The Missing HTTP Verb: STRUT

Victor Bona
Independent Researcher
Software Engineer
Blumenau, Brazil
victor.bona@hotmail.com

***Abstract*: HTTP methods encode the intent of a request, but existing creation methods leave an awkward gap for resources whose representation is mostly synthesized by the server. POST handles server-directed creation but is not idempotent by default; PUT is idempotent but expects the client to choose the target resource and provide the representation. This position paper proposes STRUT, an idempotent but unsafe HTTP method for server-driven, minimal-input creation. A STRUT request asks the server to create, or ensure the existence of, a resource from a stable target and a small amount of identifying input. We define STRUT's semantics, contrast it with POST plus Idempotency-Key, conditional PUT, WebDAV's `MKCOL`, and the `Prefer` header, and discuss security, caching, intermediaries, browser behavior, and deployment. The contribution is not a completed standard, but a candidate semantic slot for API designs that currently rely on empty POST requests, ad hoc idempotency contracts, or unsafe GET side effects.**

***Index Terms*: HTTP, REST API, HTTP Semantics, STRUT, Idempotent Methods, Resource Creation, Web Services**

I. Introduction

The Hypertext Transfer Protocol (HTTP) is the fundamental communication protocol of the World Wide Web, handling interactions between clients and servers. Over time, as web applications have grown more complex, HTTP has evolved by introducing additional methods (often informally called verbs) such as GET, POST, PUT, DELETE, and PATCH to handle different kinds of operations on web resources. These methods are key to RESTful architectures, acting as standardized actions that can be performed on resources identified by Uniform Resource Identifiers (URIs) [3]. Each standard HTTP method has well-defined semantics intended to ensure interoperability and adherence to web architecture principles.

Despite the rich set of existing HTTP methods, there are scenarios where none of the standard methods cleanly fit the requirements while preserving their intended semantics. One such scenario is *minimal-input resource creation*: cases where the client possesses only a small amount of identifying information, while the server is responsible for determining the rest of the representation from policy, defaults, identity, or internal state. Developers commonly approximate this pattern with an empty POST body, a POST plus an idempotency token, a conditional PUT when the client can choose the URI, or, in weaker designs, a GET endpoint with side effects. These approaches can work, but each pushes part of the contract outside the method semantics.

To address this gap, we propose a new HTTP method named **STRUT**. STRUT is designed for resource creation scenarios where the client supplies only minimal information, such as a stable identifier or reference, and the server generates the full resource according to its own rules. STRUT is defined as an idempotent creation method: repeating the same STRUT request must not create duplicate resources or additional side effects. This property distinguishes STRUT from POST and gives clients a clearer retry model when a response is lost or delayed.

This study has three main objectives: (1) to analyze why current HTTP methods and patterns fall short for minimal-input resource creation, and to review related work or extensions that have attempted to address similar problems; (2) to formally define the STRUT method, detailing its semantics and how it fits into the existing HTTP protocol framework (including compliance with relevant RFCs and security considerations), and to illustrate STRUT's use through real-world inspired use cases; (3) to evaluate the potential benefits and impacts of STRUT, including a discussion of performance implications (e.g., reduced payload sizes and communication overhead) and any trade-offs or challenges associated with adopting a new HTTP method at the protocol level.

The remainder of this paper is organized as follows. Section II reviews related work and the limitations of current HTTP methods in scenarios requiring server-driven resource creation, highlighting compliance and efficiency issues. Section III analyzes related standards and design patterns that approximate minimal-input creation (including the use of idempotence keys, conditional requests, WebDAV's `MKCOL` method, and the `Prefer` header), and positions STRUT relative to these approaches. Section IV presents the proposed STRUT method in detail, including its formal semantics, how it adheres to or extends HTTP standards, and representative use cases (such as one-time password generation and creation of entity relationships) that motivate its usage. Section V discusses important implementation considerations for STRUT from both client and server perspectives, including how HTTP stacks could support a new method and ensure idempotence. Section VI provides a broader discussion, including security implications, deployment challenges, and the benefits and potential hurdles of adopting STRUT in real-world systems, as well as comparisons to existing practices. Finally, Section VII concludes the paper and suggests directions for future work, such as steps toward standardization of STRUT via the IETF and exploring its adoption in web frameworks.

II. Related Work and Limitations of Current HTTP Methods

HTTP defines a standard set of methods that clients use to indicate the desired action on a resource identified by a URI [1]. The core methods standardized in the HTTP Semantics specification (RFC9110) [1] include GET, POST, PUT, DELETE, HEAD, and OPTIONS, with PATCH introduced later in RFC5789 [2]. Each method has specific rules, semantics, and intended use cases as defined by the standards. This section reviews the relevant HTTP methods and analyzes why they are not well-suited for the minimal-input resource creation scenario that motivates STRUT. We focus on how each method's standard behavior creates limitations when the client cannot or should not fully describe the resource to be created.

A. Overview of Standard HTTP Methods

GET: The GET method is used to retrieve data from a specified resource. According to RFC 9110, GET requests should only fetch data and produce no side effects on the server [1]. This makes GET a safe and idempotent operation by definition:

Safe, meaning it does not modify the server's state.

Idempotent, meaning that issuing the same GET request multiple times has the same effect as a single request.

Because GET is defined as safe and idempotent, it is intended purely for retrieval. HTTP does not forbid a request body on GET, but its semantics are undefined and widely unsupported; clients should not rely on it. Parameters are therefore typically conveyed in the target URI (path or query). *Limitations*: GET is inherently unsuited for creating new resources. Using GET to create or modify resources would violate its safe, side-effect-free contract and break HTTP compliance. Moreover, trying to encode necessary data for creation as query parameters in a GET request is both semantically incorrect and limited (especially for complex or sensitive data). In short, GET cannot be repurposed for resource creation without contravening HTTP standards.

POST: The POST method is typically used to submit data to be processed to a specified resource, often resulting in a new subordinate resource being created or an existing resource being modified on the server. RFC 9110 notes that a successful POST to a collection may create a new resource as a subordinate of the specified URI [1]. Unlike GET, POST is neither safe nor idempotent:

It is *not safe* because it can change the server's state (e.g., inserting new data).

It is *not idempotent* because sending the same POST request multiple times can have additional side effects or create multiple resources.

Limitations: POST typically expects the client to provide a request body containing the data for the new resource (such as a form submission or JSON payload). If the client only has minimal information and relies on the server to fill in the details, constructing a full request body is impractical. An empty or nearly empty POST request (just to trigger server-side logic) is semantically awkward and creates an ambiguous contract, since the request itself does not describe the resource, making the API less self-documenting. Additionally, because POST is not idempotent, if a client repeats a POST due to not receiving a response (e.g., due to a network timeout), the server might create duplicate resources or perform the action multiple times. This complicates error handling and recovery. Using POST without meaningful content deviates from its intended use and could confuse clients or developers maintaining the API, leading to poor compliance with the principle of least surprise in API design.

PUT: The PUT method requests that the server create or replace a resource at a client-defined URI with the enclosed representation. RFC 9110 specifies that PUT should create a new resource or replace the target resource with the payload provided in the request [1]. PUT is defined to be idempotent (repeating the same PUT yields the same result) but not safe (it modifies state). *Limitations*: PUT requires the client to send the full representation of the resource to be created or updated. This is problematic for our scenario because the client, by assumption, does not have all the details of the resource. If the server is supposed to determine those details, the client cannot supply a complete representation as required by PUT. Additionally, PUT generally requires the client to know and specify the exact URL of the resource to create. In many resource creation workflows, the server chooses the resource identifier (for example, generating a new ID or determining a location in a hierarchy), which means the client cannot know the final URI in advance. While PUT's idempotence is desirable, using PUT when the server will diverge from the provided content (by filling in missing pieces or deciding its own content) would violate the expectation that the same request repeated yields the same stored resource. In other words, if the server adds varying information on each creation, it would conflict with PUT's idempotent nature unless carefully managed.

DELETE: The DELETE method deletes the specified resource. It is an idempotent operation (deleting the same resource once or multiple times ends with the resource gone) but not safe, since it alters server state [1]. *Limitations*: DELETE is not relevant for resource creation at all; its purpose is the opposite, removal of resources. It offers no support for creating new resources with minimal input.

PATCH: The PATCH method, defined in RFC 5789, is used for partial modifications of an existing resource [2]. PATCH carries in its request body a set of changes to apply to the resource (often in a format like JSON Patch or an incremental diff). *Limitations*: PATCH is only applicable to resources that already exist. It cannot create new resources; it is meant to update parts of a resource identified by the client. In a scenario where the client has minimal information for a new resource, PATCH is not an option because there is nothing to "patch" until a resource exists. Furthermore, PATCH still requires the client to specify the modifications, which in our scenario the client is trying to avoid doing (relying instead on server-side logic).

HEAD, OPTIONS, TRACE: The HEAD method is identical to GET except that it returns only headers and no response body; it is safe and idempotent, used typically for obtaining metadata (e.g., checking if a resource exists or getting content length) [1]. OPTIONS is used to discover server-supported methods and capabilities for a resource, and TRACE is a diagnostic method that echoes the received request. *Limitations*: These methods are not used for resource creation. HEAD cannot create or modify resources (it doesn't even return a body), OPTIONS is merely for querying supported operations, and TRACE is a specialized debugging method often disabled for security reasons. None of these help in scenarios where a client wants to initiate creation of a new resource with minimal input.

B. Limitations in Minimal-Input Creation Scenarios

The review above shows that the primary methods used to create resources in HTTP are POST and PUT. We now summarize the specific challenges and limitations that arise when using existing methods for creating a resource with minimal client-provided data, where the server is expected to generate most of the resource content:

Dependence on Client-Provided Data: Methods like POST and PUT require the client to send the resource data in the request.

For POST, the client typically must include a request body with the data to process or store. If the client lacks detailed information (and the server is meant to supply it), an almost empty or placeholder body would be required, which is inefficient and unconventional.

PUT demands a full representation of the resource. A client that only knows an identifier (or a few parameters) cannot construct the complete representation without server help, thus cannot properly use PUT in such cases.

Implication: Current creation methods make the client responsible for data it may not have, leading to impractical API designs.

Compliance Issues with Workarounds: Lacking a suitable method, developers might misuse other methods, undermining HTTP standards:

Using GET (or HEAD) to trigger creation violates the intended safety of GET. Such misuse conflicts with the protocol's guarantees and can lead to caching or pre-fetching systems inadvertently causing changes on the server, a serious compliance and safety issue.

Using POST with an empty or minimal body deviates from POST's intended semantics. It may work in practice but creates an ambiguous contract, because the request does not clearly describe the resource, making the API less self-explanatory and potentially causing inconsistent interpretation across implementations.

Implication: These workarounds break the spirit (and sometimes the letter) of the HTTP specification, risking unpredictable behavior across different servers or user agents.

Communication Inefficiency: If a client is forced to send data just to satisfy a method's requirements:

The client may send unnecessary or dummy data, increasing payload size and network traffic with no real benefit.

The server then has to ignore or override this data with its own logic, which is extra processing. This roundabout communication wastes resources on both sides.

Implication: Not having a suitable method leads to more verbose interactions and potential performance penalties.

Mismatch with Server-Driven Resource Generation: None of the existing methods provide a clean mechanism for the server to take the lead in resource creation.

Server heuristics or internal rules might determine resource attributes (such as default values, computed fields, or relationships to other entities) when only a minimal key or trigger is provided by the client. There is no standard way for a client to signal "please create this resource using your logic" under current methods.

The lack of a standard method means developers may resort to custom endpoints (e.g., "/generateReport" as an RPC-like endpoint using POST) rather than a uniform interface. This reduces the consistency and predictability of the API ecosystem.

idempotence and Retry Challenges: In scenarios of network failure or uncertain outcomes, an idempotent operation is highly desirable so that clients can safely retry.

POST is not idempotent, so if a client times out and retries a POST request, it risks creating a duplicate resource or performing the action twice. This necessitates additional client or server logic (such as request de-duplication or unique tokens) to guard against duplicates.

PUT is idempotent by design, but as noted, it cannot be directly applied when the server generates content unpredictably. If a server attempted to provide missing content on each PUT request, repeated PUTs could actually produce different resources (violating idempotence), or the server would have to detect an existing resource and refuse changes on subsequent calls, which complicates the protocol's use.

Implication: The lack of an idempotent yet server-driven creation method leaves a gap where reliable retry semantics are hard to achieve for certain operations.

Security Considerations: Misusing HTTP methods can introduce security vulnerabilities and inconsistent behavior:

If GET is misused to perform creation, it may be cached by intermediaries or even pre-fetched by browsers or crawlers, leading to unintended resource creations without explicit client action. This violates assumptions about GET and can be dangerous.

Non-standard use of existing methods can confuse security mechanisms (like CSRF protections or firewall rules) that are geared toward typical semantics. For example, an API might not expect state-changing operations on a GET request and thus not apply the same CSRF tokens or authentication checks, potentially opening an attack vector.

Implication: A proper dedicated method for such actions can allow consistent application of security measures and clear intent, as opposed to improvised solutions that might be handled differently across implementations.

These limitations highlight the need for a new approach. Over the years, HTTP has been extended to address other gaps (for instance, the introduction of PATCH for partial updates [2] addressed the inability of PUT to do partial modifications). In the same spirit, we propose addressing the gap in minimal-input resource creation by introducing a new method, STRUT, rather than contorting existing methods into roles for which they were not intended.

III. Related Standards & Patterns

Before formalizing STRUT, it is important to compare it with existing standards and design patterns that developers have used to approximate or solve similar problems. Several approaches in current practice or proposals aim to handle resource creation with minimal client input or to enforce idempotence on creation operations. We examine the most relevant ones here and contrast them with STRUT's approach.

A. POST with Idempotency-Key Header

One common pattern to safely retry non-idempotent operations (like POST) is the use of an `Idempotency-Key` header [6]. In this approach, the client includes a unique key (e.g., a UUID) with a POST request. The server, on receiving a POST with an idempotency key, stores or reconstructs the result of that request keyed by the provided value. If the client retries the request (e.g., due to a timeout) with the same key, the server recognizes the duplicate and returns the original outcome instead of performing the action again. This mechanism, which is being standardized in an IETF draft [6], effectively makes a specific POST request idempotent from the client's perspective.

Comparison with STRUT: Both the Idempotency-Key pattern and STRUT aim to prevent duplicate side effects and allow safe retries. However, there are key differences:

The Idempotency-Key approach is a client-supplied coordination mechanism: the client must generate and supply a unique key for each operation, and the server must implement storage and lookup for those keys. STRUT, in contrast, makes idempotence part of the method semantics: the client repeats the same request (same target and parameters), and the server ensures no duplicate resource is created. This shifts duplicate detection from client key management to the server's application logic and natural resource constraints.

Using idempotence-Key still relies on POST semantics for creation. The client is typically still sending a request (potentially with a body, even if minimal), and the server's logic must handle the content. STRUT provides a more explicit, protocol-level indication that the operation is a *creation with minimal input*, rather than a general POST.

Idempotency keys are a general solution for non-idempotent requests, including payment transactions and other operations where replay must be controlled. STRUT is narrower: it is tailored for resource creation scenarios with minimal input. STRUT can therefore be seen as a specialized semantic tool, while idempotency keys remain the broader compatibility pattern.

In summary, while POST plus Idempotency-Key can achieve a similar outcome (one resource created for multiple identical attempts), STRUT provides this guarantee by design for this specific class of operation. STRUT's added value is clearer intent at the protocol level: the request itself indicates an idempotent create-if-not-exists operation, simplifying client code and making APIs more self-descriptive.

B. PUT with `If-None-Match: *`

HTTP conditional requests offer another way to achieve safe resource creation. A client can send a PUT request with an `If-None-Match: *` header (defined in HTTP's conditional request semantics) to mean "create this resource only if it does not exist" (the asterisk is a wildcard matching any current state of the target resource). If the resource at that URI already exists, the server will respond with `412 Precondition Failed` instead of overwriting it. If it does not exist, the server will create it (using the request body as the representation) and respond with `201 Created`. This pattern allows a client to ensure it doesn't accidentally overwrite an existing resource, providing a form of idempotent "create-only-once" semantics.

Comparison with STRUT: There are a few significant distinctions:

The PUT + `If-None-Match: *` approach still requires the client to provide the full resource representation in the request body, since it is a PUT. It assumes the client knows how the resource should look (aside from server-assigned identifier perhaps). STRUT, on the other hand, is explicitly for cases where the client cannot or does not provide the representation.

If the resource already exists, a repeated STRUT request would ideally return a success (200 OK or 204 No Content) indicating "nothing new was done" (idempotent outcome). In contrast, a repeated PUT+`If-None-Match:*` returns a 412 error on the second attempt (because the resource exists). From a state perspective, both are idempotent (the resource isn't duplicated), but STRUT aims to make the repeated call seamlessly successful (no error needed to indicate nothing happened). This can simplify client logic (no special case for handling a 412).

The conditional PUT pattern is limited to scenarios where the client can choose the resource URI (since it must target a specific URI with the PUT). In many creation cases (e.g., generating a new ID or letting the server decide the URI), clients use POST specifically because they *cannot* determine the URI. STRUT, like POST, allows the server to decide the resource URI (returning it in a Location header on creation), which covers use cases where `If-None-Match:*` PUT is not applicable.

Overall, `If-None-Match:*` on PUT is a useful trick to avoid duplicates when the client controls the URI and representation. STRUT generalizes the idea of a safe create-if-not-exists operation to scenarios where the client provides minimal input and expects the server to handle the rest, which cannot be handled by PUT without violating its requirement that the client provide the full resource representation.

C. WebDAV `MKCOL` Method

The Web Distributed Authoring and Versioning (WebDAV) extensions to HTTP (RFC 4918) introduced several new methods, one of which is `MKCOL` (Make Collection) [5]. The MKCOL method is used to create a new collection (analogous to a directory) at the specified URI. A MKCOL request to a URI typically creates an empty collection resource (often with no request body allowed). MKCOL is idempotent (issuing it twice on the same URI either creates the collection once and then fails or returns an error on the second attempt, but does not create two collections).

Comparison with STRUT: MKCOL is conceptually similar to STRUT in that it creates a resource (a collection) without the client supplying a representation of that collection. However:

MKCOL is specific to WebDAV and the creation of directory-like container resources. Its semantics are narrow: you can create collections (and extended MKCOL for creating collections with properties was later defined by WebDAV). STRUT is intended as a general-purpose method that could create any kind of resource for which the server can synthesize the representation.

MKCOL's idempotence works in a way that a second MKCOL usually fails with `405 Method Not Allowed` or `409 Conflict` if the collection already exists. As noted, STRUT would aim to succeed safely on a second call (with a 200/204 response), although both approaches ensure no duplicate resource state.

WebDAV methods like MKCOL are not universally supported by all HTTP clients or servers unless WebDAV is enabled, and some intermediaries might block them if not recognized. STRUT, if standardized, would stand alongside methods like PUT and POST as part of the core method set, potentially giving it wider applicability beyond the WebDAV context.

In essence, MKCOL demonstrates that new HTTP methods can be introduced to cover functionality not provided by the original methods (in MKCOL's case, creating a directory). STRUT similarly proposes a new method to cover a gap (server-driven creation). STRUT is broader in scope than MKCOL, but one could view STRUT as analogous to "MKRESOURCE" for arbitrary resources, not just collections.

D. `Prefer` Header Hints

RFC 7240 defines the `Prefer` header, which allows clients to indicate certain preferences about how the server should process a request or what kind of response it would prefer [7]. For example, a client can send `Prefer: return=representation` with a POST request to ask the server to include the created resource's representation in the response (instead of just a 201 with empty body), or `Prefer: return=minimal` to request the opposite. There are also preferences like `handling=lenient` (to ask the server to ignore minor errors).

Although not a direct mechanism for minimal-input creation, one could imagine using a header like `Prefer` to tweak existing methods. For instance, a client might send `Prefer: handling=lenient` or a hypothetical `Prefer: server-fill=true` with a POST or PUT to indicate that the server should fill in unspecified fields or create defaults. Similarly, `Prefer: respond-async` could be used if creation triggers background processing.

Comparison with STRUT: Using `Prefer` headers can influence behavior but has limitations:

The semantics of `Prefer` are intentionally hints, not strict directives; a server can ignore them. Relying on a `Prefer` value to change a fundamental behavior (like turning a POST into an idempotent create-if-not-exists) would be an out-of-band contract, likely not understood by intermediaries or tooling. In contrast, STRUT as a new method is an explicit contract understood at the protocol level.

Prefer is primarily about response formatting (minimal vs full) or processing mode (like asynchronous), not about core method semantics. There isn't a defined `Prefer` token that means "please ensure idempotence" or "use server defaults." If such a convention were invented, it would be essentially a custom contract (similar to creating a new method, but encoded in a header).

From a design standpoint, overloading POST/PUT with new semantics via headers can lead to confusion and complexity. For example, an intermediary cache or firewall might not recognize that `Prefer: server-fill=true` means the request should be treated differently, whereas a new method token STRUT is unambiguous.

STRUT's value, compared to using Prefer headers, is that it provides a uniform, explicit method for the pattern of server-side heuristic creation. It doesn't require clients and servers to negotiate new header conventions and it clearly communicates the intent. Prefer remains useful in conjunction with STRUT (for instance, a STRUT request could still use `Prefer: return=representation` to ask for the created resource's representation in the response), but STRUT defines the primary operation itself in the request line rather than hidden in headers.

Summary of Comparisons

Table I provides a comparison of STRUT with standard HTTP methods on key properties discussed, highlighting how STRUT aligns with some (idempotence like PUT) while differing on others (request body usage like POST). Overall, while existing patterns can partially address minimal-input creation (through conventions or additional headers), STRUT's introduction as a uniform method would unify these needs under one standard approach, improving clarity and consistency across systems.

IV. Proposed STRUT Method

In this section, we introduce the STRUT HTTP method as a solution to the problems identified above. We formally define the semantics of STRUT in a manner consistent with HTTP's architectural principles and describe how STRUT requests and responses should be handled. We then discuss how STRUT integrates with existing HTTP standards (ensuring compliance with the latest RFCs) and present use cases that demonstrate STRUT's practical utility.

TABLE I
COMPARISON OF HTTP METHODS AND STRUT PROPERTIES

Method	Safe?	Idempotent?	Cacheable*?	Request Body	Typical Success Codes
GET	Yes	Yes	Yes (default)	(discouraged)	200
HEAD	Yes	Yes	Yes (default)	No	200
POST	No	No	No (unless explicit)	Yes	201/200
PUT	No	Yes	No (unless explicit)	Yes (full)	201/200/204
DELETE	No	Yes	No	No	200/204
PATCH	No	No	No	Yes (diff)	200/204
STRUT (proposed)	No	Yes	No (unless explicit)	Optional (minimal)	201/200/204

*In practice, caches chiefly store GET/HEAD; other methods require explicit directives.

A. Semantics of the STRUT Method

STRUT is a proposed new HTTP method with the primary purpose of creating a resource on the server when the client possesses only an essential identifier or minimal parameters for that resource. The semantics of STRUT are carefully designed to fill the gap between the existing POST and PUT methods:

Action: A STRUT request signals to the server: "Create (or ensure the existence of) a resource identified by the given parameters, using server-side logic to determine all necessary details." In essence, the client provides a hint or key, and the server does the work of instantiating the resource.

Idempotence: STRUT is defined to be *idempotent*. This means if a client issues the same STRUT request multiple times (with identical parameters), the end state on the server will be the same as if it was performed once. In practice, the server should either create the resource on the first request and then recognize that subsequent identical requests refer to an already-created resource (and not duplicate it), or otherwise ensure that duplicate invocations do not produce additional side effects. This behavior aligns with methods like PUT and DELETE, and contrasts with POST's non-idempotence.

Safety: STRUT is *not safe* (it has side effects, since it creates or alters state on the server). This is expected, as its purpose is to create resources. Clients and intermediaries must not treat STRUT as a read-only operation.

Request Syntax: A STRUT request is an HTTP request similar in format to other methods. For example:

```
STRUT /resource/namespace/[identifier]
HTTP/1.1
```

followed by the necessary headers. In most cases, the STRUT request will have no body (the client is not supplying a representation, only using the URI and possibly query parameters or headers for identification). The target URI of the request typically represents a resource namespace or endpoint where the server knows how to create a new item. For example, the client might call `STRUT /users/userID/otp` to signal the server to create a one-time-password resource for user userID. The use of the URI path to convey the minimal identifying information is consistent with RESTful design and URI syntax guidelines [3].

Response Semantics: Upon a successful STRUT request, the server will create the resource (if not already existing) and typically return a success status. The appropriate response code for a new resource creation would be `201 Created`, along with a `Location` header pointing to the newly created resource's URI (if the creation results in a new resource with its own URI). If the resource identified by the request already exists (due to a previous STRUT invocation or other means), the server could return `200 OK` or `204 No Content` to indicate that the request was processed successfully but no new resource was created because it was unnecessary (idempotent outcome). The response may include a representation of the resource in the body (especially on the initial creation, returning the details that were generated).

Error Handling: If the server cannot fulfill the STRUT request (for example, if required parameters are missing or invalid, or the server's logic cannot create the resource for some reason), it should return an appropriate error status. `400 Bad Request` can be used for client-side issues (e.g., the provided identifier is invalid), whereas a `409 Conflict` might be appropriate if a similar resource already exists and the server disallows duplicates under a different identifier. Standard HTTP error codes would be reused where possible to indicate the nature of the failure.

These semantics position STRUT as a creation method that is closer to PUT in terms of idempotence but without requiring the client to define the resource representation, thereby differing from both PUT and POST in crucial ways. STRUT's design is intended to uphold the REST principle of a uniform interface by introducing a new uniform operation for a scenario that currently lacks one, rather than encouraging bespoke endpoint-specific behavior on top of existing methods.

It is important to note that STRUT does not carry a request body in typical use. This is by design: since the client is only providing identification and no content, all necessary information for creation should either be implied by the context (the URI and perhaps query parameters) or already known to the server. In unusual cases where a small amount of additional data is needed (beyond what can be conveyed in a URI), it could be included in headers or a minimal body, but the spirit of STRUT is that the body is generally unnecessary. This makes STRUT requests lightweight and focused.

B. Formal Definition (RFC-Style)

The key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT", and "MAY" in this section are to be interpreted as described in RFC 2119 and RFC 8174.

To define STRUT in a manner consistent with IETF standards, we use normative language (as per RFC 2119 terminology) to specify requirements:

- **Purpose**: The STRUT method is used to request that the server create a resource (or ensure it exists) identified by the target URI and minimal accompanying information. The server **MUST NOT** require the client to provide a full representation of the resource in the request.
- **Idempotence**: STRUT **MUST** be treated as an idempotent method. Multiple identical STRUT requests **MUST NOT** result in the creation of duplicate resources or side effects beyond the first request. The server **MUST** ensure that processing a duplicate STRUT request has no additional effect beyond the initial request. For example, if a STRUT request creates a resource, a subsequent STRUT request with the same parameters for that resource **SHOULD** simply acknowledge that the resource already exists (e.g., by returning a 200 OK or 204 No Content with no new resource created).
- **Safety**: STRUT is unsafe (i.e., it can modify server state), so it **MUST NOT** be considered a safe or read-only method by clients or intermediaries. Any state-changing effects of STRUT **SHOULD** be treated similarly to those of POST or PUT in terms of requiring appropriate precautions (such as authentication and CSRF protection).
- **Request Body**: A STRUT request **SHOULD NOT** contain a significant body. Servers **MUST NOT** expect a full resource representation in the request. An implementation **MAY** allow a small request payload (e.g., a JSON snippet or form fields) if needed to convey extra parameters beyond the URI, but this is optional and context-dependent. In general, the semantics of any STRUT request body are defined by the server's implementation of that particular resource factory endpoint. If a request body is present and understood, the server **SHOULD** use it as additional input for resource creation; if it is not understood or not needed, the server **MAY** ignore it or respond with `400 Bad Request` if it conflicts with the expected usage.
- **Success Responses**: On successfully processing a STRUT request:
  - If a new resource was created, the server **SHOULD** return `201 Created`. The response **MUST** include a `Location` header field containing a URI for the newly created resource. The response **MAY** include a representation of the created resource in the body (for example, a JSON representation of the new resource).
  - If the request did not result in a new resource (e.g., because it already existed), the server **SHOULD** return `200 OK` or `204 No Content` (no new content) to indicate an idempotent success with no new creation. A 200 OK response in this case **MAY** include the representation of the existing resource or a confirmation, whereas 204 would have an empty body.
- **Error Responses**: The server **MUST** handle error conditions using appropriate HTTP status codes:
  - `400 Bad Request` for syntactically or semantically invalid requests (e.g., missing required parameters, invalid identifier format).
  - `401 Unauthorized` or `403 Forbidden` if the client is not permitted to perform the creation (authentication or authorization failure).
  - `409 Conflict` if the request cannot be completed due to a state conflict (e.g., a resource with the same identifier exists and duplicates are not allowed).
  - `405 Method Not Allowed` if the target resource knows about STRUT but does not allow it (the server should include an `Allow` header listing allowed methods).
  - `501 Not Implemented` if the server (or intermediary) does not recognize or support the STRUT method at all.
  - Other 4xx or 5xx codes as appropriate (e.g., `500 Internal Server Error` or `503 Service Unavailable` for server-side failures).
- **Caching**: Responses to STRUT requests are not cacheable by default. Servers **SHOULD** include explicit cache directives (e.g., `Cache-Control`) if returning a representation. General-purpose caches **CANNOT** infer relationships between a created resource and other representations (such as collection listings); invalidation of related representations requires explicit mechanisms (short TTLs, ETag-based revalidation, cache tags/surrogate keys, or application-level purge).

This formal definition ensures that STRUT can be implemented in a manner consistent with HTTP's existing semantics, and that clients, servers, and intermediaries have a clear contract for its behavior.

C. Compliance with HTTP Standards

Any new method like STRUT must integrate with HTTP/1.1 (and HTTP/2+ in practice) in a way that is compliant with how the protocol operates. We consider here how STRUT fits into the existing framework defined by relevant standards:

HTTP Method Registration: HTTP is designed to be extensible; new methods can be introduced via the IETF process (an RFC) and registered with the Internet Assigned Numbers Authority (IANA). STRUT would follow this path, being proposed in an RFC draft for standardization. Upon standardization, it would be added to the HTTP Method Registry. Clients and servers would need to be updated to recognize the method name "STRUT". Intermediaries (like proxies, caching servers, firewalls) that strictly validate method names may require updates or configuration to allow an unknown method. However, many HTTP libraries and intermediaries treat unrecognized methods as opaque and will forward them (perhaps with

reduced functionality like not caching them). A formal standard for STRUT ensures that it can eventually be broadly recognized and properly handled by implementations.

Cache Behavior: By default, HTTP caches (per RFC 9111 [8]) only automatically cache responses to safe methods (like GET, HEAD) unless explicitly configured otherwise. Since STRUT is not safe, responses to STRUT would not be cached by standard proxies unless instructed. This is expected, as resource creation operations typically are not cacheable in any generic way. If caching of a STRUT response is desired (for example, if the server returns the representation of a newly created resource), the server must include explicit cache directives (e.g., `Cache-Control: max-age=N`) and the client or intermediary may then store it. Also, similar to PUT/POST, a STRUT request should cause invalidation of caches for any cached representations of the resource that was created or related resources (like a list).

Idempotence and Retry: Because STRUT is idempotent, clients or intermediaries (e.g., certain proxies or HTTP libraries) could automatically retry a STRUT request on timeout or error, in the same way they might retry a PUT or DELETE (which are also idempotent). This improves reliability for operations where the client isn't sure of the outcome. While automatic retries are typically done at the client application level, knowing that STRUT is idempotent allows developers to implement robust retry logic without fear of duplicating effects.

Expect/Continue: Since STRUT requests typically have no (or very small) bodies, the use of the `Expect: 100-continue` header (which is mainly for large payloads) is generally not needed. Clients **SHOULD NOT** send `Expect: 100-continue` with STRUT unless a body is present and large. Servers **MUST** handle it appropriately if they do receive it, but in practice this simplifies interactions (no multi-step expectation handshake is necessary in the common case of no body).

Security Considerations: As noted, introducing STRUT means frameworks and middleware need to treat it like other unsafe methods (e.g., protect it with CSRF tokens in web apps, enforce authentication for it as with POST/PUT, etc.). The good news is that frameworks typically have patterns for this (like lists of methods considered unsafe or state-changing) that can be extended. For example, a web framework's CSRF protection can be configured to include STRUT in the methods that require a token. In terms of the protocol, STRUT does not inherently introduce new security issues beyond those of any method that modifies server state. In fact, by eliminating the need to misuse GET for creation, STRUT can reduce certain risks (like GET side effects that might bypass security checks expecting GET to be safe).

Interoperability: A client and server both need to understand STRUT for it to work; if a STRUT request is sent to a server that does not implement it, the likely response is `501 Not Implemented`. Thus, initial adopters must handle that scenario (maybe falling back to a different method or informing the user). Over time, as support becomes widespread, this will be less of an issue. Designing STRUT to mirror known patterns (idempotent creation, similar to a PUT but without body) should make it conceptually easy for developers to adopt once available. Support can be discovered via `OPTIONS` (the `Allow` header) and documentation or hypermedia. HTTP/2 does not negotiate per-method support via ALPN or SETTINGS.

By adhering to these compliance considerations, STRUT can be introduced in a way that respects the overall design of HTTP. The intention is for STRUT to feel like a "natural" addition to the HTTP vocabulary, fitting into the client-server model without breaking any assumptions of the protocol.

It is also worth noting that in terms of REST's uniform interface [4], adding a new method is an extension of that interface. REST does not forbid adding new methods, but it encourages generality. STRUT is justified by a significant gap in functionality and is designed in a general way (not tied to a specific resource type). It preserves other REST constraints like statelessness of requests and manipulable resources (the resources created via STRUT can still be accessed with GET, modified with PUT/PATCH, etc.). In essence, STRUT extends the uniform interface to better fit a class of scenarios (server-driven creation) without violating REST principles.

D. Use Cases for STRUT

To demonstrate the practical utility of STRUT, we present several real-world inspired use cases where the client has minimal information and the server is responsible for generating the resource. These scenarios illustrate how STRUT would be used in contrast to existing methods, highlighting the improved clarity and efficiency.

1) One-Time Password (OTP) Generation: In two-factor authentication systems, a client (often a mobile app or web frontend) needs to request a one-time password to be generated and sent to a user (via SMS, email, etc.). The client itself does not generate the OTP; it only identifies the user for whom the OTP should be created.

Using current HTTP methods, an API might define an endpoint like `POST /users/123/otp` where the request body is empty or contains just a token indicating an OTP is needed. This POST request's only purpose is to tell the server "create an OTP for this user", and the server then generates a random code, saves it (perhaps associated with the user or session), and triggers the delivery (SMS/email). The response might be a `201 Created` with no body (since the OTP is secret or ephemeral).

STRUT approach: With STRUT, the client would instead issue:

```
STRUT /users/123/otp HTTP/1.1
Host: example.com
```

(plus appropriate headers like Authorization). No body is needed. This clearly indicates a creation of a new OTP resource for user 123. The server, upon receiving this, uses its internal logic to generate a secure random code, stores it (maybe as a new resource like `/users/123/otp/otp-id` or simply a record tied to user 123), and perhaps returns a `201 Created` with a Location header if the OTP is considered a resource that can be subsequently accessed or verified (some systems

might not expose it as a resource, just an action, in which case 204 No Content could be used). Importantly, the STRUT request can be made idempotent by design. The server can ensure that if the same STRUT call is repeated within a short time window, it does not issue multiple OTPs (which could confuse the user or weaken security). Instead, it could return the same result or indicate that an OTP has already been recently generated. This behavior is natural for STRUT but would be ad-hoc to implement for POST (developers would have to manually enforce idempotence with tokens or locks).

The benefits in this case include: (a) Minimal data transfer: The client only sends the user identifier (which is in the URL) and no body. (b) Clear semantics: It's obvious that this call is asking the server to create something (an OTP) using internal rules, not trying to submit data. (c) Idempotent retries: If the client doesn't get a response (network issue) and retries the STRUT, the server can recognize the duplicate attempt and simply not generate a second OTP if one was already generated moments before, returning the appropriate success response indicating the OTP request is in effect. This prevents sending two OTP codes to the user where only one was intended.

2) Server-Assigned Record Creation: Consider a service where resources have complex relationships or require server-side assignment of certain fields. For example, a client wants to create a new account in a system but the actual account record involves multiple linked entries or computed defaults. The client may only provide a username and let the server generate other details (such as a unique account number, default preferences, linkages to default groups, etc.).

With existing methods, the client might call `POST /accounts` with a JSON body that only contains the username, relying on the server to fill out the rest. This is a legitimate use of POST (since a partial data can be sent and the server completes it), but some might argue it blurs the line of responsibility for resource content. Alternatively, an API could require the client to first GET some defaults and then PUT, which is a multi-step interaction and burdens the client with data it doesn't truly know.

STRUT approach: The client issues:

```
STRUT /accounts/jdoe HTTP/1.1
```

(assuming "jdoe" is the desired username as part of the path). Optionally, if the username or other minimal fields don't fit well in the path, a very small JSON payload or query parameters could be used (e.g., `STRUT /accounts?username=jdoe`). The essence is that the client provides the key (username) and nothing else. The server then creates a new account resource for "jdoe" with all necessary details:

Generates a new account ID (which might be a database primary key or GUID).

Creates subordinate records (for preferences, profile, etc.) as needed with default or computed values.

Perhaps notifies other systems of the new account.

The server returns a `201 Created` with `Location: /accounts/accountId` (where accountId is the server-assigned identifier, not just the username) and possibly a representation of the new account containing all the details now filled in.

This pattern is effectively a server-driven sign-up process where the client only gave a minimal detail. STRUT formalizes it. The idempotence can be handled by, say, treating the username as a natural key. If the client accidentally repeats the STRUT for the same username and an account already exists, the server can simply return a `200 OK` with the existing resource representation (or a message indicating it's already created) rather than creating another. With POST, a repeated call could either create a second account with the same username (if not prevented) or error out; with STRUT, the expectation of idempotence encourages designing the operation to be repeatable without side effects beyond the first success.

3) Establishing Relationships Between Entities: In RESTful APIs, relationships between entities (such as creating a friendship between two user accounts, or linking a user to a group) are sometimes modeled as separate resource creation actions. For example, adding a friend could be seen as creating a "friendship" resource that links two user IDs. Typically, a client might POST to a `/friends` collection with a body containing the two user IDs. This requires the client to provide both IDs and possibly additional metadata.

However, consider a case where the client only directly knows one side of the relationship (itself) and an identifier for the other, and the server needs to verify or augment the relationship. For instance, a client requests to follow another user by providing the target user's username, but the server must look up internal IDs, ensure no duplicate relationship exists, and maybe generate a follow event.

STRUT approach: The client (User A) could call:

```
STRUT /users/A/follow/B HTTP/1.1
```

(with appropriate Authorization headers for A's credentials). This indicates that user A wants to follow user B. The server handling this STRUT request would:

Verify user A (from the auth token) is allowed to follow user B.

Translate the usernames or IDs as needed (maybe user B was provided as a username, the server finds the internal ID).

Check if such a follow relationship already exists (to honor idempotence and avoid duplicates).

Create a new follow relationship resource or entry if it does not exist, possibly with its own ID or timestamp.

Return `201 Created` if a new follow was established, or `200 OK` if the follow was already in place (idempotent behavior), possibly with the relationship resource or status in the body.

The STRUT method cleanly encapsulates the notion of "ensure this relationship exists, creating it if necessary." It saves the client from crafting a request body like "follower": "A", "target": "B" (which in a sense duplicates information that's already in the URL or context) and from worrying about duplicate submissions.

This pattern could be generalized to any kind of link/relationship creation where the client's job is merely to request it and the server's job is to figure out how to implement

it. STRUT could be used for creating default configurations (e.g., `STRUT /users/123/settings` to initialize a user's settings with defaults), kicking off background processes or reports where the client only triggers the action, etc., all in a standardized way.

4) Implicit Resource Initialization: In some systems, a resource might be implicitly created as a side effect of accessing a namespace. For example, the first time a user accesses a certain feature, a corresponding resource (like a settings profile or a usage log) is created server-side with defaults. If designers want to make this explicit, they might provide an endpoint to initialize or retrieve that resource.

A current approach might be: the client performs a GET on `/users/123/settings`, and if the server doesn't find settings, it creates default settings and returns them. This violates the idea that GET should not create things (though it is a pragmatic approach in some implementations).

STRUT approach: Instead, the API could expose `STRUT /users/123/settings` to explicitly initialize default settings for the user. The client calls this when it needs to ensure the settings resource exists. The server then creates the settings with defaults if not present, or does nothing if it already exists, and returns a representation of the settings (or just a 204 if it was already there). This fits perfectly with STRUT's meaning: create if necessary using server logic, and it's idempotent (calling it multiple times results in the same final state: the user has a settings resource initialized).

In summary, these use cases show how STRUT can lead to cleaner API designs:

The client code becomes simpler (no dummy payloads, no manual idempotence keys or duplicate checks).

The intent of requests is clearer (a STRUT stands out as a creation operation driven by the server).

The server can encapsulate complex creation logic behind a standard interface.

idempotence is built-in, reducing errors and duplicate processing.

Next, we examine implementation considerations for adding such a method to systems and discuss potential impacts on existing infrastructure.

V. Implementation Considerations

Implementing support for a new HTTP method like STRUT involves changes or extensions on both the client and server side. In this section, we discuss how a prototype or actual deployment of STRUT might be approached, addressing both protocol-level handling and the application logic required. We also consider how to maintain the idempotent nature of STRUT in practice, and what development frameworks and tools would need in order to accommodate this new method.

A. Server-Side Implementation

On the server side, adding STRUT means the HTTP server (or the framework in use) must recognize the method token "STRUT" in incoming requests. Many modern web frameworks allow definition of custom HTTP methods relatively easily, but some infrastructure might require updates:

HTTP Server and Framework Support: Web servers/proxies (like Apache or Nginx when acting as reverse proxies) typically forward unknown methods by default, but they might need configuration adjustments to not block them. For example, some security modules might flag unfamiliar methods. Application frameworks (e.g., Express.js, Django, ASP.NET, Java Spring) would likely treat unknown methods as not matching any route by default, so explicit route definitions for STRUT are needed. In an experimental implementation, developers can often manually add support by checking the request method string. Over time, if STRUT were standardized, frameworks would natively include it as they do GET/POST.

Routing and Handler Logic: The server application would define routes or endpoints to handle STRUT requests for relevant resources. For example, a route for `STRUT /users/:id/otp` would be implemented to call the OTP generation logic. This is analogous to how one defines handlers for POST or PUT today, but ensuring that the logic under STRUT handlers adheres to idempotence. The handler should check if the resource already exists or if the action was already taken for the given identifier, to avoid duplicating work. This might involve a quick lookup (e.g., "have we already generated an OTP recently for this user?" or "does user settings already exist?"). If yes, the handler can choose to either do nothing or update the existing resource if needed, then return a success response indicating no new creation was necessary.

idempotence Enforcement: Making an operation idempotent on the server side might involve additional state or checks. For example, to make OTP generation idempotent within a time window, the server could store a timestamp of the last OTP and refuse to generate a new one if the last one is still valid. For creating a new account by username, the server must ensure that if the account was already created, a second STRUT does not create another; this typically means the username is a unique key and the creation logic first checks for existing entry. These are application-specific details, but the key is that the method semantics push developers to consider and implement these checks up front, rather than leaving it as an afterthought.

Response Construction: The server needs to formulate the correct response per the semantics of STRUT. A successful creation would return a 201 status and include the location of the new resource. Many frameworks have utilities for returning created resources; these can be reused. If no new resource was actually created (because it was already present), the server could return 200 or 204. There is a design decision here: either return `201 Created` with `Location` on new creation, or `200 OK`/`204 No Content` when the resource already exists (idempotent completion). Status `304 Not Modified` is specific to conditional retrieval and **MUST NOT** be used for STRUT.

Logging and Monitoring: Introducing a new method means ensuring that server logs and monitoring systems properly record STRUT requests. Log parsers might need updates

to recognize the method in request logs. This is a minor consideration but important for operations teams to be aware of new kinds of traffic and to set up appropriate monitoring (for instance, tracking the rate of STRUT calls, their success/failure counts, etc., just as one would for POST/GET).

In summary, server-side implementation is feasible with moderate effort. In a controlled environment, one could implement STRUT in a custom API today by manually checking the HTTP method header and routing accordingly. The main additional burden is ensuring idempotent behavior. This is less a technical protocol issue and more a business logic guarantee, which STRUT's definition encourages developers to think about explicitly.

B. Client-Side Implementation

For clients (which could be web frontends, mobile apps, or other services calling an API), using STRUT requires that the HTTP client library or tool they use can formulate and send a request with the method set to "STRUT". Many HTTP libraries allow arbitrary method names out of the box; some might not.

Browser Support: Web browsers have a limitation in that HTML forms only support a narrow method set, but the Fetch API can send custom methods such as STRUT. Because STRUT is not a simple method (GET/POST/HEAD), a cross-origin STRUT request will cause a CORS preflight (an `OPTIONS` request) to verify that the server allows it. Thus, an API using STRUT must handle preflight requests and include `STRUT` in `Access-Control-Allow-Methods`. This is similar to the requirement when using PUT or DELETE from a web page. In summary, browsers can send STRUT, but developers must configure servers for CORS accordingly.

API Client Libraries: Many language-specific HTTP clients (Python's `requests`, JavaScript's `axios`, Java's HttpClient, etc.) allow custom methods. For example, in Python `requests`, one can call `requests.request('STRUT', url, ...)`. Provided the library doesn't restrict allowed methods, this works. Some older or more rigid frameworks might not support arbitrary methods without lower-level access. If a particular environment doesn't support STRUT, a workaround might be needed (like tunneling via POST with a header, or waiting for library support). If STRUT is standardized, one would expect updates to major libraries to list it as a known method (for instance, adding a convenience function or constant).

Developer Handling of Responses: Clients will need to handle 201 Created responses (and 200/204 for idempotent completions). For example, a client should be prepared to read the `Location` header on a 201 to get the URI of the new resource. Also, if a 501 Not Implemented is received, clients might implement a fallback (like resorting to an earlier API method if available) or at least raise a clear error. Overall, existing HTTP client logic mostly carries over, but documentation should alert developers to interpret 201 as a success (some simplistic code might only check for 200).

Tooling and Testing: Tools like Postman, cURL, and automated testing frameworks generally support custom methods. Postman, for instance, lets you type any method name. cURL supports any method via `-X`. Thus, developers can test STRUT endpoints easily. Continuous integration tests might need small adjustments if they use frameworks that validate method names (e.g., older versions of some tools might warn on unknown methods). But generally, testing infrastructure can handle it.

Overall, client-side support for STRUT is primarily a matter of awareness and using tools properly. The web ecosystem is flexible enough that adding a new method is not a showstopper. In scenarios where a client truly cannot be modified to send STRUT (e.g., a third-party webhook that only sends GET/POST), one could deploy an adaptor service or allow an alternate POST endpoint. But such cases should diminish as support for STRUT becomes commonplace.

One potential interim solution, for clients that absolutely cannot be changed to send STRUT (say a third-party integration), is to expose an alternative endpoint or allow a secondary approach. For example, a system could implement both `STRUT /resource` and `POST /resource?action=strut` as a fallback. This is not ideal long-term because it duplicates the API surface and brings back some ambiguity, but it could be a pragmatic bridge during a transition period.

C. Backward Compatibility and Incremental Adoption

When introducing a new method in a large ecosystem, not everything will support it at once. Some considerations for a smooth rollout:

API Versioning: An API might introduce STRUT in a new version of its service, while still supporting older approaches in a previous version for clients that haven't upgraded. For instance, version 2 of an API might allow STRUT on certain endpoints, whereas version 1 required POST with an empty body. This gives clients time to migrate.

Discovery: How does a client know that a server supports STRUT for a given resource? In a RESTful hypermedia context, the server could advertise allowed methods via responses to `OPTIONS` requests or in documentation (e.g., an OpenAPI specification could include "strut" as an allowed operation if extended to know about it). If a client tries a STRUT and gets `501 Not Implemented` or `405 Method Not Allowed`, it can infer lack of support and possibly attempt the fallback. In some cases, the server might include in an OPTIONS response that STRUT is allowed on a resource (since OPTIONS can return an `Allow` header listing allowed methods).

Intermediaries: A critical aspect is any proxy or gateway in between. Most proxies will forward unknown methods by default, but some security devices like certain web application firewalls might flag uncommon methods. It would be advisable to check that the infrastructure allows a method named STRUT. This might require a configuration update in API gateways or load balancers (similar to how, for example, WebDAV methods or others need enabling). The presence of an official RFC standardizing STRUT would help in convincing vendors to include it in default rulesets over time.

Standardization Process: Until STRUT is standardized, early adopters would essentially be using a custom method. There is precedent for custom methods in certain domains (WebDAV uses methods like PROPFIND, etc., which not every client knew at first). A formal RFC and registration would give implementers confidence and a reference point for proper behavior, accelerating adoption in frameworks and libraries.

In conclusion, implementing STRUT is straightforward at a technical level, given the flexibility of the HTTP protocol, but careful thought must be given to preserving idempotence and ensuring that all parts of the stack (from client to server, including security layers) are aware of the new method. The transition period where STRUT co-exists with older methods can be managed via versioning and documentation. If done well, STRUT can be introduced without disruption, and with significant payoff in simplicity and correctness for certain API interactions.

VI. Discussion

The concept of STRUT as an HTTP method raises several discussion points regarding its place in the protocol, potential challenges, and broader implications for web architecture. In this section, we address some of these points, including security implications in more depth, comparison with alternative approaches, and practical considerations for adopting STRUT in existing systems.

A. Security and Compliance Considerations

One motivating factor for STRUT was to ensure compliance with HTTP's intended usage, which in turn has security benefits. By providing a proper method for server-driven resource creation, we reduce the temptation for API designers to misuse safe methods (like GET) for side effects, which can lead to serious security issues as discussed earlier. Nonetheless, introducing STRUT means we should examine its security model:

CSRF and Side-Effect Safety: Web applications protect unsafe methods (POST, PUT, DELETE) against CSRF. STRUT **MUST** be added to the protected-methods list in frameworks and middleware; if a framework is unaware of STRUT, applications **MUST** enforce CSRF checks explicitly.

Authentication and Authorization: Similarly, any authorization logic that is method-specific should include STRUT. For example, a rule like "users can POST to /accounts to create accounts" should have an equivalent for STRUT if that's the new mechanism. A positive aspect here is that because STRUT clearly signifies creation, it might be easier to set up granular permissions (e.g., allow an API key to perform STRUT on a certain path, but not to perform arbitrary POSTs, if one wanted such differentiation).

Unknown Method Handling: Some security proxies or WAFs (Web Application Firewalls) flag unknown HTTP methods as potentially suspicious (since custom methods could be used to tunnel unwanted traffic). As mentioned in Implementation, those systems would need to be configured to allow STRUT for known good APIs. The presence of an official RFC standardizing STRUT would help in convincing vendors to include it in their default rulesets over time. Until then, this is a deployment hurdle; however, given that methods like WebDAV's methods have been managed (or sometimes explicitly disallowed for security if not needed), the same can be handled for STRUT in a controlled way.

Semantic Clarity and Maintenance: From a compliance standpoint, one might ask if STRUT could be misused for something it shouldn't do, similar to how other methods are misused. Because STRUT is specifically defined for creation with minimal input, using it for anything else (e.g., updates or deletions) would be clearly wrong. This narrow purpose is a strength in terms of maintaining clarity. If developers stick to using STRUT only when appropriate, the uniformity of interface is preserved and compliance with HTTP semantics is actually improved across the board (each method doing only what it is meant to do).

Server-Side Complexity: Ensuring idempotence on the server side might introduce complexity or new failure modes. For instance, if a STRUT request triggers a complex creation process (like provisioning an account across multiple microservices), making that whole process idempotent could be challenging. One might need to implement compensation logic or check-pointing. However, this is not fundamentally new with STRUT; any idempotent operation that spans multiple components has this challenge. STRUT simply forces us to confront it rather than ignoring it (as POST might). Careful design, such as using the resource's key as the coordination token (so each part of the system knows the operation for that key is in progress or done), can manage this. This is a design consideration for robust distributed systems rather than a flaw in STRUT itself.

B. Alternatives to Introducing a New Verb

It is worth discussing why a new HTTP method is the proposed solution, as opposed to alternative approaches:

Convention Over Configuration in APIs: One could handle minimal-input creation by simply adopting conventions with existing methods. For example, decide that `POST /resource/id` with an empty body means "create resource with id using defaults". While this could work, it is not self-documenting and relies on out-of-band knowledge. It also doesn't solve idempotence unless you impose that convention strictly and ensure the server checks for existence. Essentially, you end up implementing something like STRUT semantics via POST. The downside is that clients and new developers might not realize that a particular POST behaves idempotently or requires no body, because it's not standard. Thus, errors and misuse can occur, and tools like API documentation generators might not capture these nuances.

Using Query Parameters or Headers to Modify Semantics: Another approach could be to stick with POST but include a special header or parameter like `Prefer: idempotent` or `X-Use-Server-Defaults: true`. This again is not ideal because it overloads an existing verb with new semantics.

It may trick some middleware or caches incorrectly, and generally such approaches are considered hacky.

GraphQL/RPC Alternatives: Outside of pure REST, one might argue that if such fine server control is needed, perhaps the API should be more RPC-like or use GraphQL, where the client can call a mutation that triggers server logic. These approaches indeed offer alternatives, but they move away from the simplicity of REST and the uniform interface. The introduction of STRUT is for those who want to remain in a RESTful paradigm but address a specific shortcoming.

Extension of Existing Methods: Could we extend PUT to say "if no body, then fill with defaults"? This would blur the meaning of PUT and likely be rejected by the standards community (since it violates PUT's requirement to send a representation). Similarly, altering POST's definition to be idempotent in some cases is not feasible. Therefore, introducing a distinct method is cleaner than retrofitting semantics onto existing ones.

The decision to add a verb is not taken lightly. In HTTP's history, new methods have been added when necessary (PATCH, WebDAV's MKCOL, etc.), but the bar is high to justify them. We believe the scenarios and reasoning provided make a strong case that the benefits of STRUT justify it as a new addition rather than trying to twist current methods into this role.

C. Adoption Challenges

Even if STRUT were standardized, adoption in the real world would take time. We identify some challenges and ways to mitigate them:

Ecosystem Inertia: Many developers and architects might be initially reluctant to use a method that is not yet ubiquitous. There could be concerns about library support or simply habit. Advocacy and clear demonstration of benefits will be key. Early adopters (perhaps internal APIs within large companies) could showcase success that encourages others.

Partial Deployment and Fallbacks: As mentioned, during transition, systems might support STRUT and a fallback. This adds complexity temporarily. Tools and gateways might assist by translating or dispatching alternate flows for clients that haven't upgraded, although that must be done carefully.

Educational Aspect: Documentation, tutorials, and best practices guides will need updates to include STRUT. The concept "minimal-input creation" should become part of HTTP education, with STRUT as the solution. This paper is a step in that direction; further outreach (talks, community discussions) would help.

Tooling Updates: API design and documentation tools (like OpenAPI) would need to recognize STRUT, testing tools might add it to their method lists, etc. If the IETF standardizes STRUT, these tools likely will follow. The community should be ready to contribute to open-source tooling to add support where needed.

Despite these challenges, the alternatives (continuing with awkward or non-compliant patterns) also carry costs. We argue that STRUT's clarity and correctness provide long-term benefits that outweigh the short-term adoption effort.

D. Broader Implications for Web Architecture

Introducing STRUT has some broader implications for how we think about web APIs:

It reaffirms that servers can have more autonomy in resource creation. REST has always allowed server-driven content via hypermedia, but methods have usually required the client to supply data. STRUT tilts the balance slightly, letting the client say *what* to create and the server decide *how*. This could encourage designs where more logic resides server-side (which can centralize and simplify management).

If successful, STRUT might prompt discussions about other missing methods or patterns. We caution against a proliferation of methods, but the example of STRUT shows that carefully chosen extensions can make the interface more powerful without losing generality.

By providing a clean solution for a common pattern, STRUT might reduce the misuse of existing methods for convenience. Over time, that could lead to more consistent and predictable APIs across the web.

There might be an effect on client-side vs server-side responsibilities. With STRUT, clients may lean on servers to do more (like generate an entity). This might make clients thinner in some scenarios. Architectural preferences cycle over time, but STRUT gives an option to those who want a thinner client without giving up REST.

In summary, STRUT aims to be a practical and beneficial extension to HTTP. The challenges are manageable, and the potential improvements to API clarity, correctness, and compliance are significant. It aligns with HTTP's evolutionary path and addresses real needs without breaking existing principles.

VII. Conclusion

This paper proposed STRUT as a candidate HTTP method for scenarios where clients can only provide minimal information for resource creation and must rely on server-side intelligence to construct the resource. Through analysis of existing HTTP methods and their limitations, we identified a semantic gap in how APIs express idempotent, server-driven creation. STRUT addresses that gap by offering an idempotent, unsafe creation method, thereby streamlining interactions in RESTful systems that involve dynamic or policy-driven resource instantiation.

We presented the formal semantics of STRUT, ensuring that it adheres to HTTP's design principles (such as being idempotent and non-safe) and can be integrated via the established RFC process. Compliance with standards and security considerations were discussed in detail to show that STRUT can be adopted without compromising the integrity or safety of web interactions. We also illustrated several use cases, including OTP generation, server-assigned record creation, and establishing entity relationships, where STRUT provides cleaner and more efficient solutions compared to repurposing existing methods. These examples highlight how STRUT simplifies client logic, reduces unnecessary data transfer, and avoids protocol misuse.

An evaluation strategy was outlined, and a reasoned discussion of performance implications suggests that STRUT can reduce communication overhead and improve reliability through its idempotent nature. While primarily a semantic enhancement, STRUT has tangible effects: fewer bytes on the wire, fewer round trips in certain workflows, and more predictable outcomes on retries. The discussion addressed potential challenges in introducing a new HTTP method, concluding that while there are hurdles in terms of ecosystem support and awareness, the benefits to API design and maintenance are significant. By providing a dedicated mechanism for a common pattern, we ultimately make RESTful APIs more self-descriptive and robust.

In conclusion, STRUT is a theoretical proposal, but one grounded in practical needs observed in modern web development. We encourage the community to consider STRUT in experimental settings and provide feedback. Future work includes drafting a formal Internet-Draft for STRUT as an extension to HTTP semantics, engaging with standards bodies and industry experts to refine the concept, and testing the method in controlled APIs. More extensive real-world testing would be valuable to gather data on STRUT's impact and to uncover unforeseen issues, particularly across gateways, CORS configurations, API documentation tools, and security middleware.

By rethinking and extending the HTTP method set in a careful manner, we keep the web's evolution in step with the evolving patterns of client-server interaction. STRUT, as the "missing method," has the potential to become a natural part of the developer's toolkit for building clearer and more efficient APIs, just as POST and PUT are today. We believe the time is right to strut forward and enrich HTTP with this new capability.

REFERENCES

[1] R. Fielding, M. Nottingham, and J. Reschke, *HTTP Semantics*, RFC 9110, IETF, Jun. 2022. Available: https://doi.org/10.17487/RFC9110

[2] L. Dusseault and J. Snell, *PATCH Method for HTTP*, RFC 5789, IETF, Mar. 2010. Available: https://doi.org/10.17487/RFC5789

[3] T. Berners-Lee, R. Fielding, and L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, RFC 3986, IETF, Jan. 2005. Available: https://doi.org/10.17487/RFC3986

[4] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Doctoral dissertation, University of California, Irvine, 2000. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

[5] L. Dusseault, Ed., *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*, RFC 4918, IETF, Jun. 2007. Available: https://doi.org/10.17487/RFC4918

[6] J. Jena and S. Dalal, *The idempotence-Key HTTP Header Field*, Internet-Draft draft-ietf-httpapi-idempotence-key-header-06, IETF, Feb. 2025, *work in progress*.

[7] J. Snell, *Prefer Header for HTTP*, RFC 7240, IETF, Jun. 2014. Available: https://doi.org/10.17487/RFC7240

[8] R. Fielding, M. Nottingham, and J. Reschke, *HTTP Caching*, RFC 9111, IETF, Jun. 2022. Available: https://doi.org/10.17487/RFC9111

[9] R. Fielding, M. Nottingham, and J. Reschke, *HTTP/1.1*, RFC 9112, IETF, Jun. 2022. Available: https://doi.org/10.17487/RFC9112

[10] S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, RFC 2119, IETF, Mar. 1997. Available: https://doi.org/10.17487/RFC2119

[11] B. Leiba, *Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words*, RFC 8174, IETF, May 2017. Available: https://doi.org/10.17487/RFC8174