LEX: Lead Exchange Standard

The Open Standard for Sales Lead Interchange — An Adopter's Guide

---

Version: 1.0 (Draft)
Date: April 2026
Author: V.G.P LexStandard

Published

Table of Contents

LEX: Lead Exchange Standard

Version: 1.0 (Draft)
Status: Published
Date: April 2026
Author: V.G.P LexStandard

Abstract

LEX (Lead Exchange Standard) is an open data standard for sales lead messages. It defines what a lead contains, the states it can move through, and how systems exchange it — without locking you into a specific platform, transport, or vendor.

This whitepaper is written for adopters: the teams building integrations, the developers choosing a library, and the decision-makers asking whether LEX is right for their industry. It covers what LEX is, the value it delivers by vertical, and how to get an integration running using one of the four available client libraries.

This document is the complete adopter-facing standard. It is self-contained — no external files or repository access are required to implement LEX from this whitepaper.

The online specification, JSON Schemas, field dictionary, client libraries, and industry examples are published at https://lexstandard.org.

1. Why LEX Was Built

1.1 The Problem Every Industry Has

Every business that sells high-value assets — vehicles, aircraft, construction equipment, real estate, enterprise software — faces the same invisible friction: the moment a buyer expresses interest in one system, the data has to travel through several others before it reaches the person who can close the sale.

A buyer fills in a web form. That enquiry goes to a lead aggregator. The aggregator sends it to a dealer. The dealer's DMS receives it and creates a CRM record. If the manufacturer is involved, a copy goes there too. And if the buyer doesn't hear back, or hears back three times from three different people, or the lead arrives two days late — the sale is simply lost.

The technical cause is almost always the same: every system in that chain uses a slightly different data format, and the integrations between them are bespoke, fragile, and expensive to maintain.

1.2 The Integration Challenge

Where shared data standards exist for sales lead exchange, they are technically rigid. They define a fixed set of fields — so anything beyond that set requires proprietary extensions. Those extensions diverge across platform implementations until interoperability that existed in theory disappears in practice.

The governance challenge is equally significant. When dominant platform vendors gain control over a shared standard — through proprietary variants, integration brokerage fees, or dues-weighted governance bodies — the standard becomes a revenue mechanism rather than shared infrastructure. Participants end up paying for integrations that should be free.

Every other industry — heavy equipment, marine, aviation, real estate — has no shared standard at all. Lead exchange in those verticals is done by email, spreadsheet, or phone call.

1.3 What LEX Changes

LEX is an open standard for lead exchange that any system can implement, in any language, at no cost. It solves these structural problems:

- **A complete lifecycle model** — from first browsing session through to sale or closure, with defined states your whole supply chain can act on

- **Four wire formats** — JSON, XML, X12, EDIFACT — so you connect to what your counterparty already runs, without forcing anyone to rebuild their infrastructure
- **An extension mechanism** — your custom fields live in a named namespace; they travel safely through systems that don't understand them yet
- **Cross-industry by design** — automotive, heavy equipment, maritime, aviation, real estate, and technology all use the same message structure with industry-specific extension blocks
- **Apache 2.0 licensed, no gatekeeping** — no membership fee, no certification body, no vendor that controls access to the ecosystem

1.4 Who This Whitepaper Is For

This document is for teams and organisations that want to **adopt** LEX — to send or receive leads using the standard, integrate with counterparties who use it, or build products on top of it.

If you are a:

- **Dealer or retailer** receiving leads from portals and manufacturers
- **Manufacturer or OEM** routing leads to dealer networks
- **DMS or CRM vendor** adding LEX support to your integration layer
- **Lead aggregator or portal** sending leads to multiple downstream systems
- **Technology team** in any industry that sells high-value assets

— this whitepaper explains what LEX is, what it means for your industry, and exactly how to start using it.

2. What LEX Is

2.1 The Short Version

LEX is an open data standard for sales lead messages. It defines what a lead message contains, what states it can be in, and how it moves between systems. You implement it once; every counterparty that also implements it becomes a zero-friction connection.

Libraries are available for Python, JavaScript/Node.js, Java, and C#. Copy one file into your project and you are producing and consuming LEX messages.

2.2 Five Message Types

LEX defines five message types that cover the full lead exchange lifecycle:

Message	What it carries	When you use it
LEAD	A buyer's enquiry, status, contact details, product interest	Every time a lead is created or updated
ASSET	Product specification and availability data	When pushing inventory data to a partner system
ACKNOWLEDGMENT	Receipt confirmation, validation result, rejection reason	Sent back after receiving any message
SUBSCRIPTION	A request to receive future messages matching a filter	Setting up a lead routing rule
LEAD_CLOSURE	The final outcome — sold, lost, duplicate, expired	When a lead reaches a terminal state

For most integrations you will work primarily with LEAD and LEAD_CLOSURE. The others are available when you need them.

2.3 Four Wire Formats — One Data Model

LEX does not force a wire format. The same lead data can be serialised as:

Format	Best for
JSON-EDI	REST APIs, cloud integrations, any modern system
XML-EDI	Enterprise middleware, legacy DMS, ESB pipelines
X12	EDI VANs, automotive retail networks, contractual X12 requirements
EDIFACT	European markets, maritime trade, existing EDIFACT infrastructure

The LEX libraries handle all four. You work with the same Python/Java/JS/C# objects regardless of which format your partner requires.

2.4 What You Do Not Need to Change

LEX is designed to sit alongside existing infrastructure:

- **No new transport layer.** HTTPS, message queues, SFTP batch — all work.
- **No new database.** LEX messages are data records; store them however you store things now.
- **No certification.** You declare your own conformance level against public test criteria.
- **No membership.** The specification and libraries are Apache 2.0 licensed. Free to use, forever.

2.5 What Happens in a Typical Integration

A buyer submits an enquiry on a portal. The portal creates a LEX LEAD message and sends it to your endpoint. Your system receives it, validates it using the LEX library, creates a record in your CRM, and sends back an ACKNOWLEDGMENT. When the deal closes — whether won or lost — a LEAD_CLOSURE message is sent back to the portal so their records are complete.

That exchange works the same way whether you are a car dealer, a yacht broker, a mining equipment distributor, or a commercial real estate firm. The data model is the same; only the product fields differ.

2.6 Versioning and Stability

LEX uses semantic versioning. Optional fields and new message types are additive (minor version). Changes that break existing messages require a major version bump with a documented migration path. The current version is `1.0.0`.

2.7 Namespacing for Extensions

Your industry-specific fields live in a named extension block:

```
{org-identifier}.{division}.{purpose}
```

Example: `acme.fleet.preferredDeliveryPort`

Register your namespace at https://lexstandard.org/registry. Registration is free and confers no special rights over the core specification.

3. LEX Across Industries

The data model is the same for every industry. What changes is the product being enquired about and, where needed, a small block of industry-specific fields added as an extension. The core integration — receiving a lead, acknowledging it, updating its status, closing it — is identical.

3.1 Automotive Retail

Who participates: OEM platforms, national dealer networks, DMS providers, independent dealers, used-vehicle portals.

The problem LEX solves here: Lead integrations in the automotive sector are bespoke, fragile, and expensive to maintain. A dealer with five lead sources typically runs five different integrations, each maintained separately, with integration costs borne by the dealer rather than the platform.

What LEX changes: One integration handles all sources. The lead lifecycle (SHOPPING through ORDER) maps directly to a modern automotive sales funnel. Trade-in data, finance preferences, and EV specifications are all carried as standard extension blocks.

Industry-specific fields: - `desiredProduct.vin` — specific vehicle requested - `lex.ev` extension — battery capacity, charging standard, range requirements - `lex.financial` extension — trade-in valuation, monthly payment target, deposit - `lex.captive` extension — manufacturer finance offer data

Example scenario: A buyer configures a vehicle on an OEM portal. A LEAD with status `EXPLORING` is sent to the headquarters platform and simultaneously routed to the nearest dealer's DMS. When the buyer books a test drive, the dealer updates the lead to `APPOINTMENT_REQUEST`. The outcome — sale or lost — comes back via LEAD_CLOSURE to the OEM portal, completing the loop.

3.2 Heavy Equipment

Who participates: Manufacturers (construction, mining, agricultural), regional distributors, fleet procurement teams, rental companies.

The problem LEX solves here: Fleet buyers purchasing 20 excavators have no standard way to submit a structured procurement enquiry. Enquiries arrive by email, get manually re-keyed into the distributor's CRM, and the manufacturer never knows the outcome.

What LEX changes: A procurement team sends a single structured LEAD with `assetClass: HEAVY_EQUIPMENT` and a `fleetSize` of 20. The distributor's system receives it, routes to the right regional

team, and closure data flows back to the manufacturer for forecasting.

Industry-specific fields: - `desiredProduct.assetClass: HEAVY_EQUIPMENT` - `desiredProduct.genericSerialNumber` — specific unit identifier - `lex.financial.tenderReference` — procurement tender number - `organisationBuyer.buyerType: FLEET` — company, government, or NGO buyer

---

3.3 Maritime

Who participates: Shipyards, yacht brokers, vessel distributors, leasing companies, port procurement offices.

The problem LEX solves here: Vessel procurement is largely relationship-driven with no digital standard. Large orders (ferries, cargo ships) involve lengthy negotiation with no tracked lifecycle visible to the manufacturer.

What LEX changes: Enquiries from brokers and fleet operators arrive as structured LEAD messages. The full negotiation lifecycle — EXPLORING, RESERVATION, IN_NEGOTIATION, ORDER — is tracked. The manufacturer's platform can push ASSET messages with updated vessel specifications back to the broker's system.

Industry-specific fields: - `desiredProduct.assetClass: MARITIME` - `desiredProduct.maritimeSpecs.vesselType` — cargo, ferry, patrol, yacht, etc. - `desiredProduct.maritimeSpecs.grossTonnage` - `desiredProduct.maritimeSpecs.flagState`

3.4 Aviation

Who participates: Aircraft manufacturers, MRO organisations, charter operators, government procurement offices, leasing companies.

The problem LEX solves here: Aircraft acquisition involves long lead times, multiple stakeholders, and complex financing. There is no shared digital format for the initial enquiry-to-order flow between operator and manufacturer.

What LEX changes: Operators submit structured enquiries with full specification requirements. The lifecycle tracks from initial interest through to delivery reservation. Government procurement rules (tender references, ECA financing) are carried in standard extension fields.

Industry-specific fields: - `desiredProduct.assetClass: AVIATION` - `desiredProduct.aviationSpecs.aircraftType` — fixed-wing, rotary, UAV -

`desiredProduct.aviationSpecs.seatingCapacity` - `desiredProduct.aviationSpecs.rangeNm` - `lex.financial.ecaFinancing` — export credit agency financing flag

3.5 Real Estate

Who participates: Developers, agencies, property portals, commercial brokers, property management platforms.

The problem LEX solves here: Property portals generate leads in their own format. Agency CRMs receive them in another. The developer's sales team works in a third. None of these systems close the loop — the portal never learns whether its lead converted.

What LEX changes: Portals send structured LEX LEADs with property identifiers and buyer requirements. Agencies update status as the process progresses. LEAD_CLOSURE returns the outcome — enabling the portal to measure true conversion, not just lead volume.

Industry-specific fields: - `desiredProduct.assetClass: REAL_ESTATE` - `desiredProduct.realEstateSpecs.propertyType` — residential, commercial, industrial - `desiredProduct.realEstateSpecs.tenure` — freehold, leasehold - `desiredProduct.realEstateSpecs.floorAreaSqm`

3.6 Enterprise Technology

Who participates: Software vendors, hardware OEMs, managed service providers, channel partners, enterprise procurement teams.

The problem LEX solves here: Partner channels generate leads that sales operations teams manually process. There is no standard for a reseller to programmatically submit a qualified enterprise opportunity to a vendor's platform.

What LEX changes: Channel partners submit structured leads with qualification data. Vendor platforms acknowledge, route, and update status back to the partner. Deal registration and protection is evidence-based rather than email-based.

Industry-specific fields: - `desiredProduct.assetClass: TECHNOLOGY` - `desiredProduct.technologySpecs.productCategory` — software, hardware, services - `desiredProduct.technologySpecs.deploymentModel` — cloud, on-premise, hybrid - `organisationBuyer.companySize` — employee count band - `lex.financial.annualContractValue`

4. Getting Started

You need two things to start using LEX: the library for your language, and an understanding of what a message looks like. This chapter covers both.

4.1 Install the Library

Install the LEX library from your language's standard package registry. All four libraries are compiled from the same Haxe source and behave identically.

Python

```
pip install lexstandard
```

JavaScript / Node.js

```
# npm
npm install @lexstandard/lex-js

# or yarn
yarn add @lexstandard/lex-js
```

Java

```
<!-- Maven -->
<dependency>
 <groupId>io.lexstandard</groupId>
 <artifactId>lex-java</artifactId>
 <version>1.0.0</version>
</dependency>
```

```
// Gradle
implementation 'io.lexstandard:lex-java:1.0.0'
```

C# (.NET)

```
# NuGet
dotnet add package LexStandard
```

Pre-built binaries for all four platforms are also available at `https://lexstandard.org/libraries`.

4.2 Receive Your First Lead

The most common first integration: your system receives an inbound lead from a portal or aggregator, validates it, and acknowledges receipt.

Python

```python
from lex_client import LexClient

client = LexClient()

# Receive and validate
result = client.parse_json(raw_json_string)

if result['valid']:
    lead = result['message']['lex']['payload']['lead']
    print(f"New lead: {lead['leadId']} | {lead['status']}")
    print(f"Customer: {lead['customer']['firstName']} {lead['customer']['lastName']}")
    print(f"Product:  {lead['desiredProduct']['assetClass']} - {lead['desiredProduct'].get('make', '')}")
else:
    for error in result['errors']:
        print(f"Validation error: {error['message']}")
```

JavaScript

```javascript
const LexClient = require('./LexClient');
const client = new LexClient();

client.parseJson(rawJsonString).then(result => {
    if (result.valid) {
        const lead = result.message.lex.payload.lead;
        console.log(`New lead: ${lead.leadId} | ${lead.status}`);
        console.log(`Customer: ${lead.customer.firstName} ${lead.customer.lastName}`);
    } else {
        result.errors.forEach(e => console.error('Validation error:', e.message));
    }
});
```

Java

```java
LexClient client = new LexClient();
LexClient.ParseResult result = client.parseJson(rawJsonString);

if (result.isValid()) {
```

```
    JsonObject lead = result.getMessage()
        .getAsJsonObject("lex")
        .getAsJsonObject("payload")
        .getAsJsonObject("lead");
    System.out.println("New lead: " + lead.get("leadId").getAsString());
} else {
    result.getErrors().forEach(System.err::println);
}
```

C# (.NET)

```
var client = new LexClient();
var result = client.ParseJson(rawJsonString);

if (result.IsValid) {
    var lead = result.Message["lex"]["payload"]["lead"];
    Console.WriteLine($"New lead: {lead["leadId"]} | {lead["status"]}");
} else {
    foreach (var error in result.Errors)
        Console.Error.WriteLine($"Validation error: {error}");
}
```

4.3 Send a Lead

Creating and sending a new lead — for example, from a portal to a dealer's system.

Python

```
from lex_client import LexClient
import json

client = LexClient()

lead_data = {
    "leadId":  "LEAD-2026-001234",
    "status":  "EXPRESSED_INTEREST",
    "customer": {
        "firstName": "Alex",
        "lastName":  "Rivera",
        "email":     "alex.rivera@example.com",
        "phone":     "+1-555-0100"
    },
    "desiredProduct": {
        "assetClass": "VEHICLE",
        "make":  "Example",
        "model": "Crossover",
        "year":  2026
    },
```

```
    "source": {
        "channel": "WEB",
        "originatingSystem": "portal-west"
    }
}

message = client.create_message(lead_data, "LEAD", "portal-west")
print(client.serialize_json(message, prettify=True))
```

JavaScript

```
const message = client.createMessage(
    {
        leadId: "LEAD-2026-001234",
        status: "EXPRESSED_INTEREST",
        customer: { firstName: "Alex", lastName: "Rivera", email: "alex@example.com" },
        desiredProduct: { assetClass: "VEHICLE", make: "Example", model: "Crossover", year: 2026 },
        source: { channel: "WEB", originatingSystem: "portal-west" }
    },
    "LEAD",
    "portal-west"
);

const json = client.serializeJson(message, true);
// POST json to your partner's endpoint
```

4.4 Update a Lead's Status

As the buyer moves through the sales process, send updated LEAD messages with the new status. Each update carries the same `leadId` so all parties can correlate the history.

```
# Buyer booked a test drive
update_data = {
    "leadId": "LEAD-2026-001234",
    "status": "APPOINTMENT_REQUEST",
    "appointmentDetails": {
        "requestedDate": "2026-04-05",
        "requestedTime": "10:00"
    }
    # customer and desiredProduct carried forward unchanged
}

update_message = client.create_message(update_data, "LEAD", "dms-dealer-42")
```

4.5 Close a Lead

When a lead reaches its final state, send a LEAD_CLOSURE. This completes the loop for all upstream systems — portal, aggregator, manufacturer — that originated or forwarded the lead.

```
closure_data = {
   "leadId":      "LEAD-2026-001234",
   "closureType": "SALE_COMPLETED",
   "closedAt":    "2026-04-08T14:30:00Z",
   "notes":       "Vehicle delivered. Customer satisfied."
}

closure = client.create_message(closure_data, "LEAD_CLOSURE", "dms-dealer-42")
```

See the complete closure type reference in **section 10.3**.

5. The Lead Lifecycle

Understanding the lifecycle is the most important part of any LEX integration. It determines what status values you send, how your counterparties react, and when a lead is considered done.

5.1 The Fifteen Stages

A lead moves through defined stages. Each stage represents a real milestone in the buyer's journey that downstream systems — DMS, CRM, manufacturer platform — can act on.

Stage	What it means for your business
`CART`	Buyer has started a configurator or added a product to a cart. Earliest trackable intent signal.
`SHOPPING`	Buyer is actively browsing. Useful for analytics; no action required yet.
`EXPLORING`	Buyer has identified a product and is actively comparing options. Assign to a sales rep.
`TEST_DRIVE_REQUESTED`	Buyer has submitted a test drive booking. Schedule and confirm.
`TEST_DRIVE_COMPLETED`	Test drive has taken place. Follow up for feedback and next steps.
`TRADE_IN`	Buyer has submitted a trade-in for valuation. Trigger an appraisal workflow.
`EXPRESSED_INTEREST`	Formal enquiry. This is the classic "lead" — full contact details are expected here.
`RESERVATION`	Buyer has placed a provisional hold. Reserve the specific unit in inventory.
`APPOINTMENT_REQUEST`	Buyer wants to visit or speak to someone. Schedule the appointment.
`IN_NEGOTIATION`	Active deal in progress. Assign to a senior closer or finance team.

`ORDER`	Purchase order placed. Trigger fulfilment, handover, and finance workflows.
`ORDER_CONFIRMED`	Order accepted and confirmed by the selling party.
`IN_DELIVERY`	Asset is in transit to the customer.
`DELIVERED`	Asset delivered to the customer. Post-delivery follow-up may apply.
`ARCHIVED`	Lead closed. A LEAD_CLOSURE message records the outcome.

5.2 Which Stages Do You Need?

You do not need to use all fifteen stages. The minimum for an L1 integration is:

- Start leads at `EXPRESSED_INTEREST` (or wherever you first have contact details); use `CART` if you capture intent earlier
- Move to `ORDER` then `ORDER_CONFIRMED` → `IN_DELIVERY` → `DELIVERED` for a completed sale
- Send `LEAD_CLOSURE` with the final outcome

The intermediate stages are available when your business process supports them and you want to give counterparties richer signal. An OEM platform that receives `TEST_DRIVE_COMPLETED` or `APPOINTMENT_REQUEST` knows the dealer is engaged; one that only receives `EXPRESSED_INTEREST` followed by `LEAD_CLOSURE: SALE_COMPLETED` learns less.

5.3 Stage Progression Rules

Leads move forward, not backward. Valid progressions:

```
CART                 ->  SHOPPING, EXPLORING, EXPRESSED_INTEREST, ARCHIVED
SHOPPING             ->  EXPLORING, TEST_DRIVE_REQUESTED, EXPRESSED_INTEREST, ARCHIVED
EXPLORING            ->  TEST_DRIVE_REQUESTED, TRADE_IN, EXPRESSED_INTEREST, RESERVATION, ARCHIVED
TEST_DRIVE_REQUESTED ->  TEST_DRIVE_COMPLETED, ARCHIVED
TEST_DRIVE_COMPLETED ->  EXPRESSED_INTEREST, RESERVATION, APPOINTMENT_REQUEST, IN_NEGOTIATION, ARCHIVED
TRADE_IN             ->  EXPRESSED_INTEREST, RESERVATION, IN_NEGOTIATION, ARCHIVED
EXPRESSED_INTEREST   ->  RESERVATION, APPOINTMENT_REQUEST, IN_NEGOTIATION, ARCHIVED
RESERVATION          ->  APPOINTMENT_REQUEST, IN_NEGOTIATION, ORDER, ARCHIVED
APPOINTMENT_REQUEST  ->  IN_NEGOTIATION, ARCHIVED
IN_NEGOTIATION       ->  ORDER, ARCHIVED
```

```
ORDER                ->  ORDER_CONFIRMED, ARCHIVED  (cancellation only)
ORDER_CONFIRMED      ->  IN_DELIVERY, ARCHIVED
IN_DELIVERY          ->  DELIVERED
DELIVERED            ->  ARCHIVED
ARCHIVED             ->  (terminal)
```

If your system receives a transition that is not in this list, reject it with an ACKNOWLEDGMENT and include the reason `INVALID_STATUS_TRANSITION`. Sending back a clear error — rather than silently accepting bad data — is what makes the network reliable.

5.4 The Ownership Chain

Every system that handles a lead appends to the `ownershipChain` array:

```
"ownershipChain": [
 { "systemId": "portal-west",   "receivedAt": "2026-04-01T10:00:00Z", "action": "CREATED"  },
 { "systemId": "dms-dealer-42", "receivedAt": "2026-04-01T10:00:05Z", "action": "RECEIVED" },
 { "systemId": "oem-platform",  "receivedAt": "2026-04-01T10:00:09Z", "action": "ROUTED"   }
]
```

You get a full audit trail of every hand-off, including timestamps, without building any centralised tracking service. If a lead goes missing or arrives with corrupted data, the ownership chain tells you exactly which system was last responsible for it.

5.5 Duplicate Detection

If your system receives a lead that is already in your CRM — same buyer, same product, different source — send a LEAD_CLOSURE with `closureType: DUPLICATE` rather than creating a second record. Include the `leadId` of the original in the closure message.

The upstream system that sent the duplicate receives the closure, knows what happened, and does not chase you for an update that will never come.

6. What a LEX Message Looks Like

You do not need to read the full specification to build your first integration. This chapter shows you the minimum message structure and explains each field in plain English.

6.1 The Minimum Viable Lead

Six fields are required for an L1-conformant LEAD message:

```
{
 "lex": {
   "header": {
     "messageId":   "550e8400-e29b-41d4-a716-446655440000",
     "messageType": "LEAD",
     "lexVersion":  "1.0.0",
     "timestamp":   "2026-04-01T10:00:00Z",
     "senderId":    "my-portal",
     "receiverId":  "dealer-dms-42"
   },
   "payload": {
     "lead": {
       "leadId": "LEAD-2026-001234",
       "status": "EXPRESSED_INTEREST",
       "customer": {
         "firstName": "Alex",
         "lastName":  "Rivera",
         "email":     "alex.rivera@example.com",
         "phone":     "+1-555-0100"
       },
       "desiredProduct": {
         "assetClass": "VEHICLE",
         "make":  "Example",
         "model": "Crossover",
         "year":  2026
       },
       "source": {
         "channel":           "WEB",
         "originatingSystem": "my-portal"
       }
     }
   }
 }
}
```

6.2 What Each Field Does

Header

Field	Plain English
`messageId`	A unique ID for this specific message. Use a UUID (any UUID generator works).
`messageType`	Always `"LEAD"` for a lead message.
`lexVersion`	The version of the LEX spec you used. Start with `"1.0.0"`.
`timestamp`	When you created this message. UTC, ISO 8601.
`senderId`	A stable identifier for your system. You choose this; tell your partner what to expect.
`receiverId`	The identifier of the system you are sending to. Your partner tells you this.

Lead payload

Field	Plain English
`leadId`	A stable ID for this lead. Stays the same across all status updates.
`status`	Where the buyer is in the sales process. See Chapter 5 for the full list.
`customer`	Buyer contact details. `email` or `phone` — at least one is required.
`desiredProduct`	What the buyer wants. `assetClass` tells the receiver what type of product.
`source`	Where this lead came from. Helps the receiver route it correctly.

6.3 Asset Classes

The `assetClass` field tells the receiver what industry vertical the lead belongs to:

Value	Industry
`VEHICLE`	Automotive (cars, trucks, motorcycles)
`HEAVY_EQUIPMENT`	Construction, mining, agriculture machinery
`MARITIME`	Vessels, yachts, commercial shipping
`AVIATION`	Aircraft, rotary, UAV
`REAL_ESTATE`	Residential, commercial, industrial property
`TECHNOLOGY`	Software, hardware, enterprise services

6.4 Adding an Acknowledgment

When you receive a lead, send an ACKNOWLEDGMENT back. This tells the sender their message arrived and passed validation:

```
{
 "lex": {
   "header": {
     "messageId":     "660e9500-f30c-52e5-b827-557766551111",
     "messageType":   "ACKNOWLEDGMENT",
     "lexVersion":    "1.0.0",
     "timestamp":     "2026-04-01T10:00:05Z",
     "senderId":      "dealer-dms-42",
     "receiverId":    "my-portal",
     "correlationId": "550e8400-e29b-41d4-a716-446655440000"
   },
   "payload": {
     "acknowledgment": {
       "status":        "ACCEPTED",
       "referenceId":   "LEAD-2026-001234",
       "processedAt":   "2026-04-01T10:00:05Z"
     }
   }
 }
}
```

The `correlationId` in the header links this acknowledgment to the original lead's `messageId`. The sender can confirm receipt and stop any retry logic.

6.5 A Closure Message

When the lead ends, send a LEAD_CLOSURE:

```
{
 "lex": {
   "header": {
     "messageId":     "770fa600-g41d-63f6-c938-668877662222",
     "messageType":   "LEAD_CLOSURE",
     "lexVersion":    "1.0.0",
     "timestamp":     "2026-04-08T14:30:00Z",
     "senderId":      "dealer-dms-42",
     "receiverId":    "my-portal",
     "correlationId": "LEAD-2026-001234"
   },
   "payload": {
     "leadClosure": {
       "leadId":      "LEAD-2026-001234",
       "closureType": "SALE_COMPLETED",
       "closedAt":    "2026-04-08T14:30:00Z"
     }
   }
 }
}
```

- An L1 implementation can be built in a day
- Core fields never change (no forced migration)
- Every participant can understand any message without industry-specific knowledge

Business requirements that go beyond the core are handled through the extension mechanism.

6.2 Extension Namespaces

An extension namespace is a dot-delimited prefix that uniquely identifies the organisation and purpose of a set of custom fields:

```
{organisation-id}.{division}.{purpose}
```

Examples:

Namespace	Owner	Purpose
`lex.ev`	LEX working group	Electric vehicle specifications
`lex.captive`	LEX working group	Captive finance products

`acme.fleet`	ACME dealer group	Fleet procurement fields
`portx.maritime`	PortX platform	Maritime vessel procurement

Namespaces prefixed with `lex.*` are reserved for the LEX working group. All other namespaces are open and self-assigned.

6.3 Registering a Namespace

Namespace registration is submitted through the online registry at https://lexstandard.org/registry:

```
{
 "namespace": "acme.fleet",
 "owner": "ACME Dealer Group",
 "contact": "techstandards@acme.example",
 "description": "Fleet procurement extension fields",
 "registeredAt": "2026-03-01",
 "specUrl": "https://acme.example/lex-fleet-extension"
}
```

Registration is approved if: (a) the namespace follows the format; (b) it does not conflict with an existing registration; (c) it does not impersonate another organisation. No fee, no editorial review of the extension content itself.

6.4 Using Extensions in Messages

Extensions are placed in the `extensions` block of the payload, keyed by namespace:

```
"payload": {
 "lead": { ... },
 "extensions": {
   "acme.fleet": {
     "fleetSize": 50,
     "preferredDeliveryPort": "PORT-FELIXSTOWE",
     "tenderReference": "TENDER-2026-0042"
   },
   "lex.ev": {
     "batteryCapacityKwh": 77,
     "chargingStandardPreference": "CCS2"
   }
 }
}
```

6.5 Passthrough Obligation

A participant that does not recognise an extension namespace MUST:

- Preserve all unrecognised extension blocks when forwarding the message
- NOT reject the message because of an unrecognised extension
- NOT modify or truncate extension block contents

A participant that recognises an extension MUST validate it against the extension's own schema if one is published.

6.6 Standard Extensions

The LEX working group maintains a set of standard extensions that are part of the LEX specification but are not part of the L1 conformance requirement:

Extension	Category	Description
`lex.ev`	Automotive	Electric vehicle battery capacity, charging standard, range, and subscription data
`lex.captive`	Finance	Manufacturer captive finance offer — rate, term, balloon payment, and residual value
`lex.financial`	Finance	Trade-in valuation, deal pricing, incentives, taxes, and net deal value
`lex.lineage`	Audit	Historical deal terms, negotiation audit trail, and attribution chain
`lex.governance`	Compliance	Data residency, retention period, and processing jurisdiction rules

`lex.consent`	Compliance	GDPR, UK GDPR, CCPA, India DPDPA, China PIPL, Brazil LGPD, South Africa POPIA, Canada PIPEDA, Singapore PDPA, Japan APPI, Switzerland nFADP, South Korea PIPA-K, Saudi Arabia PDPL, Thailand PDPA, and US state privacy law consent evidence and buyer preferences
`lex.intelligence`	Analytics	Lead scoring, intent signals, behavioural attributes, and predictive data

Full extension field definitions are published at https://lexstandard.org/spec.

7. Extending LEX for Your Industry

The LEX core is intentionally small. Industry-specific fields are added as named extension blocks rather than growing the core. This chapter shows you how.

7.1 The Extension Principle

Core fields cover what every industry needs. Extension blocks add what your specific business requires. Any receiver that does not understand your extension must pass it through unchanged — it cannot reject your message because of it.

This means you can start using extensions immediately without waiting for every counterparty to upgrade.

7.2 Registering Your Namespace

Before sending extension data, register a namespace at https://lexstandard.org/registry:

```
{
 "namespace": "acme",
 "owner":     "ACME Corporation",
 "contact":   "lex-admin@acme.example.com",
 "industries": ["HEAVY_EQUIPMENT", "MARITIME"],
 "description": "ACME fleet and procurement extension fields"
}
```

Registration is free. It namespaces your fields so they never conflict with another organisation's extensions.

7.3 Adding Extension Fields

Extension data is added in the `extensions` block at the payload level:

```
"extensions": {
 "acme": {
   "fleet": {
     "preferredDeliveryPort": "PORT-HOUSTON-TX",
     "maintenanceContractType": "FULL_SERVICE"
   }
 }
}
```

In code (Python):

```
lead_data = {
   "leadId": "LEAD-2026-001234",
   "status": "EXPRESSED_INTEREST",
   # ... core fields ...
   "extensions": {
       "acme": {
           "fleet": {
               "preferredDeliveryPort":     "PORT-HOUSTON-TX",
               "maintenanceContractType":   "FULL_SERVICE",
               "fleetSize":                 12
           }
       }
   }
}
```

7.4 Industry Extension Blocks

Full extension field definitions for each supported industry are in Chapter 6 of this document and published at https://lexstandard.org/spec.

Industry	Extension namespace	Key extension fields
Automotive — EV	`lex.ev`	Battery capacity, charging standard, range requirements, MSRP
Automotive — Finance	`lex.financial`	Finance type, LTV ratio, trade-in valuation, monthly payment target
Captive Finance	`lex.captive`	Manufacturer finance offer, interest rate, balloon payment, residual
Heavy Equipment	`lex.financial` + custom	Fleet size, tender reference, delivery port, maintenance contract type
Maritime	custom	Vessel type, gross tonnage, flag state, charter type, delivery port

Aviation	`lex.financial` + custom	Aircraft type, seating capacity, range (nm), ECA financing flag
Real Estate	custom	Property type, tenure (freehold/leasehold), floor area (sqm), zone
Technology	`lex.financial` + custom	Product category, deployment model, ACV, company size band

7.5 Consuming Extensions You Did Not Write

When you receive a message with an extension block your system does not recognise:

1. **Do not reject** the message.
2. **Do not modify** the extension block.
3. If forwarding the message, **preserve** the extension block unchanged.
4. Optionally log unknown namespaces for monitoring.

```
message = client.parse_json(raw_json)

# Core fields you process
lead = message['lex']['payload']['lead']

# Unknown extensions — store and pass through
extensions = message['lex']['payload'].get('extensions', {})

# When forwarding
outbound['lex']['payload']['extensions'] = extensions
```

7.6 Transport and Delivery

LEX does not require a specific transport mechanism. Common patterns:

Pattern	When to use
HTTPS POST (REST)	Most integrations. Simple, debuggable, firewall-friendly.

Message queue (AMQP, SQS)	High-volume feeds, async delivery guarantees.
SFTP batch	Legacy DMS, batch-mode EDI workflows.
Webhook fan-out	Distributing one lead to multiple dealer endpoints.

For burst handling, add a `ttl` block to the message header specifying how long the message should be considered valid. Receivers that cannot process within the TTL should return a `REJECTED` acknowledgment with reason `MESSAGE_EXPIRED`.

Transport	Suitable for
HTTPS (REST)	Real-time point-to-point integration
AMQP / RabbitMQ	Internal enterprise message bus
Apache Kafka	High-throughput streaming integrations
AS2	EDI VAN connections (X12, EDIFACT)
SFTP batch	Scheduled batch lead file delivery

The LEX message envelope is the same regardless of transport. A message sent over Kafka and a message sent over HTTPS REST carry the same `header` + `payload` structure.

7.2 Message TTL

Every LEX message may carry a `ttl` block in the header that specifies how long it should be held for delivery before being considered expired.

```
"ttl": {
 "maxAgeSeconds": 3600,
 "expiresAt": "2026-03-01T11:00:00Z",
 "onExpiry": "DLQ"
}
```

Default TTL values by message type:

Message Type	Default TTL	Rationale

`LEAD`	3600 s (1 h)	Stale leads damage buyer experience
`LEAD_CLOSURE`	86400 s (24 h)	Closure data is less time-critical
`ACKNOWLEDGMENT`	300 s (5 min)	Real-time signal
`SUBSCRIPTION`	2592000 s (30 d)	Long-lived registration
`ASSET`	43200 s (12 h)	Product data has a natural refresh cadence

7.3 Retry Strategy

Transient failures (network timeout, HTTP 5xx) should be retried with exponential backoff. The `retryContext` header block signals to the receiver that a message is a retry attempt:

```
"retryContext": {
 "attemptNumber": 2,
 "firstAttemptAt": "2026-03-01T10:00:00Z",
 "lastAttemptAt":  "2026-03-01T10:05:00Z",
 "retryReason": "HTTP_503"
}
```

Recommended backoff schedule:

Attempt	Delay
1 (initial)	Immediate
2	30 s
3	2 min
4	10 min
5	30 min
6+	Dead-letter queue

7.4 Dead-Letter Queue

A message that cannot be delivered after exhausting the retry schedule MUST be moved to a dead-letter queue (DLQ) rather than silently discarded. The DLQ entry must include:

- The original message verbatim
- The final delivery error
- All retry attempt timestamps
- The `ttl.onExpiry` value from the original message

DLQ contents must be inspectable by operations staff and must not be purged automatically within the message's `maxAgeSeconds` window.

7.5 Failure Categories

Category	Recovery
Transient failure (network, 5xx)	Retry with backoff
Permanent failure (parse error, invalid schema)	Route to DLQ immediately — no retry
Business rejection (validation fail, unknown receiver)	Send `ACKNOWLEDGMENT` with `REJECTED`
TTL expiry	Route to DLQ with `reason: TTL_EXPIRED`
Duplicate detected	Send `LEAD_CLOSURE` with `DUPLICATE` — do not discard original

7.6 Idempotency

All message receivers MUST be idempotent with respect to `messageId`. If a message with a previously seen `messageId` is received, the receiver MUST return the same ACKNOWLEDGMENT it returned the first time and MUST NOT create a duplicate record.

This allows senders to retry any message without risk of data duplication.

8. Conformance — What Level Do You Need?

LEX defines three conformance levels. You choose the level that matches what your business does, then self-declare. There is no external certification body. The published test vectors let you verify your own implementation.

8.1 The Three Levels

Level	Who it is for	What you implement
L1 — Basic	Portals, simple lead senders and receivers	Core message structure, LEAD + ACKNOWLEDGMENT, JSON-EDI only
L2 — Standard	DMS vendors, CRM platforms, aggregators	All L1 + lifecycle transitions, LEAD_CLOSURE, deduplication
L3 — Full	OEM platforms, enterprise lead networks	All L2 + ASSET, SUBSCRIPTION, extensions, multi-format

8.2 Which Level Is Right for You?

You are a property portal that sends leads to estate agencies. You create leads and want an acknowledgment back. Aim for **L1**.

You are a DMS vendor receiving leads from multiple portals. You need to validate, route, deduplicate, and close leads. Aim for **L2**.

You are an OEM platform routing leads to hundreds of dealers and receiving outcome data. You need subscriptions, asset messaging, full lifecycle tracking, and multi-format support. Aim for **L3**.

You are connecting two internal systems. Start with L1. Add what you need as your integration matures.

8.3 L1 Checklist

- [ ] Accept and validate inbound LEAD messages (JSON-EDI)

- [ ] Send ACKNOWLEDGMENT on receipt (ACCEPTED or REJECTED)
- [ ] Create outbound LEAD messages with at least: `leadId`, `status`, `customer`, `desiredProduct`, `source`
- [ ] Reject unknown status values gracefully
- [ ] Set `senderId` and `receiverId` in every message header

8.4 L2 Additions

- [ ] Enforce status transition rules (see Chapter 5.3)
- [ ] Send LEAD_CLOSURE when a lead is terminated
- [ ] Detect and reject duplicate leads using `customerFingerprint`
- [ ] Maintain the `ownershipChain` array
- [ ] Pass through unknown extension blocks unchanged

8.5 L3 Additions

- [ ] Send and receive ASSET messages
- [ ] Implement SUBSCRIPTION filtering
- [ ] Support at least one non-JSON format (X12, EDIFACT, or XML-EDI)
- [ ] Implement the `consentRecord` block for GDPR/CCPA compliance
- [ ] Handle message TTL and retry logic

8.6 Security Practices

LEX messages carry buyer personal data. Apply standard data-handling rules:

- Transport messages over HTTPS/TLS 1.2+ or equivalent encrypted channel.
- Validate `senderId` against an allow-list on receipt. Do not process messages from unknown senders.
- Store only the personal data fields you need to process the lead. The `consentRecord` block records the buyer's data processing consent — preserve it.

- For GDPR / CCPA: the buyer's right to erasure applies to the `customer` block. If you store LEX messages and receive a deletion request, clear `customer` fields while preserving the lead record skeleton for audit purposes.

8.7 Declaring Conformance

Once your implementation passes your own testing against the published test vectors, update your capability advertisement:

```
{
 "systemId":   "dealer-dms-42",
 "lexVersion": "1.0.0",
 "conformance": "L2",
 "formats":    ["JSON-EDI", "XML-EDI"],
 "messageTypes": ["LEAD", "ACKNOWLEDGMENT", "LEAD_CLOSURE"]
}
```

Share this with your integration partners so they know the capabilities of your endpoint.

environment, and the trust relationship between participants.

The security model covers four concerns:

1. **Transit security** — protecting messages in motion
2. **Message integrity** — detecting tampering
3. **Consent** — evidence that the buyer agreed to data sharing
4. **Data governance** — rules about where data can go and how long it can be kept

8.2 Transit Security

Minimum requirement (L1): TLS 1.2 or higher on all connections.
Recommended: TLS 1.3 with certificate pinning for production integrations.

For batch file transfer (SFTP, AS2), files must be encrypted at rest using AES-256 or equivalent before transmission.

8.3 Message Integrity

The optional `securityMetadata` header block allows a sender to sign a message and a receiver to verify it has not been tampered with in transit:

```
"securityMetadata": {
 "encryptionAlgorithm": "AES-256-GCM",
 "signatureAlgorithm": "RS256",
 "signature": "<base64-encoded-signature>",
 "keyId": "sender-key-2026-03",
 "encryptedFields": ["customer.email", "customer.phone"]
}
```

Field-level encryption (`encryptedFields`) allows PII to be encrypted while the envelope (routing fields, messageId, status) remains readable by intermediaries.

8.4 Consent Model

Any LEX message that carries personally identifiable information (PII) should include a `consentRecord` in the extensions block:

```
"extensions": {
 "lex.consent": {
   "consentId": "CONSENT-2026-0042",
   "consentedAt": "2026-02-28T09:15:00Z",
   "consentExpiresAt": "2027-02-28T09:15:00Z",
   "purposes": ["LEAD_PROCESSING", "MARKETING"],
   "jurisdiction": "EU",
   "withdrawalUrl": "https://portal.example/consent/withdraw/CONSENT-2026-0042"
 }
}
```

L3 conformance requires `consentRecord` on all PII-bearing messages. L1 and L2 implementations are encouraged to include it but are not required to.

8.5 Data Governance

The `lex.governance` extension allows the sender to specify constraints on how data may be processed by downstream participants:

Field	Description
`dataResidency`	Country or region where data must remain (e.g. `EU`, `US-CA`)
`retentionDays`	Maximum period receiver may hold the record
`allowRemarketing`	Whether PII may be used for marketing purposes

`allowThirdPartySharing`	Whether data may be forwarded to unlisted parties
`anonymiseAfterDays`	Receiver must anonymise PII after this period

Receivers that cannot honour a `dataResidency` constraint MUST reject the message with `ACKNOWLEDGMENT status: REJECTED` and `reason: DATA_RESIDENCY_VIOLATION` rather than silently accepting data they cannot lawfully process.

8.6 Audit Trail

The `ownershipChain` field (see Section 4.4) serves as the primary audit trail. For regulated industries or high-value transactions, the `dealLineage` extension (`lex.lineage`) provides a structured record of every change to deal terms, with timestamps and actor identifiers.

8.7 Regulatory Mapping

Regulation	LEX Coverage
GDPR (EU)	`lex.consent`, `lex.governance` data residency and retention fields
CCPA (California)	`lex.consent` opt-out and purpose fields
Consumer Duty (UK)	`ownershipChain` for accountability; closure reason codes
FCA (UK finance)	`lex.captive` and `lex.financial` audit fields

LEX provides the data structures. Compliance with a specific regulation requires that all participants in a message chain correctly populate and honour the relevant fields — the standard cannot enforce compliance, only make it possible.

9. Governance and License

9.1 License

LEX is published under the **Apache License, Version 2.0** (`SPDX-License-Identifier: Apache-2.0`).

You can use it commercially. You can embed it in products. You can build proprietary services on top of it. You do not need to pay anyone. You do not need to ask permission. Required conditions: preserve the copyright notice in any distribution; if you contribute and then sue anyone for patent infringement relating to your contribution, your license terminates.

Apache 2.0 was chosen over MIT because it includes a **patent retaliation clause** — the protection that matters for a technical standard with an open contribution chain.

9.2 Why Open?

Data standards only work if all participants are willing to adopt them. A standard that charges licensing fees, grants special rights to founding members, or can be withheld from competitors will never achieve the network coverage needed to be useful.

LEX is designed to be adopted by a fragmented ecosystem. For that to happen, every nervous legal team at every potential adopter needs to be able to confirm: no cost, no lock-in, no control risk.

9.3 No Standards Body Required

You do not join a body to use LEX. You do not pay membership fees. You do not attend working groups unless you want to influence the next version.

If you want to propose a change or a new extension, the process is:

1. Open an issue in the LEX repository describing the need.
2. Propose a solutions in a pull request, referencing the issue.
3. Changes go through a public review period before merging.

All changes are public before they are accepted.

9.4 Anti-Capture

The specification explicitly protects against capture by a single dominant actor:

- No contributor earns special rights over existing content.
- Extension namespaces are registered, not granted — every sender has equal access.
- The registry files (`LEX_EXTENSION_REGISTRY.json`, `LEX_ORGANIZATION_REGISTRY.json`) are part of the repository, not a private service operated by any single entity.

9.5 Versioning Policy

LEX follows semantic versioning. The promise to adopters:

Change	What we guarantee
New optional fields (MINOR)	Your existing code continues to work without changes.
New required fields (MAJOR)	Migration guide published. Deprecation period of 12 months minimum before old version is non-normative.
Breaking removal (MAJOR)	Same as above. No silent breaking changes.

9.6 Reporting Issues

Problems with the specification — ambiguities, contradictions, missing cases — are reported through the repository issue tracker. This creates a public record of known issues so every adopter benefits from the same corrections.

9.2 Level 1 — Basic

The minimum for any production system that sends or receives LEX messages.

Requirement	Description
Valid envelope	Can produce and parse `lex.header` + `lex.payload`
Required fields	All required fields present in every message
MessageID uniqueness	No duplicate `messageId` values within a session

Timestamp validity	Timestamps within ±60 s of wall clock
ACKNOWLEDGMENT response	Sends ACK for every received LEAD message
Error response	Returns structured error for malformed input
Idempotency	Duplicate `messageId` returns same ACK, no duplicate record
TLS 1.2+	All connections encrypted

Target implementers: Any DMS, CRM, portal, or aggregator that wants to send or receive LEX leads.

9.3 Level 2 — Standard

Required for DMS-to-manufacturer and DMS-to-aggregator integrations.

All L1 requirements, plus:

Requirement	Description
Full lifecycle	Can send/receive all 15 lead status values
Transition validation	Rejects invalid status transitions with correct error code
Business rule validation	Email, phone, VIN, financing field rules — see Section 9.2 above
Lead Closure	Can send/receive `LEAD_CLOSURE` messages
Bidirectional flow	Can both initiate and respond to lead messages
Deduplication	Sends `customerFingerprint`; handles `DUPLICATE` closure
TTL handling	Honours `ttl.expiresAt`; routes expired messages to DLQ

Target implementers: DMS vendors, manufacturer platforms, large dealer groups.

9.4 Level 3 — Full

Required for OEM platform certification and multi-industry deployments.

All L2 requirements, plus:

Requirement	Description
Consent records	`lex.consent` present and valid on all PII-bearing messages
EV extensions	Can parse and produce `lex.ev` block
Lead intelligence passthrough	Preserves `lex.intelligence` unmodified when forwarding
Retry context headers	Populates `retryContext` on retried messages
DLQ routing	Routes permanently failed messages to DLQ; DLQ is inspectable
Multi-format support	Can parse both JSON-EDI and XML-EDI
Subscription management	Full `SUBSCRIPTION` message send/receive
Organisation context	`organizationContext` correctly populated

Target implementers: OEM platforms, national aggregators, super-DMS providers.

9.5 Test Vectors

The full test case library is maintained at lexstandard.org/conformance. Each test case specifies:

- Input message (or input state)
- Expected output / response
- The conformance level it tests
- Whether it is a MUST (failure = non-conformant) or SHOULD (failure = advisory)

A conformance test runner is included in the reference implementation (`src/LEX_VALIDATION_ENGINE.ts`) that accepts a message and returns a conformance report against any of the three levels.

9.6 Capability Advertisement

An L3-conformant system can publish a capability advertisement message — a structured declaration of which optional extensions and features it supports. Counterparties use this to determine which fields they can safely include in messages sent to that system.

See the online specification at lexstandard.org/spec.

10. Quick Reference

10.1 Library Methods by Language

Methods follow each language's idiomatic casing. JavaScript and Java share identical method names.

Convention	Languages
`snake_case`	Python
`camelCase`	JavaScript, Java
`PascalCase`	C# (.NET)

Operation	Python	JavaScript / Java	C# (.NET)
Create client	`LexClient()`	`new LexClient()`	`new LexClient()`
Parse JSON	`parse_json(s)`	`parseJson(s)`	`ParseJson(s)`
Build message	`create_message(data, type, sender)`	`createMessage(data, type, sender)`	`CreateMessage(data, type, sender)`
Validate	`validate(msg)`	`validate(msg)`	`Validate(msg)`
Serialise to JSON	`serialize_json(msg, prettify)`	`serializeJson(msg, pretty)`	`SerializeJson(msg, pretty)`
Save to file	`save_to_file(msg, path, fmt)`	—	—

10.2 Lead Status Codes

Code	Stage in the buyer journey
`CART`	Configurator started or product added to cart
`SHOPPING`	Browsing; no explicit product selected

`EXPLORING`	Product identified; actively comparing
`TEST_DRIVE_REQUESTED`	Test drive booking submitted
`TEST_DRIVE_COMPLETED`	Test drive has taken place
`TRADE_IN`	Trade-in asset submitted for valuation
`EXPRESSED_INTEREST`	Formal enquiry — full contact details present
`RESERVATION`	Provisional hold on a specific unit
`APPOINTMENT_REQUEST`	Buyer requested a visit or call
`IN_NEGOTIATION`	Active deal in progress
`ORDER`	Purchase order placed
`ORDER_CONFIRMED`	Order accepted and confirmed
`IN_DELIVERY`	Asset in transit to customer
`DELIVERED`	Asset delivered to customer
`ARCHIVED`	Lead closed (see closure types below)

10.3 Closure Types

Code	Meaning
`SALE_COMPLETED`	Successfully sold
`LOST_TO_COMPETITOR`	Buyer purchased elsewhere
`UNCONTACTABLE`	No response after multiple attempts
`BUDGET_INSUFFICIENT`	Financing or budget did not qualify
`PRODUCT_UNAVAILABLE`	Product could not be sourced
`DUPLICATE`	Already processed under another lead ID

`BUYER_WITHDREW`	Buyer explicitly cancelled
`EXPIRED`	Exceeded maximum age without activity
`OTHER`	None of the above; `notes` field required

10.4 Acknowledgment Statuses

Code	Meaning
`ACCEPTED`	Message received and valid
`REJECTED`	Message received but validation failed
`DEFERRED`	Message received; processing will happen asynchronously

10.5 Asset Classes

`VEHICLE`, `HEAVY_EQUIPMENT`, `MARITIME`, `AVIATION`, `REAL_ESTATE`, `TECHNOLOGY`

10.6 Required Header Fields

`messageId`, `messageType`, `lexVersion`, `timestamp`, `senderId`, `receiverId`

10.7 Required Lead Fields

`leadId`, `status`, `customer` (at least email or phone), `desiredProduct` (at least `assetClass`), `source`

10.8 Rejection Reason Codes

Code	Triggered when
`MISSING_REQUIRED_FIELD`	A required field is absent
`INVALID_STATUS_TRANSITION`	Status jump is not in the allowed transitions table
`UNKNOWN_SENDER`	`senderId` not in the receiver's allow-list

`MESSAGE_EXPIRED`	Message received after its `ttl` deadline
`DUPLICATE_MESSAGE`	`messageId` already processed
`SCHEMA_VALIDATION_FAILED`	Message does not match the JSON Schema

10.9 Wire Format MIME Types

Format	Content-Type
JSON-EDI	`application/lex+json`
XML-EDI	`application/lex+xml`
X12	`application/edi-x12`
EDIFACT	`application/edifact`

10.10 Reference Implementation

The reference implementation is the authoritative source for resolving ambiguities in validation behaviour. It is authored in Haxe and compiled to four target platforms:

Platform	Artifact	Download
JavaScript / Node.js	`lex.js`	https://lexstandard.org/libraries
Python 3	`lex.py`	https://lexstandard.org/libraries
Java (JVM 11+)	`Lex.jar`	https://lexstandard.org/libraries
C# (.NET 6+)	`Lex.dll`	https://lexstandard.org/libraries

All four are compiled from the same Haxe source, ensuring identical behaviour across platforms.

10.11 What the Reference Implementation Provides

Parsers

Four format parsers — JSON-EDI, XML-EDI, X12, EDIFACT — that produce the canonical in-memory LEX data model from any of the four wire formats.

Validators

A validation engine (`ValidationEngine.hx`) that checks:

- Schema conformance: required fields, type constraints, enum values
- Business rules: valid status transitions, email/phone format, VIN format
- Lifecycle rules: ownership chain continuity, closure type appropriateness
- Extension schemas: validation of known `lex.*` extensions

Data Models

Strongly-typed data models for all five message types and all extension blocks, available in all four target languages.

Registries

- `OrganizationRegistry.hx` — organisation type definitions and extension namespaces
- `ProductRegistry.hx` — asset classes and product type seed data
- `LexSecurity.hx` — security metadata construction and verification helpers

10.12 Building the Libraries

Prerequisites: Haxe 4.3+, Java 11+ (for Java target), .NET 6+ (for C# target)

```
# Interactive build menu (Windows)
lex.bat

# Build all targets non-interactively
lex.bat -Build all

# Build a specific target
lex.bat -Build js
lex.bat -Build python
lex.bat -Build java
lex.bat -Build csharp

# Package compiled artifacts into libraries/
lex.bat -Package
```

10.13 Language Wrapper Usage

Each compiled library ships with a thin idiomatic wrapper that exposes the LEX API in the conventions of the target language.

Python

```python
from lex import LexParser, LexValidator

message = LexParser.parse_json(raw_json)
result  = LexValidator.validate(message)

if result.valid:
   print(f"Lead {message.payload.lead.leadId} — {message.payload.lead.status}")
else:
   for error in result.errors:
       print(f"  [{error.code}] {error.message}")
```

JavaScript / Node.js

```javascript
const { LexParser, LexValidator } = require('./lex');

const message = LexParser.parseJson(rawJson);
const result  = LexValidator.validate(message);

if (result.valid) {
 console.log(`Lead ${message.payload.lead.leadId} — ${message.payload.lead.status}`);
}
```

Full wrapper documentation is in `haxe/wrappers/README.md`.

10.14 Validation Engine

The TypeScript validation engine (`src/LEX_VALIDATION_ENGINE.ts`) provides conformance testing against all three conformance levels and is usable independently of the compiled Haxe libraries:

```javascript
import { LexValidationEngine } from './LEX_VALIDATION_ENGINE';

const engine = new LexValidationEngine();
const report = engine.validateConformance(message, 'L2');

console.log(`Conformance L2: ${report.conformant}`);
report.failures.forEach(f => console.log(`  FAIL: ${f.rule} — ${f.detail}`));
```

11. API Definitions

LEX publishes two machine-readable API definitions alongside the specification. They allow you to generate clients, drive API explorers, run contract tests, and integrate with event infrastructure without building anything by hand.

11.1 OpenAPI 3.1 — REST API

File: `api/lex-openapi.yaml`
URL: https://lexstandard.org/api/lex-openapi.yaml

The OpenAPI definition covers the LEX REST transport layer — the primary integration method for web-connected systems. It is the right choice if your counterparty uses HTTP, webhooks, or a REST-based API gateway.

Endpoints

Method	Path	Description
`POST`	`/messages`	Submit any LEX message (LEAD, ASSET, ACK, SUBSCRIPTION, LEAD_CLOSURE)
`GET`	`/messages/{messageId}`	Retrieve a previously submitted message
`GET`	`/messages/{messageId}/acknowledgment`	Retrieve the acknowledgment for a message
`POST`	`/messages/batch`	Submit up to 100 messages in one call
`POST`	`/leads`	LEAD-specific shorthand endpoint
`POST`	`/leads/{leadId}/closure`	Submit a LEAD_CLOSURE for a lead

`POST`	`/assets`	Submit an ASSET inventory message
`POST`	`/subscriptions`	Register a webhook subscription
`GET`	`/subscriptions`	List subscriptions for the authenticated organisation
`DELETE`	`/subscriptions/{subscriptionId}`	Cancel a subscription
`POST`	`/conformance/run`	Run the LEX conformance test suite
`GET`	`/schema/{messageType}`	Retrieve the JSON Schema for a message type
`GET`	`/health`	Health check (no auth required)

Authentication

LEX requires OAuth 2.0 Client Credentials (machine-to-machine) for all endpoints. Your platform implements the token endpoint — LEX defines the pattern but does not host an auth service. Set your token URL in the `securitySchemes.oauth2.flows.clientCredentials.tokenUrl` field of the OpenAPI file before importing it into your toolchain.

Available scopes: `lex:leads:read`, `lex:leads:write`, `lex:assets:read`, `lex:assets:write`, `lex:closures:write`, `lex:subscriptions:manage`, `lex:conformance`.

Idempotency

Every `POST` endpoint is idempotent on `messageId`. Retry the same request safely — duplicate `messageId` submissions return `200 OK` without reprocessing. Use the optional `Idempotency-Key` request header to detect duplicates at the HTTP layer before message parsing.

Using the OpenAPI File

```
# Try with Swagger UI (Docker)
docker run -p 8080:8080 \
 -e SWAGGER_JSON_URL=https://lexstandard.org/api/lex-openapi.yaml \
 swaggerapi/swagger-ui
```

```
# Import into Postman
# File > Import > Link > https://lexstandard.org/api/lex-openapi.yaml

# Generate a client (openapi-generator)
openapi-generator-cli generate \
 -i https://lexstandard.org/api/lex-openapi.yaml \
 -g python -o lex-client-python
```

11.2 AsyncAPI 3.0 — Event-Driven Messaging

File: `api/lex-asyncapi.yaml`
URL: https://lexstandard.org/api/lex-asyncapi.yaml

The AsyncAPI definition covers the LEX event-driven transport layer. Use this when your infrastructure is message-broker-based: Kafka, RabbitMQ, Azure Service Bus, or webhook fan-out.

Channels and Message Types

Channel	Direction	Message	Description
`lex/leads/inbound`	receive	`LEAD`	Leads arriving from portals, OEMs, DMS platforms
`lex/leads/outbound`	send	`LEAD`	Status updates from OEM/manufacturer back to dealer
`lex/assets/inbound`	receive	`ASSET`	Inventory updates across all supported asset classes
`lex/acknowledgments`	send	`ACKNOWLEDGMENT`	Receipt confirmation — MUST be sent for every received LEAD or ASSET
`lex/subscriptions`	send	`SUBSCRIPTION`	Register filters for which events your system wants to receive

`lex/closures`	send	`LEAD_CLOSURE`	Final deal outcome (WON, LOST, ABANDONED …) reported by the dealer
`lex/dlq`	receive	DLQ	Messages that failed all delivery retries

Supported Transport Bindings

Transport	Protocol	Best for
REST/Webhook	HTTPS POST	Web portals and SaaS platforms
AMQP	RabbitMQ / Azure Service Bus	Enterprise middleware and on-premise DMS
Kafka	Topic-per-channel	High-volume OEM platforms
SFTP	File-drop EDI	Legacy batch systems

Using the AsyncAPI File

```
# Explore with AsyncAPI Studio (browser)
# Open https://studio.asyncapi.com and paste the URL

# Generate a client
asyncapi generate fromTemplate \
 https://lexstandard.org/api/lex-asyncapi.yaml \
 @asyncapi/nodejs-template -o lex-async-client

# Mock broker for contract testing (Microcks)
# Import via Microcks > Importers > AsyncAPI URL
```

11.3 CloudEvents Alignment

File: `api/lex-cloudevents-mapping.md`

URL: https://lexstandard.org/api/lex-cloudevents-mapping

LEX message headers map to the CNCF CloudEvents v1.0.2 envelope. This means LEX events can flow through AWS EventBridge, Azure Event Grid, Google Eventarc, and Kafka Schema Registry without a translation layer.

Event type naming convention:

```
org.lexstandard.{messageType}.{action}

org.lexstandard.lead.submitted
org.lexstandard.lead.status.updated
org.lexstandard.lead.closed
org.lexstandard.asset.submitted
org.lexstandard.message.acknowledged
```

The CloudEvents mapping document includes structured content mode examples, binary mode HTTP headers, EventBridge rule patterns, Azure Event Grid filters, Kafka producer configuration, and JavaScript conversion functions.

11.4 JSON Schemas

All five LEX message types have JSON Schemas (Draft-07) in the `schemas/` directory and at `https://lexstandard.org/schemas`:

Schema file	Message type
`LEX_MESSAGE_SCHEMA.json`	Message envelope (root)
`LEX_LEAD_SCHEMA.json`	LEAD
`LEX_ASSET_SCHEMA.json`	ASSET (all 18 asset classes)
`LEX_ACKNOWLEDGMENT_SCHEMA.json`	ACKNOWLEDGMENT
`LEX_SUBSCRIPTION_SCHEMA.json`	SUBSCRIPTION
`LEX_LEAD_CLOSURE_SCHEMA.json`	LEAD_CLOSURE

The OpenAPI and AsyncAPI files reference these schemas. You can use them independently with any JSON Schema validator:

```
import jsonschema, json

schema = json.load(open("schemas/LEX_LEAD_SCHEMA.json"))
message = json.load(open("my-lead.json"))
jsonschema.validate(message["lex"]["payload"], schema)
```

12. Next Steps

12.1 Start Your Integration

Choose a library. Download the client library for your language from https://lexstandard.org/libraries into your project. No package manager required for Python or JavaScript.

Send a test lead. Use the minimal example from Chapter 6.1. Validate it with `client.validate()` before sending. If it passes, send it and check the ACKNOWLEDGMENT.

Implement closure. Wire up LEAD_CLOSURE for your won and lost outcomes. This is the data that makes the network useful for every upstream participant.

Pick your conformance level. If you are receiving leads, L1 is sufficient to start. If you are a DMS or aggregator, target L2. See Chapter 8 for the checklists.

12.2 Useful Resources

Resource	Where to find it
Online specification & field dictionary	https://lexstandard.org/spec
Conformance test suite	https://lexstandard.org/conformance
Extension registry	https://lexstandard.org/registry
Client libraries (Python, JS, Java, C#)	https://lexstandard.org/libraries
JSON Schemas (all 5 message types)	https://lexstandard.org/schemas
OpenAPI 3.1 definition (REST)	https://lexstandard.org/api/lex-openapi.yaml
AsyncAPI 3.0 definition (event-driven)	https://lexstandard.org/api/lex-asyncapi.yaml
Industry examples repository	https://github.com/lexstandard/examples

12.3 Register an Extension Namespace

If you need industry-specific fields not covered by the core or existing extensions, submit a registration request at https://lexstandard.org/registry. Registration is free and reviewed within two weeks.

12.4 Report a Problem

If you find an ambiguity, a contradiction, or a missing case in this specification, open an issue at https://lexstandard.org/issues. Every adopter benefits from corrections being public.

12.5 Industry Examples

Complete worked examples for each supported industry are published at https://github.com/lexstandard/examples:

- **Automotive** — Retail vehicle leads in JSON-EDI, XML-EDI, X12, and EDIFACT
- **Heavy Equipment** — Construction, mining, and agricultural fleet procurement
- **Maritime** — Vessel, yacht, and commercial shipping enquiries
- **Aviation** — Aircraft, rotary, and UAV acquisition flows
- **Real Estate** — Residential, commercial, and industrial property enquiries
- **Technology** — Enterprise software, hardware, and managed services channel leads

```
Apache License, Version 2.0
SPDX-License-Identifier: Apache-2.0
Copyright (c) 2026 LEX Lead Exchange Standard Contributors

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

   http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND.

CANONICAL SPECIFICATION: https://lexstandard.org
Any derivative must disclose it is not the canonical specification.
```

Any entity — DMS provider, OEM, aggregator, or individual — may implement LEX freely without payment, approval, or membership.

11.2 Open Participation

Contributions to the specification are open to all via public pull request. No financial contribution, dues payment, or membership fee is required to:

- Propose changes to the core specification
- Register extension namespaces
- Be credited as a contributor
- Build commercial products on LEX

11.3 Anti-Capture Provisions

Any entity that introduces mandatory participation fees, access restrictions, proprietary data requirements, or vendor-specific dependencies into the core LEX specification is acting in violation of LEX governance.

This clause exists because shared standards in high-value sales ecosystems have historically been captured by dominant platform vendors. Proprietary variants, integration brokerage fees, and incumbent governance control have turned open standards into revenue mechanisms. Some governance bodies have evolved into dues-paying organisations where the largest payers control the roadmap, at the expense of the broader ecosystem. LEX explicitly rejects both of these patterns.

11.4 No Mandatory Certification

Conformance levels (L1, L2, L3) are self-declared against published test vectors. Any third-party organisation may offer certification services, but no certification body is mandatory. No implementation is blocked from the LEX ecosystem because it has not paid for assessment.

11.5 Extension Registry Governance

The extension namespace registry (lexstandard.org/registry) accepts all registrations that follow the naming convention. Grounds for rejection are limited to:

1. Collision with the `lex.core.*` reserved namespace
2. Demonstrable impersonation of another organisation
3. Format does not match the documented schema

There is no editorial veto, no review fee, and no approval delay beyond format validation. Namespace registration is not an endorsement by any governing body.

11.6 Simplicity Budget

The LEX core message structure (header + payload) must remain understandable by a competent developer without reading more than two specification documents. This is an explicit design constraint, not an aspiration.

Every new field proposed for the LEX core must be justified against this budget. Optional extension blocks are additive — a developer does not need to understand `lex.captive`, `lex.governance`, or `lex.lineage` to build a conformant L1 system.

When a proposed addition to the core is rejected in favour of an optional block, the specification will document the reason so future contributors understand the constraint.

11.7 Versioning and Deprecation

No field may be removed from the core specification without:

1. A MAJOR version bump
2. A minimum 18-month deprecation period during which the field is marked `deprecated` but still valid
3. A published migration guide

This constraint exists because participants in the LEX network operate on independent release cycles. A field removal that breaks an L1 receiver causes silent data loss in production — the worst possible failure mode for a lead interchange standard.

Bibliography

The following standards, specifications, and informational references are cited or directly relevant to the LEX Lead Exchange Standard.

Data Format Standards

[JSON] Bray, T. (Ed.). *The JavaScript Object Notation (JSON) Data Interchange Format.* IETF RFC 8259, December 2017. https://www.rfc-editor.org/rfc/rfc8259

[JSON-Schema] Wright, A., Andrews, H., Hutton, B. (Eds.). *JSON Schema: A Media Type for Describing JSON Documents.* Internet-Draft draft-bhutton-json-schema-01, December 2020. https://json-schema.org/specification

[XML] Bray, T. et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation, November 2008. https://www.w3.org/TR/xml/

[EDIFACT] *ISO 9735: Electronic Data Interchange for Administration, Commerce and Transport (EDIFACT) — Application level syntax rules.* International Organization for Standardization, 2002.

[X12] *ASC X12 Standards for Electronic Data Interchange.* Accredited Standards Committee X12, Washington D.C. https://www.x12.org/

[ISO-8601] *ISO 8601-1:2019 — Date and time — Representations for information interchange — Part 1: Basic rules.* International Organization for Standardization, 2019.

Transport and Security Standards

[HTTPS] Fielding, R. et al. *HTTP/1.1: Message Syntax and Routing.* IETF RFC 7230, June 2014. Updated by RFC 9110 (HTTP Semantics), June 2022. https://www.rfc-editor.org/rfc/rfc9110

[TLS13] Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.3.* IETF RFC 8446, August 2018. https://www.rfc-editor.org/rfc/rfc8446

[JWT] Jones, M., Bradley, J., Sakimura, N. *JSON Web Token (JWT).* IETF RFC 7519, May 2015. https://www.rfc-editor.org/rfc/rfc7519

[WEBHOOK] Webhooks are a widely adopted, informally standardised pattern. For a community-maintained description see: https://www.webhooks.fyi/

[AMQP] *OASIS AMQP Version 1.0 — Advanced Message Queuing Protocol.* OASIS Standard, October 2012. https://docs.oasis-open.org/amqp/core/v1.0/

API and Event Specifications

[OpenAPI] *OpenAPI Specification 3.1.0.* OpenAPI Initiative, February 2021. https://spec.openapis.org/oas/v3.1.0

[AsyncAPI] Ponchon, F. et al. *AsyncAPI Specification 3.0.0.* AsyncAPI Initiative, 2023. https://www.asyncapi.com/docs/reference/specification/v3.0.0

[CloudEvents] *CloudEvents Specification v1.0.2.* Cloud Native Computing Foundation (CNCF), 2022. https://cloudevents.io/

Privacy and Regulatory Frameworks

[GDPR] *Regulation (EU) 2016/679 of the European Parliament and of the Council — General Data Protection Regulation.* Official Journal of the European Union, May 2018. https://gdpr-info.eu/

[CCPA] *California Consumer Privacy Act of 2018 (CCPA), Cal. Civ. Code § 1798.100 et seq.* State of California, amended by CPRA (Prop. 24), effective January 2023. https://oag.ca.gov/privacy/ccpa

---

Informational References

[RFC2119] Bradner, S. *Key Words for Use in RFCs to Indicate Requirement Levels.* IETF RFC 2119, March 1997. LEX uses MUST/SHOULD/MAY with the semantics defined here. https://www.rfc-editor.org/rfc/rfc2119

[RFC8174] Leiba, B. *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words.* IETF RFC 8174, May 2017. https://www.rfc-editor.org/rfc/rfc8174

[SemVer] Preston-Werner, T. *Semantic Versioning 2.0.0.* https://semver.org/ — LEX version numbers follow SemVer conventions.

LEX Project Resources

Resource	Location
Online specification & field dictionary	https://lexstandard.org/spec
JSON Schemas	https://lexstandard.org/schemas
Conformance test suite	https://lexstandard.org/conformance
Extension registry	https://lexstandard.org/registry
Client libraries (Python, JS, Java, C#)	https://lexstandard.org/libraries
GitHub repository	https://github.com/lexstandard/