - the route's specification

Returns `{Api}` - a new Api with the route added

webapi.queryData(ctx as web.Context, fields as list of string)

The named query parameters as a string map (absent names omitted).

Parameters

- `ctx` `{web.Context}` - the request context
- `fields` `{list of string}` - the parameter names to read

Returns `{map of string to string}` - the present query values

webapi.sendJson(ctx as web.Context, status as int, value as json.Value)

Send an envelope-consistent JSON response.

Parameters

- `ctx` `{web.Context}` - the request context
- `status` `{int}` - the HTTP status
- `value` `{json.Value}` - the response body

webapi.sendPage(ctx as web.Context, items as json.Value, p as Page, total as int)

Send a paged list response: `{"items": [...], "offset", "limit", "total"}` . `items` must already be a `json.Value` list (the page slice the handler built).

Parameters

- `ctx` `{web.Context}` - the request context
- `items` `{json.Value}` - the page's items, as a JSON list
- `p` `{Page}` - the page window
- `total` `{int}` - the total item count across all pages

webapi.unauthorized(ctx as web.Context, message as string)

Send a `401` error envelope with a `WWW-Authenticate: Bearer` header.

Parameters

- `ctx` `{web.Context}` - the request context
- `message` `{string}` - the error message

webapi.validated(a as Api, ctx as web.Context)

Inside a handler: the request data the route's rules were checked against (query + body scalars). The guard already validated it, so a handler that got this far can trust it. Takes the `Api` because Jennifer has no per-request state to stash it in.

Parameters

- a `{Api}` - the built API
- `ctx` `{web.Context}` - the request context

Returns `{map of string to string}` - the request data

webapi.wants(ctx as web.Context)

What the caller wants from the `Accept` header: `"json"` or `"html"` . JSON is the default when the header is absent or `*/*` .

Parameters

- `ctx` `{web.Context}` - the request context

Returns `{string}` - "json" or "html"

Structs

webapi.Api

The built API description: its routes, mount points, and the authenticator / limiter `func` values. Value-semantic; every builder returns a fresh `Api` . The authenticator / limiter are entry-program `func` values, called through their home interpreter so they resolve their own imports and can construct a `webapi.Identity` across the module boundary.

Field	Type	Description
routes	list of RouteDef	the registered routes
mounts	list of Mount	the version mount points
authenticator	func	func(token as string) -> Identity (only when hasAuthenticator)
hasAuthenticator	bool	whether an authenticator is set (else every route is public)
limiter	func	func(key as string, limit as int) -> bool (only when hasLimiter)
hasLimiter	bool	whether a rate limiter is set (else no limiting)

webapi.Decision

The outcome of evaluating a `Spec` against a request: proceed, or halt with the status + message (and any validation failures) to answer. The return of the pure `webapi.evaluate` .

Field	Type	Description
proceed	bool	true if the request may run the handler
status	int	the HTTP status to answer when not proceeding
message	string	the error message to answer
failures	list of validate.Failure	the field-level failures (validation only)

webapi.Identity

The result of authenticating a request. `ok` is false when the credential was absent or rejected; the other fields describe the caller when it is true.

Field	Type	Description
ok	bool	whether the credential was accepted
subject	string	a stable identifier for the caller
display	string	a human-readable label
scopes	list of string	the permissions this caller holds

webapi.Mount

One version mount point. Exported only because `Api.mounts` exposes a list of them; programs build these through `webapi.mount` / `alias` , never directly.

Field	Type	Description
version	int	the version label
path	string	the base path it serves under ("" = root)
sunset	string	the deprecation date ("" = not deprecated)

webapi.Page

A page window parsed from a request's `offset` / `limit` query parameters, clamped by `webapi.page` .

Field	Type	Description
offset	int	the zero-based start index
limit	int	the page size

webapi.RouteDef

One registered route. Exported only because `Api.routes` exposes a list of them; programs build these through `webapi.get` / `post` / ..., never directly.

Field	Type	Description
method	string	the HTTP method
pattern	string	the un-prefixed route pattern
handler	func	the entry-program handler func value
feature	string	the discovery feature label (defaults to "METHOD /pattern")
spec	Spec	the route's specification

webapi.Spec

The metadata attached to a route: what it needs and what it makes. A zero `Spec` (from `webapi.public()`) is an unauthenticated JSON route with no rules.

Field	Type	Description
summary	string	a one-line human description (used in discovery)
auth	Auth	the authentication requirement
scopes	list of string	the permissions the identity must hold
rules	map of string to list of validate.Rule	per-field request validation
rateLimit	int	allowed requests per identity per window (0 = unlimited)
produces	Produces	the response content type policy

Enums

webapi.Auth

How a route authenticates. `None` is public (no credential); `Bearer` requires an `Authorization: Bearer <token>` the authenticator accepts.

webapi.Produces

What a route produces, driving content negotiation. `Json` always answers JSON, `Html` always HTML, `Negotiate` picks from the request's `Accept` .

webhook API reference

Sign and verify HMAC-signed webhooks - the GitHub-style `X-Hub-Signature-256` convention. A sender computes `sha256=<hex>` , the hex HMAC-SHA256 of the exact request body keyed by a shared secret, and sends it in a header; a receiver recomputes it and compares, confirming the delivery is authentic and untampered. `sign` / `verify` are pure and run on **both** binaries; `send` POSTs a payload with the signature header and needs the default binary (`net` via `http`).

Import with `import "webhook.j" as webhook;` . See the `webhook` guide for prose and examples.

Functions

webhook.send(url as string, payload as string, secret as string)

POST a payload to a webhook URL with the `X-Hub-Signature-256` header set, returning the receiver's HTTP response. The body is sent as `application/json` (the common webhook content type). Inspecting the result needs `import "http.j"` for the `http.Response` type.

Parameters

- `url` `{string}` - the receiver URL
- `payload` `{string}` - the request body
- `secret` `{string}` - the shared secret

Returns `{http.Response}` - the receiver's response (status / headers / body)

Throws

- `{Error}` - on a network failure (a positioned `http` / `net` error)

webhook.sign(payload as string, secret as string)

Sign a payload: `sha256=` followed by the hex HMAC-SHA256 of the payload keyed by the shared secret - the value a receiver checks in `X-Hub-Signature-256` .

Parameters

- `payload` `{string}` - the exact request body
- `secret` `{string}` - the shared secret

Returns `{string}` - the signature, e.g. "sha256=757107ea..."

webhook.slackSign(secret as string, body as string, timestamp as int)

Sign a payload the Slack way: the signed base string is `"v0:"` + `<timestamp>` + `":"` + `<body>` , HMAC-SHA256, hex-encoded, and the returned value is `v0=<hexsig>` (the `X-Slack-Signature` shape). The timestamp travels separately in `X-Slack-Request-Timestamp` .

Parameters

- `secret` `{string}` - the Slack signing secret
- `body` `{string}` - the exact request body
- `timestamp` `{int}` - the unix timestamp (seconds), the request timestamp

Returns `{string}` - the signature, e.g. "v0=a2114d57..."

webhook.slackVerify(secret as string, body as string, timestamp as int, signature as string, toleranceSeconds as int, now as int)

Verify a Slack `v0=...` signature against a body, secret, and the request timestamp (from `X-Slack-Request-Timestamp`). Recomputes the HMAC-SHA256 over `"v0:"` + `<timestamp>` + `":"` + `<body>` and constant-time compares. Returns true only if the signature matches AND the timestamp is within

toleranceSeconds of now (replay protection, e.g. 300s).

Parameters

- `secret` `{string}` - the Slack signing secret
- `body` `{string}` - the exact request body received
- `timestamp` `{int}` - the request timestamp (seconds) from the header
- `signature` `{string}` - the received `v0=...` header value
- `toleranceSeconds` `{int}` - the freshness window in seconds (e.g. 300)
- `now` `{int}` - the current unix time (seconds)

Returns `{bool}` - true if the signature is valid and the timestamp is fresh

webhook.stripeSign(secret as string, body as string, timestamp as int)

Sign a payload the Stripe way: the signed base string is `<timestamp>` + `"."` + `<body>` , HMAC-SHA256, hex-encoded, and the returned header value is `t=<timestamp>,v1=<hexsig>` (the `Stripe-Signature` shape).

Parameters

- `secret` `{string}` - the endpoint signing secret
- `body` `{string}` - the exact request body
- `timestamp` `{int}` - the unix timestamp (seconds) to stamp and sign

Returns `{string}` - the header value, e.g. "t=1492774800,v1=5257a8..."

webhook.stripeVerify(secret as string, body as string, header as string, toleranceSeconds as int, now as int)

Verify a Stripe `t=...,v1=...` signature header against a body and secret. Parses the timestamp and the (possibly several) `v1=` signatures, recomputes the HMAC-SHA256 over `<t>.<body>` , and constant-time compares. Returns true only if a signature matches AND the timestamp is within toleranceSeconds of now (replay protection).

Parameters

- `secret` `{string}` - the endpoint signing secret
- `body` `{string}` - the exact request body received
- `header` `{string}` - the received `t=...,v1=...` header value
- `toleranceSeconds` `{int}` - the freshness window in seconds (e.g. 300)
- `now` `{int}` - the current unix time (seconds)

Returns `{bool}` - true if a signature is valid and the timestamp is fresh

webhook.timestampedSign(secret as string, body as string, timestamp as int, algo as string, encoding as string)

Generic timestamped HMAC signature: sign `<timestamp>` + `"."` + `<body>` with a caller-chosen digest and text encoding. Covers GitHub-style `sha1` / `sha256` and hex/base64 schemes from one composable primitive.

Parameters

- `secret` `{string}` - the shared secret
- `body` `{string}` - the exact request body
- `timestamp` `{int}` - the unix timestamp (seconds) to stamp and sign
- `algo` `{string}` - the HMAC digest: "sha1" or "sha256"
- `encoding` `{string}` - the text encoding: "hex" or "base64"

Returns `{string}` - the encoded signature (no scheme prefix)

webhook.timestampedVerify(secret as string, body as string, timestamp as int, signature as string, algo as string, encoding as string, toleranceSeconds as int, now as int)

Verify a generic timestamped HMAC signature (see `timestampedSign`) with a constant-time compare and a replay-protecting freshness check. Returns true only if the signature matches AND the timestamp is within toleranceSeconds of `now` .

Parameters

- `secret` `{string}` - the shared secret
- `body` `{string}` - the exact request body received
- `timestamp` `{int}` - the request timestamp (seconds)
- `signature` `{string}` - the received signature (encoded, no prefix)
- `algo` `{string}` - the HMAC digest: "sha1" or "sha256"
- `encoding` `{string}` - the text encoding: "hex" or "base64"
- `toleranceSeconds` `{int}` - the freshness window in seconds
- `now` `{int}` - the current unix time (seconds)

Returns `{bool}` - true if the signature is valid and the timestamp is fresh

webhook.verify(payload as string, signature as string, secret as string)

Verify a signature against a payload and secret, with a constant-time compare.

Parameters

- `payload` `{string}` - the exact request body received
- `signature` `{string}` - the received signature, e.g. "sha256=..."
- `secret` `{string}` - the shared secret

Returns `{bool}` - true if the signature is valid

websocket API reference

An RFC 6455 WebSocket client over `net` . `connect` performs the HTTP `Upgrade` handshake (verifying the server's `Sec-WebSocket-Accept` , which is SHA-1 + base64 of the client key), then `send` / `sendBytes` write masked text / binary frames and `receive` reads the next message, transparently answering pings with pongs and reassembling fragmented messages. `close` sends a close frame and shuts the socket. `ws://` is plain TCP, `wss://` is TLS.

Client-to-server frames are masked (required by the spec); server frames are not. Needs the default `jennifer` binary (`net`); a protocol error or a dropped connection throws `Error{kind: "websocket"}` . A server-side upgrade is out of scope (it needs an `httpd` connection-hijack hook).

Import with `import "websocket.j" as websocket;` . See the `websocket` guide for prose and examples.

Functions

websocket.close(c as Conn)

Send a close frame and shut the connection.

Parameters

- `c {Conn}` - the connection

websocket.connect(url as string)

Open a WebSocket connection to a ws:// or wss:// URL (default receive timeout).

Parameters

- `url {string}` - the ws:// or wss:// URL

Returns `{Conn}` - the open connection

Throws

- `{Error}` - kind "websocket" on a failed handshake

websocket.connectWith(url as string, timeoutMs as int)

Open a WebSocket connection with an explicit per-receive read timeout.

Parameters

- `url {string}` - the ws:// or wss:// URL
- `timeoutMs {int}` - the per-receive read timeout in milliseconds

Returns `{Conn}` - the open connection

Throws

- `{Error}` - kind "websocket" on a failed handshake

websocket.ping(c as Conn)

Send a ping (empty payload).

Parameters

- `c {Conn}` - the connection

websocket.receive(c as Conn)

Receive the next message. Pings are answered with pongs and skipped; fragmented text / binary messages are reassembled. A close frame is returned as kind "close".

Parameters

- `c {Conn}` - the connection

Returns `{Message}` - the next message

Throws

- `{Error}` - kind "websocket" on a protocol error or dropped connection

websocket.send(c as Conn, text as string)

Send a text message.

Parameters

- `c {Conn}` - the connection
- `text {string}` - the message text

websocket.sendBytes(c as Conn, data as bytes)

Send a binary message.

Parameters

- `c {Conn}` - the connection
- `data {bytes}` - the message payload

Structs

websocket.Conn

An open WebSocket connection.

Field	Type	Description
socket	net.Conn	the underlying (plain or TLS) connection
timeoutMs	int	the per-receive read timeout in milliseconds

websocket.Message

A received message.

Field	Type	Description
kind	string	"text", "binary", "close", or "pong"
text	string	the decoded text (for a "text" message; "" otherwise)
data	bytes	the raw payload bytes