INTERNET-DRAFT
Intended status: Best Current Practice
Expires: 17 November 2026

A. Jurkovikj
16 May 2026

HTTP Profile for Synchronized Resource State (Agentic State Transfer) draft-jurkovikj-httpapi-agentic-state-01

Abstract

HTTP resources are frequently exposed in multiple representations (e.g., text/html for rendering and application/json for processing) via Content Negotiation. Standard HTTP validators such as ETags identify selected representations, while many applications need concurrency control over a shared logical resource state. This creates a synchronization gap where a client modifying one representation cannot guarantee consistency with the underlying state of another representation, leading to race conditions and "lost updates" in multi-client environments.

This document specifies Agentic State Transfer (AST), an HTTP profile for managing Canonical Resource State (CRS) across multiple synchronized representations of the same resource. It defines a State-Bearing Representation (SBR) whose strong ETag acts as the AST semantic validator for the CRS. It further mandates the use of Optimistic Concurrency Control via If-Match headers for state-changing operations, ensuring that mutations applied by agents (whether automated clients, scripts, or human-driven applications) are atomic and consistent across all views.

Table of Contents

1. Introduction

The architecture of the World Wide Web allows a single logical resource (identified by a URI) to be represented in multiple formats. A common pattern in modern systems is to expose:

* a *Rendered Representation* (e.g., text/html) for human agents, and
* a *Structural Representation* (e.g., application/json) for automated agents.

In AST terminology, rendered representations are typically Projected Representations, while a structural representation MAY serve as the State-Bearing Representation (SBR) when it exposes the full Canonical Resource State.

HTTP provides mechanisms for content negotiation [RFC9110], and defines validators such as ETag for caching and conditional requests. However, in most deployments:

* validators are derived from a particular representation’s bytes, and
* concurrency control, if present at all, is scoped to a single representation.

This creates a synchronization gap: two clients interacting with different representations of the same logical resource can unwittingly overwrite each other’s work.

AST addresses this problem by treating the Canonical Resource State (CRS) as the primary object of synchronization, and by standardizing how the SBR ETag is used as the profile-specific semantic validator for that CRS.

1.1. The Synchronization Problem

Consider a resource /article/123 exposed both as HTML and JSON.

1. Client A (Browser) retrieves the HTML representation.
2. Client B (Automated Agent) retrieves the JSON representation.
3. Client A submits an edit (e.g., via the JSON API or a form handler that writes to the CRS), updating the underlying database record.
4. Client B, unaware of the change, computes an update based on its stale JSON representation and submits a PUT or PATCH request.

In typical HTTP APIs, at least one of the following is true:

* there is no ETag at all;
* the ETag is tied to the bytes of a specific representation (e.g., JSON only); or
* the server does not enforce preconditions (If-Match).

As a result, Client B's request may succeed, even though:

* its view of the resource is stale, and
* the update logically conflicts with Client A's change.

This is the classic Lost Update problem, now exacerbated by the growing use of automated agents acting as first-class editors.

1.2. Canonical Resource State (CRS)

To address this, AST introduces the concept of Canonical Resource State (CRS):

CRS is the authoritative, logical state of the resource, independent of any particular representation or serialization.

For example, the CRS might be:

* a database row (or set of rows),
* a document or abstract syntax tree,
* a domain object in application memory, or
* in a WordPress CMS, the post record in wp_posts with fields like post_content, post_title, post_status, and associated metadata.

Representations such as HTML, JSON, or Markdown are derived from the CRS and are considered projections of that state.

AST defines a profile where:

1. Shared Identity -- All HTTP representations of an AST resource are projections of the same CRS.
2. Semantic Validation -- Validators (ETag values) are derived from the CRS, not a specific representation's bytes, so that a change to the CRS invalidates all representations together.
3. Mandatory Concurrency -- State-changing operations on AST resources MUST provide a precondition (If-Match) based on the CRS validator.

This enables safe interaction by automated agents and human clients, ensuring they participate in the same state transition model.

1.3. Deployment Models

AST accommodates two common deployment patterns:

Single-URI (Content Negotiation): The SBR and Projected Representations are served from the same URI based on Accept headers. The SBR ETag is returned when the structural media type (e.g., application/json) is selected; a different representation-specific ETag MAY be returned for other media types. Discovery of the SBR media type is implicit in the content negotiation.

Multi-URI (Separate Endpoints): The SBR is served from a dedicated API endpoint (e.g., /api/article/123) while Projected

Representations
are served from other URIs (e.g., /article/123). The rel="state"
link header connects Projected Representations to the SBR
endpoint.

In both models, the SBR ETag serves as the Semantic Validator for
the
shared CRS. Clients MUST use the SBR ETag (not any projection-
specific
ETag) for If-Match preconditions on state-changing operations.

In multi-URI deployments, the SBR endpoint and Projected endpoints
MAY
reside on different authorities, base paths, or even different
servers.
The rel="state" link is the canonical connector that allows
clients to
discover the SBR regardless of deployment topology.

AST is defined as a profile - a set of constraints and conventions
layered
on standard HTTP semantics. Conforming implementations remain
fully
interoperable with generic HTTP clients and servers; AST does not
introduce breaking changes to HTTP.

1.4. Relationship to Dual-Native Pattern

This specification defines an HTTP profile of the Dual-Native
Pattern,
an architectural approach where resources provide both human-
optimized and
machine-optimized representations with formal semantic equivalence
guarantees.

The Dual-Native Pattern is defined in the Core Architecture and
Requirements
specification and demonstrated across multiple domains (HTTP,
databases,
streaming systems, healthcare). AST specifically addresses the
HTTP binding
with focus on safe concurrency control for state-changing
operations.

1.4.1. Terminology Mapping

AST terminology aligns with Dual-Native Pattern concepts as
follows:

- AST Term / Dual-Native Term: Description
- State-Bearing Representation (SBR) / Machine Representation
(MR): Canonical, structured representation for programmatic access
- Projected Representation / Human Representation (HR):
Presentation-optimized interface (typically HTML)
- Semantic Validator (SBR ETag) / Content Identity (CID): Version-
specific strong validator for change detection
- Canonical Resource State (CRS) / Underlying resource state:
Authoritative logical state independent of serialization
- Resource URI / Resource Identity (RID): Stable identifier shared
across representations
- If-Match preconditions / Safe writes: Optimistic concurrency
control via validator preconditions
- If-None-Match conditional requests / Zero-fetch reads: Bandwidth
optimization via validator revalidation
- Content-Digest / Integrity digest: Verification of exact byte-
level payload parity

- rel="state" discovery / Dual-Native Catalog (DNC) entry:
Discovery mechanism connecting representations

1.4.2. Critical Properties Coverage

AST implements the critical properties identified in the Dual-Native Pattern:

1. Canonical MR: The SBR serves as the deterministic, machine-readable
representation (Section 3.1)

2. Strong CID: The SBR ETag is a strong validator that changes whenever the CRS changes (Section 3.2)

3. Safe writes: If-Match preconditions provide optimistic concurrency
control with explicit conflict signals (Section 3.4)

4. Zero-fetch reads: If-None-Match conditional requests enable bandwidth
optimization when content is unchanged (Section 3.3)

5. Integrity: Content-Digest provides byte-level verification independent
of semantic validation (Section 5)

6. Discovery: rel="state" links enable clients to discover the SBR from
Projected Representations (Section 6)

1.4.3. Scope: HTTP-Specific Binding

AST is specifically an HTTP profile. It defines:

- How to map Dual-Native concepts to HTTP headers (ETag, If-Match, Content-Digest, Link)
- HTTP-specific status codes (412, 428, 304)
- Content negotiation patterns (single-URI vs multi-URI)
- HTTP caching semantics for SBR and Projected Representations

AST does not address:

- Database, streaming, or IoT bindings (see Dual-Native Implementation Guide)
- Full catalog/registry specifications (DNC is out of scope; AST covers
point-to-point discovery only)
- Schema evolution or version migration strategies
- Authentication/authorization mechanisms (domain-specific)

For the broader architectural context, see the Dual-Native Pattern Whitepaper.
For formal cross-domain requirements, see the Core Architecture and
Requirements specification.

2. Terminology

Note: AST terminology is consistent with the Dual-Native Pattern specification family. See Section 1.4.1 for detailed mapping.

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.

Canonical Resource State (CRS)
: The underlying, authoritative logical state of a resource (e.g., the
database record or abstract syntax tree). The CRS is the atomic unit
of concurrency.

State-Bearing Representation (SBR)
: A specific representation (e.g., application/json) that exposes the
CRS in a lossless, deterministic format. The SBR serves as the Concurrency Token for the resource. Its ETag represents the version of
the CRS.

State-Bearing Endpoint
: The HTTP endpoint (URI) that serves the State-Bearing Representation.
This is the primary target for state-changing operations and the authoritative source of the Semantic Validator.

Projected Representation
: A representation derived from the CRS for a specific consumption pattern (e.g., text/html for rendering, text/csv for analysis).
Projected Representations reflect the CRS but do not govern it.
State-changing operations SHOULD be directed to the State-Bearing Endpoint; see Write Semantics for handling writes at other endpoints.

Semantic Validator
: The strong ETag associated with the SBR. It MUST change whenever the
CRS changes. It SHOULD NOT change when the CRS has not changed, although implementation factors (for example, canonicalization or hashing algorithm changes, migrations, or salting strategies) may occasionally cause validator changes without CRS changes. In this document, "Semantic Validator" means the AST profile’s SBR ETag. The
separate Semantic-ETag field defined in
[I-D.jurkovikj-http-semantic-validator] is a related HTTP extension but
is not required by this profile.

AST Resource
: A logical Canonical Resource State (CRS) that may be exposed via one or
more HTTP endpoints (URIs). Endpoints that share the same CRS share the
same Semantic Validator (SBR ETag) for concurrency purposes.

3. The AST Profile

This section defines the Agentic State Transfer (AST) profile. It specifies how clients synchronize state across multiple representations
by pivoting on the State-Bearing Representation (SBR).

3.1. Representation Roles

For a given AST Resource, the server MUST designate exactly one representation as the State-Bearing Representation (SBR). All

other
representations are considered Projected Representations.

* The SBR acts as the source of truth for state synchronization.
* Projected Representations are derivative views. While they MAY have
their own caching validators (ETags) based on their specific byte sequences, they MUST NOT be used as the basis for state-changing preconditions on the CRS.

Endpoints that expose distinct CRS values (i.e., different logical resources) MUST use distinct Semantic Validators. Conversely, endpoints
that share the same CRS MUST be bound to the same Semantic Validator for
concurrency control, meaning they MUST direct clients to the same SBR
ETag for If-Match preconditions. Projected endpoints MAY emit their
own representation-specific ETags for caching purposes, but these MUST
NOT be used for CRS concurrency control.

3.2. Semantic Validators (The SBR ETag)

The server MUST assign a Strong ETag to the SBR. This ETag serves as the
AST Semantic Validator for the resource. Because If-Match uses the strong comparison function [RFC9110], weak entity-tags are not suitable
for AST concurrency control; a weak ETag will never satisfy an If-Match precondition.

* Determinism: The SBR ETag MUST change whenever the CRS changes.
* Scope: The SBR ETag MUST NOT be reused as the ETag for Projected Representations, as this would confuse standard HTTP caches.

Canonicalization: Implementations SHOULD compute the SBR ETag from a
deterministic canonical serialization of the CRS. When the SBR uses
JSON, servers MUST canonicalize using the JSON Canonicalization Scheme
([RFC8785]) or a profile-specified, equivalently deterministic canonicalization before computing the validator. This ensures that logically identical states produce identical validators, regardless of
field ordering or formatting variations. See the Canonicalization subsection in Computing and Managing Semantic Validators for detailed
guidance.

3.2.1. ETag Computation (Informative)

The SBR ETag MUST be a strong validator as defined in [RFC9110]. Implementations MAY compute the SBR ETag using any method that satisfies
both the determinism and strong validator requirements. Common approaches
include:

* Content Hash: SHA-256 or similar cryptographic hash over the canonicalized SBR bytes (e.g., "sha256-YWJjMTIz..."). This is RECOMMENDED for deployment predictability, as it supports reproducible
validator generation and naturally produces strong validators.

* Version Counter: Monotonic version number from the data store (e.g., "v42" or "rev-1234"). This is simple but requires centralized version tracking. The version must increment on every CRS change to maintain strong validator semantics.

The content hash approach provides the strongest guarantee that identical CRS values produce identical ETags within the same canonicalization and validator-construction regime, which aids debugging and distributed deployments.

3.2.2. Relationship to Semantic-ETag

This profile uses the standard ETag field on the SBR as the AST Semantic Validator because If-Match and If-None-Match already define strong validator behavior for state-changing and conditional requests.

The separate Semantic-ETag extension [I-D.jurkovikj-http-semantic-validator] defines an HTTP field for carrying a server-defined semantic validator independently of the selected representation. Deployments that support both profiles can expose Semantic-ETag on Projected Representations as an additional signal for clients that understand it. Such use does not replace the AST requirement that CRS mutations use the SBR ETag, discovered through the SBR endpoint, as the If-Match precondition unless a later AST revision explicitly defines otherwise.

A future revision of AST might define If-Semantic-Match as an alternative to SBR discovery for deployments that expose Semantic-ETag; this revision does not define that alternative.

3.2.3. Encodings and ETags

SBR endpoints MUST NOT apply content-codings to SBR responses; the Content-Encoding header field MUST be absent. (The identity token is reserved in HTTP and MUST NOT appear in Content-Encoding [RFC9110].) This requirement ensures the SBR ETag remains a single, stable strong validator suitable for If-Match concurrency control.

SBR endpoints SHOULD include Cache-Control: no-transform to prevent intermediaries from applying transformations in transit.

Clients SHOULD indicate that no content-codings are acceptable when retrieving the SBR, for example by sending Accept-Encoding: identity or by omitting Accept-Encoding entirely if the client does not support any codings.

Deployments that cannot avoid content-coding at a given URI (for example, due to platform or intermediary constraints) SHOULD NOT designate

that URI
as the SBR. Instead, they SHOULD expose a separate SBR endpoint
that can
be served without content-coding and link to it using rel="state".
The
content-coded URI MAY be treated as a Projected Representation.

3.3. Read Semantics

3.3.1. Reading the SBR

Clients SHOULD use the SBR ETag for conditional requests
(If-None-Match) on the SBR URI. The server MUST return 304 Not
Modified if the CRS has not changed. On a 304 response, the server
SHOULD include the current SBR ETag in the response headers.

Clients SHOULD NOT use the SBR ETag as If-None-Match against
Projected
endpoints. Projected endpoints have their own byte-specific
validators;
using the SBR ETag would result in unnecessary cache misses and
conflates
semantic validation with byte-level caching.

Example: HEAD Request to Fetch SBR Validator

Clients MAY use HEAD to obtain the current SBR ETag without
transferring
the full representation body:

```
HEAD /api/article/123 HTTP/1.1
Host: example.com
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
Content-Length: 4521
```

This allows clients to check for state changes with minimal
bandwidth
usage before deciding whether to fetch the full representation.

3.3.2. Reading Projections

Clients MAY cache Projected Representations using their specific
ETags.
However, clients MUST recognize that a Projected Representation
may be
stale relative to the CRS if the SBR ETag has changed.

3.3.3. Low-Overhead Change Detection (Informative)

Clients monitoring resources for changes (e.g., polling for
updates) can
minimize bandwidth by using conditional requests with HEAD or GET.

Pattern 1: HEAD + If-None-Match (Minimal Overhead)

```
HEAD /api/article/123 HTTP/1.1
Host: example.com
Accept: application/json
If-None-Match: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"

HTTP/1.1 304 Not Modified
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
```

If the resource has changed:

```
HEAD /api/article/123 HTTP/1.1
Host: example.com
Accept: application/json
If-None-Match: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"

HTTP/1.1 200 OK
ETag: "sha256-NEWVALIDATOR456"
Content-Length: 4721
```

The client detects the ETag change without transferring the body, then performs a GET to retrieve the updated representation only when needed.

Pattern 2: GET + If-None-Match (Conditional Transfer)

```
GET /api/article/123 HTTP/1.1
Host: example.com
Accept: application/json
If-None-Match: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"

HTTP/1.1 304 Not Modified
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
```

This pattern combines checking and fetching in one request. If unchanged, the server returns 304 with no body. If changed, it returns 200 with the new representation, saving a round trip compared to HEAD-then-GET.

Recommendation for Polling: Use Pattern 2 (GET + If-None-Match) when the client expects frequent changes and will fetch the body most of the time. Use Pattern 1 (HEAD + If-None-Match) when changes are infrequent and the client needs the ETag for other purposes (e.g., preparing an If-Match precondition for a future write).

3.4. Write Semantics (The Safe Write Protocol)

To modify the Canonical Resource State, clients MUST perform the mutation against the SBR Endpoint (or an endpoint explicitly bound to the SBR's validator).

Clients MUST NOT use a Projected Representation's ETag as an If-Match precondition for CRS mutations. Only the SBR's own ETag is valid for concurrency control.

1. Precondition: The client obtains the current SBR ETag (via GET or HEAD on the SBR URI).

2. Mutation: The client sends the state-changing request (PUT, POST, PATCH, DELETE) including the If-Match header set to the SBR ETag.

3. Verification:

* If the If-Match header is missing, servers SHOULD respond with

428 Precondition Required [RFC6585]. Servers MAY respond with 400 Bad Request if 428 is not supported by the implementation environment, provided the error body clearly indicates the missing precondition requirement.
* If the provided ETag does not match the current CRS version, the server MUST respond with 412 Precondition Failed.
* If the provided ETag matches the current CRS version, the server MAY perform the write if the request is otherwise authorized and valid. If the server performs the write, it MUST update the CRS and ensure subsequent Projected Representation responses are derived from the updated state. Projected validators SHOULD change when the projection bytes change.

Note: HTTP provides no general mechanism for servers to actively invalidate cached Projected Representations in shared caches or CDNs.
Clients that perform CRS mutations SHOULD invalidate their own locally
cached projections; intermediary caches will expire based on their Cache-Control directives.

Note: For endpoints that do not implement AST semantics, servers MAY accept unconditional writes as in traditional HTTP APIs.
See the "Non-AST Resources and Fallback Behavior" section for guidance on mixed deployments.

4. Response: Upon a successful write, the server SHOULD convey the
new SBR ETag to allow the client to chain subsequent updates. The mechanism depends on the response type:

* If the response body is the SBR representation (e.g., the server returns 200 OK with Content-Type: application/json containing the updated state): include ETag: <new-sbr-etag> in the response headers.

* If the response body is a Projected Representation or empty: convey
the updated SBR validator via the Link header’s state-etag parameter, the 204 No Content with Content-Location pattern, or the 303 See Other redirect pattern (see "Returning the Updated Validator from Non-SBR Endpoints" below).

The ETag header in a response MUST describe the selected representation in that response [RFC9110]; it MUST NOT be used to convey the SBR validator when the response body is not the SBR.

3.4.1. Normative Foundations

The AST safe write protocol builds on standard HTTP conditional request
semantics and status codes:

* Entity Tags and Strong Comparison [RFC9110] Section 8.8.3 and Section 13.1.1: Define entity-tag syntax and the strong comparison function used by If-Match. AST requires strong validators because weak entity-tags never satisfy If-Match preconditions under strong comparison.

* If-Match Precondition [RFC9110] Section 13.1.1: Defines precondition
evaluation. The server MUST perform strong comparison between the provided entity-tag(s) and the current validator. If none match, the
server MUST respond with 412 Precondition Failed.

* 428 Precondition Required [RFC6585] Section 3: Defines the status code for requests that require a precondition but lack one. AST servers SHOULD use 428 when If-Match is missing on state-changing operations.

* 412 Precondition Failed [RFC9110] Section 15.5.13: Indicates that precondition evaluation failed. AST servers MUST return 412 when If-Match does not match the current SBR ETag.

* Problem Details [RFC9457]: Defines application/problem+json for machine-readable error responses. AST servers SHOULD return Problem Details bodies for 412, 428, and 400 (missing precondition) responses to assist automated conflict resolution.

This normative alignment ensures AST concurrency control integrates cleanly with existing HTTP implementations and intermediaries.

3.4.2. Error Handling

For 412, 428, and 400 (when used for missing preconditions) responses, servers SHOULD return a Problem Details body [RFC9457] to assist automated clients in resolving the conflict.

When rejecting a request due to Digest Fields validation failures (unsupported algorithm, malformed digest value, or digest mismatch), servers MAY return 400 Bad Request with a Problem Details body using the problem types defined in [I-D.ietf-httpapi-digest-fields-problem-types]. Servers SHOULD NOT include the server-computed digest value in error responses to avoid potential oracle attacks.

3.4.3. Write Location

State-changing operations SHOULD be performed against the State-Bearing Endpoint. Projected endpoints (e.g., HTML rendering URIs) SHOULD reject state-changing requests with 405 Method Not Allowed and include a Link header with rel="state" pointing to the SBR endpoint.

A server MAY designate additional endpoints as accepting state-changing operations (e.g., for compatibility with existing form submissions or legacy APIs). Such endpoints MUST:

* enforce If-Match against the same Semantic Validator as the SBR,
* advertise the SBR via a Link header with rel="state", and
* return the updated Semantic Validator in a manner consistent with HTTP semantics (see below).

Returning the Updated Validator from Non-SBR Endpoints:

In HTTP, the ETag header in a response describes the selected representation returned in that response [RFC9110]. If a non-SBR

endpoint
returns a Projected Representation (e.g., HTML) after a successful
write,
the ETag header MUST describe that projection, not the SBR.

To provide the updated SBR validator without violating HTTP
semantics,
servers SHOULD use one of the following patterns:

* 303 See Other Redirect (RECOMMENDED): Return 303 See Other with
Location pointing to the SBR URI. The client obtains the updated
ETag from the subsequent GET. This adds a round-trip but is
unambiguous and widely understood.

http
HTTP/1.1 303 See Other
Location: /api/article/123
Link: </api/article/123>; rel="state"; type="application/json"

* 204 No Content with Content-Location: Return 204 No Content with
Content-Location set to the SBR URI and the ETag header set to
the updated Semantic Validator. Because Content-Location
identifies
the representation described by the response metadata, this
approach
is semantically defensible, though some implementations may find
it
unusual.

http
HTTP/1.1 204 No Content
Content-Location: /api/article/123
ETag: "sha256-NEWVALIDATOR123"
Link: </api/article/123>; rel="state"; type="application/json"

* Projection Response with Link Metadata: Return the projection
with
its own ETag, and convey the updated SBR validator via the Link
header using an extension parameter:

http
HTTP/1.1 200 OK
Content-Type: text/html
ETag: "html-v2"
Link: </api/article/123>; rel="state"; type="application/json";
state-etag="\"sha256-NEWVALIDATOR123\""

In this pattern, the response ETag describes the HTML projection,
while the Link header’s state-etag parameter conveys the SBR
validator.
Clients MUST use the state-etag parameter value (not the response
ETag) for subsequent If-Match preconditions.

Definition of state-etag Link Extension Parameter:

The state-etag parameter is an extension parameter as permitted by
[RFC8288] Section 3.4.2. Its value is an RFC 8288 quoted-string
whose
decoded content is an HTTP entity-tag as defined in [RFC9110]
Section
8.8.3 (including the surrounding DQUOTE characters). The backslash
escaping is required because Link parameters use the quoted-string
production.

For example, if the SBR ETag is "sha256-ABC123", the Link
parameter

is serialized as state-etag="\"sha256-ABC123\"". Clients extracting
the entity-tag MUST:

1. Parse the parameter value as an RFC 8288 quoted-string (removing
outer quotes and unescaping backslashes), yielding "sha256-ABC123".
2. Use this value directly in If-Match or If-None-Match headers
without additional quoting changes.

This ensures that regardless of which endpoint processes the write, all
clients can obtain the updated Semantic Validator while maintaining
correct HTTP semantics.

3.4.4. Patch Formats (Informative)

When using PATCH [RFC5789] for partial updates, servers SHOULD support one or
more of the following standard patch formats:

* JSON Merge Patch ([RFC7396]): application/merge-patch+json for simple field-level merging. Suitable for most partial updates.
* JSON Patch ([RFC6902]): application/json-patch+json for operation-based updates (including array manipulation).

Servers SHOULD document which patch formats they accept. Clients SHOULD
use the Accept-Patch header (if provided) to discover supported formats.

Example: 428 Response for Missing Precondition

When a client attempts a state-changing operation without If-Match, the
server SHOULD return a 428 response:

```
PATCH /api/article/123 HTTP/1.1
Host: example.com
Content-Type: application/json

HTTP/1.1 428 Precondition Required
Content-Type: application/problem+json

{
  "type": "https://example.com/errors/precondition-required",
  "title": "Precondition Required",
  "status": 428,
  "detail": "If-Match header is required for CRS mutations"
}
```

Alternative: 400 Response When 428 Is Not Available

If the server environment does not support 428, it MAY use 400 with a
clear error message:

```
PATCH /api/article/123 HTTP/1.1
Host: example.com
Content-Type: application/json

HTTP/1.1 400 Bad Request
Content-Type: application/problem+json
```

```
{
  "type": "https://example.com/errors/missing-precondition",
  "title": "Missing Required Precondition",
  "status": 400,
  "detail": "If-Match header is required for state-changing
  operations on AST resources"
}
```

Example: 412 Response with Problem Details

When a client's If-Match precondition fails, the server SHOULD return a
Problem Details response indicating the current state:

```
PATCH /api/article/123 HTTP/1.1
Host: example.com
Content-Type: application/json
If-Match: "sha256-ABC123OLDVALUE456789STALE0123"

HTTP/1.1 412 Precondition Failed
Content-Type: application/problem+json
Link: </api/article/123>; rel="state"; type="application/json";
  state-etag="\"sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F\""

{
  "type": "https://example.com/errors/precondition-failed",
  "title": "Precondition Failed",
  "status": 412,
  "detail": "The resource state has changed since you last read
  it.",
  "current-etag": "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F",
  "provided-etag": "sha256-ABC123OLDVALUE456789STALE0123"
}
```

The current SBR validator is conveyed via both the state-etag Link parameter (for header-level access) and the current-etag member in the Problem Details body. This allows clients to immediately retry with the correct If-Match value or refetch the latest state.

3.4.5. Conflict Resolution (Informative)

When a client receives a 412 Precondition Failed response, it SHOULD
follow one of these resolution strategies:

1. Refetch and Retry: Fetch the current SBR to obtain the new ETag and state, then reapply the intended mutation and retry with the updated ETag.

2. Three-Way Merge: If the client maintains the original state it read, the changes it intended to make, and can obtain the current server state, it MAY attempt an automatic merge (similar to version
control systems). This is only appropriate when changes affect non-overlapping fields or can be deterministically combined.

3. User Intervention: Present the conflict to the user (human or automated decision system) for manual resolution, showing both the client's intended changes and the server's current state.

4. Abort: Abandon the update if the conflict indicates the client's
view is too stale or the changes are incompatible.

Servers MAY assist conflict resolution by including the current state or

a conflict diff in the 412 response body (see Problem Details example
above). This reduces round trips and provides clients with immediate
context for resolution.

Retry Example:

```
Client receives 412:
HTTP/1.1 412 Precondition Failed
Content-Type: application/problem+json
Link: </api/article/123>; rel="state"; type="application/json";
  state-etag="\"sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F\""

Client immediately refetches:
GET /api/article/123 HTTP/1.1
If-None-Match: "sha256-ABC123OLDVALUE456789STALE0123"

HTTP/1.1 200 OK
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
{ ...current state... }

Client reapplies mutation with fresh ETag:
PATCH /api/article/123 HTTP/1.1
If-Match: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
```

3.4.6. Idempotency

Servers MAY support an Idempotency-Key header for POST and PATCH to
allow safe retries by automated agents. When a duplicate request is
received with the same idempotency key and identical semantics, servers
SHOULD respond with the original status code, headers (including ETag),
and, if applicable, response body.

Example: Idempotency-Key Usage for Resource Creation

```
POST /api/articles HTTP/1.1
Host: example.com
Content-Type: application/json
Idempotency-Key: client-txn-abc123

{"title": "New Article", "body": "..."}

HTTP/1.1 201 Created
Location: /api/articles/456
ETag: "sha256-NEWVALUE789XYZ"
```

Note on Resource Creation: The If-Match requirement applies to
mutations of existing resources. For creating new resources, clients
typically cannot use If-Match on the target URI (which does not yet
exist). Servers MAY support creation patterns such as:

* POST to a collection endpoint, returning the new resource URI and
SBR ETag in the response (as shown above), or
* PUT with If-None-Match: * to create a resource only if it does
not already exist at that URI.

Once created, subsequent mutations to the resource follow the standard

If-Match protocol.

Note: Idempotency-Key is an opaque token. Per HTTP header grammar, both
token and quoted-string forms are acceptable; the exact format is
implementation-defined.

Idempotency-Key behavior is intentionally profile-neutral.
Deployments that support this header SHOULD document replay windows,
storage scope, and duplicate-detection semantics.

3.4.7. Asynchronous Updates

For mutations that cannot complete synchronously, servers MAY
respond with
202 Accepted and a Location header pointing to a status resource.

When the mutation completes:

* The status resource SHOULD indicate completion and provide the
URI of
the updated resource.
* A subsequent GET on the updated resource MUST return the new SBR
ETag.
* Clients SHOULD poll the status resource or use server-sent
events /
webhooks if available.

The original If-Match precondition applies at the time the 202 is
issued; the server MUST reject the operation with 412 Precondition
Failed if the precondition fails at that point, even if processing
is
deferred.

3.5. SBR Endpoint Headers

To ensure correct caching and transfer behavior, SBR endpoints
SHOULD
include the following headers:

* Cache-Control: SBR responses SHOULD include Cache-Control:
no-transform to prevent intermediaries from modifying the
representation (which would invalidate the Semantic Validator).

* Vary: For single-URI deployments serving multiple media types
via
content negotiation (e.g., JSON and HTML from the same URI), set
Vary: Accept to ensure caches differentiate by requested media
type.
Since SBR endpoints MUST NOT apply content-coding (per Encodings
and ETags),
Vary: Accept-Encoding is not applicable to the SBR itself.

Note on Content Codings: The Semantic Validator (SBR ETag) is
computed from the canonical, uncompressed representation bytes.
SBR
endpoints MUST NOT apply content-coding; the Cache-Control: no-
transform
directive prevents intermediaries from introducing encoding in
transit.

* Accept-Ranges: SBR endpoints SHOULD include Accept-Ranges: none
if range requests would break the semantic integrity of the
representation (e.g., partial JSON structures).

Example SBR Response Headers:

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
Cache-Control: no-cache, no-transform
Vary: Accept
Accept-Ranges: none
Content-Digest:
  sha-256=:RK63H5GH1GQJQM8A7LN7RP8VNR7FVXWH1TQKJM8QZH4=:
```

3.6. Caching Considerations

3.6.1. SBR Caching

AST caching behavior follows HTTP caching semantics [RFC9111].

SBR endpoints SHOULD follow these caching practices:

* Non-Negotiated SBR: Prefer a single canonical media type for the SBR to avoid cache fragmentation.

* Strong ETag: The SBR MUST expose a strong ETag as the Semantic Validator.

* Cache-Control: Use Cache-Control: no-cache, no-transform to require revalidation while allowing storage. If the SBR contains sensitive or user-specific data, use Cache-Control: private or no-store as appropriate.

* Vary: For single-URI deployments serving multiple media types via content negotiation, set Vary: Accept to ensure caches differentiate by media type. Since SBR endpoints MUST NOT apply content-coding (per Encodings and ETags), Vary: Accept-Encoding is not applicable to the SBR.

* Accept-Ranges: Consider Accept-Ranges: none to prevent partial-byte range requests that could create ambiguity in state validation.

* 304 Revalidation: Support conditional requests with If-None-Match and respond with 304 Not Modified when the SBR ETag matches.

* ETag Scope: Do NOT reuse the SBR ETag on Projected endpoints, as this would conflate semantic and byte-level validation.

3.6.2. Projected Representation Caching

Projected Representations SHOULD be cached using standard HTTP caching mechanisms:

* Representation-Specific ETags: Use their own ETags based on the byte content of each projection.

* Standard Cache-Control: Apply appropriate Cache-Control directives (e.g., max-age, public, private) based on the projection’s volatility and sensitivity.

* Client-Side Invalidation: Clients that perform CRS mutations SHOULD invalidate their own cached Projected Representations after a successful write or upon receiving a 412 Precondition Failed

response. Note that shared caches and intermediaries will not automatically invalidate projections based on SBR changes.

4. Discovery of Synchronized Representations

To enable agents to discover different representations of the same AST resource, servers SHOULD expose relationships between representations using the HTTP Link header [RFC8288].

4.1. The state Relation (SBR Discovery)

Projected Representations SHOULD include a Link header with rel="state" pointing to the State-Bearing Representation (SBR). This allows clients consuming a Projected Representation to discover the SBR for synchronization or editing.

The state relation identifies the authoritative SBR for concurrency control purposes. Clients MUST use the ETag from the state target (the SBR) for all If-Match and If-None-Match preconditions on state-changing operations.

Example (Projected Representation response):

```
HTTP/1.1 200 OK
Content-Type: text/html
ETag: "html-v1"
Link: </api/article/123>; rel="state"; type="application/json"
```

4.2. The alternate Relation (General Cross-Representation Navigation)

The alternate link relation type is RECOMMENDED for general navigation between synchronized representations of the same AST resource.

When providing an alternate link:

* the type attribute MUST be present to indicate the media type of the target representation (for example, application/json, text/html); and
* the profile attribute MAY be used to indicate a specific schema or capability set of the target representation.

Example (SBR response linking to Projected):

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
Link: </article/123.html>; rel="alternate"; type="text/html"
```

Note: The state relation specifically identifies the SBR for concurrency control purposes. In contrast, rel="alternate" is for general cross-representation navigation and does not imply concurrency semantics. Clients MUST NOT assume that a representation linked via alternate participates in the AST concurrency protocol.

4.3. Bidirectional Linking

To ensure agents can navigate between representations regardless of

their entry point, linking SHOULD be bidirectional.

* If Representation A links to Representation B, then Representation B SHOULD link back to Representation A (or to a canonical URI that resolves to A).

This ensures that an agent encountering a structural representation (for example, JSON) can identify the corresponding rendered representation (for example, HTML) for attribution, verification, or human inspection.

4.4. Profile Advertisement

To enable clients to detect whether a resource implements AST semantics, servers SHOULD advertise this using the profile link relation [RFC6906] via one of the following mechanisms:

Link Header with Profile Relation:

```
HTTP/1.1 200 OK
Content-Type: application/json
Link: <https://datatracker.ietf.org/doc/draft-jurkovikj-
  httpapi-agentic-state/>; rel="profile"
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
```

Content-Type Profile Parameter:

```
HTTP/1.1 200 OK
Content-Type: application/json;
  profile="https://datatracker.ietf.org/doc/draft-jurkovikj-
  httpapi-agentic-state/"
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
```

This allows clients to detect AST resources and apply appropriate concurrency control workflows (If-Match preconditions, 412 handling, etc.) instead of assuming traditional HTTP semantics.

Implementations MAY advertise either a versioned profile URI (e.g., .../draft-jurkovikj-httpapi-agentic-state-01) or the versionless profile URI (e.g., .../draft-jurkovikj-httpapi-agentic-state/). Clients SHOULD accept both as equivalent identifiers for the purposes of feature detection.

5. Integrity and Transport

In standard HTTP usage, ETags often serve a dual purpose: validating the semantic state for caching/concurrency, and validating the integrity of the response bytes (to detect transport corruption).

In the AST profile, the State-Bearing Representation (SBR) ETag serves as the Semantic Validator for the Canonical Resource State (CRS). Projected Representations MAY have their own representation-specific ETags for caching purposes, but these MUST NOT be used as Semantic

Validators for concurrency control.

5.1. Separation of State and Integrity

To allow clients to verify the integrity of the payload bytes independently of the semantic state, AST servers SHOULD use the Content-Digest header field defined in [RFC9530].

* ETag (SBR): validates the *Canonical Resource State* (logical truth) and is used with If-Match and If-None-Match for concurrency control.

* ETag (Projected): MAY be used for caching individual representations, but MUST NOT be used for state-changing preconditions.

* Content-Digest: validates HTTP content according to [RFC9530], independently of CRS validation.

Clients MUST NOT send If-Match or If-None-Match preconditions using a Projected Representation’s ETag when interacting with the SBR
endpoint. Only the SBR’s own ETag participates in concurrency control
logic.

Example Response (SBR):

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "sha256-RK63H5GH1GQJQM8A7LN7RP8VNR7F"
Content-Digest:
  sha-256=:RK63H5GH1GQJQM8A7LN7RP8VNR7FVXWH1TQKJM8QZH4=:
```

Example Response (Projected):

```
HTTP/1.1 200 OK
Content-Type: text/html
ETag: "html-v1"
Link: </api/article/123>; rel="state"; type="application/json"
Content-Digest: sha-256=:X8Z9Y7W6V5U4T3S2R1Q0P9O8N7M6L5K4=:
```

This separation allows intermediaries and clients to detect payload
integrity failures without conflating them with state changes.

In AST, the SBR ETag serves as a semantic validator for the Canonical
Resource State. Clients SHOULD use Content-Digest for verifying HTTP
content integrity. Outside of AST contexts, traditional ETag usage for caching and integrity remains unchanged.

When content-coding is applied to non-SBR (Projected) representations,
servers MAY include Content-Digest and/or Repr-Digest [RFC9530] according to the Digest Fields content-coding rules. Clients that need
to verify the unencoded representation MAY use Unencoded-Digest and
express preferences with Want-Unencoded-Digest [I-D.ietf-httpbis-unencoded-digest], if provided/supported by the server.

5.2. Request Integrity (Client-Sent Digests)

Clients MAY include Content-Digest or Repr-Digest [RFC9530] on state-changing requests to allow servers to verify payload integrity before processing.

Servers that choose to validate request digests:

* SHOULD document which digest algorithms they accept;
* MUST reject requests with unsupported algorithms or malformed digests with an appropriate error response; and
* SHOULD NOT echo the server-computed digest in error responses to avoid potential oracle attacks.

When returning errors for digest validation failures, servers MAY use the problem types defined in [I-D.ietf-httpapi-digest-fields-problem-types]. These problem types will be registered in the HTTP Problem Types registry upon publication of that specification; servers SHOULD use the type member of the Problem Details response to reference the appropriate URI as defined in that specification.

Example digest validation error:

```
HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

{
  "type": "https://iana.org/assignments/http-problem-
  types#digest-invalid-values",
  "title": "Invalid Digest Values",
  "status": 400,
  "detail": "The sha-256 digest value has incorrect length"
}
```

Request digest validation is independent of AST concurrency control; If-Match remains the mechanism for preventing lost updates, while digests verify payload integrity.

6. Computing and Managing Semantic Validators

AST does not mandate a single algorithm for computing Semantic Validators. However, deployment predictability benefits from consistent approaches.

6.1. Canonicalization

A common pattern is to derive the Semantic Validator from a canonical serialization of the CRS. For example:

1. Serialize the CRS to a structured format (for example, JSON).
2. Canonicalize the serialization (for example, by sorting keys and controlling numeric formats).
3. Compute a hash over the canonical form (for example, SHA-256).
4. Encode and prefix the hash to produce the ETag value.

Servers SHOULD ensure that:

* fields that are purely operational or volatile (for example, last
access timestamps) are excluded from the canonical form; and
* the canonicalization process is deterministic across deployments.

Servers MAY use [RFC8785] JSON Canonicalization Scheme (JCS) or an equivalent deterministic procedure. When the SBR is JSON, servers MUST
use [RFC8785] or a profile-specified, equivalently deterministic canonicalization prior to computing the SBR validator.

6.2. Storage Models

Servers MAY:

* compute Semantic Validators on demand from the CRS; or
* store precomputed Semantic Validators alongside the CRS.

When validators are stored:

* they SHOULD be updated atomically with the CRS; and
* they MUST be invalidated or recomputed whenever the CRS changes.

When validators are computed on demand, implementations SHOULD ensure that the cost of computing the validator is small relative to the
cost of generating the representation.

6.3. Relationship to Application Backends

AST is agnostic to how applications store and manage CRS. For example:

* a SQL application might treat a row (or set of rows) as CRS, with the
Semantic Validator derived from a version column and relevant fields;
* a document store might treat a document body plus metadata as CRS;
* a CMS might treat a block tree or content graph as CRS.

The only requirement is that the Semantic Validator change whenever the
logical CRS changes, regardless of how many representations exist.

6.4. Non-AST Resources and Fallback Behavior

Servers MAY implement AST for some resources and not others.

For resources that do not follow AST:

* servers MAY continue to use representation-specific ETags as byte
validators;
* servers MAY accept unconditional writes (no If-Match required), as
in traditional HTTP APIs; and
* clients MUST NOT assume AST semantics.

Servers SHOULD clearly document which resources implement AST.

7. Conformance Requirements Summary

This section provides a checklist of normative requirements for

AST
conformance. Implementations MUST satisfy all items marked
MUST/REQUIRED;
items marked SHOULD/RECOMMENDED are strongly advised.

7.1. Server Requirements

7.1.1. State-Bearing Representation (SBR)

- [ ] MUST designate exactly one representation as the SBR for
each
AST resource
- [ ] MUST assign a strong ETag to the SBR that changes whenever
the
CRS changes
- [ ] MUST NOT apply content-coding to SBR responses (Content-
Encoding
MUST be absent)
- [ ] SHOULD compute SBR ETag from canonicalized CRS (using RFC
8785
for JSON)
- [ ] SHOULD include Cache-Control: no-transform on SBR responses

7.1.2. Concurrency Control

- [ ] MUST evaluate If-Match preconditions on state-changing
operations
using strong comparison [RFC9110]
- [ ] MUST respond with 412 Precondition Failed when If-Match does
not match current SBR ETag
- [ ] SHOULD respond with 428 Precondition Required when If-Match
is missing on state-changing operations
- [ ] SHOULD return Problem Details [RFC9457] body for 412, 428,
and 400 (missing precondition) responses

7.1.3. Discovery and Linking

- [ ] SHOULD include Link: rel="state" header on Projected
Representations pointing to the SBR endpoint
- [ ] SHOULD advertise AST profile using Link: rel="profile" or
Content-Type profile parameter
- [ ] SHOULD include bidirectional rel="alternate" links between
representations

7.1.4. Prohibited Actions

- [ ] MUST NOT reuse the SBR ETag as the ETag for Projected
Representations
- [ ] MUST NOT use weak entity-tags for the SBR validator
- [ ] MUST NOT use Content-Encoding on SBR responses

7.2. Client Requirements

7.2.1. Reading State

- [ ] SHOULD use If-None-Match with SBR ETag for conditional
requests to the SBR
- [ ] SHOULD NOT use SBR ETag as If-None-Match against Projected
endpoints

7.2.2. Writing State

- [ ] MUST obtain current SBR ETag before performing state-
changing
operations
- [ ] MUST include If-Match header with SBR ETag on all state-

changing
requests to AST resources
- [ ] MUST NOT use Projected Representation ETags as If-Match
preconditions for CRS mutations
- [ ] SHOULD handle 412 responses by refetching current state and
retrying or resolving conflicts

7.2.3. Discovery

- [ ] SHOULD follow rel="state" links from Projected
Representations
to discover SBR endpoints
- [ ] SHOULD check for rel="profile" links to detect AST resources

7.3. HTTP Status Codes

AST-conforming implementations MUST use these status codes as
specified:

Code - Usage
200 OK - Successful read or write
204 No Content - Successful write with no body (when using
Content-Location pattern)
303 See Other - Redirect to SBR after successful write (optional
pattern)
304 Not Modified - Conditional request matched current ETag
400 Bad Request - Digest validation failure or missing
precondition (if 428 unavailable)
412 Precondition Failed - If-Match precondition did not match
current ETag
428 Precondition Required - If-Match missing on state-changing
operation

7.4. Required Headers

7.4.1. SBR Response Headers

- ETag: Strong validator (e.g., "sha256-ABC123...")
- Cache-Control: Should include no-transform
- Vary: Accept for single-URI negotiated deployments
- Link: rel="alternate" to Projected Representations (recommended)

7.4.2. Projected Response Headers

- ETag: Representation-specific validator (optional)
- Link: rel="state" pointing to SBR endpoint (recommended)

7.4.3. Write Request Headers

- If-Match: SBR ETag for precondition evaluation (required)
- Content-Type: Appropriate media type for request body
- Idempotency-Key: For idempotent POST/PATCH (optional)

This checklist is informative; the normative requirements remain
those
specified in the body of this document using RFC 2119 keywords.

8. Security Considerations

This profile relies on existing HTTP security mechanisms. However,
the
synchronization of multiple representations introduces specific
security
considerations.

8.1. Consistency and Race Conditions

The primary security benefit of AST is the prevention of race conditions (Lost Updates). By enforcing If-Match checks against the CRS, the server ensures that an agent cannot overwrite state changes made by another client, even if those changes were made via a different representation.

Applications SHOULD treat failed preconditions (412, 428) as an expected outcome in concurrent editing scenarios and provide appropriate conflict resolution mechanisms.

8.2. Information Leakage via Divergence

If the access control policies for different representations diverge, there is a risk of information leakage.

* Servers MUST ensure that the authorization requirements for accessing a structural representation (for example, JSON) are at least as strict as those for the rendered representation (for example, HTML).

* Servers MUST ensure that a structural representation does not expose sensitive data that is redacted in the rendered representation, unless the client is explicitly authorized to see the raw state.

If different audiences see different subsets of the CRS, the server SHOULD either:

* maintain separate CRS domains with separate Semantic Validators; or
* apply view-specific filtering consistently to all representations exposed to that audience.

8.3. Validator Guessing

Because Semantic Validators are often derived from content (for example, a hash), they may be susceptible to offline guessing attacks if the content has low entropy.

Implementations SHOULD ensure that:

* validators have sufficient entropy; or
* validators are combined with a secret salt; or
* validators are derived from internal versioning mechanisms rather than directly from plaintext content,

if the content version itself is sensitive.

Validators MUST NOT be treated as authentication or authorization tokens.

9. IANA Considerations

9.1. Link Relation Type Registration

This document requests registration of the "state" link relation

type in
the "Link Relation Types" registry maintained by IANA per
[RFC8288]:

* Relation Name: state
* Description: Refers to the State-Bearing Representation (SBR) of
the current resource for concurrency control. The target is the
authoritative CRS view and exposes the strong ETag used for If-
Match
and If-None-Match on CRS-affecting operations.
* Reference: This document (Sections 4.1 and 9.1)
* Notes: The target's media type SHOULD be suitable for machine
processing (e.g., application/json). Clients MUST use the target's
strong ETag (not the source representation's ETag) as the
validator
for CRS preconditions. The type attribute SHOULD be present on the
link to indicate the target's media type.

Rationale for Name Selection:

The name state was chosen over alternatives such as edit,
canonical,
or source because:

* edit (defined in [RFC5023]) implies write capability, which may
not
apply to all clients;
* canonical [RFC6596] identifies the preferred URI for a resource,
not specifically its concurrency-control representation;
* source suggests origin content rather than authoritative state.

The term state directly conveys that the target exposes the
resource's
authoritative state for concurrency purposes, aligning with the
"State-Bearing Representation" terminology used throughout this
specification.

Registration Process and Timing:

This registration is requested upon adoption or publication of
this
specification in the IETF stream. Per IANA registration policy,
Link
Relation Type registrations typically occur after Working Group
adoption
or RFC publication, not from unadopted Internet-Drafts.

Interim Usage:

Implementations deploying this profile prior to IANA registration
SHOULD
use the extension relation type URI:

https://datatracker.ietf.org/doc/draft-jurkovikj-httpapi-agentic-
state/rels/state

This URI-based extension relation is immediately usable without
registry
conflicts and follows [RFC8288] extensibility mechanisms. Once the
state
relation type is registered by IANA, implementations SHOULD
migrate to
the registered simple string form (rel="state").

Example using URI-based extension relation:

```
Link: </api/article/123>;
  rel="https://datatracker.ietf.org/doc/draft-jurkovikj-
  httpapi-agentic-state/rels/state"; type="application/json"
```

Appendix A. Implementation Status

Note to RFC Editor: Please remove this section before publication.

This section records the status of known implementations of the AST profile at the time of publication. Based on [RFC7942], reviewers are invited to report implementation experiences.

WordPress Dual-Native Implementation

Organization: Independent implementation

Description: A WordPress plugin implementing the AST profile for content management and template synchronization.

Implementation Details:

* SBR Endpoint: Exposes WordPress posts and templates as application/json representations at /wp-json/dual-native/v1/posts/{id} and /wp-json/dual-native/v1/design/templates/...

* Semantic Validators: Uses SHA-256 content hashes as strong ETags, formatted as "sha256-{base64}", computed from canonicalized JSON representation of the CRS.

* Concurrency Control: Enforces If-Match preconditions on all state-changing operations (POST, PATCH). Returns 412 Precondition Failed on ETag mismatches and 428 Precondition Required when If-Match is missing.

* Projected Representations: HTML rendering at canonical WordPress URLs with rel="state" Link headers pointing to the SBR endpoint.

* Discovery: Implements rel="state" link relations on HTML responses and rel="alternate" on SBR responses.

* Multi-Bot Testing: Demonstrated with multiple concurrent agents. Reported testing showed reduced monitoring bandwidth and no lost updates in the tested concurrent-agent workload.

Maturity: Experimental implementation reported in active use.

Coverage: Implements all normative requirements of this specification including SBR designation, strong ETag generation, If-Match enforcement, 412/428 error handling with Problem Details (RFC 9457), and Link header discovery.

Licensing: Open source

Contact: antunjurkovic@gmail.com

Appendix B. Canonicalization Test Vectors (Informative)

This appendix provides test vectors for implementers to verify

canonicalization and ETag computation.

B.1. JSON Canonicalization (RFC 8785)

Input (arbitrary formatting):

```
{
  "status": "published",
  "id": 123
}
```

Canonical form (JCS per RFC 8785):

```
{"id":123,"status":"published"}
```

SHA-256 (hex):

```
f8b47f69857655c0c84feb9427d443933d91891589f5eee6e51
```

Resulting ETag (base64-encoded, example):

```
"sha256-+LR/aYV2VcDIT+uUJ9RD2Sx8DhtEOTPZGJFYn17ublE="
```

The ETag value shown above is the Base64-encoded SHA-256 digest of the
canonical JSON form.

B.2. Numeric Precision

Input:

```
{"value": 1.0}
```

Canonical form (JCS normalizes 1.0 to 1):

```
{"value":1}
```

Implementers MUST ensure their JSON canonicalization produces JCS-compliant output before hashing.

Appendix C. Single-URI Deployment Example (Informative)

This appendix demonstrates a complete workflow using the Single-URI deployment model, where both the Projected Representation (HTML) and the State-Bearing Representation (JSON) are served from the same URI via content negotiation.

Scenario: A blog article at https://example.com/article/123 serves both HTML (for browsers) and canonical JSON (for API clients). An agent wants to read the human-friendly HTML, extract the Semantic Validator, and perform a safe write operation.

C.1. Reading the Projected Representation

The agent first requests the HTML Projected Representation:

```
GET /article/123 HTTP/1.1
Host: example.com
Accept: text/html
```

Server response:

```
HTTP/1.1 200 OK
Content-Type: text/html
ETag: "html-abc123"
Vary: Accept
```

```
Link: </article/123>; rel="state"; type="application/json"

<!DOCTYPE html>
<html>
<head><title>Understanding HTTP Caching</title></head>
<body>
  <article>
    <h1>Understanding HTTP Caching</h1>
    <p class="meta">Status: published | Author: Jane Smith</p>
    <p>HTTP caching is a fundamental optimization...</p>
  </article>
</body>
</html>
```

The Link: </article/123>; rel="state" header indicates the SBR is available at the same URI by requesting application/json.

C.2. Obtaining the SBR Validator

To obtain the Semantic Validator for safe writes, the agent requests the SBR:

```
GET /article/123 HTTP/1.1
Host: example.com
Accept: application/json
```

Server response:

```
HTTP/1.1 200 OK
Content-Type: application/json
ETag: "sha256-K8N9P3M2L5H7"
Vary: Accept
Link: </article/123>; rel="alternate"; type="text/html"
Link: <https://datatracker.ietf.org/doc/draft-jurkovikj-
  httpapi-agentic-state/>; rel="profile"

{
  "id": 123,
  "title": "Understanding HTTP Caching",
  "status": "published",
  "author": "Jane Smith",
  "body": "HTTP caching is a fundamental optimization..."
}
```

The SBR response uses rel="alternate" to link to the HTML projection, enabling bidirectional navigation. The rel="state" link is omitted because this response is the SBR, clients already have the authoritative validator in the ETag header.

The agent now has the Semantic Validator "sha256-K8N9P3M2L5H7" which represents the current Canonical Resource State.

C.3. Performing a Safe Write

The agent wants to update the article status to "draft". It sends a conditional write using the Semantic Validator:

```
PATCH /article/123 HTTP/1.1
Host: example.com
Content-Type: application/merge-patch+json
If-Match: "sha256-K8N9P3M2L5H7"

{
  "status": "draft"
```

```
}
```

Success response (303 See Other pattern):

```
HTTP/1.1 303 See Other
Location: /article/123
Link: </article/123>; rel="state"; type="application/json";
  state-etag="\"sha256-R9T4Q8P1N7M3\""
```

The agent can now follow the Location header to retrieve the updated SBR, or extract the new Semantic Validator "sha256-R9T4Q8P1N7M3" from the state-etag parameter for subsequent operations.

Alternative success response (204 No Content pattern):

```
HTTP/1.1 204 No Content
Content-Location: /article/123
Link: </article/123>; rel="state"; type="application/json";
  state-etag="\"sha256-R9T4Q8P1N7M3\""
```

Conflict response (state changed):

If another agent modified the article between steps C.2 and C.3:

```
HTTP/1.1 412 Precondition Failed
Content-Type: application/problem+json
Link: </article/123>; rel="state"; type="application/json";
  state-etag="\"sha256-X2Y9Z4W1V8U5\""

{
  "type": "https://example.com/errors/precondition-failed",
  "title": "Precondition Failed",
  "status": 412,
  "detail": "The article has been modified by another agent.",
  "current-etag": "sha256-X2Y9Z4W1V8U5",
  "provided-etag": "sha256-K8N9P3M2L5H7"
}
```

The agent can extract the current Semantic Validator from the state-etag parameter and retry with the updated state.

C.4. Optional: Zero-Fetch Optimization

For monitoring scenarios, the agent can efficiently poll for changes without fetching the full representation:

Pattern 1: HEAD request with If-None-Match

```
HEAD /article/123 HTTP/1.1
Host: example.com
Accept: application/json
If-None-Match: "sha256-R9T4Q8P1N7M3"
```

If unchanged:

```
HTTP/1.1 304 Not Modified
ETag: "sha256-R9T4Q8P1N7M3"
Vary: Accept
```

If changed:

```
HTTP/1.1 200 OK
ETag: "sha256-T5U8V1W4X7Y0"
Vary: Accept
Content-Type: application/json
```

```
Content-Length: 234
```

Pattern 2: GET request with If-None-Match

```
GET /article/123 HTTP/1.1
Host: example.com
Accept: application/json
If-None-Match: "sha256-R9T4Q8P1N7M3"
```

Server returns 304 if unchanged, or 200 with updated content if changed.

This zero-fetch pattern minimizes bandwidth while maintaining accurate change detection based on the Semantic Validator.

10. References

10.1. 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.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017.

[RFC6585] Nottingham, M. and R. Fielding, "Additional HTTP Status Codes", RFC 6585, DOI 10.17487/RFC6585, April 2012.

[RFC9530] Polli, R. and L. Pardue, "Digest Fields", RFC 9530, DOI 10.17487/RFC9530, February 2024.

[RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023.

10.2. Informative References

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022.

[RFC6906] Wilde, E., "The ’profile’ Link Relation Type", RFC 6906, DOI 10.17487/RFC6906, March 2013.

[RFC5023] Gregorio, J., Ed., and B. de hOra, Ed., "The Atom Publishing Protocol", RFC 5023, DOI 10.17487/RFC5023, October 2007.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010.

[RFC6596] Ohye, M. and J. Kupke, "The Canonical Link Relation", RFC 6596, DOI 10.17487/RFC6596, April 2012.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020.

[I-D.jurkovikj-http-semantic-validator] Jurkovikj, A., "Semantic

Validators for HTTP", Work in Progress, Internet-Draft, draft-jurkovikj-http-semantic-validator-00, May 2026.

[RFC6902] Bryan, P., Ed. and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Patch", RFC 6902, DOI 10.17487/RFC6902, April 2013.

[RFC7396] Hoffman, P. and J. Snell, "JSON Merge Patch", RFC 7396, DOI 10.17487/RFC7396, October 2014.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016.

[I-D.ietf-httpbis-unencoded-digest] Pardue, L. and M. West, "HTTP Unencoded Digest", Work in Progress, Internet-Draft, draft-ietf-httpbis-unencoded-digest-04, March 2026.

[I-D.ietf-httpapi-digest-fields-problem-types] Kleidl, M., Pardue, L., and R. Polli, "HTTP Problem Types for Digest Fields", Work in Progress, Internet-Draft, draft-ietf-httpapi-digest-fields-problem-types-05, March 2026.

Author’s Address

Antun Jurkovikj
Email: antunjurkovic@gmail.com