Workgroup:	Network Working Group
Internet-Draft:	draft-nearintents-charge-00
Published:	30 June 2026
Intended Status:	Informational
Expires:	1 January 2027
Author:	I. Alustiza *Near One*

NEAR Intents Charge Intent for HTTP Payment Authentication

Abstract

This document defines the "charge" intent for the "nearintents" payment method in the Payment HTTP Authentication Scheme [I-D.httpauth-payment]. It specifies how clients and servers exchange one-time, cross-chain payments settled through the NEAR Intents 1Click Swap API [ONECLICK-API], in which a client pays a specified amount of a source asset on any supported origin chain and the resource server (merchant) receives an exact amount of a destination asset on any supported destination chain, with the NEAR Intents solver network executing the cross-chain swap in between.

One credential type is supported: `type="hash"` (push mode), where the client broadcasts the deposit transaction to the origin chain itself and presents the confirmed transaction hash for server verification. Because the 1Click backend issues a unique, single-use deposit address for each challenge, the recipient address is itself challenge-specific, which strengthens the challenge binding relative to generic hash-based credentials.

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 1 January 2027.

Copyright Notice

Table of Contents

1. Introduction 3
   1.1. Push Mode 4
   1.2. Cross-Chain Settlement Model 5
   1.3. Relationship to the Charge Intent 6
2. Requirements Language 6
3. Terminology 6
4. Request Schema 7
   4.1. Shared Fields 7
   4.2. Asset Identifiers 8
   4.3. Method Details 8
5. Credential Schema 10
   5.1. Credential Structure 10
   5.2. Hash Payload — Push Mode (type="hash") 11
6. Challenge Expiry and Deposit Deadline 11
7. Challenge Binding 12
8. Verification 12
9. Error Codes 13
10. Settlement Procedure 14
   10.1. Client Recovery 14

10.2. Settlement Finality 14
11. Receipt 15
12. Replay Protection 16
13. Security Considerations 16
13.1. Transport Security 16
13.2. Amount and Asset Verification 17
13.3. Hash Credential Binding 17
13.4. Deposit Address Authenticity 17
13.5. Trust Model 17
13.6. Idempotency and Side Effects 17
14. IANA Considerations 18
14.1. Payment Method Registration 18
14.2. Payment Intent Registration 18
15. References 18
15.1. Normative References 18
15.2. Informative References 19
Appendix A. ABNF Collected 20
Appendix B. Examples 20
B.1. Cross-Chain Charge — USDC on Arbitrum to merchant on NEAR 20
B.2. Native BTC origin 22
B.3. Multiple origin chains via Accept-Payment 23
Appendix C. Acknowledgements 23
Author's Address 23

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 "nearintents" payment method.

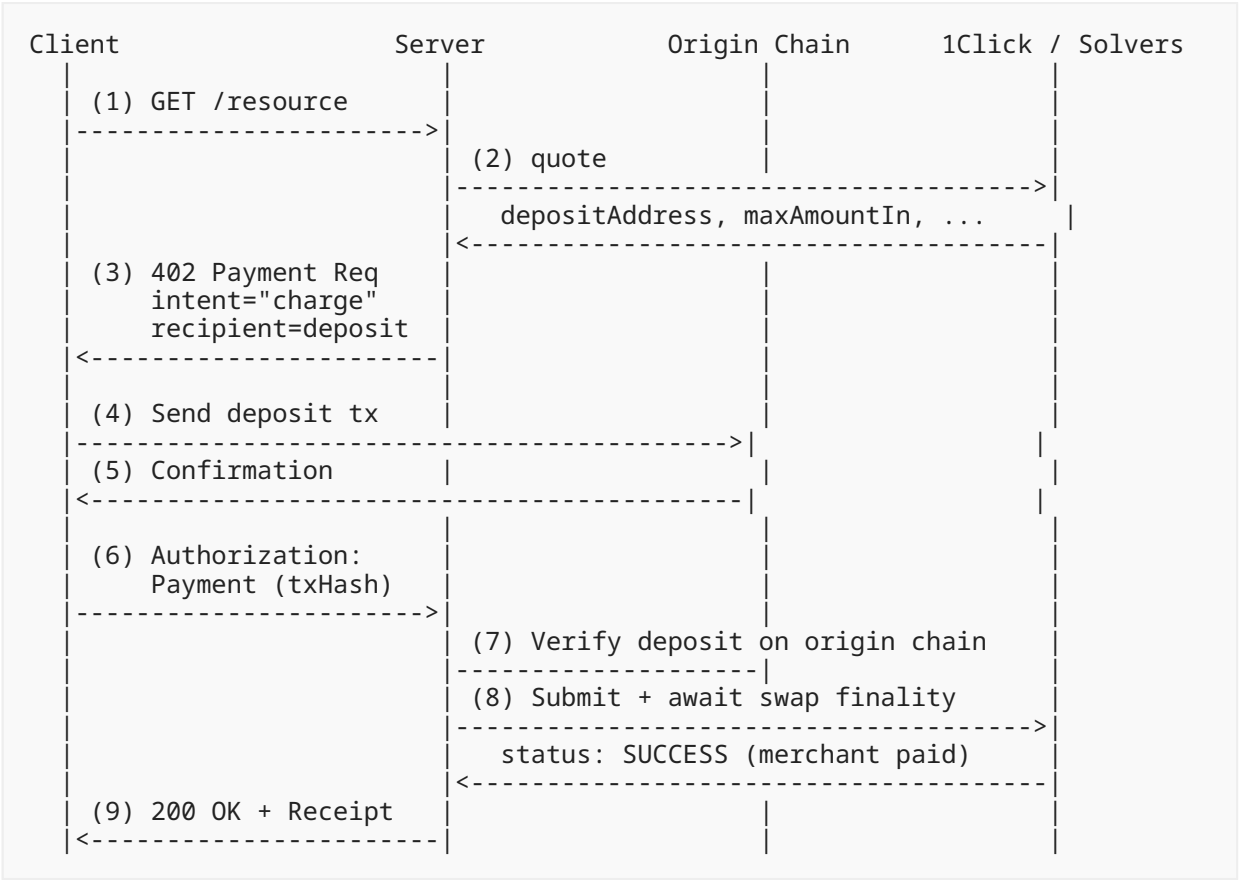
NEAR Intents is an intent-based settlement system in which a network of competing solvers fulfills user-expressed swap intents. The 1Click Swap API [ONECLICK-API] is a hosted interface to this system: a caller requests a quote for converting a source asset into a destination asset, receives a unique deposit address on the origin chain, sends the source asset to that address, and the solver network delivers the destination asset to a specified recipient — potentially on a different chain. A list of supported origin and destination chains is maintained at [NEAR-INTENTS-CHAINS].

This document specifies how to use that settlement system to fulfill the "charge" intent: a one-time payment of a specified amount, as defined in [I-D.payment-intent-charge]. The server may settle the payment any time before the challenge `expires` auth-param timestamp.

This specification inherits the shared request semantics of the "charge" intent from [I-D.payment-intent-charge]. It defines only the NEAR Intents-specific `methodDetails`, `payload`, verification, and settlement procedures for the "nearintents" payment method.

1.1. Push Mode

This method uses a single settlement flow, called "push mode", which uses `type="hash"` credentials. The client "pushes" the deposit transaction to the origin chain itself and presents the confirmed transaction hash; the server verifies the deposit and then drives the cross-chain swap to completion.



Unlike methods that offer a server-submitted ("pull") flow for fee sponsorship, this method has no pull mode: the source asset must be deposited to the 1Click address before settlement can begin, and the client therefore always pays its own origin-chain network fee. A future revision **MAY** define an additional credential type for clients that hold balances on the NEAR Intents settlement layer and wish to delegate submission of a signed intent; such a credential is out of scope for this document.

1.2. Cross-Chain Settlement Model

Single-chain charge methods describe one transfer: the client sends `amount` of `currency` to `recipient`, and the recipient receives exactly that. This method describes a payment whose two sides may diverge across chains and assets.

To remain compatible with the shared "charge" request schema and with push-mode verification, the standard fields describe the **payment the client makes on the origin chain**, and the **merchant's destination leg** is carried in `methodDetails`:

- `recipient` is the 1Click **deposit address** on the origin chain. It is the payee of the client's on-chain transfer and the address whose receipt of funds the server verifies from the transaction hash. It is controlled by the NEAR Intents settlement system for the duration of the swap, not by the merchant.

- `currency` and `amount` describe the **source asset** and the **deposit amount** the client sends to `recipient`.
- `methodDetails.originNetwork` is the **origin chain** where `recipient` lives and where the transaction hash is anchored.
- `methodDetails.destinationNetwork`, `methodDetails.destinationAsset`, `methodDetails.destinationRecipient`, and `methodDetails.amountOut` describe the asset, chain, beneficiary, and exact amount the **merchant** receives once the swap completes.

This mapping keeps push-mode verification — confirm an on-chain transfer to `recipient` of the `currency` the client was asked to send — directly applicable: the verified deposit is the payment, and the cross-chain swap is an extension that executes afterward and delivers to the merchant.

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 NEAR Intents-specific `methodDetails`, `payload`, and verification procedures for the "nearintents" 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

1Click Swap API A hosted interface to the NEAR Intents settlement system [ONECLICK-API]. Exposes quote generation, deposit notification, and status endpoints used by this method.

Deposit Address A unique address generated by the 1Click backend for a single quote. The client sends the source asset to this address. Each address corresponds to exactly one quote and is single-use.

Solver Network The set of competing agents that fulfill NEAR Intents swap intents by sourcing the destination asset and delivering it to the recipient.

EXACT_OUTPUT A 1Click quote mode in which the destination output amount is fixed and the maximum source input amount required to guarantee it is computed by the quote. This method uses `EXACT_OUTPUT` so the merchant receives a deterministic amount.

Origin Chain The chain on which the client holds the source asset and broadcasts the deposit transaction. Identified by a CAIP-2 [CAIP-2] network identifier.

Destination Chain The chain on which the merchant receives the destination asset. Identified by a CAIP-2 [CAIP-2] network identifier.

Settlement Finality For this method, the point at which the merchant has received the destination asset, signalled by the 1Click terminal status `SUCCESS`. See Section 10.

Base Units The smallest transferable unit of an asset, determined by its decimal precision (e.g., USDC with 6 decimals uses 1,000,000 base units per 1 USDC).

4. Request Schema

The `request` parameter in the `WWW-Authenticate` challenge contains a base64url-encoded [RFC4648] JSON [RFC8259] object. The JSON **MUST** be serialized using JSON Canonicalization Scheme (JCS) [RFC8785] before base64url encoding, per [I-D.httpauth-payment].

This specification implements the shared request fields defined in [I-D.payment-intent-charge].

4.1. Shared Fields

Field	Type	Presence	Description
`amount`	string	**REQUIRED**	Deposit amount the client is asked to send, in base units of `currency` (the 1Click quote `maxAmountIn`): the upper bound that guarantees the merchant receives `methodDetails.amountOut`. A requested/display value — the verifier actually requires `methodDetails.minAmountIn`, not `amount` (see Section 8).
`currency`	string	**REQUIRED**	Source asset the client pays with, as a CAIP-19 [CAIP-19] asset identifier (see Section 4.2); chain component **MUST** equal `methodDetails.originNetwork`.
`recipient`	string	**REQUIRED**	1Click deposit address on the origin chain. The payee of the client's on-chain transfer.
`description`	string	**OPTIONAL**	Human-readable payment description. **MUST NOT** be relied upon for verification per [I-D.httpauth-payment].
`externalId`	string	**OPTIONAL**	Merchant reference (order ID, invoice number, etc.).

Table 1

Challenge expiry is conveyed by the `expires` auth-param in `WWW-Authenticate` per [I-D.httpauth-payment] (see Section 6).

4.2. Asset Identifiers

Both `currency` and `methodDetails.destinationAsset` **MUST** be CAIP-19 [CAIP-19] asset identifiers. CAIP-19 encodes the chain and the asset in one self-describing string, so both legs of the swap use the same convention and either side can be parsed without external context:

Chain type	CAIP-19 form	Example
EVM chains	`eip155:<chainId>/erc20:<address>`	`eip155:42161/erc20:0xaf88d065e77c8cC2239327C5EDb3A432268e`
Solana	`solana:<genesisHash>/token:<mint>`	`solana:5eykt4UsFv8P8NJdTREpY1vzqKqZKvdp/token:EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v`
NEAR (NEP-141)	`near:mainnet/nep141:<account>`	`near:mainnet/nep141:17208628f84f5d6ad33f0da3bbbeb27ffcb398eac501a31bd6`
Native assets (no contract)	`<caip2>/slip44:<coinType>`	`bip122:000000000019d6689c085ae165831e93/slip44:0` (BTC)

Table 2

Implementations **MUST NOT** use informal short identifiers (e.g., `"arb"`, `"BTC"`) or bare contract addresses without the CAIP-19 chain prefix in `currency` or `destinationAsset`. The chain component of `currency` **MUST** equal `methodDetails.originNetwork`, and the chain component of `destinationAsset` **MUST** equal `methodDetails.destinationNetwork`. CAIP-19 identifiers **MUST** be compared by their parsed components, with the address or reference component compared in the relevant chain's canonical form (for example, EVM addresses compared case-insensitively).

4.3. Method Details

Field	Type	Presence	Description
`methodDetails.originNetwork`	string	**REQUIRED**	CAIP-2 [CAIP-2] identifier of the origin chain (where `recipient` lives and the deposit tx is anchored).
`methodDetails.destinationNetwork`	string	**REQUIRED**	CAIP-2 [CAIP-2] identifier of the chain where the merchant receives the destination asset.

Field	Type	Presence	Description
`methodDetails.destinationAsset`	string	**REQUIRED**	Destination asset the merchant receives, as a CAIP-19 [CAIP-19] asset identifier (see Section 4.2); chain component **MUST** equal `methodDetails.destinationNetwork`.
`methodDetails.destinationRecipient`	string	**REQUIRED**	Merchant address on the destination chain that receives `amountOut`.
`methodDetails.amountOut`	string	**REQUIRED**	Exact amount the merchant receives, in base units of `destinationAsset`. Guaranteed via `EXACT_OUTPUT`.
`methodDetails.minAmountIn`	string	**REQUIRED**	Minimum deposit the backend accepts, in base units of `currency`. Threshold checked at verification (Section 8, step 3): a confirmed deposit **MUST** be at least `minAmountIn`. Deposits below it yield `INCOMPLETE_DEPOSIT` and are refunded to `refundTo`.
`methodDetails.depositMemo`	string | null	**OPTIONAL**	Deposit memo required by some origin chains (e.g., Stellar). Absent or `null` when not required.
`methodDetails.slippageTolerance`	number	**OPTIONAL**	Slippage tolerance in basis points (e.g., `100` = 1%).
`methodDetails.timeEstimate`	number	**OPTIONAL**	Estimated swap completion time in seconds, from the 1Click quote.
`methodDetails.refundTo`	string	**REQUIRED**	Address on the origin chain to which the backend refunds the deposit if the swap fails or excess exists.
`methodDetails.settlementBackend`	string	**OPTIONAL**	Discloses the settlement backend. Value `"near-intents"` for this method.
`methodDetails.credentialTypes`	array	**OPTIONAL**	Ordered list of accepted credential types. For this method, the only valid value is `"hash"`.

Table 3

If `methodDetails.credentialTypes` is omitted, servers **MUST** accept `"hash"`. Servers **MUST NOT** advertise credential types other than `"hash"` for this method.

Example (decoded `request`):

```
{
  "amount": "1005000",
  "currency": "eip155:42161/
erc20:0xaf88d065e77c8cC2239327C5EDb3A432268e5831",
  "recipient": "0x76b4c56085ED136a8744D52bE956396624a730E8",
  "description": "Cross-chain premium data access",
  "externalId": "order_12345",
  "methodDetails": {
    "originNetwork": "eip155:42161",
    "destinationNetwork": "near:mainnet",
    "destinationAsset": "near:mainnet/
nep141:17208628f84f5d6ad33f0da3bbbeb27ffcb398eac501a31bd6ad2011e36133a1",
    "destinationRecipient": "merchant.near",
    "amountOut": "1000000",
    "minAmountIn": "1000000",
    "depositMemo": null,
    "slippageTolerance": 100,
    "timeEstimate": 120,
    "refundTo": "0x2527D02599Ba641c19FEa793cD0F9a6e8457C317",
    "settlementBackend": "near-intents",
    "credentialTypes": ["hash"]
  }
}
```

This requests a deposit of 1.005 USDC on Arbitrum (`eip155:42161`) so that the merchant receives exactly 1.00 of the destination asset on NEAR (`near:mainnet`).

5. Credential Schema

The credential in the `Authorization` header contains a base64url-encoded JSON object per [I-D.httpauth-payment].

5.1. Credential Structure

Field	Type	Presence	Description
`challenge`	object	**REQUIRED**	Echo of the challenge auth-params from `WWW-Authenticate` per [I-D.httpauth-payment].
`payload`	object	**REQUIRED**	NEAR Intents-specific payload (see Section 5.2).
`source`	string	**OPTIONAL**	Payer DID.

Table 4

The `source` field, if present, **SHOULD** use the `did:pkh` method [DID-PKH] with the CAIP-2 origin network identifier and the payer's origin-chain address (e.g., `did:pkh:eip155:42161:0x2527...`).

5.2. Hash Payload — Push Mode (type="hash")

In push mode, the client has already broadcast the deposit transaction to the origin chain. The `hash` field contains the transaction hash for the server to verify.

Field	Type	Presence	Description
`type`	string	**REQUIRED**	`"hash"`
`hash`	string	**REQUIRED**	Transaction hash of the client's deposit on the origin chain (`methodDetails.originNetwork`). Format is origin-chain-native.

Table 5

The address the deposit must pay is not repeated in the payload: it is the challenge `recipient`, which is bound into the challenge `id` (see Section 7). The server resolves which deposit to verify from the echoed challenge.

Example:

```
{
  "challenge": {
    "id": "qB3wErTyU7iOpAsD9fGhJk",
    "realm": "api.example.com",
    "method": "nearintents",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-06-25T15:10:00Z"
  },
  "payload": {
    "type": "hash",
    "hash":
"0x9bcff372aee89b648c922b850573b22387c31d693079f5e37cd255814e2d615a"
  },
  "source": "did:pkh:eip155:42161:0x2527D02599Ba641c19FEa793cD0F9a6e8457C317"
}
```

6. Challenge Expiry and Deposit Deadline

The challenge `expires` auth-param ([I-D.httpauth-payment]) **MUST** be set to the 1Click quote deadline, or to a slightly earlier time, so that a deposit made before `expires` is still within the quote's validity window.

Clients **MUST NOT** broadcast a deposit transaction after the challenge `expires` timestamp. A deposit that arrives at the deposit address after the quote deadline may be refunded to `methodDetails.refundTo` rather than swapped.

Servers **SHOULD** set `expires` such that the remaining window before the quote deadline accommodates origin-chain confirmation plus `methodDetails.timeEstimate`. Because the deposit address is time-limited, servers **SHOULD** cache the challenge for a given resource until the quote deadline and reissue the same challenge (including the same `recipient`) for repeated requests, regenerating only when the deadline passes.

7. Challenge Binding

The challenge `id` binds the challenge parameters per [I-D.httpauth-payment], including `request`, which carries `recipient` (the deposit address), `amount`, and `currency`. Servers **MUST** verify that the credential echoes a challenge whose recomputed binding matches the `id` they issued (for example, via the HMAC-SHA256 mechanism in [I-D.httpauth-payment]), and **MUST** reject credentials presenting an unknown, expired, modified, or already-settled `id`.

Because the 1Click backend issues a **unique deposit address per quote**, the `recipient` in each challenge is challenge-specific. A deposit transaction observed at that address is therefore implicitly bound to the single challenge that advertised it — an attacker cannot present the same deposit against a different challenge, because a different challenge has a different `recipient`. This gives push mode under this method a stronger practical binding than hash credentials whose recipient is a static merchant address shared across many challenges. See Section 13.3.

8. Verification

Upon receiving a request with a credential, the server **MUST** first validate that `payload.type` is `"hash"`, then perform the following checks. If any check fails, the server **MUST** return a `verification-failed` error per [I-D.httpauth-payment] (or a more specific code from Section 9).

If the origin-chain RPC or the 1Click status endpoint is unavailable for a required check, servers **MUST** treat this as a server error (HTTP 5xx) rather than a `verification-failed` response, and **MUST NOT** settle the credential.

1. The challenge `id` matches an outstanding, unsettled challenge issued by this server, the recomputed challenge binding matches `id` (Section 7), and the current time is before the challenge `expires` auth-param.
2. `payload.hash` has not been previously consumed (see Section 12).
3. The transaction identified by `payload.hash` exists on `methodDetails.originNetwork`, is confirmed to that chain's policy (see Section 10), and transfers at least `methodDetails.minAmountIn` of `currency` to the challenge `recipient` (the deposit address), including `methodDetails.depositMemo` when non-null. This confirmation **MUST** be performed either by querying the origin chain directly, or by querying the 1Click status endpoint and observing that the backend has detected a qualifying deposit at `recipient`. Relying solely on client-asserted payload fields does not satisfy this requirement.

4. The deposit address (`recipient`) corresponds to an active, non-expired, non-settled quote in the server's state.
5. Claim `payload.hash` as **in-flight** for this challenge, rejecting any concurrent or later submission of the same hash while settlement runs. The hash is **permanently consumed only on a terminal settlement state** (see Section 10 and Section 12); verification alone does not burn the credential. On a non-success terminal state the client recovers with a fresh challenge (see Section 10.1).

Verification confirms that the client has paid the origin-chain leg. The cross-chain swap that delivers the destination asset to the merchant is driven during settlement (Section 10); a `Payment-Receipt` is issued only after that delivery completes.

9. Error Codes

This specification defines the following additional error code beyond those in [I-D.httpauth-payment]:

Code	HTTP	Description
`settlement-failed`	402	The deposit was verified but the cross-chain swap did not complete (1Click terminal status `FAILED` or `REFUNDED`); the deposit is refunded to `refundTo`.

Table 6

Other failure conditions map to the standard problem types of [I-D.httpauth-payment]:

Condition	Problem type
Challenge unknown, modified, already settled	`invalid-challenge`
Challenge or quote deadline passed	`payment-expired`
Deposit not found, wrong recipient, wrong asset/memo	`verification-failed`
Deposit below `methodDetails.minAmountIn`	`payment-insufficient`
Swap failed or refunded after a verified deposit	`settlement-failed`
Credential malformed (bad base64url or JSON)	`malformed-credential`

Table 7

As required by [I-D.httpauth-payment], every rejection returns HTTP 402 with a fresh `WWW-Authenticate: Payment` challenge and a Problem Details [RFC9457] body with `Content-Type: application/problem+json`.

10. Settlement Procedure

After successful verification, the server drives the swap to finality before granting access. The server:

1. Notifies the 1Click backend of the deposit (optional but recommended, as it accelerates processing; the backend also detects deposits automatically). This corresponds to the 1Click deposit-notification endpoint, supplying the transaction hash and deposit address.
2. Polls the 1Click status endpoint for the deposit address until a terminal status is reached, or until the time implied by the challenge `expires` window is exceeded. Terminal statuses are `SUCCESS`, `FAILED`, `REFUNDED`, and `INCOMPLETE_DEPOSIT`.
3. On `SUCCESS` — the merchant has received `methodDetails.amountOut` of `methodDetails.destinationAsset` on `methodDetails.destinationNetwork` — returns the resource with a `Payment-Receipt` header per Section 11.
4. On `FAILED` or `REFUNDED`, returns a `settlement-failed` error per Section 9. On `INCOMPLETE_DEPOSIT`, returns `payment-insufficient`. In all non-success terminal cases the backend refunds the deposit to `methodDetails.refundTo`; the server **MUST NOT** grant access.
5. On reaching any terminal status, permanently consume `payload.hash` (see Section 12). After a terminal state the quote and its deposit address are spent — the deposit was either delivered (`SUCCESS`) or refunded — so the same hash **MUST NOT** be accepted again.

10.1. Client Recovery

A non-success terminal state (`FAILED`, `REFUNDED`, `INCOMPLETE_DEPOSIT`, or a settlement timeout) is terminal for this credential: the deposit is refunded to `methodDetails.refundTo`, and because the quote and deposit address are single-use, the same `payload.hash` cannot be retried. To pay again, the client re-requests the resource to obtain a **fresh challenge** (a new quote, hence a new deposit address), sends a new deposit, and submits a new credential. Consistent with Section 9, the server returns each non-success terminal state as a 402 carrying that fresh `WWW-Authenticate: Payment` challenge, which the client uses to begin a new attempt. Burning the original hash is therefore safe: it can never deliver the resource, and the funds it represents have been refunded.

10.2. Settlement Finality

This method has two candidate finality boundaries: (a) confirmation of the client's deposit on the origin chain, and (b) delivery of the destination asset to the merchant. This method defines finality as **(b) destination delivery** (1Click terminal status `SUCCESS`). The server **MUST NOT** return a `Payment-Receipt` before `SUCCESS`.

Per-origin-chain confirmation depth (boundary a) is a policy of the settlement backend: the 1Click backend applies chain-appropriate confirmation depth before executing a swap, including on chains with probabilistic finality (e.g., Bitcoin block depth, Solana commitment levels).

Servers **MUST NOT** treat a deposit observed below the backend's confirmation threshold as final; in practice this is automatic, since `SUCCESS` cannot be reached before the backend confirms the deposit.

The time from deposit to `SUCCESS` varies by origin chain and current conditions and is approximated by `methodDetails.timeEstimate`. Servers **SHOULD NOT** assume a fixed latency, and **SHOULD** calibrate the challenge `expires` window per origin chain (short for fast chains, substantially longer for slow-finality origins such as Bitcoin). For long-running settlements, servers **MAY** return `202 Accepted` with a retry location or offer webhook notification instead of holding the connection open; specifying such an asynchronous delivery mechanism is otherwise out of scope for this document.

11. Receipt

Upon successful settlement, servers **MUST** return a `Payment-Receipt` header per [I-D.httpauth-payment]. Servers **MUST NOT** include a `Payment-Receipt` header on error responses.

The receipt payload fields:

Field	Type	Presence	Description
`method`	string	**REQUIRED**	`"nearintents"`
`challengeId`	string	**REQUIRED**	The `id` from the original challenge.
`reference`	string	**REQUIRED**	Settlement reference: the destination-chain transaction hash of the merchant delivery.
`status`	string	**REQUIRED**	`"success"`
`timestamp`	string	**REQUIRED**	[RFC3339] settlement time.
`originTxHash`	string	**REQUIRED**	The client's origin-chain deposit transaction hash (`payload.hash`).
`destinationNetwork`	string	**OPTIONAL**	CAIP-2 identifier of the chain where the merchant was paid.
`externalId`	string	**OPTIONAL**	Echoed from the challenge request.

Table 8

Example (decoded):

```
{
  "method": "nearintents",
  "challengeId": "qB3wErTyU7iOpAsD9fGhJk",
  "reference": "FtChYxxQh1k6vKjQ9wq5q1f8s2n3p4r5t6u7v8w9x0yz",
  "status": "success",
  "timestamp": "2026-06-25T15:12:11Z",
  "originTxHash":
"0x9bcff372aee89b648c922b850573b22387c31d693079f5e37cd255814e2d615a",
  "destinationNetwork": "near:mainnet",
  "externalId": "order_12345"
}
```

12. Replay Protection

Per [I-D.httpauth-payment], payment proofs **MUST** be single-use. This method enforces replay protection on two layers:

- **Deposit address as nonce**: each quote yields a unique deposit address, and the 1Click backend processes at most one swap per address. The server **MUST** reject any credential whose `recipient` (deposit address) does not correspond to an active, non-expired, non-settled quote in its state.
- **Single-use transaction hashes**: the server **MUST** track each `payload.hash` and reject any hash that is already in-flight or consumed. At verification the hash is claimed as in-flight (atomically, satisfying the concurrency requirement below); it is **permanently consumed only when settlement reaches a terminal state** — `SUCCESS` or any failure/refund — so a credential is never burned before its outcome is known and is never reusable afterward. On a non-success terminal state the client obtains a fresh challenge to retry (see Section 10.1).

When a single origin transaction is one of several deposits aggregated by the backend to one address (e.g., a top-up after an under-deposit), each individual transaction hash remains single-use; the server **MAY** accept any one qualifying hash provided the aggregate meets `methodDetails.minAmountIn`.

Servers **MUST** also satisfy the concurrency requirement of [I-D.httpauth-payment]: concurrent requests bearing the same credential **MUST** result in at most one settlement and one resource delivery.

13. Security Considerations

13.1. Transport Security

All communication **MUST** use TLS 1.2 or higher per [I-D.httpauth-payment]. Credentials and receipts **MUST** only be transmitted over HTTPS connections.

13.2. Amount and Asset Verification

Before broadcasting a deposit, clients **MUST** decode and verify the challenge `request` per [I-D.httpauth-payment]: that `amount`, `currency`, and `recipient` (the deposit address) are as expected, that `methodDetails.originNetwork` is the intended origin chain, and that the destination leg (`destinationNetwork`, `destinationAsset`, `destinationRecipient`, `amountOut`) matches what the client intends the merchant to receive. Clients **MUST NOT** rely on `description`.

13.3. Hash Credential Binding

Hash credentials generally provide weaker challenge binding than signature-based credentials: the server confirms a matching on-chain payment exists but, for a static recipient, cannot prove the payment was created for a specific challenge instance. Under this method that gap is narrowed by the unique-per-quote deposit address: the `recipient` is itself challenge-specific and is bound into the challenge `id` (Section 7), so a deposit observed at that address is bound to the one challenge that advertised it. Servers nonetheless **MUST** enforce consumed-hash tracking (Section 12) and **MUST** reject credentials referencing an unknown, expired, or settled challenge, and **SHOULD** use unique `externalId` values per challenge.

13.4. Deposit Address Authenticity

Servers **MUST** only advertise deposit addresses obtained from authenticated calls to the 1Click API, and **MUST NOT** relay deposit addresses from untrusted sources. Because the client pays before learning whether the resource will be delivered, clients **SHOULD** apply heightened scrutiny to the server's identity (origin, TLS certificate) before depositing, and browser-based wallets **SHOULD** require explicit user confirmation per [I-D.httpauth-payment].

13.5. Trust Model

Settlement in this method is **not trustless**. For the duration of the swap, the deposited funds are custodied by the NEAR Intents settlement system, which is trusted to either deliver the destination asset to the merchant or refund the deposit to `methodDetails.refundTo`. The `recipient` (deposit address) ↔ merchant pairing is fixed in the quote and is enforced by the backend, not by an on-chain contract that the client co-signs. This is comparable to entrusting a payment processor with a transfer. The automatic refund path bounds the client's downside: every non-success terminal state refunds the deposit. Clients and autonomous agents applying per-method risk policies can identify this trust model from the `method` (`nearintents`) and `methodDetails.settlementBackend`.

13.6. Idempotency and Side Effects

Per [I-D.httpauth-payment], servers **MUST NOT** perform side effects for unpaid requests, and **SHOULD** honour `Idempotency-Key` for non-idempotent methods so that client retries with the same credential do not trigger a second settlement.

14. IANA Considerations

14.1. Payment Method Registration

This document registers the following payment method in the "HTTP Payment Methods" registry established by [I-D.httpauth-payment]:

Method Identifier	Description	Reference
`nearintents`	Cross-chain payment settled via the NEAR Intents 1Click Swap API	This document

Table 9

Contact: Iker Alustiza (iker.alustiza@nearone.org).

14.2. Payment Intent Registration

This document registers the following payment intent usage in the "HTTP Payment Intents" registry established by [I-D.httpauth-payment]:

Intent	Applicable Methods	Description	Reference
`charge`	`nearintents`	One-time cross-chain payment delivered to the merchant	This document

Table 10

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>.

[CAIP-2] Chain Agnostic Standards Alliance, "CAIP-2: Blockchain ID Specification", 2020, <https://chainagnostic.org/CAIPs/caip-2>.

[CAIP-19] Chain Agnostic Standards Alliance, "CAIP-19: Asset Type and Asset ID Specification", 2021, <https://chainagnostic.org/CAIPs/caip-19>.

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

[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/>.

15.2. Informative References

[DID-PKH] W3C CCG, "did:pkh Method Specification", n.d., <https://github.com/w3c-ccg/did-pkh>.

[ONECLICK-API] NEAR Intents, "NEAR Intents 1Click Swap API", n.d., <https://docs.near-intents.org/integration/distribution-channels/1click-api/about-1click-api>.

[NEAR-INTENTS-CHAINS] NEAR Intents, "NEAR Intents Supported Chains", n.d., <https://docs.near-intents.org/resources/chain-support>.

Appendix A. ABNF Collected

```
nearintents-charge-challenge = "Payment" 1*SP
  "id=" quoted-string ","
  "realm=" quoted-string ","
  "method=" DQUOTE "nearintents" DQUOTE ","
  "intent=" DQUOTE "charge" DQUOTE ","
  "request=" base64url-nopad
  [ "," "expires=" quoted-string ]

nearintents-charge-credential = "Payment" 1*SP base64url-nopad

; Base64url encoding without padding per RFC 4648 Section 5
base64url-nopad = 1*( ALPHA / DIGIT / "-" / "_" )
```

Appendix B. Examples

B.1. Cross-Chain Charge — USDC on Arbitrum to merchant on NEAR

1. Challenge (402 response):

```
HTTP/1.1 402 Payment Required
Cache-Control: no-store
Content-Type: application/problem+json
WWW-Authenticate: Payment id="qB3wErTyU7iOpAsD9fGhJk",
  realm="api.example.com",
  method="nearintents",
  intent="charge",
  request="eyJ...",
  expires="2026-06-25T15:10:00Z"

{
  "type": "https://paymentauth.org/problems/payment-required",
  "title": "Payment Required",
  "status": 402,
  "detail": "Payment required for access."
}
```

Decoded `request`:

```
{
  "amount": "1005000",
  "currency": "eip155:42161/
erc20:0xaf88d065e77c8cC2239327C5EDb3A432268e5831",
  "recipient": "0x76b4c56085ED136a8744D52bE956396624a730E8",
  "description": "Cross-chain premium data access",
  "externalId": "order_12345",
  "methodDetails": {
    "originNetwork": "eip155:42161",
    "destinationNetwork": "near:mainnet",
    "destinationAsset": "near:mainnet/
nep141:17208628f84f5d6ad33f0da3bbbeb27ffcb398eac501a31bd6ad2011e36133a1",
    "destinationRecipient": "merchant.near",
    "amountOut": "1000000",
    "minAmountIn": "1000000",
    "depositMemo": null,
    "slippageTolerance": 100,
    "timeEstimate": 120,
    "refundTo": "0x2527D02599Ba641c19FEa793cD0F9a6e8457C317",
    "settlementBackend": "near-intents",
    "credentialTypes": ["hash"]
  }
}
```

The client sends 1.005 USDC to `0x76b4c560...` on Arbitrum, then presents the deposit hash.

2. Credential:

```
GET /resource HTTP/1.1
Host: api.example.com
Authorization: Payment eyJ...
```

Decoded credential:

```
{
  "challenge": {
    "id": "qB3wErTyU7iOpAsD9fGhJk",
    "realm": "api.example.com",
    "method": "nearintents",
    "intent": "charge",
    "request": "eyJ...",
    "expires": "2026-06-25T15:10:00Z"
  },
  "payload": {
    "type": "hash",
    "hash":
"0x9bcff372aee89b648c922b850573b22387c31d693079f5e37cd255814e2d615a"
  },
  "source": "did:pkh:eip155:42161:0x2527D02599Ba641c19FEa793cD0F9a6e8457C317"
}
```

3. Response (after swap reaches `SUCCESS`):

```
HTTP/1.1 200 OK
Cache-Control: private
Payment-Receipt: eyJ...
Content-Type: application/json

{"data": "..."}
```

Decoded receipt:

```
{
  "method": "nearintents",
  "challengeId": "qB3wErTyU7iOpAsD9fGhJk",
  "reference": "FtChYxxQh1k6vKjQ9wq5q1f8s2n3p4r5t6u7v8w9x0yz",
  "status": "success",
  "timestamp": "2026-06-25T15:12:11Z",
  "originTxHash":
"0x9bcff372aee89b648c922b850573b22387c31d693079f5e37cd255814e2d615a",
  "destinationNetwork": "near:mainnet",
  "externalId": "order_12345"
}
```

B.2. Native BTC origin

A challenge advertising a Bitcoin-origin deposit. The longer `expires` window accommodates Bitcoin confirmation plus swap time.

Decoded `request`:

```
{
  "amount": "38000",
  "currency": "bip122:000000000019d6689c085ae165831e93/slip44:0",
  "recipient": "bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh",
  "methodDetails": {
    "originNetwork": "bip122:000000000019d6689c085ae165831e93",
    "destinationNetwork": "near:mainnet",
    "destinationAsset": "near:mainnet/
nep141:17208628f84f5d6ad33f0da3bbbeb27ffcb398eac501a31bd6ad2011e36133a1",
    "destinationRecipient": "merchant.near",
    "amountOut": "1000000",
    "minAmountIn": "37500",
    "depositMemo": null,
    "slippageTolerance": 150,
    "timeEstimate": 1800,
    "refundTo": "bc1qm34lsc65zpw79lxes69zkqmk6ee3ewf0j77s3h",
    "settlementBackend": "near-intents",
    "credentialTypes": ["hash"]
  }
}
```

B.3. Multiple origin chains via Accept-Payment

A client declares it can pay on Arbitrum or Bitcoin and prefers Arbitrum:

```
GET /resource HTTP/1.1
Host: api.example.com
Accept-Payment: nearintents/charge
```

The server **MAY** return several `nearintents/charge` challenges, each on a different origin `methodDetails.originNetwork` with its own `recipient` deposit address, allowing the client to choose:

```
HTTP/1.1 402 Payment Required
Cache-Control: no-store
WWW-Authenticate: Payment id="qB3wErTyU7iOpAsD9fGhJk",
realm="api.example.com", method="nearintents", intent="charge",
request="eyJ...arbitrum...", expires="2026-06-25T15:10:00Z"
WWW-Authenticate: Payment id="nH6xJkLpO3qRtYsA6wDcVb",
realm="api.example.com", method="nearintents", intent="charge",
request="eyJ...bitcoin...", expires="2026-06-25T15:40:00Z"
```

Each challenge corresponds to a distinct 1Click quote and deposit address. The client selects one and returns a single `Authorization: Payment` credential per [I-D.httpauth-payment].

Appendix C. Acknowledgements

The author thanks the Tempo Labs and Stripe teams for the Machine Payments Protocol framework, and the NEAR Intents and Near One teams for the 1Click Swap settlement system.

Author's Address

Iker Alustiza
Near One
Email: iker.alustiza@nearone.org