
Workgroup: Network Working Group
Internet-Draft: draft-solana-charge-00
Published: 21 April 2026
Intended Status: Informational
Expires: 23 October 2026
Authors: L. Galabru I. Gitter
Solana Foundation Solana Foundation

Solana Charge Intent for HTTP Payment Authentication

Abstract

This document defines the "charge" intent for the "solana" payment method within the Payment HTTP Authentication Scheme [I-D.httpauth-payment]. The client constructs and signs a native SOL or SPL token transfer on the Solana blockchain; the server verifies the payment and presents the transaction signature as proof of payment.

Two credential types are supported: `type="transaction"` (default), where the client sends the signed transaction to the server for broadcast, and `type="signature"` (fallback), where the client broadcasts the transaction itself and presents the on-chain transaction signature for server verification.

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 23 October 2026.

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.

Table of Contents

1. Introduction	4
1.1. Pull Mode (Default)	4
1.2. Push Mode (Fallback)	5
1.3. Relationship to the Charge Intent	6
2. Requirements Language	6
3. Terminology	6
4. Intent Identifier	7
5. Intent: "charge"	7
6. Encoding Conventions	7
7. Request Schema	8
7.1. Shared Fields	8
7.2. Method Details	8
7.2.1. Native SOL Example	10
7.2.2. SPL Token Example	10
7.2.3. Fee Sponsorship Example	10
7.2.4. Payment Splits Example	11
8. Credential Schema	11
8.1. Transaction Payload — Pull Mode	11
8.2. Signature Payload — Push Mode	12
8.3. Limitations of Push Mode	13
9. Fee Sponsorship	13
9.1. Server-Paid Fees	13
9.2. Client-Paid Fees	14
9.3. Server Requirements	14
9.4. Client Requirements	14

10. Verification Procedure	14
10.1. Pull Mode Verification	14
10.2. Push Mode Verification	15
10.3. Native SOL Verification	16
10.4. SPL Token Verification	16
10.5. Replay Protection	16
11. Settlement Procedure	17
11.1. Pull Mode Settlement (type="transaction")	17
11.2. Push Mode Settlement (type="signature")	18
11.3. Client Transaction Construction	19
11.3.1. Native SOL	19
11.3.2. SPL Tokens	19
11.3.3. Fee Payer Configuration	19
11.4. Confirmation Requirements	20
11.5. Finality	20
11.6. Receipt Generation	20
12. Error Responses	21
13. Security Considerations	21
13.1. Transport Security	21
13.2. Replay Protection Considerations	21
13.3. Client-Side Verification	22
13.4. RPC Trust	22
13.5. Front-running (Push Mode)	22
13.6. Fee Payer Risks	23
13.7. Transaction Payload Security	23
13.8. Blockhash Freshness	23
14. IANA Considerations	24
14.1. Payment Method Registration	24
14.2. Payment Intent Registration	24

15. References	24
15.1. Normative References	24
15.2. Informative References	25
Appendix A. Examples	25
A.1. Native SOL Charge (Pull Mode)	25
A.2. SPL Token (USDC) Charge with Fee Sponsorship	27
A.3. Push Mode (type="signature")	28
A.4. Payment Splits	28
Appendix B. Acknowledgements	29
Authors' Addresses	29

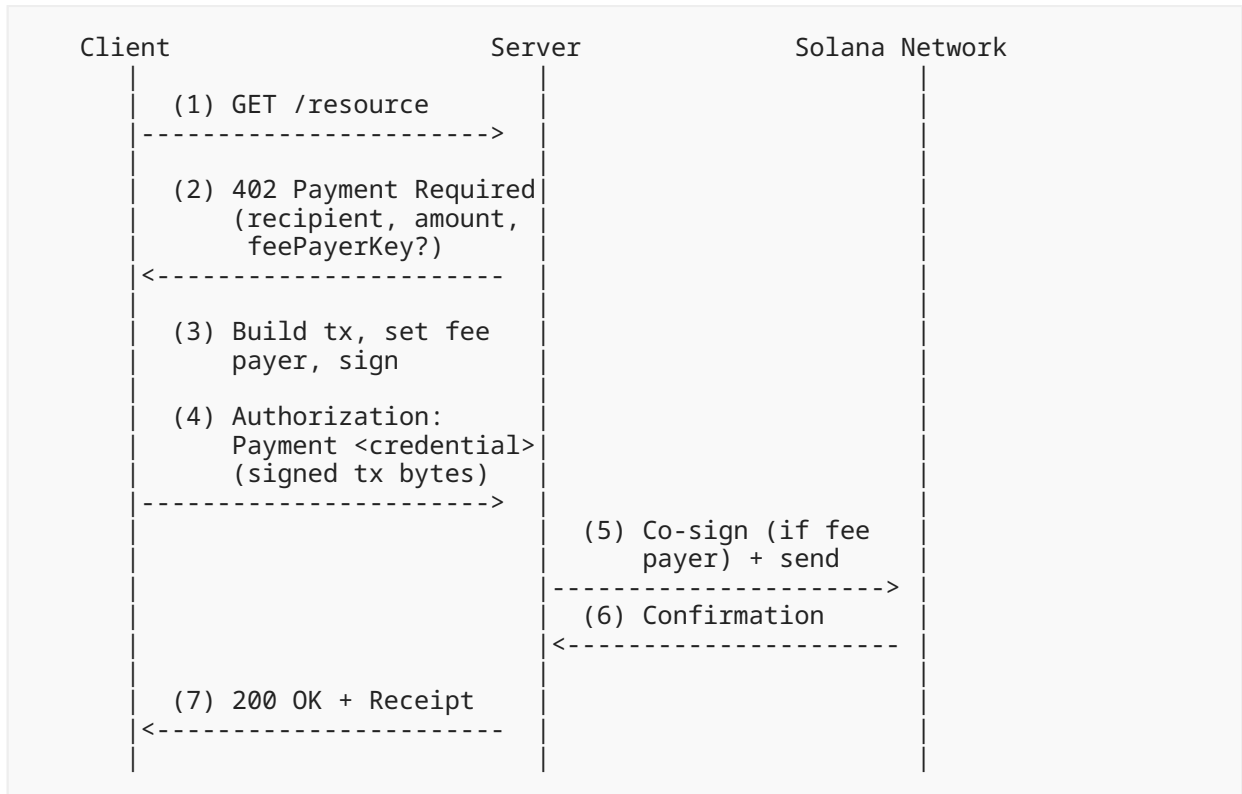
1. Introduction

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

Solana is a high-throughput blockchain with sub-second finality and low transaction fees [[SOLANA-DOCS](#)]. This specification supports payments in both native SOL and SPL tokens (including Token-2022 [[SPL-TOKEN-2022](#)]), making it suitable for micropayment use cases where fast confirmation and low overhead are important.

1.1. Pull Mode (Default)

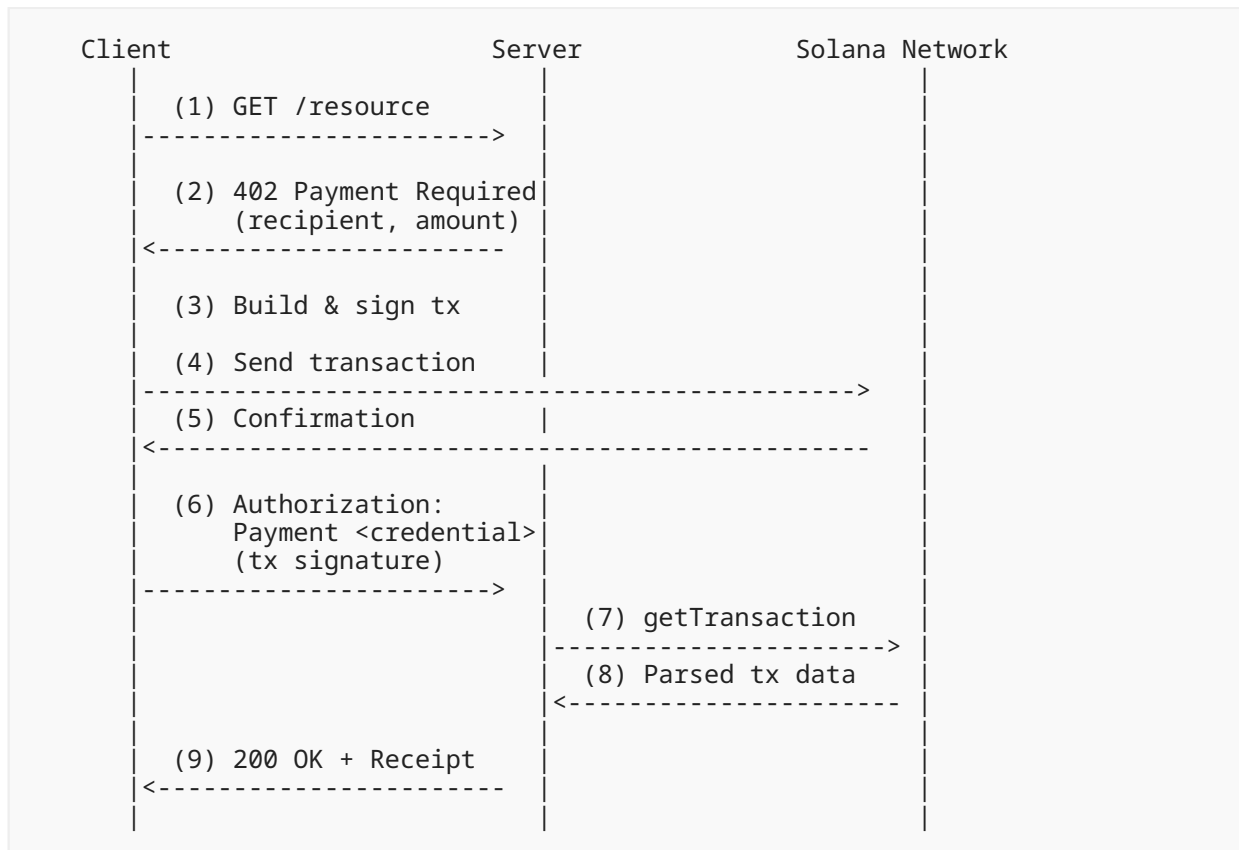
The default flow, called "pull mode", uses `type="transaction"` credentials. The client signs the transaction and the server "pulls" it for broadcast to the Solana network:



In this model the server controls transaction broadcast, enabling fee sponsorship (Section 9) and server-side retry logic. When `feePayer` is `true`, the challenge includes `feePayerKey` so the client sets the server as fee payer. The server co-signs with its fee payer key before broadcasting.

1.2. Push Mode (Fallback)

The fallback flow, called "push mode", uses `type="signature"` credentials. The client "pushes" the transaction to the network itself and presents the confirmed signature. The client broadcasts the transaction itself and presents the confirmed transaction signature:



This flow is useful when the client cannot or does not wish to delegate broadcast to the server. The server verifies the payment by fetching and inspecting the on-chain transaction via RPC.

1.3. Relationship to the Charge Intent

This document inherits the shared request semantics of the "charge" intent from [I-D.payment-intent-charge]. It defines only the Solana-specific methodDetails, payload, and verification procedures for the "solana" payment method.

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

Transaction Signature

A base58-encoded [BASE58] unique identifier for a Solana transaction, produced by the first signer. Serves as both the transaction identifier and proof of payment in this specification.

SPL Token A fungible token on Solana conforming to the SPL Token program [SPL-TOKEN] or the Token-2022 program [SPL-TOKEN-2022].

Associated Token Account (ATA) A deterministically derived token account for a given owner and mint, per the Associated Token Program. The address is a Program Derived Address (PDA) seeded by the owner's public key, the token mint, and the token program ID.

Lamports The smallest unit of native SOL. 1 SOL = 1,000,000,000 lamports.

Base Units The smallest transferable unit of an SPL token, determined by the token's decimal precision. For example, USDC uses 6 decimals, so 1 USDC = 1,000,000 base units.

Fee Payer An account that pays Solana transaction fees. When the server acts as fee payer, it adds its signature to the transaction before broadcasting, covering the transaction fee on behalf of the client.

Pull Mode The default settlement flow where the client signs the transaction and the server broadcasts it (`type="transaction"`). The server "pulls" the signed transaction from the credential. Enables fee sponsorship and server-side retry logic.

Push Mode The fallback settlement flow where the client broadcasts the transaction itself and presents the confirmed signature (`type="signature"`). The client "pushes" the transaction to the network directly. Cannot be used with fee sponsorship.

4. Intent Identifier

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

5. Intent: "charge"

The "charge" intent represents a one-time payment gating access to a resource. The client builds and signs a Solana transfer transaction, then either sends the signed transaction bytes to the server for broadcast (`type="transaction"`) or broadcasts the transaction itself and sends the on-chain signature (`type="signature"`). The server verifies the transfer details and returns a receipt.

6. 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.

7. Request Schema

7.1. Shared Fields

The request auth-param of the WWW-Authenticate: Payment header contains a JCS-serialized, base64url-encoded JSON object (see Section 6). The following shared fields are included in that object:

amount **REQUIRED**. The payment amount in base units, encoded as a decimal string. For native SOL, the amount is in lamports. For SPL tokens, the amount is in the token's smallest unit (e.g., for USDC with 6 decimals, "1000000" represents 1 USDC). The value **MUST** be a positive integer that fits in a 64-bit unsigned integer (max 18,446,744,073,709,551,615).

currency **REQUIRED**. For native SOL, **MUST** be the lowercase string "sol". For SPL tokens, **MUST** be the base58-encoded mint address (e.g., "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v" for USDC). The mint address uniquely identifies the token and is used by the client to construct the transfer instruction. **MUST NOT** exceed 128 characters.

description **OPTIONAL**. A human-readable memo describing the resource or service being paid for. **MUST NOT** exceed 256 characters.

recipient **REQUIRED**. The base58-encoded public key of the account receiving the payment. For native SOL transfers, this is the destination account. For SPL token transfers, this is the owner of the destination associated token account, not the ATA address itself.

externalId **OPTIONAL**. Merchant's reference (e.g., order ID, invoice number), per [I-D.payment-intent-charge]. May be used for reconciliation or idempotency. **MUST NOT** exceed 566 bytes (Solana Memo Program limit). When present, clients **SHOULD** include this value as a Memo Program instruction in the transaction, making it visible on-chain for auditing and reconciliation. Servers **MAY** verify the memo matches the externalId from the challenge.

7.2. Method Details

The following fields are nested under methodDetails in the request JSON:

network **OPTIONAL**. Identifies which Solana cluster the payment should be made on. **MUST** be one of "mainnet", "devnet", or "localnet". Defaults to "mainnet" if omitted. Clients **MUST** reject challenges whose network does not match their configured cluster.

`decimals` Conditionally **REQUIRED**. The number of decimal places for the token (0–9). **MUST** be present when currency is a mint address; **MUST** be absent when currency is "sol". Used by the client to construct a `TransferChecked` instruction.

`tokenProgram` **OPTIONAL**. The base58-encoded program ID of the token program governing the token. **MUST** be either the Token Program (TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA) or the Token-2022 Program (TokenzQdBNbLqP5VEhdkAS6EPFLC1PHnBqCXEpPxuEb). If omitted, clients **MUST** determine the correct token program by fetching the mint account from the network and inspecting its owner program. If that lookup fails, returns an unexpected owner, or cannot be verified, clients **MUST** reject the challenge rather than falling back to the Token Program. Servers **SHOULD** include this field as a hint to avoid the extra RPC lookup. **MUST NOT** be present when currency is "sol".

`feePayer` **OPTIONAL**. A boolean indicating whether the server will pay transaction fees on behalf of the client. Defaults to `false` if omitted. When `true`, the `feePayerKey` field **MUST** also be present. See [Section 9](#).

`feePayerKey` Conditionally **REQUIRED**. The base58-encoded public key of the server's fee payer account. **MUST** be present when `feePayer` is `true`; **MUST** be absent when `feePayer` is `false` or omitted. The client uses this key as the transaction fee payer when constructing the transaction.

`splits` **OPTIONAL**. An array of at most 8 additional payment splits. Each entry is a JSON object with the following fields:

- `recipient` (**REQUIRED**): Base58-encoded public key of the split recipient.
- `amount` (**REQUIRED**): Amount in the same base units and asset as the primary amount.
- `memo` (**OPTIONAL**): Human-readable label for this split (e.g., "platform fee", "referral"). **MUST NOT** exceed 566 bytes (Solana Memo Program limit).

When present, the client **MUST** include a transfer instruction for each split in addition to the primary transfer to `recipient`. All splits use the same asset as the primary payment (native SOL or the token from currency).

The top-level amount is the total the client pays. The sum of all split amounts **MUST NOT** exceed amount. The primary `recipient` receives amount minus the sum of all split amounts; this remainder **MUST** be greater than zero. Servers **MUST** reject challenges where splits consume the entire amount. Servers **MUST** verify each split transfer on-chain during credential verification. If the same recipient appears more than once in `splits`, each occurrence is a distinct payment leg and **MUST** be verified separately; servers **MUST NOT** implicitly aggregate such entries.

This mechanism is a Solana-specific extension to the base charge intent. It can be used for fee payer cost recovery, platform fees, revenue sharing, or referral commissions.

recentBlockhash **OPTIONAL**. A base58-encoded recent blockhash for the client to use when constructing the transaction. When provided, clients **SHOULD** use this blockhash instead of fetching one from an RPC node. This avoids an extra RPC round-trip and ensures the server can verify blockhash freshness. This field is advisory and short-lived; it **MUST NOT** be assumed to remain valid for the full lifetime of the payment challenge. If omitted, clients **MUST** fetch a recent blockhash themselves.

7.2.1. Native SOL Example

```
{
  "amount": "10000000",
  "currency": "sol",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Weather API access",
  "methodDetails": {
    "network": "mainnet"
  }
}
```

This requests a transfer of 0.01 SOL (10,000,000 lamports).

7.2.2. SPL Token Example

```
{
  "amount": "1000000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Premium API call",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "tokenProgram": "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA"
  }
}
```

This requests a transfer of 1 USDC (1,000,000 base units).

7.2.3. Fee Sponsorship Example

```
{
  "amount": "10000000",
  "currency": "sol",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Weather API access",
  "methodDetails": {
    "network": "mainnet",
    "feePayer": true,
    "feePayerKey": "9aE3Fg7HjKlMnOpQr5TuVwXyZ2AbCdEf8GhIjKlMnOp1R"
  }
}
```

This requests a transfer of 0.01 SOL where the server pays transaction fees.

7.2.4. Payment Splits Example

```
{
  "amount": "1050000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Marketplace purchase",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "splits": [
      { "recipient": "3pF8Kg2aHbNvJKLMwEqR7YtDxZ5sGhJn4UV6mWcXrT9A",
        "amount": "50000", "memo": "platform fee" }
    ]
  }
}
```

This requests a total payment of 1.05 USDC. The platform receives 0.05 USDC and the primary recipient (seller) receives 1.00 USDC.

8. Credential Schema

The Authorization header carries a single base64url-encoded JSON token (no auth-params). The decoded object contains the following top-level fields:

challenge **REQUIRED**. An echo of the challenge auth-params from the WWW-Authenticate header: id, realm, method, intent, request, and (if present) expires. This binds the credential to the exact challenge that was issued.

source **OPTIONAL**. A payer identifier string, as defined by [I-D.httpauth-payment]. Solana implementations **MAY** use the payer's base58-encoded public key or a DID.

payload **REQUIRED**. A JSON object containing the Solana-specific credential fields. The type field determines which additional fields are present. Two payload types are defined: "transaction" (default) and "signature" (fallback).

8.1. Transaction Payload — Pull Mode

In pull mode (type="transaction"), the client sends the signed transaction bytes to the server for broadcast. The transaction field contains the base64-encoded serialized signed transaction.

Field	Type	Required	Description
type	string	REQUIRED	"transaction"

Field	Type	Required	Description
transaction	string	REQUIRED	Base64-encoded serialized signed transaction bytes (max 1232 bytes decoded)

Table 1

The transaction **MUST** be a valid Solana versioned transaction that does not exceed the 1232-byte transaction size limit. containing the transfer instruction(s) matching the challenge parameters. The client **MUST** sign the transaction with the transfer authority key. When `feePayer` is false or absent, the client **MUST** also be the fee payer and the transaction **MUST** be fully signed. When `feePayer` is true, the transaction **MUST** set the server's `feePayerKey` as fee payer, and the client signs only as transfer authority; the server adds the fee payer signature before broadcasting (see [Section 9](#)).

Example (decoded):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "solana",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "transaction",
    "transaction": "AQAAAA...base64-encoded-signed-tx..."
  }
}
```

8.2. Signature Payload — Push Mode

In push mode (`type="signature"`), the client has already broadcast the transaction to the Solana network. The `signature` field contains the base58-encoded transaction signature for the server to verify on-chain.

Field	Type	Required	Description
type	string	REQUIRED	"signature"
signature	string	REQUIRED	Base58-encoded Solana transaction signature

Table 2

Example (decoded):

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "solana",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "signature",
    "signature": "5UfDuX7hXbPjGUpTmt9PHRLsNGJe4dEny..."
  }
}
```

8.3. Limitations of Push Mode

The type="signature" credential has the following limitations:

- **MUST NOT** be used when `feePayer` is `true` in the challenge request. Since the client has already broadcast the transaction, the server cannot add its fee payer signature. Servers **MUST** reject type="signature" credentials when the challenge specifies `feePayer: true`.
- The server cannot modify or enhance the transaction (e.g., add priority fees, adjust compute units, or retry with different parameters).

9. Fee Sponsorship

When a challenge includes `feePayer: true` in `methodDetails`, the server commits to paying Solana transaction fees on behalf of the client. This section describes the fee sponsorship mechanism.

9.1. Server-Paid Fees

When `feePayer` is `true`:

1. **Client constructs transaction:** The client builds the transfer transaction with the server's `feePayerKey` set as the transaction fee payer. The client's account is the transfer authority but NOT the fee payer.
2. **Client partially signs:** The client signs the transaction with only its own key (the transfer authority). The fee payer signature slot remains empty.
3. **Client sends credential:** The client sends the partially signed transaction as a type="transaction" credential.
4. **Server adds fee payer signature:** The server verifies the transaction contents, then signs with the fee payer key to complete the transaction.
5. **Server broadcasts:** The fully signed transaction (containing both the client's transfer authority signature and the server's fee payer signature) is broadcast to the Solana network.

9.2. Client-Paid Fees

When `feePayer` is `false` or omitted, the client **MUST** set itself as the fee payer and fully sign the transaction. The server broadcasts the transaction as-is without adding any signatures.

9.3. Server Requirements

When acting as fee payer, servers:

- **MUST** maintain sufficient SOL balance in the fee payer account to cover transaction fees
- **MUST** verify the transaction contents before signing (see [Section 10.1](#))
- **SHOULD** implement rate limiting to mitigate fee exhaustion attacks (see [Section 13.6](#))

9.4. Client Requirements

- When `feePayer` is `true`: clients **MUST** set `feePayerKey` from `methodDetails` as the transaction fee payer and **MUST** sign only with the transfer authority key. Clients **MUST** use `type="transaction"` credentials.
- When `feePayer` is `false` or omitted: clients **MUST** set themselves as the fee payer and fully sign the transaction. Clients **MAY** use either `type="transaction"` or `type="signature"` credentials.

10. Verification Procedure

Upon receiving a request with a credential, the server **MUST**:

1. Decode the `base64url` credential and parse the JSON.
2. Verify that `payload.type` is present and is either `"transaction"` or `"signature"`.
3. Look up the stored challenge using `credential.challenge.id`. If no matching challenge is found, reject the request.
4. Verify that all fields in `credential.challenge` exactly match the stored challenge `auth-params`.
5. If `payload.type` is `"signature"` and the challenge specifies `feePayer: true`, reject the request (see [Section 8.3](#)).
6. Proceed with type-specific verification:
 - For `type="transaction"`: see [Section 10.1](#).
 - For `type="signature"`: see [Section 10.2](#).

10.1. Pull Mode Verification

For credentials with `type="transaction"`:

1. Decode the `base64 payload.transaction` value.

2. Deserialize the transaction and verify that it structurally matches the challenge request:
 - the fee payer matches the challenge policy;
 - the transfer authority is signed by the client;
 - the transaction contains only expected transfer, ATA-creation, memo, and compute-budget instructions;
 - the payment semantics match the challenge request, as described in [Section 10.3](#) or [Section 10.4](#).
3. If `feePayer` is true, add the server's fee payer signature using the `feePayerKey` and re-serialize. The transaction **MUST** have the server's `feePayerKey` set as the fee payer account.
4. If `feePayer` is true, simulate the transaction using the `simulateTransaction` RPC method. The server **MUST** reject the credential if simulation fails. If `feePayer` is false or omitted, the server **SHOULD** simulate the transaction before broadcast and **SHOULD** reject the credential if simulation indicates the transaction will fail. This catches invalid transactions without spending fees, which is especially important in fee payer mode (see [Section 13.6](#)).
5. Broadcast the transaction to the Solana network using `sendTransaction`.
6. Wait for confirmation at the required commitment level.
7. Fetch the confirmed transaction using `getTransaction` with `jsonParsed` encoding and verify the transfer details still match the challenge request, as described in [Section 10.3](#) or [Section 10.4](#).
8. Record the transaction signature as consumed to prevent replay (see [Section 10.5](#)).
9. Return the resource with a Payment-Receipt header.

10.2. Push Mode Verification

For credentials with `type="signature"`:

1. Verify that `payload.signature` is present and is a valid base58-encoded string.
2. Verify the transaction signature has not been previously consumed (see [Section 10.5](#)).
3. Fetch the transaction from the Solana network using the RPC `getTransaction` method with `jsonParsed` encoding and the `confirmed` commitment level.
4. Verify the transaction was successful (no error in the transaction metadata).
5. Verify the transfer details match the challenge request, as described in [Section 10.3](#) or [Section 10.4](#).
6. Mark the transaction signature as consumed to prevent replay.
7. Return the resource with a Payment-Receipt header.

Note: both credential types reuse the same on-chain transfer verification logic defined in [Section 10.3](#) and [Section 10.4](#).

10.3. Native SOL Verification

For native SOL payments (currency is "sol"), the server **MUST**:

1. Compute the primary payment amount as the top-level amount minus the sum of all splits, if any.
2. Locate a System Program transfer instruction in the transaction's parsed instructions whose destination matches the top-level recipient and whose lamports field matches that primary payment amount.
3. For each split in splits, if any, locate an additional System Program transfer instruction whose destination and lamports fields match that split.

Each required payment leg **MUST** be matched to a distinct transfer instruction. A single transfer instruction **MUST NOT** satisfy more than one required payment leg, even if multiple legs share the same recipient.

If any required transfer instruction is missing, the server **MUST** reject the credential.

10.4. SPL Token Verification

For SPL token payments (currency is a mint address, not "sol"), the server **MUST**:

1. Compute the primary payment amount as the top-level amount minus the sum of all splits, if any.
2. Locate a transferChecked instruction from the appropriate token program (Token Program or Token-2022) in the transaction's parsed instructions whose mint field matches the top-level currency field from the challenge request.
3. Derive the expected destination associated token account for the top-level recipient from the recipient, currency, and tokenProgram in the challenge request. Verify that at least one matching transferChecked instruction uses that derived ATA as destination and has tokenAmount.amount equal to the primary payment amount.
4. For each split in splits, if any, derive the expected destination ATA for that split recipient and verify that at least one additional transferChecked instruction uses that ATA as destination and has tokenAmount.amount equal to the split amount.

Each required payment leg **MUST** be matched to a distinct transferChecked instruction. A single instruction **MUST NOT** satisfy more than one required payment leg, even if multiple legs resolve to the same destination ATA.

If any required transferChecked instruction is missing, the server **MUST** reject the credential.

10.5. Replay Protection

Servers **MUST** maintain a set of consumed transaction signatures. Before accepting a credential, the server **MUST** check whether the signature has already been consumed. After successful verification, the server **MUST** atomically mark the signature as consumed.

The transaction signature is globally unique on the Solana network, making it a natural replay prevention token. A signature that has been consumed **MUST NOT** be accepted again, even if presented with a different challenge ID.

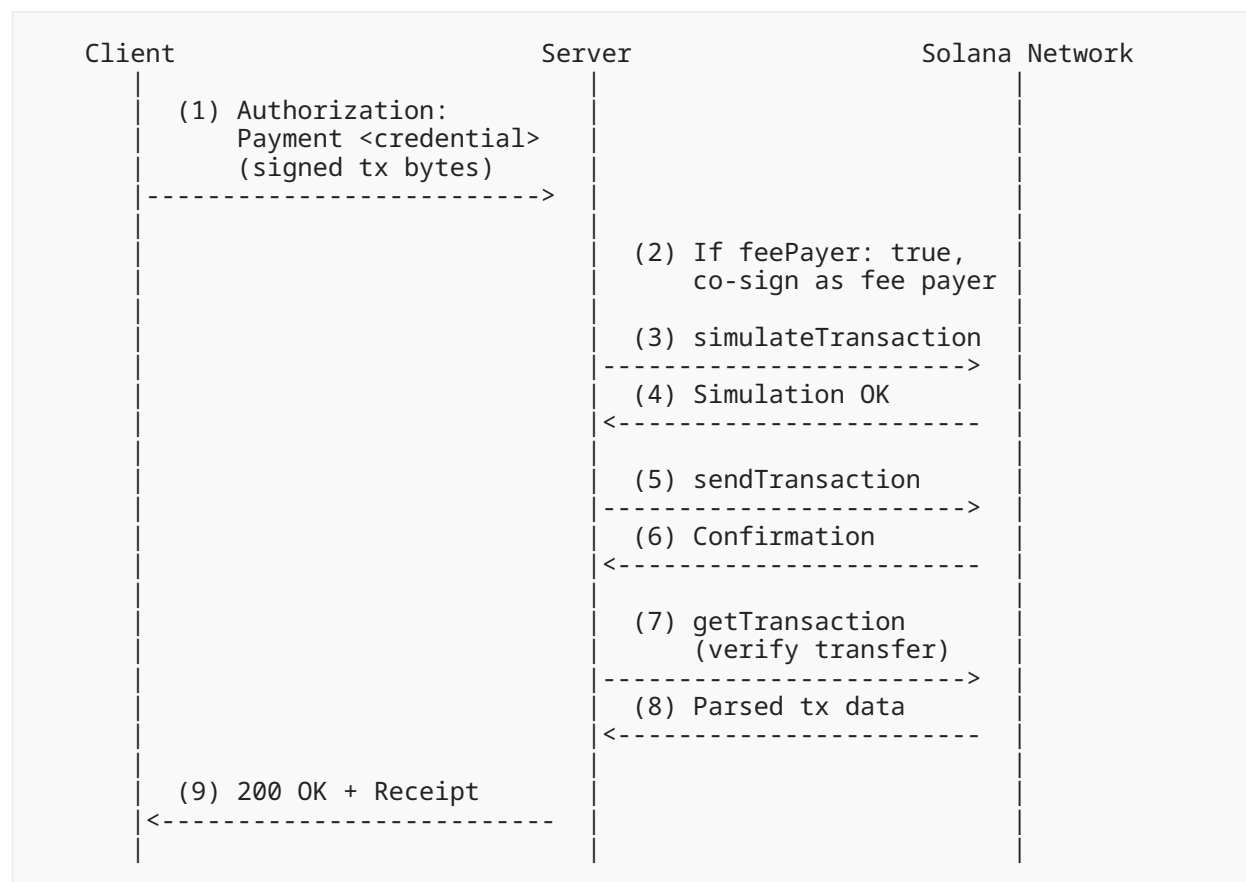
For `type="transaction"` credentials, the transaction signature is derived after broadcast. For `type="signature"` credentials, the signature is provided directly by the client.

11. Settlement Procedure

Two settlement flows are supported, corresponding to the two credential types.

11.1. Pull Mode Settlement (`type="transaction"`)

For `type="transaction"` credentials, the client signs the transaction and sends it to the server. The server optionally adds a fee payer signature and broadcasts:

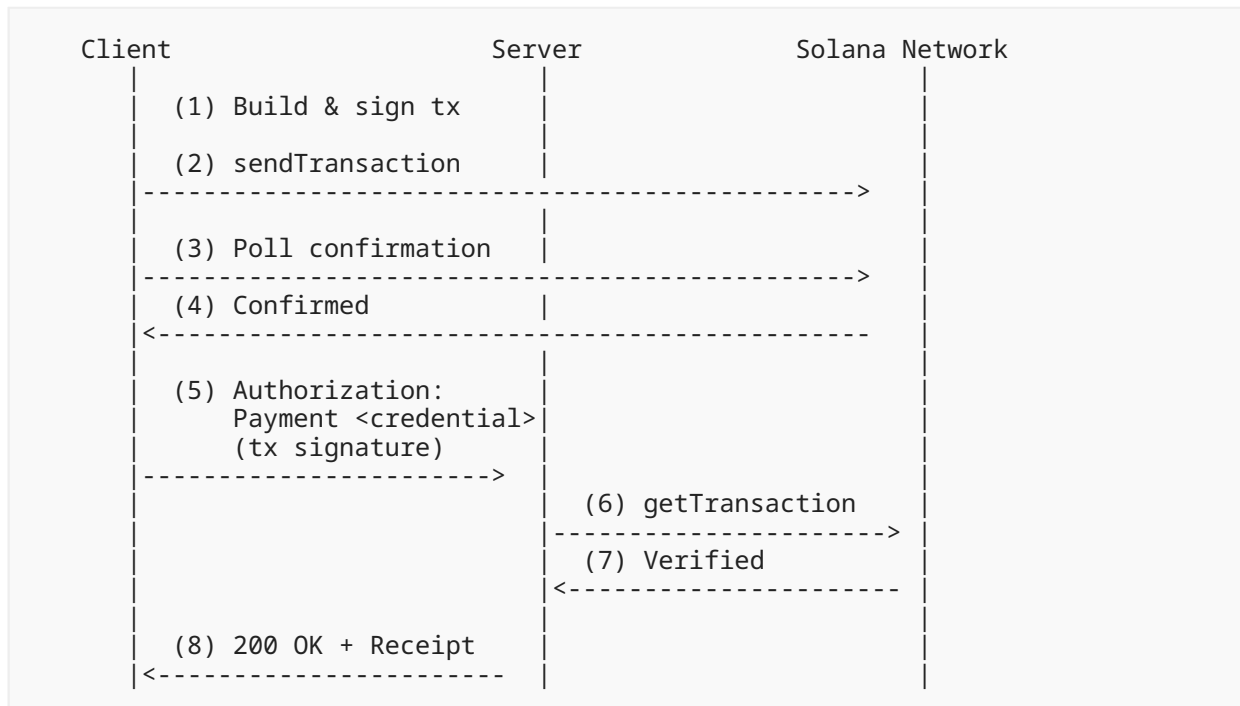


1. Client submits credential containing signed transaction bytes.
2. If `feePayer` is `true`, the server co-signs with its fee payer key.
3. Server simulates the transaction to catch failures without spending fees.
4. Server broadcasts the transaction to Solana.

5. Transaction reaches the required commitment level.
6. Server fetches the confirmed transaction and verifies the transfer details match the challenge request.
7. Server records the signature as consumed and returns the resource with a Payment-Receipt header whose reference field is the transaction signature.

11.2. Push Mode Settlement (type="signature")

For type="signature" credentials, the client broadcasts the transaction itself and presents the confirmed signature:



1. Client builds a transfer transaction and signs it.
2. Client sends the transaction to the Solana network.
3. Client polls for confirmation status.
4. Transaction reaches confirmed commitment level.
5. Client presents the transaction signature as the credential.
6. Server fetches the transaction via RPC and verifies transfer details.
7. Server confirms the payment matches the challenge.
8. Server returns the resource with a Payment-Receipt.

11.3. Client Transaction Construction

11.3.1. Native SOL

The client **MUST** construct a transaction containing a System Program transfer instruction with:

- `source`: the client's signing account
- `destination`: the recipient from the challenge
- `lamports`: the amount from the challenge

11.3.2. SPL Tokens

The client **MUST** construct a transaction containing:

1. An idempotent Associated Token Account creation instruction for the recipient's ATA, ensuring payment succeeds even if the recipient has never held the token. The payer covers the rent-exempt minimum (~0.002 SOL) if the account does not exist.
2. A `transferChecked` instruction on the appropriate token program with:
 - `source`: the client's associated token account
 - `mint`: the currency field
 - `destination`: the recipient's derived ATA
 - `authority`: the client's signing account
 - `amount`: the amount from the challenge
 - `decimals`: the decimals from `methodDetails`

11.3.3. Fee Payer Configuration

When `feePayer` is true in the challenge:

- The client **MUST** set the server's `feePayerKey` as the transaction fee payer.
- The client **MUST** sign the transaction only with its own key (transfer authority).
- The fee payer signature slot **MUST** be left empty for the server to fill.

When `feePayer` is false or absent:

- The client **MUST** set itself as the transaction fee payer.
- The client **MUST** fully sign the transaction.

Clients **SHOULD** set a compute unit limit and priority fee appropriate for current network conditions.

11.4. Confirmation Requirements

For type="signature" credentials, clients **MUST** wait for at least the confirmed commitment level before presenting the credential. Servers **MUST** fetch the transaction with at least confirmed commitment. Servers **MAY** require finalized commitment for high-value transactions.

For type="transaction" credentials, the server controls the broadcast and confirmation process. Servers **MUST** wait for at least confirmed commitment before returning the receipt.

11.5. Finality

Solana provides two commitment levels relevant to payment verification:

- **confirmed**: optimistic confirmation from a supermajority of validators (~400ms). Sufficient for most payment use cases.
- **finalized**: deterministic finality after ~31 slots (~12 seconds). Required for high-value transactions where rollback risk is unacceptable.

In theory, a confirmed transaction could be rolled back if validators shift consensus to a competing fork that excludes the confirmed block. In practice, this has never occurred on Solana mainnet. The confirmed level is **RECOMMENDED** as the default for payment verification to minimize latency.

11.6. Receipt Generation

Upon successful verification, the server **MUST** include a Payment-Receipt header in the 200 response.

The receipt payload for Solana charge:

Field	Type	Description
method	string	"solana"
challengeId	string	The challenge id from WWW-Authenticate
reference	string	The transaction signature (base58-encoded)
status	string	"success"
timestamp	string	[RFC3339] verification time

Table 3

Example (decoded):

```
{
  "method": "solana",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "5UfDuX7hXbPjGUpTmt9PHRLsNGJe4dEny...",
  "status": "success",
  "timestamp": "2026-03-10T21:00:00Z"
}
```

12. 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. Servers **MUST** use the standard problem types defined in [I-D.httpauth-payment]: malformed-credential, invalid-challenge, and verification-failed. The detail field **SHOULD** contain a human-readable description of the specific failure (e.g., "Transaction not found", "Amount mismatch", "Signature already consumed").

All error responses **MUST** include a fresh challenge in WWW-Authenticate.

Example error response body:

```
{
  "type": "https://paymentauth.org/problems/verification-failed",
  "title": "Transfer Mismatch",
  "status": 402,
  "detail": "Destination token account does not belong to expected recipient"
}
```

13. Security Considerations

13.1. Transport Security

All communication **MUST** use TLS 1.2 or higher. Solana credentials **MUST** only be transmitted over HTTPS connections.

13.2. Replay Protection Considerations

Servers **MUST** track consumed transaction signatures and reject any signature that has already been accepted. The check-and-consume operation **MUST** be atomic to prevent race conditions where concurrent requests present the same signature. Transaction signatures are globally unique on the Solana network (derived from the signer's key and the blockhash), making them natural replay prevention tokens.

13.3. Client-Side Verification

Clients **MUST** verify the challenge before signing:

1. `amount` is reasonable for the service
2. `currency` matches the expected asset
3. `recipient` is the expected party
4. If `currency` is a mint address, verify it is a known token
5. `splits`, if present, contain expected recipients and amounts — malicious servers could add splits to redirect funds
6. `feePayerKey`, if present, is the expected server

Malicious servers could request excessive amounts, direct payments to unexpected recipients, or add hidden splits.

13.4. RPC Trust

The server relies on its Solana RPC endpoint to provide accurate transaction data for on-chain verification. A compromised RPC could return fabricated transaction data, causing the server to accept payments that were never made. Servers **SHOULD** use trusted RPC providers or run their own nodes.

13.5. Front-running (Push Mode)

In push mode, the client broadcasts the transaction before presenting the credential, making it visible on-chain. A party monitoring the chain could attempt to present the same signature to the server. The challenge binding (the credential echoes the challenge `id`, which is HMAC-verified) and single-use signature enforcement mitigate this: only the party that received the challenge can construct a valid credential.

Push mode does not require the on-chain transaction to carry a challenge-specific marker. It proves that a payment matching the challenged terms was made, but not necessarily that the payment was created for one unique challenge instance. If multiple valid challenges have identical terms, the same confirmed transaction could satisfy any one of them, and the first accepted presentation wins.

Requiring an on-chain marker such as a Memo carrying the challenge `id` would provide stronger binding, but would also reveal extra correlation metadata on chain. This specification does not require such a marker in the base flow, but implementations **MAY** define a backward-compatible profile that does.

Pull mode is not susceptible to front-running because the transaction is not broadcast until the server receives and validates the credential.

13.6. Fee Payer Risks

Servers acting as fee payers accept financial risk in exchange for providing a seamless payment experience.

Denial of Service via Bad Transactions Malicious clients could submit transactions that fail on-chain (insufficient balance, invalid instructions), causing the server to pay ~5,000 lamports per failed transaction. Mitigations:

- **Transaction simulation:** `simulateTransaction` catches most failures before broadcast, without spending fees. Servers **MUST** simulate fee-sponsored pull mode transactions before broadcasting. Servers **SHOULD** simulate non-fee-sponsored pull mode transactions before broadcasting.
- **Rate limiting:** per client address, per IP, or per time window.
- **Balance verification:** check the client's balance covers the transfer amount before signing.
- **Client authentication:** require API keys or OAuth tokens before accepting fee-sponsored transactions.

ATA Rent Drain When the fee payer funds creation of an Associated Token Account (ATA), it pays ~0.002 SOL in rent. The recipient can close the ATA to reclaim rent, then the next payment re-creates it at the fee payer's expense. Servers **SHOULD** verify the recipient's ATA exists before co-signing, or factor rent cost into the payment amount via `splits`.

Fee Payer Balance Exhaustion Servers **MUST** monitor fee payer balance and reject new fee-sponsored requests when insufficient. The server **SHOULD** return a 402 with `feePayer: false`, allowing the client to pay its own fees as fallback.

13.7. Transaction Payload Security

In pull mode, the server receives raw transaction bytes from the client. A malicious client could craft a transaction that transfers funds FROM the server's fee payer account rather than simply paying fees.

Servers **MUST** verify that the transaction contains only the expected instructions: transfer instruction(s) matching the challenge parameters, ATA creation (idempotent), and optionally compute budget instructions. Any unexpected instructions **MUST** cause rejection.

13.8. Blockhash Freshness

When the server provides `recentBlockhash` in the challenge, clients **SHOULD** verify it is plausible (not obviously stale). A malicious server could provide an expired blockhash, causing the client to sign a transaction that will never land — wasting the signing effort. However, since the transaction is not broadcast by the client in pull mode, the practical risk is limited to a failed payment attempt that the client can retry.

14. IANA Considerations

14.1. Payment Method Registration

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

Method Identifier	Description	Reference
solana	Solana blockchain native SOL and SPL token transfer	This document

Table 4

14.2. 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
charge	solana	One-time SOL or SPL token transfer	This document

Table 5

15. References

15.1. 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>>.

[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>>.

[I-D.payment-intent-charge] Moxey, J., Ryan, B., and T. Meagher, "'charge' Intent for HTTP Payment Authentication", 2026, <<https://datatracker.ietf.org/doc/draft-payment-intent-charge/>>.

[I-D.httpauth-payment] Moxey, J., "The 'Payment' HTTP Authentication Scheme", January 2026, <<https://datatracker.ietf.org/doc/draft-ryan-httpauth-payment/>>.

15.2. Informative References

[SOLANA-DOCS] Solana Foundation, "Solana Documentation", 2026, <<https://solana.com/docs>>.

[SPL-TOKEN] Solana Foundation, "SPL Token Program", 2026, <<https://solana.com/docs/tokens>>.

[SPL-TOKEN-2022] Solana Foundation, "SPL Token-2022 Program", 2026, <<https://solana.com/docs/tokens/extensions>>.

[BASE58] Sporny, M., "Base58 Encoding Scheme", 2023, <<https://datatracker.ietf.org/doc/html/draft-msporny-base58-03>>.

Appendix A. Examples

The following examples illustrate the complete HTTP exchange for each flow. Base64url values are shown with their decoded JSON below.

A.1. Native SOL Charge (Pull Mode)

A 0.01 SOL charge for weather API access.

1. Challenge (402 response):

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="kM9xPqWvT2nJrHsY4aDfEb",
  realm="api.example.com",
  method="solana",
  intent="charge",
  request="eyJhbW91bnQiOiIiXMDAwMDAwMCIsImN1cnJlbnN5Ij
    oiU09MIiwizGVzY3JpcHRpb24iOiJXZWZ0aGVyIEFQSSBhY2
    Nlc3MiLCJtZXRob2REZXRhaWxzIjp7Im5ldHdvcm50Ij01JmN1
    dHdYzEwYi01JmN1dHdYzEwYi01JmN1dHdYzEwYi01JmN1dHdYz
    LTQzNzItYTU2Ny0wZTAyYjJmM2Q0NzkiIj01JmN1dHdYzEwYi01
    joiN3hLWHRnMkNXODdkOTdUWEpTRHBiRDVqQmtoZVRxQTgzVF
    pSdUpvc2dBc1UifQ",
  expires="2026-03-15T12:05:00Z"
Cache-Control: no-store
```

Decoded request:

```
{
  "amount": "10000000",
  "currency": "sol",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Weather API access",
  "methodDetails": {
    "network": "mainnet"
  }
}
```

2. Credential (retry with signed transaction):

```
GET /weather HTTP/1.1
Host: api.example.com
Authorization: Payment <base64url-encoded credential>
```

Decoded credential:

```
{
  "challenge": {
    "id": "kM9xPqWvT2nJrHsY4aDfEb",
    "realm": "api.example.com",
    "method": "solana",
    "intent": "charge",
    "request": "<base64url-encoded request>",
    "expires": "2026-03-15T12:05:00Z"
  },
  "payload": {
    "type": "transaction",
    "transaction": "<base64-encoded signed transaction>"
  }
}
```

3. Response (with receipt):

```
HTTP/1.1 200 OK
Payment-Receipt: <base64url-encoded receipt>
Content-Type: application/json

{"temperature": 72, "condition": "sunny"}
```

Decoded receipt:

```
{
  "method": "solana",
  "challengeId": "kM9xPqWvT2nJrHsY4aDfEb",
  "reference": "4vJ9YFuPzUgdLkWYJf3KqfNM8cTnBp3jXx...",
  "status": "success",
  "timestamp": "2026-03-15T12:04:58Z"
}
```

A.2. SPL Token (USDC) Charge with Fee Sponsorship

A 1 USDC charge where the server sponsors transaction fees and includes a `recentBlockhash` to eliminate client RPC dependency.

Decoded request:

```
{
  "amount": "1000000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Premium API call",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "tokenProgram": "TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA",
    "feePayer": true,
    "feePayerKey": "Gh9ZwEmdLJ8DscKNTkTqPbNwLNNBjuSzaG9Vp2KGtKJr",
    "recentBlockhash": "EkSnNWid2cvwEVnVx9aBqawnmiCniDgp3gUdkDPTKN1N"
  }
}
```

The client uses `recentBlockhash` from the challenge (no RPC call needed), sets `feePayerKey` as the transaction fee payer, and partially signs with its own key only. The server verifies the transaction contents, co-signs as fee payer, and broadcasts.

Decoded credential:

```
{
  "challenge": { "...": "echoed challenge" },
  "payload": {
    "type": "transaction",
    "transaction": "<base64-encoded partially-signed tx>"
  }
}
```

A.3. Push Mode (type="signature")

The client broadcasts the transaction itself and presents the confirmed signature. Cannot be used with fee sponsorship.

Decoded credential:

```
{
  "challenge": { "...": "echoed challenge" },
  "payload": {
    "type": "signature",
    "signature": "4vJ9YFuPzUgdLkWyJf3KqfNM8cTnBp3jXx..."
  }
}
```

A.4. Payment Splits

A marketplace charge of 1.05 USDC where 0.05 USDC goes to the platform as a fee.

Decoded request:

```
{
  "amount": "1050000",
  "currency": "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v",
  "recipient": "7xKXtg2CW87d97TXJSDpbD5jBkheTqA83TZRuJosgAsU",
  "description": "Marketplace purchase",
  "methodDetails": {
    "network": "mainnet",
    "decimals": 6,
    "splits": [
      {
        "recipient": "3pF8Kg2aHbNvJKLMwEqR7YtDxZ5sGhJn4UV6mWcXrT9A",
        "amount": "50000",
        "memo": "platform fee"
      }
    ]
  }
}
```

The client builds a transaction with two transfers: 1,000,000 base units to the primary recipient and 50,000 to the platform. The total paid remains 1,050,000 base units, matching the top-level amount.

Appendix B. Acknowledgements

The authors thank the Tempo team for their input on this specification.

Authors' Addresses

Ludo Galabru

Solana Foundation

Email: ludo.galabru@solana.org

Ilan Gitter

Solana Foundation

Email: ilan.gitter@solana.org