
Workgroup:	Network Working Group		
Internet-Draft:	draft-lightning-session-00		
Published:	9 September 2026		
Intended Status:	Informational		
Expires:	13 March 2027		
Authors:	K. Zhang	J. Klein	Z. Lu
	<i>Lightspark</i>	<i>Lightspark</i>	<i>Lightspark</i>

Lightning Network Session Intent for HTTP Payment Authentication

Abstract

This document defines the "session" intent for the "lightning" payment method using BOLT11 invoices on the Lightning Network, within the Payment HTTP Authentication Scheme. It specifies a prepaid session model for incremental, metered payments suitable for streaming services such as LLM token generation, where the per-request cost is unknown upfront.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 13 March 2027.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	4
1.1. Use Case: LLM Token Streaming	4
2. Requirements Language	6
3. Terminology	6
4. Intent Identifier	6
5. Encoding Conventions	6
6. Session Lifecycle	7
7. Challenge Request	7
8. Credential Schema	8
8.1. Structure	8
8.2. Open Action	9
8.3. Bearer Action	9
8.4. TopUp Action	10
8.5. Close Action	11
9. Deposit Amount	13
10. Verification Procedure	13
10.1. Open Verification	13
10.2. Bearer Verification	13
10.3. TopUp Verification	14
10.4. Close Verification	14
11. Idempotency	14
12. Server-Initiated Close	15
13. Streaming and Per-Chunk Billing	15
13.1. payment-need-topup Event	16
13.2. session-timeout Event	16
13.3. Client Top-Up Flow	17
14. Session State	18

15. Receipt	18
15.1. Streaming Receipt Event	19
16. Error Responses	19
17. Security Considerations	20
17.1. Bearer Token Exposure	20
17.2. Preimage Verification	20
17.3. Session Isolation	20
17.4. Double-Spend Prevention	21
17.5. Return Invoice Validation	21
17.6. Top-Up Invoice Binding	21
17.7. Replay Prevention	21
18. IANA Considerations	21
18.1. Payment Intent Registration	21
18.2. Problem Type Registrations	22
19. References	22
19.1. Normative References	22
19.2. Informative References	23
Appendix A. Examples	24
A.1. Challenge (Unauthenticated Request)	24
A.2. Open Credential	24
A.3. Bearer Credential	25
A.4. Insufficient Balance SSE Event	25
A.5. TopUp Credential	25
A.6. Close Credential and Response	25
Appendix B. Schema Definitions	26
B.1. Challenge Request Schema	26
B.2. Credential Payload Schema	27
B.3. Receipt Schema	28
Appendix C. Acknowledgements	28
Authors' Addresses	28

1. Introduction

HTTP Payment Authentication [[I-D.httpauth-payment](#)] defines a challenge-response mechanism that gates access to HTTP resources behind micropayments. This document registers the "session" intent for the "lightning" payment method.

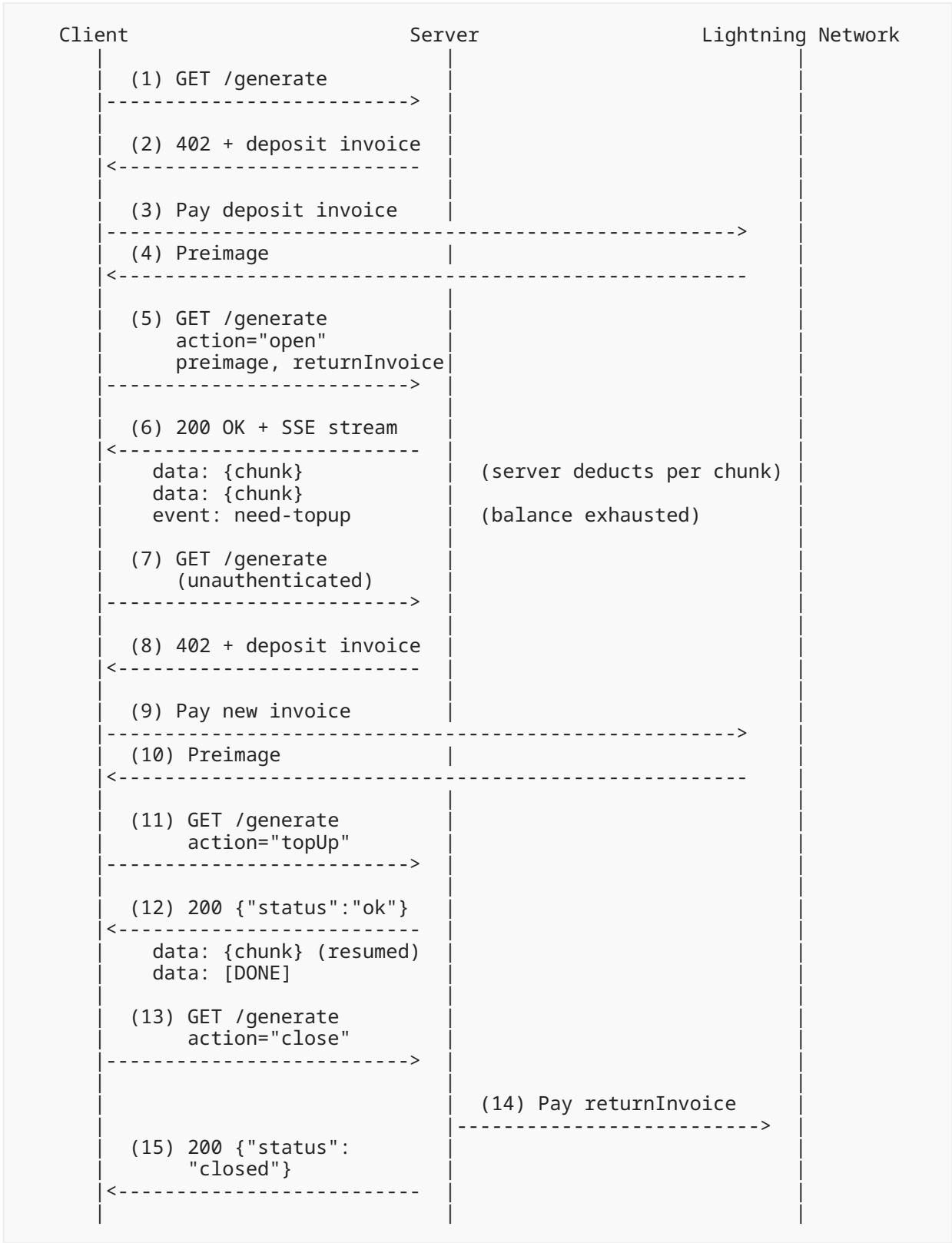
Unlike the "charge" intent, which requires a full per-request Lightning payment, the session intent allows clients to pre-deposit a lump sum and then authenticate subsequent requests by presenting the payment preimage as a bearer token. The server tracks a running balance and deducts the configured cost per unit of service. When the session closes, the server refunds any unspent balance via a client-supplied return invoice.

This model is well-suited to streaming responses (e.g., Server-Sent Events for LLM token generation) where the total cost is unknown upfront and per-request Lightning round-trips would introduce unacceptable latency. This design is inspired by the session intent defined for EVM payment channels in [[draft-tempo-session-00](#)]. The proof mechanism differs: Lightning session uses a prepaid balance with the deposit preimage as a bearer token, while Tempo session uses cumulative vouchers against on-chain escrow. Both implementations share the same abstract intent: prepaid deposit, per-unit billing, and refund of unspent balance on close.

1.1. Use Case: LLM Token Streaming

The typical session flow proceeds as follows:

1. Client requests a streaming completion (SSE response)
2. Server returns 402 with a session challenge containing a deposit invoice
3. Client pays the deposit invoice and opens a session
4. Server begins streaming; deducts cost per chunk from session balance
5. If balance is exhausted mid-stream, server emits a payment-need-topup SSE event and holds the connection open; client pays a new deposit invoice and sends a topUp credential; server resumes the stream on the same connection
6. Client closes the session; server refunds unspent balance via the return invoice provided at open time



2. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Terminology

Session A prepaid payment relationship between a client and server, identified by the paymentHash of the deposit invoice.

Session Balance The running total of deposited satoshis minus satoshis spent against the session.

Deposit A [BOLT11] payment from the client to the server that establishes or extends the session balance.

Payment Preimage A 32-byte random secret whose SHA-256 hash equals the paymentHash of the deposit invoice. Revealed to the payer upon Lightning payment settlement. Used as a bearer token for the lifetime of the session.

Return Invoice A BOLT11 invoice with no encoded amount, created by the client at session open. The server pays this invoice with the unspent session balance on close, specifying the refund amount explicitly via amountSatsToSend.

Top-Up An additional Lightning payment to an existing session. The client pays a new deposit invoice and submits the top-up preimage; the server adds the amount to the session balance.

4. Intent Identifier

The intent identifier for this specification is "session". It **MUST** be lowercase.

5. Encoding Conventions

All JSON [RFC8259] objects carried in auth-params or HTTP headers in this specification **MUST** be serialized using the JSON Canonicalization Scheme (JCS) [RFC8785] before encoding. JCS produces a deterministic byte sequence, which is required for any digest or signature operations defined by the base spec [I-D.httpauth-payment].

The resulting bytes **MUST** then be encoded using base64url [RFC4648] Section 5 without padding characters (=). Implementations **MUST NOT** append = padding when encoding, and **MUST** accept input with or without padding when decoding.

This encoding convention applies to: the request auth-param in WWW-Authenticate, the credential token in Authorization, and the receipt token in Payment-Receipt.

6. Session Lifecycle

A session progresses through three phases, each corresponding to one or more credential actions:

1. **Open**: Client pays the deposit invoice and submits an open action. The server verifies the preimage, stores session state, and begins serving requests against the deposit balance.
2. **Streaming**: Client submits bearer actions to authenticate requests without additional Lightning payments. The streaming layer deducts the per-unit cost from the session balance for each chunk delivered. When the balance is exhausted mid-stream, the server emits a payment-need-topup event and holds the connection open. Top-up actions extend the balance and allow the stream to resume.
3. **Closed**: Client submits a close action. The server verifies session ownership, pays any unspent balance to the return invoice, and marks the session closed. No further actions are accepted.

Sessions have no built-in expiry and remain open until explicitly closed by the client or by the server. Servers **SHOULD** enforce an idle timeout on open sessions to bound operational liability and storage requirements, and **MUST** document their timeout policy. The **RECOMMENDED** idle timeout is 5 minutes of inactivity. When a server closes a session without a client close action, it **MUST** follow the same refund procedure described in [Section 12](#).

7. Challenge Request

When a request arrives without a valid credential, the server **MUST** respond with HTTP 402 [[RFC9110](#)], a Cache-Control: no-store [[RFC9111](#)] header, and a WWW-Authenticate: Payment header of the form:

```
WWW-Authenticate: Payment id="<challenge-id>",
  realm="<realm>",
  method="lightning",
  intent="session",
  request="<base64url(JCS-serialized JSON)>",
  expires="<RFC3339 timestamp>"
```

The request auth-param contains a JCS-serialized, base64url-encoded JSON object (see [Section 5](#)) with the following fields:

amount **REQUIRED**. Cost per unit of service in base units (satoshis), as a decimal string (e.g., "2"). For streaming responses, this is the cost per emitted chunk. The value **MUST** be a positive integer.

currency **REQUIRED**. Identifies the unit for amount. **MUST** be the string "sat" (lowercase). "sat" denotes satoshis, the base unit used for Lightning/Bitcoin amounts.

description **OPTIONAL**. Human-readable description of the service. This field is carried inside the request JSON and is distinct from any description auth-param that the base [I-D.httpauth-payment] scheme may include at the header level. Servers **MAY** instead (or additionally) convey the description as the base spec's description auth-param.

unitType **OPTIONAL**. Human-readable label for the unit being priced (e.g., "token", "chunk", "request"). Informational only; clients **MAY** display this to users.

depositInvoice **CONDITIONAL**. BOLT11 invoice the client must pay to open or top up the session. **REQUIRED** for open and topUp challenges; **MUST** be absent for bearer and close challenges, where no payment is required.

paymentHash **REQUIRED**. SHA-256 hash of the deposit invoice preimage, as a lowercase hex string. Used by the server to verify open and topUp credentials.

depositAmount **OPTIONAL**. Exact deposit amount in satoshis, as a decimal string. When present, **MUST** equal the amount encoded in depositInvoice. Informs the client of the deposit size before it inspects the invoice. The BOLT11 invoice encodes a fixed amount; the client pays exactly that amount. This document uses "depositAmount" (not "suggestedDeposit") intentionally: the client cannot deposit a different amount than the invoice specifies.

idleTimeout **OPTIONAL**. The server's idle timeout policy for open sessions, in seconds, as a decimal string (e.g., "300" for 5 minutes). When present, informs the client how long the server will retain an open session without activity before initiating a server-side close. Clients **SHOULD** ensure their return invoice expiry exceeds this value. This field is informational; the server's actual timeout behavior is authoritative.

Servers **MUST** generate a fresh BOLT11 invoice for every unauthenticated request. The invoice amount is the deposit amount as described in [Section 9](#).

8. Credential Schema

8.1. Structure

The Authorization header carries a single base64url-encoded JSON token (no auth-params) per [I-D.httpauth-payment]. The decoded object contains three top-level fields:

challenge **REQUIRED**. An echo of the challenge auth-params from the most recent WWW-Authenticate: Payment header: id, realm, method, intent, request, and (if present) expires. This binds the credential to a specific, unexpired, single-use challenge issued by the server. For the open and topUp actions, where a new deposit invoice was issued, this also binds the preimage to the specific invoice.

source **OPTIONAL**. A payer identifier string, as defined by [I-D.httpauth-payment]. The **RECOMMENDED** format is a Decentralized Identifier (DID) per [W3C-DID]. Servers **MUST NOT** require this field.

payload **REQUIRED**. A JSON object containing the session action. The action field discriminates the type; action-specific fields are placed alongside it. Implementations **MUST** ignore unknown fields to allow forward-compatible extensions.

8.2. Open Action

The open action proves the deposit invoice was paid and registers the client's return invoice for future refunds.

action **REQUIRED**. The string "open".

preimage **REQUIRED**. Hex-encoded payment preimage. SHA-256(preimage) **MUST** equal challenge.request.paymentHash.

returnInvoice **REQUIRED**. BOLT11 invoice with no encoded amount. The server pays the unspent session balance to this invoice on close. **MUST** have no amount in the BOLT11 human-readable part. **SHOULD** have an expiry of at least 30 days to remain valid when the session eventually closes.

Example credential (decoded):

```
{
  "challenge": {
    "id": "nX7kPqWvT2mJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "preimage": "a3f1e2d4b5c6a7e8...",
    "returnInvoice": "lnbcrt1p5abc..."
  }
}
```

8.3. Bearer Action

The bearer action authenticates a request against an existing session without any Lightning payment. The client proves session ownership by presenting the preimage that was revealed when the deposit invoice was paid.

action **REQUIRED**. The string "bearer".

sessionId **REQUIRED**. The paymentHash of the original deposit invoice, identifying the session.

preimage **REQUIRED**. Same preimage as the open action. SHA-256(preimage) **MUST** equal session.paymentHash.

The server verifies `SHA-256(preimage) == session.paymentHash` and that the session is open. Balance checking and deduction are handled by the streaming layer (see [Section 13](#)); bearer verification itself does not modify session.spent.

Security note: The payment preimage is a 32-byte random secret revealed only to the payer upon Lightning payment settlement. Using it directly as a bearer token allows the server to verify ownership with a single SHA-256 check against the stored paymentHash, without ever storing the secret. An alternative design using per-request HMAC tokens would require the server to store the preimage, which is a worse security posture. Implementations **MUST** use TLS (TLS 1.2 or higher; TLS 1.3 **RECOMMENDED**); the preimage carries the same threat model as any API bearer token.

Example credential (decoded):

```
{
  "challenge": {
    "id": "pR4mNvKqU8wLsYtZ1bCdFg",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ..."
  },
  "payload": {
    "action": "bearer",
    "sessionId": "7f3a1b2c4d5e6f...",
    "preimage": "a3f1e2d4b5c6a7e8..."
  }
}
```

8.4. TopUp Action

The topUp action proves payment of a new deposit invoice and adds the deposited amount to the existing session balance.

action **REQUIRED**. The string "topUp".

sessionId **REQUIRED**. The paymentHash of the original deposit invoice, identifying the session.

topUpPreimage **REQUIRED**. Preimage of the top-up invoice. SHA-256(topUpPreimage) **MUST** equal challenge.request.paymentHash of the fresh invoice issued for this top-up.

To obtain a challenge for a top-up, the client submits an unauthenticated request (no Authorization header) to a protected endpoint. The request **MAY** target the same resource URI that required the top-up (e.g., the paused stream) or a different protected resource URI in the

same realm; the server issues a fresh deposit invoice in either case. The server assigns a new challenge id and returns 402. The client pays the invoice, obtains the preimage, and submits a topUp credential echoing that challenge. The server **MUST** verify the echoed challenge .id exists, has not expired, and has not been previously consumed before crediting the balance. This ensures each top-up invoice can only be used once.

Upon successful verification, the server **MUST** atomically add the top-up amount to session.depositSats and return HTTP 200 with body { "status": "ok" }. The top-up credits the shared session balance; any streams that were paused waiting for sufficient balance **SHOULD** observe the updated balance and resume delivery autonomously. The server **MUST NOT** initiate a new stream in response to a topUp credential.

Example credential (decoded):

```
{
  "challenge": {
    "id": "qS5n0wLrV9xMtZuA2cDeGh",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-03-15T12:10:00Z"
  },
  "payload": {
    "action": "topUp",
    "sessionId": "7f3a1b2c4d5e6f...",
    "topUpPreimage": "b9c3a4e1d2f5..."
  }
}
```

8.5. Close Action

The close action terminates the session and triggers a refund of the unspent balance to the client's return invoice.

action **REQUIRED**. The string "close".

sessionId **REQUIRED**. The paymentHash of the original deposit invoice, identifying the session.

preimage **REQUIRED**. Same preimage as the open action. Proves session ownership.

After verifying the preimage and marking the session closed, the server **MUST**:

1. Compute refundSats = session.depositSats - session.spent
2. If refundSats > 0, attempt to pay session.returnInvoice with amountSatsToSend = refundSats. If refundSats is zero, the server **MUST NOT** attempt to pay the return invoice.
3. Return HTTP 200 with a Payment-Receipt header (per [Section 15](#)) and body {"status": "closed", "refundSats": N, "refundStatus": "succeeded" | "failed" | "skipped"}. If refundSats is zero, set refundStatus to "skipped". If the refund payment succeeded, set

refundStatus to "succeeded". If the refund payment failed (e.g., the return invoice has expired or cannot be routed), set refundStatus to "failed"; the server **MUST** still close the session and **MUST NOT** reopen it. The server **SHOULD** log failed refunds for auditability. The server **MUST NOT** retry the refund indefinitely; a single best-effort attempt is sufficient.

Clients **MUST NOT** submit further actions on a closed session. Servers **MUST** reject any action on a closed session.

Example credential (decoded):

```
{
  "challenge": {
    "id": "rT6oQxMsW0yNuAvB3dEfHi",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ..."
  },
  "payload": {
    "action": "close",
    "sessionId": "7f3a1b2c4d5e6f...",
    "preimage": "a3f1e2d4b5c6a7e8..."
  }
}
```

Example response (decoded receipt in Payment-Receipt header, body). Success:

```
{ "status": "closed", "refundSats": 140, "refundStatus": "succeeded" }
```

Refund payment failed (session still closed):

```
{ "status": "closed", "refundSats": 140, "refundStatus": "failed" }
```

No refund owed:

```
{ "status": "closed", "refundSats": 0, "refundStatus": "skipped" }
```

The Payment-Receipt header **MUST** be present. For close, the receipt **MUST** also include refundSats and refundStatus (see [Section 15](#)). Example decoded receipt for close (success):

```
{"method":"lightning","reference":"7f3a1b2c4d5e6f...","status":"success","timestamp":"2026-03-10T21:00:00Z","refundSats":140,"refundStatus":"succeeded"}
```

Close with refund failed:

```
{"method":"lightning","reference":"7f3a1b2c4d5e6f...","status":"success","timestamp":"2026-03-10T21:00:00Z","refundSats":140,"refundStatus":"failed"}
```

9. Deposit Amount

The server determines the deposit amount when generating the challenge invoice. The deposit **MUST** be at least 1 unit of service (i.e., at least amount satoshis). The **RECOMMENDED** formula is:

```
depositSats = configured_depositAmount ?? (amount * 20)
```

The default multiplier of 20 is a recommendation that gives clients ~20 units of service before a top-up is required, balancing upfront cost against top-up frequency. Servers **MAY** use a different multiplier and **MUST** document their deposit policy.

Clients **SHOULD** inspect the depositAmount field in the challenge request before paying to confirm the expected deposit size. The deposit amount is encoded in the BOLT11 invoice and cannot be changed by the client.

10. Verification Procedure

10.1. Open Verification

1. Look up the stored challenge using `credential.challenge.id`. If not found or already consumed, reject.
2. Verify the echoed `credential.challenge` exactly matches the stored challenge params.
3. Decode preimage from hex
4. Compute SHA-256(preimage) and verify it equals the paymentHash stored with the challenge
5. Decode deposit amount from the BOLT11 invoice human-readable part
6. Verify `depositSats >= amount` (at least one unit of service)
7. Verify the returnInvoice is a valid BOLT11 invoice on the same network as the deposit invoice. Decode the returnInvoice and verify that the encoded amount resolves to 0 satoshis. The invoice **MAY** omit the amount field entirely, or **MAY** encode an explicit 0-satoshi amount; both are accepted. Invoices encoding a non-zero amount **MUST** be rejected, as the refund amount is determined by the server at close time.
8. Store session state: {paymentHash, depositSats, spent: 0, returnInvoice, status: "open"}
9. Return a receipt with reference = paymentHash (the session ID)

10.2. Bearer Verification

1. Look up the session by `payload.sessionId`
2. Verify `session.status == "open"`
3. Compute SHA-256(payload.preimage) and verify it equals `session.paymentHash`

4. Return a receipt

Bearer verification does not deduct from the session balance. All billing is handled by the streaming layer (see [Section 13](#)).

10.3. TopUp Verification

1. Look up the stored challenge using `credential.challenge.id`. If not found, expired, or already consumed, reject.
2. Verify the echoed `credential.challenge` exactly matches the stored challenge params. Mark the challenge consumed.
3. Look up the session by `payload.sessionId`
4. Verify `session.status == "open"`
5. Compute `SHA-256(payload.topUpPreimage)` and verify it equals the `paymentHash` stored with the challenge
6. Decode top-up amount from the BOLT11 invoice in the stored challenge's `depositInvoice`
7. Atomically update `session.depositSats += topUpSats`
8. Return a receipt

10.4. Close Verification

1. Look up the session by `payload.sessionId`
2. Verify `session.status == "open"`
3. Compute `SHA-256(payload.preimage)` and verify it equals `session.paymentHash`
4. Mark `session.status = "closed"`
5. If `refundSats > 0`, attempt to pay `session.returnInvoice` with `amountSatsToSend = refundSats`. If `refundSats` is zero, the server **MUST NOT** attempt to pay the return invoice.
6. Return HTTP 200 with a Payment-Receipt header and body
`{"status":"closed","refundSats":N,"refundStatus":"succeeded" | "failed" | "skipped"}`. The server **SHOULD** log failed refunds. The server **MUST NOT** retry the refund indefinitely.

11. Idempotency

Retrying a credential after a network failure **MUST NOT** result in double-crediting or double-refunding. Servers **MUST** enforce idempotency keyed by `credential.challenge.id`:

- A challenge that has already been successfully consumed **MUST** return the original HTTP response (same status code and body) when presented again, without re-executing the action.
- Servers **MUST** store the result of each consumed challenge (status code, response body) for at least the duration of the challenge's expires window, or for a minimum of 5 minutes if no expiry was set.

- Servers **MUST** atomically mark a challenge consumed and record its result in a single operation, so that a crash between verification and response does not leave the challenge in an ambiguous state on retry.

12. Server-Initiated Close

A server **MAY** close an open session without a client close action. Common triggers include idle timeout (no bearer or topUp action received within the server's configured inactivity window) and planned maintenance. When a server initiates a close, it **MUST**:

1. Mark `session.status = "closed"` atomically before attempting the refund, to prevent concurrent bearer actions from spending against a session that is being closed.
2. Compute `refundSats = session.depositSats - session.spent`
3. If `refundSats > 0`, attempt to pay `session.returnInvoice` with `amountSatsToSend = refundSats`, subject to the same fee constraints as a client-initiated close. If `refundSats` is zero, the server **MUST NOT** attempt to pay the return invoice.

If the refund payment fails (e.g., the return invoice has expired or cannot be routed), the server **MUST NOT** reopen the session. The server **SHOULD** log the failed refund attempt including the `sessionId` and `refundSats` for auditability. The server **MUST NOT** retry indefinitely; a single best-effort attempt is sufficient.

After a server-initiated close, subsequent bearer or close actions referencing that session will be rejected per the normal closed-session handling (see [Section 17.7](#)). Clients discover the server-initiated close when their next bearer action is rejected and **SHOULD** treat the rejection as a signal to open a new session.

To reduce the risk of return invoice expiry, clients **SHOULD** supply a return invoice with an expiry that comfortably exceeds the server's advertised `idleTimeout`. The **RECOMMENDED** return invoice expiry is at least twice the `idleTimeout` value.

13. Streaming and Per-Chunk Billing

For streaming responses, the challenge request's `amount` field is the cost per emitted chunk in satoshis. Bearer verification does not deduct from the session balance; instead, the streaming layer deducts `amount` sats from the balance for each chunk delivered, reading the per-unit cost directly from the echoed challenge request.

When a streaming response exhausts the available session balance mid-stream, the server **MUST**:

1. Stop delivering additional metered content immediately
2. Emit a payment-need-topup SSE event on the existing connection
3. Hold the connection open and pause delivery
4. Poll or await an increase in `session.depositSats`. Once a topUp credential is successfully verified and the session balance is sufficient to cover the next unit of service, resume delivery on the paused connection.

A top-up credits the shared session balance; the server does not need to be explicitly notified which connection to resume. Any paused streams that observe sufficient balance after a top-up **SHOULD** resume autonomously. Servers **SHOULD** close any held connection if the balance does not become sufficient within a reasonable timeout (**RECOMMENDED**: 60 seconds). When the timeout fires, the server **MUST** close the SSE connection. The server **SHOULD** emit a final SSE event (e.g., event: session-timeout) with session balance details before closing (see [Section 13.2](#)), so the client does not only observe a dropped connection.

Holding the connection open preserves any upstream state the server maintains (e.g., an in-flight request to an upstream LLM provider). Clients benefit because they do not need to replay the original request or deduplicate partially-delivered content.

13.1. payment-need-topup Event

For SSE [[SSE](#)] responses, the payment-need-topup event **MUST** be formatted as follows:

```
event: payment-need-topup
data: {"sessionId":"7f3a...", "balanceSpent":300, "balanceRequired":2}
```

The event data **MUST** be a JSON object containing:

sessionId **REQUIRED**. The session identifier (paymentHash of the deposit invoice).

balanceSpent **REQUIRED**. Total satoshis spent from the current deposit at the point of exhaustion.

balanceRequired **REQUIRED**. Satoshis needed for the next unit of service (i.e., the amount that could not be covered).

13.2. session-timeout Event

When the server closes a held SSE connection because the balance did not become sufficient within the configured timeout, the server **SHOULD** emit a session-timeout event immediately before closing the connection. This allows the client to distinguish a timeout from a generic network drop and to show session balance details to the user.

```
event: session-timeout
data: {"sessionId":"7f3a...", "balanceSpent":300, "balanceRequired":2}
```

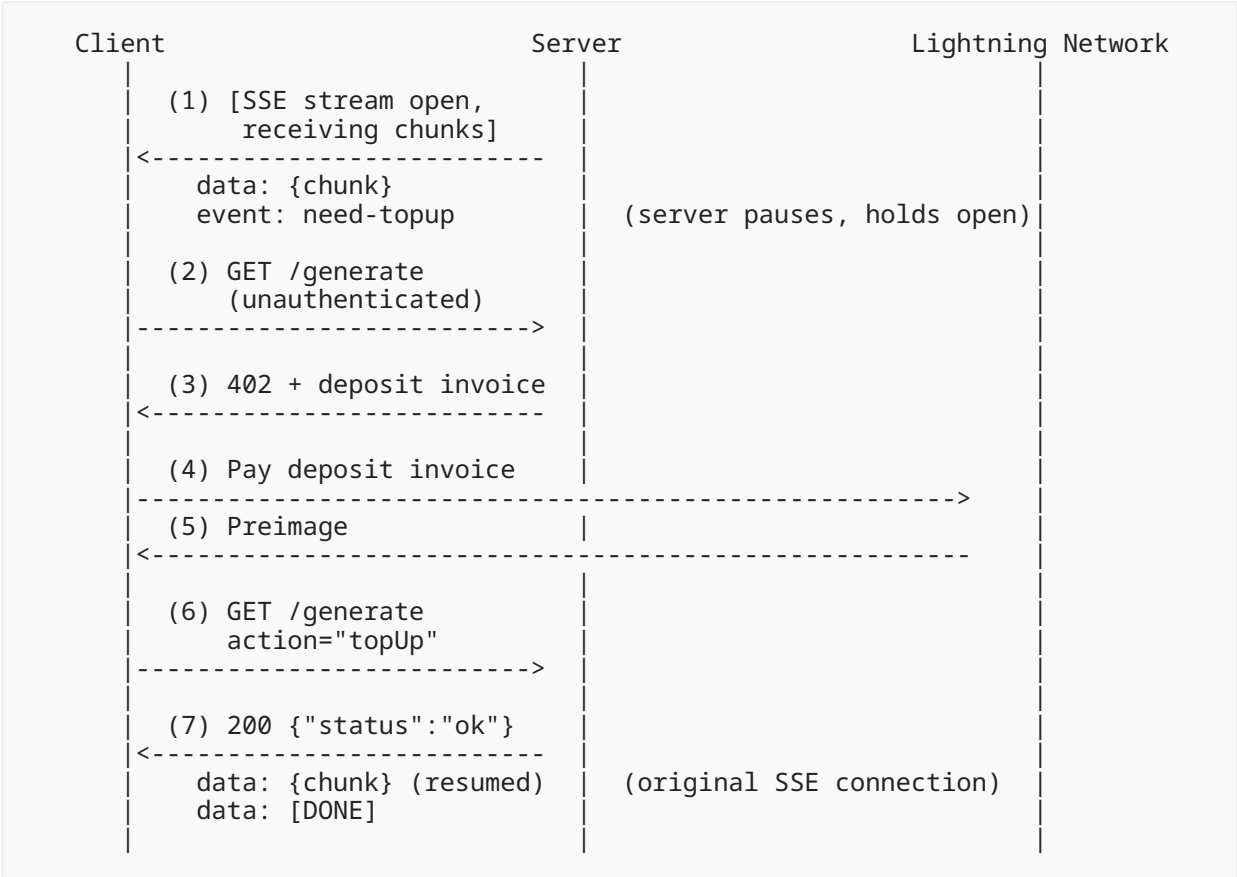
The event data **MUST** be a JSON object containing the same fields as the payment-need-topup event: sessionId, balanceSpent, and balanceRequired (see [Section 13.1](#)). After emitting this event, the server **MUST** close the SSE connection.

13.3. Client Top-Up Flow

Upon receiving a payment-need-topup event, the client **MUST**:

- 1. Continue holding the original SSE response reader open (do not close the connection)
- 2. Submit an unauthenticated request to the same resource URI to obtain a new 402 challenge with a fresh deposit invoice
- 3. Pay the deposit invoice and obtain the preimage
- 4. Retry the request with a topUp credential; the server verifies the preimage, credits the session balance, and returns HTTP 200
- 5. Continue reading from the original SSE reader; the paused stream observes the updated balance and resumes automatically

The topUp request returns HTTP 200 with body {"status": "ok"} to acknowledge the balance credit. The actual stream content resumes on the original connection, not the topUp response.



14. Session State

Servers **MUST** persist session state in a durable store keyed by sessionId. The minimum required state is:

paymentHash SHA-256 hash of the deposit preimage. Serves as the session identifier.

depositSats Total satoshis deposited. Increases with each successful top-up.

spent Running total of satoshis charged against the session.

returnInvoice BOLT11 return invoice for refunds on close.

status Either "open" or "closed".

Servers **MUST** serialize all balance updates to prevent race conditions. Concurrent bearer and topUp requests for the same session **MUST NOT** result in double-spending the same satoshis.

15. Receipt

Upon successful verification of any action (including close), the server **MUST** attach a payment receipt to the response via the Payment-Receipt header per [\[I-D.httpauth-payment\]](#) {#receipt}. The header receipt contains:

method **REQUIRED**. The string "lightning".

reference **REQUIRED**. The session ID (paymentHash).

status **REQUIRED**. The string "success".

timestamp **REQUIRED**. Settlement time in [\[RFC3339\]](#) format.

refundSats For close actions only. **REQUIRED** in the receipt when the action is close. The unspent balance that was (or was attempted to be) refunded, as a number. Omitted for non-close actions.

refundStatus For close actions only. **REQUIRED** in the receipt when the action is close. One of "succeeded" (refund payment completed), "failed" (refund payment did not complete, e.g., return invoice expired or could not be routed), or "skipped" (refundSats was zero, no payment attempted). Omitted for non-close actions.

15.1. Streaming Receipt Event

For streaming responses, the Payment-Receipt header is attached at response initiation before any chunks are delivered. Because the final consumption totals are not yet known at that point, the server **MUST** also emit a final payment-receipt SSE [SSE] event once the stream completes, with the following JSON data:

method **REQUIRED**. The string "lightning".

reference **REQUIRED**. The session ID (paymentHash).

status **REQUIRED**. The string "success".

timestamp **REQUIRED**. Completion time in [RFC3339] format.

spent **REQUIRED**. Total satoshis deducted from the session balance during this stream, as a number.

units **REQUIRED**. Number of chunks delivered in this stream.

The payment-receipt event **MUST** be emitted before the [DONE] sentinel. Example:

```
event: payment-receipt
data: {"method":"lightning","reference":"7f3a...","status":"success",
      "timestamp":"2026-03-11T00:00:00Z","spent":202,"units":101}

data: [DONE]
```

16. Error Responses

When rejecting a credential, the server **MUST** return HTTP 402 (Payment Required) with a fresh WWW-Authenticate: Payment challenge per [I-D.httpauth-payment]. The server **SHOULD** include a response body conforming to RFC 9457 [RFC9457] Problem Details, with Content-Type: application/problem+json. The following problem types are defined for this intent:

https://paymentauth.org/problems/lightning/malformed-credential HTTP 402. The credential token could not be decoded or parsed, or required fields are absent or have the wrong type. A fresh challenge **MUST** be included in WWW-Authenticate.

https://paymentauth.org/problems/lightning/unknown-challenge HTTP 402. The credential.challenge.id does not match any challenge issued by this server, or has already been consumed. A fresh challenge **MUST** be included in WWW-Authenticate.

`https://paymentauth.org/problems/lightning/invalid-preimage` HTTP 402.

SHA-256(payload.preimage) or SHA-256(payload.topUpPreimage) does not equal the paymentHash stored for the identified challenge. A fresh challenge **MUST** be included in WWW-Authenticate.

`https://paymentauth.org/problems/lightning/session-not-found` HTTP 402. The

payload.sessionId does not match any session stored by this server. A fresh challenge **MUST** be included in WWW-Authenticate.

`https://paymentauth.org/problems/lightning/session-closed` HTTP 402. The session identified by

payload.sessionId has status "closed". No further actions are accepted. A fresh challenge **MUST** be included in WWW-Authenticate.

`https://paymentauth.org/problems/lightning/insufficient-balance` HTTP 402. The session balance

is insufficient to cover the requested operation. The client **SHOULD** top up the session. A fresh challenge **MUST** be included in WWW-Authenticate.

`https://paymentauth.org/problems/lightning/challenge-expired` HTTP 402. The challenge

identified by credential.challenge.id has passed its expiry time. The client **MUST** obtain a fresh challenge. A fresh challenge **MUST** be included in WWW-Authenticate.

`https://paymentauth.org/problems/lightning/invalid-return-invoice` HTTP 402. The

payload.returnInvoice in an open action is not a valid BOLT11 invoice, or encodes a non-zero amount. A fresh challenge **MUST** be included in WWW-Authenticate.

17. Security Considerations

17.1. Bearer Token Exposure

The preimage is transmitted in every bearer request. Implementations **MUST** use TLS (HTTPS) for all endpoints protected by this method (TLS 1.2 or higher; TLS 1.3 **RECOMMENDED**). The preimage has the same exposure risk as any API bearer token; its security properties are equivalent to a 256-bit random secret transmitted over an encrypted channel.

17.2. Preimage Verification

Servers **MUST** verify `SHA-256(preimage) == paymentHash` for all open, bearer, and close actions. Failure to verify allows any party to impersonate a session they did not fund.

17.3. Session Isolation

The paymentHash of the deposit invoice serves as the session identifier. Payment hashes are globally unique within the Lightning Network for a given invoice. Servers **MUST NOT** allow session IDs to be guessed or reused across sessions.

17.4. Double-Spend Prevention

Balance deduction is the exclusive responsibility of the streaming layer (see [Section 13](#)); bearer credential verification itself does not modify `session.spent`. The streaming layer **MUST** atomically deduct amount sats from the available balance for each chunk delivered, to prevent concurrent chunk deliveries from overdrawing the session balance. TopUp actions **MUST** atomically increment `session.depositSats` so that any paused streams observe the updated balance immediately after the credit.

17.5. Return Invoice Validation

Servers **MUST** verify that `returnInvoice` is a valid BOLT11 invoice with no encoded amount before storing it. Servers **SHOULD** reject return invoices that encode an amount, as the refund amount is determined by the server at close time and must not be constrained by the invoice.

17.6. Top-Up Invoice Binding

The `topUpPreimage` **MUST** be verified against `challenge.request.paymentHash` — the payment hash of the fresh invoice the server issued for this specific top-up request. This prevents a client from replaying an old top-up preimage to fraudulently inflate their session balance.

17.7. Replay Prevention

Closed sessions **MUST** be retained in the store (with status: "closed") to prevent replay attacks using the preimage of a closed session. Servers **MUST** reject any action submitted against a closed session.

18. IANA Considerations

The "lightning" payment method is registered in the "HTTP Payment Methods" registry by [[I-D.lightning-charge](#)]; this document does not register it again.

18.1. Payment Intent Registration

This document requests registration of the following entry in the "HTTP Payment Intents" registry established by [[I-D.httpauth-payment](#)]:

Intent	Applicable Methods	Description	Reference	Contact
session	lightning	Prepaid session with per-unit streaming billing and refund on close	This document	Lightspark (contact@lightspark.com)

Table 1

18.2. Problem Type Registrations

This document defines the following problem type URIs under the `https://paymentauth.org/problems/lightning/` namespace, for use with RFC 9457 [RFC9457] Problem Details:

Type URI	HTTP Status	Description	Reference
lightning/malformed-credential	402	Credential is unparseable or missing required fields	This document
lightning/unknown-challenge	402	Challenge ID not found or already consumed	This document
lightning/invalid-preimage	402	Preimage does not match stored payment hash	This document
lightning/session-not-found	402	Session ID not found	This document
lightning/session-closed	402	Session is closed; no further actions accepted	This document
lightning/insufficient-balance	402	Session balance insufficient for requested operation	This document
lightning/challenge-expired	402	Challenge has passed its expiry time	This document
lightning/invalid-return-invoice	402	Return invoice invalid or encodes non-zero amount	This document

Table 2

19. References

19.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <<https://www.rfc-editor.org/info/rfc3339>>.

- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/info/rfc4648>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <<https://www.rfc-editor.org/info/rfc8785>>.
- [RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <<https://www.rfc-editor.org/info/rfc9110>>.
- [RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <<https://www.rfc-editor.org/info/rfc9111>>.
- [RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <<https://www.rfc-editor.org/info/rfc9457>>.
- [BOLT11] Lightning Network Developers, "BOLT #11: Invoice Protocol for Lightning Payments", 2024, <<https://github.com/lightning/bolts/blob/master/11-payment-encoding.md>>.
- [I-D.httpauth-payment] Moxey, J., "The 'Payment' HTTP Authentication Scheme", January 2026, <<https://datatracker.ietf.org/doc/draft-ryan-httpauth-payment/>>.

19.2. Informative References

- [I-D.lightning-charge] Lightspark, "Lightning Network Charge Intent for HTTP Payment Authentication", 2026, <<https://datatracker.ietf.org/doc/draft-lightning-charge/>>.
- [SSE] WHATWG, "Server-Sent Events", 2024, <<https://html.spec.whatwg.org/multipage/server-sent-events.html>>.
- [draft-tempo-session-00] Tempo Labs, "Tempo Session Intent for HTTP Payment Authentication", March 2026, <<https://datatracker.ietf.org/doc/draft-tempo-session/>>.
- [W3C-DID] W3C, "Decentralized Identifiers (DIDs) v1.0", 2022, <<https://www.w3.org/TR/did-core/>>.

Appendix A. Examples

A.1. Challenge (Unauthenticated Request)

```
GET /generate HTTP/1.1
Host: api.example.com

HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="nX7kPqWvT2mJrHsY4aDfEb",
    realm="api.example.com",
    method="lightning",
    intent="session",

request="eyJhbW91bnQiOiIyIiwia3VycmVuY3kiOiJCVEMiLCJkZXBvc2l0SW52b2ljZSI6Imxu
YmNyYDFwNW16ZnNhLi4uIiwicGF5bWVudEhhc2giOiI3ZjNhLi4uIiwiaGVhZGVwb3NpdEFtb3VudCI6I
jMwMCJ9",
    expires="2026-03-15T12:05:00Z"
Cache-Control: no-store
```

Decoded request:

```
{
  "amount": "2",
  "currency": "sat",
  "description": "LLM token stream",
  "depositInvoice": "lnbcrt1p5mzfsa...",
  "paymentHash": "7f3a1b2c4d5e6f...",
  "depositAmount": "300"
}
```

A.2. Open Credential

```
{
  "challenge": {
    "id": "nX7kPqWvT2mJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "action": "open",
    "preimage": "a3f1e2d4b5c6a7e8...",
    "returnInvoice": "lnbcrt1p5abc..."
  }
}
```

A.3. Bearer Credential

```
{
  "challenge": {
    "id": "pR4mNvKqU8wLsYtZ1bCdFg",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ..."
  },
  "payload": {
    "action": "bearer",
    "sessionId": "7f3a1b2c4d5e6f...",
    "preimage": "a3f1e2d4b5c6a7e8..."
  }
}
```

A.4. Insufficient Balance SSE Event

```
event: payment-need-topup
data: {"sessionId":"7f3a...", "balanceSpent":300, "balanceRequired":2}
```

A.5. TopUp Credential

```
{
  "challenge": {
    "id": "qS5n0wLrV9xMtZuA2cDeGh",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ...",
    "expires": "2026-03-15T12:10:00Z"
  },
  "payload": {
    "action": "topUp",
    "sessionId": "7f3a1b2c4d5e6f...",
    "topUpPreimage": "b9c3a4e1d2f5..."
  }
}
```

A.6. Close Credential and Response

Credential (decoded):

```
{
  "challenge": {
    "id": "rT6oQxMsw0yNuAvB3dEfHi",
    "realm": "api.example.com",
    "method": "lightning",
    "intent": "session",
    "request": "eyJ..."
  },
  "payload": {
    "action": "close",
    "sessionId": "7f3a1b2c4d5e6f...",
    "preimage": "a3f1e2d4b5c6a7e8..."
  }
}
```

Response body:

```
{ "status": "closed", "refundSats": 140 }
```

Appendix B. Schema Definitions

The schemas in this appendix use JSON Schema vocabulary for illustrative purposes only. They are informational; the normative definitions are in the body of this document. No specific JSON Schema draft version is required or assumed.

B.1. Challenge Request Schema

```
{
  "type": "object",
  "required": [
    "amount", "currency", "paymentHash"
  ],
  "properties": {
    "amount": { "type": "string" },
    "currency": { "type": "string", "const": "sat" },
    "description": { "type": "string" },
    "unitType": { "type": "string" },
    "depositInvoice": { "type": "string" },
    "paymentHash": { "type": "string" },
    "depositAmount": { "type": "string" },
    "idleTimeout": { "type": "string" }
  }
}
```

B.2. Credential Payload Schema

```
{
  "oneOf": [
    {
      "type": "object",
      "required": ["action", "preimage", "returnInvoice"],
      "properties": {
        "action": { "const": "open" },
        "preimage": { "type": "string" },
        "returnInvoice": { "type": "string" }
      }
    },
    {
      "type": "object",
      "required": ["action", "sessionId", "preimage"],
      "properties": {
        "action": { "const": "bearer" },
        "sessionId": { "type": "string" },
        "preimage": { "type": "string" }
      }
    },
    {
      "type": "object",
      "required": ["action", "sessionId", "topUpPreimage"],
      "properties": {
        "action": { "const": "topUp" },
        "sessionId": { "type": "string" },
        "topUpPreimage": { "type": "string" }
      }
    },
    {
      "type": "object",
      "required": ["action", "sessionId", "preimage"],
      "properties": {
        "action": { "const": "close" },
        "sessionId": { "type": "string" },
        "preimage": { "type": "string" }
      }
    }
  ]
}
```

B.3. Receipt Schema

```
{
  "type": "object",
  "required": ["method", "reference", "status", "timestamp"],
  "properties": {
    "method": { "type": "string", "const": "lightning" },
    "reference": { "type": "string" },
    "status": { "type": "string", "const": "success" },
    "timestamp": { "type": "string", "format": "date-time" }
  }
}
```

Appendix C. Acknowledgements

The authors thank the Spark SDK team, the Tempo Labs team for their prior work on the session intent for EVM-based payment channels, and the broader Lightning Network developer community. The authors also thank Brendan Ryan for his review of this document.

Authors' Addresses

Kevin Zhang

Lightspark

Email: keviz@lightspark.com**Jeremy Klein**

Lightspark

Email: jeremy@lightspark.com**Zhen Lu**

Lightspark

Email: zhenlu@lightspark.com