LLD: Middleware Lifecycle Flow

Middleware Lifecycle — Low-Level Design

Document: LLD-MIDDLEWARE **Status:** Approved **Last updated:** 2026-02-21 **Author:** Standards Architect **Applies to:** Drop API (Hono) — `src/drop-api/src/app.ts`

Overview

The Drop API uses Hono as its HTTP framework. Middleware is organized into two layers: **global middleware** applied to every request via `app.use("*")`, and **per-route middleware** applied within individual route handlers. This document describes the complete execution order.

Source of truth: `src/drop-api/src/app.ts`

Middleware Execution Order

```
flowchart TD
    A["Incoming HTTP Request"] --> B["1. CORS Middleware\n(global)"]
    B --> C["2. Request ID Middleware\n(global)"]
    C --> D["3. Client IP Middleware\n(global)"]
    D --> E["4. Route Matching\n(/v1/* or /api/*)"]

    E --> F{Route found?}
    F -->|No| G["404 Not Found"]
    F -->|Yes| H["5. Per-Route Middleware\n(auth / rateLimit / featureGate)"]

    H --> I["6. Route Handler"]
    I --> J["Response"]
```

```
    I -.->|Error thrown| K["Global Error Handler\n(app.onError)"]
    K --> J

    style A fill:#f5f5f5,stroke:#333
    style B fill:#ffd93d,stroke:#333
    style C fill:#ffd93d,stroke:#333
    style D fill:#ffd93d,stroke:#333
    style H fill:#6bcb77,stroke:#333
    style I fill:#4d96ff,stroke:#333,color:#fff
    style K fill:#ff6b6b,stroke:#333,color:#fff
```

Global Middleware (Applied to Every Request)

These are registered in `app.ts` with `app.use("*")` and execute in registration order, top-to-bottom.

1. CORS (`hono/cors`)

Source: `app.ts:23-30`

Configures Cross-Origin Resource Sharing headers for browser-based clients.

Setting	Value
Allowed origins	`http://localhost:3000`, `http://localhost:3001`, `process.env.APP_URL`
Credentials	`true` (cookies sent cross-origin)

The `credentials: true` setting is required because the web app sends JWT tokens in httpOnly cookies. Empty strings from unset env vars are filtered out.

2. Request ID

Source: `app.ts:33-38`

Generates or propagates a unique request identifier for distributed tracing.

Behavior	Detail
Header checked	`x-request-id`

Behavior	Detail
Fallback	`crypto.randomUUID()`
Context variable	`c.get("requestId")`
Response header	`x-request-id` (echoed back)

Downstream middleware and route handlers access the request ID via `c.get("requestId")` for structured logging and audit trails.

3. Client IP

Source: `app.ts:41-47`

Extracts the originating client IP address from proxy headers.

Priority	Header	Processing
1st	`x-real-ip`	Trimmed
2nd	`x-forwarded-for`	First entry, trimmed
Fallback	—	`127.0.0.1`

The extracted IP is stored as `c.get("clientIp")` and used by rate limiting and audit logging.

Note: The `rate-limit.ts` module also exports a `getClientIp(c)` helper that performs the same extraction. Some route handlers use `getClientIp(c)` directly instead of `c.get("clientIp")`.

4. Global Error Handler

Source: `app.ts:50`, `middleware/error-handler.ts:16-23`

Registered via `app.onError(globalErrorHandler)`. This is not middleware in the traditional sense — it is an error boundary that catches any unhandled exceptions thrown during request processing.

Error Type	Response
`HTTPException` (Hono)	Returns the exception's status and message
All other errors	Logs via `logger.error`, reports to Sentry via `captureError`, returns `500 Internal Server Error` with generic message

The error handler never leaks stack traces or internal details to the client.

Route Mounting

Source: `app.ts:53-72`

All API routes are mounted under a versioned prefix:

Mount Point	Purpose
`/v1/*`	Primary API path (mobile + new clients)
`/api/*`	Backward compatibility during migration

Both mount points serve the identical route handlers — `/api` is an alias for `/v1`.

Mounted Route Groups

Path	Route Module	Primary Middleware
`/v1/auth`	`authRoutes`	Rate limiting (inline)
`/v1/health`	`healthRoutes`	None (public)
`/v1/transactions`	`transactionRoutes`	`authMiddleware` + rate limiting
`/v1/recipients`	`recipientRoutes`	`authMiddleware`
`/v1/rates`	`rateRoutes`	None (public)
`/v1/cards`	`cardRoutes`	`authMiddleware` + feature gate
`/v1/merchants`	`merchantRoutes`	`merchantMiddleware`
`/v1/settings`	`settingsRoutes`	`authMiddleware`
`/v1/notifications`	`notificationRoutes`	`authMiddleware`
`/v1/user`	`userRoutes`	`authMiddleware`
`/v1/admin`	`adminRoutes`	`adminMiddleware` + rate limiting
`/v1/consents`	`consentRoutes`	`authMiddleware`
`/v1/complaints`	`complaintRoutes`	`authMiddleware`
`/v1/cron`	`cronRoutes`	Varies
`/v1/withdrawal`	`withdrawalRoutes`	`authMiddleware`

Per-Route Middleware

Per-route middleware is applied within individual route files, not globally. It executes **after** the global middleware chain.

Authentication Middleware (`middleware/auth.ts`)

Three variants, all following the same pattern: extract JWT, verify token + session, set `c.set("user", ...)`.

Middleware	Role Check	Used By
`authMiddleware`	Any authenticated user	Most routes (transactions, recipients, settings, etc.)
`merchantMiddleware`	`role === 'merchant'`	Merchant routes
`adminMiddleware`	`role === 'admin'`	Admin routes (audit, screening, STR)

Flow:

1. Extract bearer token from `Authorization` header or cookie
2. Verify JWT signature (HS256) and check session in `sessions` table
3. If invalid or expired: return `401 Unauthorized`
4. If role mismatch (merchant/admin variants): return `403 Forbidden`
5. Set `c.set("user", authUser)` for downstream handlers

Rate Limiting (`middleware/rate-limit.ts`)

Rate limiting is **not** a Hono middleware function — it is a utility called inline within route handlers.

```
// Example from transactions.ts
if (!(await rateLimit(ip, 10))) {
  return c.json({ error: "rate_limited", message: "Too many requests" }, 429);
}
```

Parameter	Description
`ip`	Rate limit key (usually client IP, sometimes `user:{id}`)
`limit`	Maximum requests per window
`windowMs`	Window duration in ms (default: 60000 = 1 minute)

Rate limit state is persisted in the `rate_limits` database table (SQLite/PostgreSQL). Expired entries are cleaned up every 100 checks.

Per-endpoint limits:

Endpoint	Key	Limit	Window

`POST /transactions/remittance`	IP	10/min	60s
`POST /transactions/remittance`	`user:{id}`	3/min	60s
`POST /transactions/qr-payment`	IP	10/min	60s
`POST /transactions/qr-payment`	`user:{id}`	3/min	60s
`GET /admin/audit`	IP	30/min	60s
`GET /admin/screening`	IP	30/min	60s
`POST /admin/screening`	IP	10/min	60s
`GET /admin/str`	IP	30/min	60s
`POST /admin/str`	IP	10/min	60s
`PATCH /admin/str`	IP	10/min	60s

Feature Gates (`lib/feature-flags.ts`)

Feature gates control access to unreleased functionality. Like rate limiting, they are called inline within route handlers, not as Hono middleware.

```
// Example from cards.ts
if (!isEnabled("virtualCards")) {
  return c.json({ error: "not_found", message: "Feature not available" }, 404);
}
```

Flag	Default	Controls
`virtualCards`	`false`	Card creation, listing, detail, cancellation
`physicalCards`	`false`	Physical card ordering
`cardDetails`	`false`	Card detail endpoint
`cardFreeze`	`false`	Card freeze/unfreeze
`cardPin`	`false`	Card PIN management
`spendingLimits`	`false`	Spending limit management
`notifications`	`true`	Notification endpoints
`merchantDashboard`	`true`	Merchant dashboard

Flags are read from environment variables (`FF_VIRTUAL_CARDS=true`) with fallback to compiled defaults. The `featureGate()` helper throws an `HTTPException(404)` for disabled features, which the global error handler catches.

Complete Request Lifecycle (Sequence Diagram)

```
sequenceDiagram
    participant Client
    participant CORS as CORS Middleware
    participant ReqID as Request ID Middleware
    participant IP as Client IP Middleware
    participant Router as Hono Router
    participant Auth as Auth Middleware
    participant RL as Rate Limiter
    participant FG as Feature Gate
    participant Handler as Route Handler
    participant DB as Database
    participant ErrH as Error Handler

    Client->>CORS: HTTP Request
    CORS->>CORS: Check origin, set CORS headers
    CORS->>ReqID: next()
    ReqID->>ReqID: Extract/generate x-request-id
    ReqID->>IP: next()
    IP->>IP: Extract client IP from headers
    IP->>Router: next()

    Router->>Router: Match route (/v1/* or /api/*)

    alt Public route (health, rates)
        Router->>Handler: Direct execution
    else Authenticated route
        Router->>Auth: authMiddleware / adminMiddleware / merchantMiddleware
        Auth->>DB: Verify JWT + session
        alt Token invalid
            Auth-->>Client: 401 Unauthorized
        else Token valid
            Auth->>Auth: Set c.user
            Auth->>RL: Check rate limit (inline)
            alt Rate exceeded
```

```
            RL-->>Client: 429 Too Many Requests
        else Within limit
            RL->>FG: Check feature flag (if applicable)
            alt Feature disabled
                FG-->>Client: 404 Feature not available
            else Feature enabled
                FG->>Handler: Execute route logic
                Handler->>DB: Query/mutation
                Handler-->>Client: JSON response
            end
        end
    end
end

Note over Handler,ErrH: If any error is thrown
Handler-->>ErrH: Unhandled error
ErrH->>ErrH: Log + Sentry report
ErrH-->>Client: 500 Internal Server Error
```

Input Validation

Input validation is not middleware — it is a collection of utility functions in `middleware/validation.ts` called directly by route handlers.

Function	Purpose	Used By
`sanitizeText(text, maxLength)`	Strip HTML tags, control characters, truncate	All text input fields
`validatePhone(phone)`	International phone format (`+` prefix, 8-15 digits)	User profile
`validateAmount(amount)`	Positive number, max 2 decimal places	Transactions
`validateIBAN(iban)`	ISO 13616 IBAN checksum validation	Bank accounts
`validatePIN(pin)`	Exactly 4 digits	Card PIN
`validateEmail(email)`	Basic email format	Registration
`validateCurrency(currency)`	Whitelist: EUR, USD, GBP, BAM, CHF, PLN, NOK, RSD, TRY, PKR	Transactions
`validateName(name)`	Non-empty, contains letters, no script injection	Recipients

Function	Purpose	Used By
`validateLanguage(lang)`	Whitelist: nb, en, bs, sq	Settings
`auditLog(...)`	Insert audit trail record	All significant actions

Cross-References

- Security Architecture — Trust boundaries, STRIDE, application security controls
- Authentication — JWT, session management, BankID OIDC
- API Reference — Endpoint specifications and security requirements
- Login Authentication Flow — BankID OIDC authentication detail

---

Revision #3
Created 2026-02-24 10:13:00 UTC by John
Updated 2026-05-23 10:56:22 UTC by John