JMAP
Internet-Draft
Intended status: Standards Track
Expires: 6 May 2026

M. De Gennaro
Stalwart Labs

2 November 2025

JMAP Enhanced Result References
draft-degennaro-jmap-refplus-00

Abstract

This document specifies an extension to the JSON Meta Application Protocol (JMAP) that enhances the result reference mechanism defined in [RFC8620]. The extension allows result references to be used in additional contexts beyond method call arguments, specifically within object properties during JMAP /set operations and within FilterCondition objects used in /query operations. Additionally, this specification extends result references to support JSON Path expressions ([RFC9535]) as an alternative to JSON Pointer ([RFC6901]), providing more expressive query capabilities for extracting values from previous method call results. These enhancements enable more efficient data reuse patterns across JMAP method calls, reducing the need for multiple round trips between client and server.

Status of This Memo

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

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

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

This Internet-Draft will expire on 6 May 2026.

Copyright Notice

Table of Contents

1. Introduction

JMAP ([RFC8620] — JSON Meta Application Protocol) is a generic protocol for synchronizing data, such as mail, calendars or contacts, between a client and a server. It is optimized for mobile and web environments, and aims to provide a consistent interface to different data types.

One of JMAP's core design principles is its ability to build efficient request chains that minimize the number of round trips between client and server. This is achieved through the result reference mechanism defined in Section 3.7 of [RFC8620], which allows method call arguments to reference the results of previous method calls within the same request. When processing a request, the server executes method calls sequentially, and any argument prefixed with "#" (an octothorpe) is resolved by extracting the specified value from a previous method call's result before the current method is executed.

Since the publication of JMAP Core ([RFC8620]) and JMAP for Mail ([RFC8621]), the JMAP ecosystem has expanded significantly with the definition of multiple additional data types and extensions, including JMAP for Calendars ([I-D.ietf-jmap-calendars]), JMAP for Contacts ([RFC9610]), and others. This growth has revealed practical limitations in the current result reference mechanism. Many real-world use cases require creating JMAP objects that incorporate data from other JMAP objects. For example, when creating a CalendarEvent, it is often necessary to copy "Participant" or "Location" objects from an existing event. Similarly, when creating Email objects, reusing attachment information or recipient lists from other messages would be beneficial.

The current result reference mechanism, however, is limited to method call arguments. This restriction prevents result references from being used within the object structures passed to /set methods for creating or updating records, or within FilterCondition objects used in /query methods. These limitations force clients to either make additional round trips to fetch intermediate data or to duplicate data within the request, both of which reduce efficiency.

Furthermore, JMAP Core exclusively uses JSON Pointer ([RFC6901]) syntax for navigating result structures. While JSON Pointer is simple and efficient, it has limited expressive power for certain selection patterns. JSON Path ([RFC9535]), a more recent standard, provides a richer query language for extracting values from JSON structures, including support for filters, array slicing, and more sophisticated selection patterns.

Therefore, this document aims to extend the result references defined in [RFC8620] to be usable in multiple additional contexts, specifically within JMAP /set method calls for object property values and within FilterCondition objects used in /query methods. Additionally, this specification defines support for JSON Path ([RFC9535]) expressions as an optional alternative to JSON Pointer, enabling more powerful value extraction capabilities while maintaining backward compatibility with existing implementations.

1.1. Notational Conventions

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

1.2. Addition to the Capabilities Object

The capabilities object is returned as part of the JMAP Session object; see Section 2 of [RFC8620]. This document defines one additional capability URI.

1.2.1. urn:ietf:params:jmap:refplus

This capability indicates that the server supports enhanced result references as defined in this specification. When this capability is present, clients may use result references within /set method object properties and FilterCondition objects, in addition to the standard method call argument usage defined in [RFC8620].

The value of this property in the JMAP Session "capabilities" property is an empty object.

The value of this property in an account’s "accountCapabilities" property is an object that MUST contain the following information on server capabilities and permissions for that account:

jsonPath: Boolean

Whether the server supports JSON Path expressions in result references. If true, result references may use JSON Path ([RFC9535]) syntax in addition to JSON Pointer ([RFC6901]) syntax. If false or absent, only JSON Pointer syntax is supported.

2. Enhanced Result References

Enhanced result references extend the existing result reference mechanism defined in Section 3.7 of [RFC8620] to enable broader applicability and more expressive value extraction. Specifically, enhanced result references support two key extensions to the base mechanism:

First, enhanced result references may be used in additional contexts beyond method call arguments. They may be used anywhere within a "Foo" object when creating or updating such an object via the Foo/set method, allowing property values to be derived from previous method call results. They may also be used within FilterCondition objects in Foo/query method calls, enabling dynamic query construction based on prior results.

Second, enhanced result references optionally support JSON Path ([RFC9535]) expressions as an alternative to JSON Pointer ([RFC6901]) for extracting values from result structures. JSON Path provides more sophisticated query capabilities, including array filtering, recursive descent, and complex selection patterns, while JSON Pointer remains the baseline requirement for all implementations.

The syntax and resolution mechanism for enhanced result references follows the same fundamental pattern as standard result references, with extensions to accommodate the new contexts and optional JSON Path support. Backward compatibility is maintained: clients that do not use enhanced result references, and servers that do not advertise the urn:ietf:params:jmap:refplus capability, interoperate normally using the base result reference mechanism from [RFC8620].

2.1. JSON Path Support

JSON Path ([RFC9535]) is a query language for JSON that provides powerful capabilities for selecting and extracting values from JSON structures. While JSON Pointer ([RFC6901]) provides a simple notation for referencing a specific value within a JSON document using a sequence of reference tokens, JSON Path offers a more expressive syntax that supports complex queries including filters, recursive descent, array slicing, and wildcard selections.

2.1.1. Comparison with JSON Pointer

The key differences between JSON Pointer and JSON Path include:

* JSON Pointer uses a simple path syntax with "/" separators and identifies exactly one value. JSON Path uses a richer syntax with multiple selector types and can return zero, one, or multiple values.

* JSON Pointer is deterministic and always produces the same result for a given document. JSON Path queries may be order-dependent when multiple values match.

* JSON Pointer has no conditional or filtering capabilities. JSON Path supports filter expressions, enabling selection based on value predicates.

* JSON Pointer requires exact knowledge of the structure. JSON Path supports recursive descent (..) allowing queries to find values at any depth.

2.1.2. Support and Capabilities

Enhanced result references extend the base result reference mechanism to optionally support JSON Path expressions. This support is indicated by the "jsonPath" property in the account capabilities being set to true. Servers MAY implement JSON Path support, as it can be more resource-intensive than JSON Pointer due to the potential for complex query evaluation.

When a server supports JSON Path, enhanced result references may use JSON Path syntax as an alternative to JSON Pointer syntax in the "path" property of a ResultReference object. To indicate that a path should be interpreted as a JSON Path expression rather than a JSON Pointer, the path MUST begin with "$" (dollar sign), which is the root node identifier in JSON Path syntax. Paths not beginning with "$" are interpreted as JSON Pointer expressions following the rules in [RFC8620].

2.1.3. Validation Requirements

If the JSON Path expression is syntactically invalid according to [RFC9535], the server MUST reject the method call with an "invalidResultReference" error. The error description SHOULD indicate the syntax error where possible.

If a client attempts to use a JSON Path expression (path beginning with "$") but the server does not support JSON Path (jsonPath capability is false or absent), the server MUST reject the method call with an "invalidResultReference" error. The error description SHOULD indicate that JSON Path is not supported.

Servers supporting JSON Path SHOULD implement the complete JSON Path specification as defined in [RFC9535], including all standard function extensions. However, servers MAY limit the complexity of JSON Path expressions for security or performance reasons, for example by restricting maximum expression length, maximum result set size, or computational complexity. If a server rejects a valid JSON Path expression due to such limitations, it SHOULD return an "invalidResultReference" error with a description indicating the specific limitation encountered.

2.2. Resolution Algorithm

This section defines the algorithm for resolving enhanced result references, which extends the base resolution mechanism defined in Section 3.7 of [RFC8620] to handle both JSON Path expressions and extended JSON Pointer expressions (including wildcard support).

When processing a ResultReference object, the server MUST follow these steps:

1. Resolve the "resultOf" and "name" properties to identify the target method response, as defined in Section 3.7 of [RFC8620].

2. Identify the second element of the method response array (the arguments object) as the target JSON structure for path evaluation.

3. Determine whether the "path" property uses JSON Path or JSON Pointer syntax:

   * If the path begins with "$" (dollar sign), treat it as a JSON Path expression.

   * Otherwise, treat it as a JSON Pointer expression.

4. Evaluate the path expression against the target JSON structure according to the appropriate specification ([RFC9535] for JSON Path, [RFC6901] for JSON Pointer).

5. Apply the type-specific resolution rules defined in this section to produce the final resolved value.

2.2.1. Resolving JSON Path References

When processing a ResultReference with a JSON Path expression, the server MUST:

1. Apply the JSON Path expression (the "path" value) to the arguments object of the identified response.

2. Evaluate the JSON Path expression according to [RFC9535] to produce a nodelist.

3. Determine the expected type of the property being set and resolve the nodelist according to the following type-specific rules.

The nodelist produced by JSON Path evaluation MUST be resolved according to the expected type of the target property:

For properties that expect *JSON primitive types* (String, Boolean, Number as defined in [RFC8259], or null) or a *single JMAP object* type:

* If the nodelist contains exactly one node, use that node’s value as the resolved value.

* If the nodelist contains zero nodes, use null as the resolved value.

* If the nodelist contains more than one node, the result reference resolution fails with an "invalidResultReference" error.

For properties that expect an *array type* (denoted as "A[]" in Section 1.1 of [RFC8620], where A[] is an array of values of type A):

* Map the nodelist to an array containing the value of each node in the nodelist, in the order specified by [RFC9535].

* If the nodelist contains zero nodes, use an empty array as the resolved value.

* If the nodelist contains one or more nodes, use an array containing all node values as the resolved value.

For properties that expect a *map type* (denoted as "A[B]" in Section 1.1 of [RFC8620], where the keys are of type A and the values are of type B):

* If the nodelist contains exactly one node and that node’s value is a JSON object conforming to the A[B] type, use that object as the resolved value.

* If the nodelist contains zero nodes, use an empty object {} as the resolved value.

* If the nodelist contains more than one node, or if the single node’s value is not a JSON object, the result reference resolution fails with an "invalidResultReference" error.

2.2.2. Resolving JSON Pointer References

JSON Pointer expressions, as defined in [RFC6901], provide a deterministic path to a single value within a JSON structure. However, [RFC8620] extends JSON Pointer syntax to support wildcards ("*") in certain contexts, allowing a single pointer to match multiple values. This extension is maintained in enhanced result references.

When processing a ResultReference with a JSON Pointer expression, the server MUST:

1. Apply the JSON Pointer expression (the "path" value) to the arguments object of the identified response.

2. Evaluate the JSON Pointer expression according to [RFC6901], with wildcard extensions as defined in [RFC8620].

3. Determine the expected type of the property being set and resolve according to the following type-specific rules.

The result produced by JSON Pointer evaluation (which may be a single value or multiple values when wildcards are used) MUST be resolved according to the expected type of the target property:

For properties that expect *JSON primitive types* (String, Boolean, Number as defined in [RFC8259], or null) or a *single JMAP object* type:

* If the pointer identifies exactly one value, use that value as the resolved value.

* If the pointer contains wildcards and matches zero values, use null as the resolved value.

* If the pointer contains wildcards and matches more than one value, the result reference resolution fails with an "invalidResultReference" error.

* If the pointer does not contain wildcards and does not match any value, the result reference resolution fails with an "invalidResultReference" error (per [RFC6901]).

For properties that expect an *array type* (denoted as "A[]" in Section 1.1 of [RFC8620], where A[] is an array of values of type A):

* If the pointer identifies a single value that is already an array, use that array as the resolved value.

* If the pointer contains wildcards and matches multiple values, collect all matched values into an array in the order they appear in the document structure.

* If the pointer contains wildcards and matches zero values, use an empty array as the resolved value.

* If the pointer identifies a single value that is not an array, create an array containing that single value as the resolved value.

For properties that expect a *map type* (denoted as "A[B]" in Section 1.1 of [RFC8620], where the keys are of type A and the values are of type B):

* If the pointer identifies exactly one value and that value is a JSON object conforming to the A[B] type, use that object as the resolved value.

* If the pointer contains wildcards and matches zero values, use an empty object {} as the resolved value.

* If the pointer contains wildcards and matches more than one value, or if the single identified value is not a JSON object, the result reference resolution fails with an "invalidResultReference" error.

2.2.3. Type Validation

After applying the type-specific resolution rules, the server MUST validate that the resolved value conforms to the expected type for the property. If type validation fails after resolution, the server MUST reject the operation with an "invalidProperties" error (for /set operations) or an "invalidArguments" error (for /query operations).

2.3. Usage in /set Methods

Enhanced result references may be used anywhere within object property values when creating or updating objects via the Foo/set method. This enables efficient reuse of data from previous method calls when constructing new objects or modifying existing ones.

To use an enhanced result reference within an object property, the property name is prefixed with "#" (an octothorpe), and the value is set to a ResultReference object as defined in Section 3.7 of [RFC8620]. This follows the same pattern as using result references in method call arguments. For example, to set a property "participants" to a value from a previous result, the object would include a property "#participants" with a ResultReference value.

Enhanced result references may be used at any depth within the object structure, not just at the top level. For nested objects and arrays, result references can be applied to properties at any level of nesting. The server MUST recursively process all levels of object and array nesting to resolve all result references before validating the complete object structure.

When processing a Foo/set method call, the server MUST examine all objects in the "create" and "update" arguments for properties with names beginning with "#". For each such property found, the server MUST:

1. Remove the "#" prefix to determine the actual property name.

2. Resolve the ResultReference value following the resolution algorithm defined in Section 3.7 of [RFC8620], with the extensions defined in this specification.

3. Set the actual property to the resolved value.

4. Remove the property with the "#" prefix from the object.

If a result reference fails to resolve, the server MUST reject the creation or update of that specific object with a SetError of type "invalidResultReference". The SetError SHOULD include a description indicating which property reference failed and why.

If an object contains both a property name with "#" prefix and the same property name without the prefix (e.g., both "#participants" and "participants"), the server MUST reject the creation or update with a SetError of type "invalidProperties", as this represents an ambiguous specification.

The resolved value MUST be of a type appropriate for the property being set. If the resolved value does not match the expected type for the property, the server MUST reject the creation or update with a SetError of type "invalidProperties". Type validation occurs after result reference resolution but before other property validation.

2.3.1. Usage in Patch Objects

Enhanced result references may also be used as values within patch objects, as defined in Section 5.3 of [RFC8620]. A patch object is used to update specific properties of an existing object without having to send the entire object. When using result references in patch objects, special syntax rules apply:

The patch object key (the path to the property being patched) MUST be a JSON Pointer as specified in [RFC6901], and MUST be prefixed with "#" to indicate that the value is a result reference. For example, to patch a property "locations/a1/city" with a value from a previous result, the patch object would include a key "#/locations/a1/city" with a ResultReference value.

The value associated with the prefixed JSON Pointer key is a ResultReference object. This ResultReference may use either JSON Pointer or JSON Path (if supported by the server) in its "path" property to extract the value from the previous method result.

When processing patch objects, the server MUST:

1. Identify keys in the patch object that begin with "#".
2. Remove the "#" prefix to obtain the JSON Pointer path to be patched.
3. Resolve the ResultReference value according to the resolution algorithm.
4. Apply the resolved value to the location specified by the JSON Pointer path in the target object.

If a result reference in a patch object fails to resolve, the server MUST reject the update with a SetError of type "invalidResultReference".

It is important to note that while the patch object key (the path) MUST use JSON Pointer syntax, the ResultReference value's "path" property may use either JSON Pointer or JSON Path syntax (if the server supports JSON Path).

2.4. Usage in FilterCondition Objects

Enhanced result references may be used within FilterCondition objects in Foo/query method calls. This enables dynamic query construction where filter criteria are derived from the results of previous method calls in the same request.

To use an enhanced result reference within a FilterCondition, the filter property name is prefixed with "#" (an octothorpe), and the value is set to a ResultReference object. For example, to filter based on a dynamically determined mailbox id, the FilterCondition would include "#inMailbox" rather than "inMailbox".

When processing a Foo/query method call, the server MUST examine the "filter" argument and any nested FilterCondition objects for properties with names beginning with "#". For each such property found, the server MUST:

1. Remove the "#" prefix to determine the actual filter property name.

2. Resolve the ResultReference value following the resolution algorithm defined in Section 3.7 of [RFC8620], with the extensions defined in this specification.

3. Replace the property with "#" prefix with a property of the actual name, having the resolved value.

If a result reference in a FilterCondition fails to resolve, the server MUST reject the entire query method call with an error response of type "invalidResultReference". The error response SHOULD include a description indicating which filter property reference failed and why.

If a FilterCondition contains both a property name with "#" prefix and the same property name without the prefix (e.g., both "#inMailbox" and "inMailbox"), the server MUST reject the query with an "invalidArguments" error, as this represents an ambiguous filter specification.

The resolved value MUST be of a type appropriate for the filter property being set. If the resolved value does not match the expected type for that filter property, the server MUST reject the query with an "invalidArguments" error.

Enhanced result references in FilterConditions support the full range of filter compositions defined by JMAP specifications, including nested FilterCondition objects combined with FilterOperators. The server MUST recursively process all levels of FilterCondition nesting to resolve all result references before evaluating the query.

3. Examples

This section provides additional examples demonstrating the use of enhanced result references in various contexts.

3.1. Usage in /set

Example using JSON Path (assuming server supports jsonPath capability):

```
[
  ["CalendarEvent/query", {
    "accountId": "a1",
    "filter": {
      "uid": "meeting-template-001"
    }
  }, "c0"],
  ["CalendarEvent/get", {
    "accountId": "a1",
    "#ids": {
      "resultOf": "c0",
      "name": "CalendarEvent/query",
      "path": "/ids"
    },
    "properties": ["participants", "locations"]
  }, "c1"],
  ["CalendarEvent/set", {
    "accountId": "a1",
    "create": {
      "new-event": {
        "calendarIds": {"cal-1": true},
        "title": "Team Sync",
        "start": "2025-11-01T14:00:00Z",
        "duration": "PT1H",
        "#participants": {
          "resultOf": "c1",
          "name": "CalendarEvent/get",
          "path": "$.list[0].participants"
        },
        "#locations": {
          "resultOf": "c1",
          "name": "CalendarEvent/get",
          "path": "$.list[0].locations"
        }
      }
    }
  }, "c2"]
]
```

In this example, the new calendar event reuses the participants and locations from a template event retrieved in previous calls. The server would resolve the result references before creating the event, effectively copying those complex object values without requiring the client to extract and reformat them.

3.2. Usage in /set patch objects

Example of using result references in a patch object (assuming server supports jsonPath capability):

```
[
  ["CalendarEvent/get", {
    "accountId": "a1",
    "ids": ["event-template"],
    "properties": [ "locations",
                    "organizerCalendarAddress",
                    "participants"]
  }, "c0"],
  ["CalendarEvent/set", {
    "accountId": "a1",
    "update": {
      "event-123": {
        "#/locations/a1": {
          "resultOf": "c0",
          "name": "CalendarEvent/get",
          "path": "$.list[0].locations.loc1"
        },
        "#/organizerCalendarAddress": {
          "resultOf": "c0",
          "name": "CalendarEvent/get",
          "path": "$.list[0].organizerCalendarAddress"
        },
        "#/participants/p1": {
          "resultOf": "c0",
          "name": "CalendarEvent/get",
          "path": "$.list[0].participants[?@.roles.chair]"
        }
      }
    }
  }, "c1"]
]
```

In this example, the patch object updates specific properties of event-123 using values extracted from a template event. The patch updates a single location entry ("locations/a1"), copies the organizer's calendar address, and adds a participant ("participants/p1") by extracting a participant with the "chair" role from the template using a JSON Path filter expression. The patch keys "#/locations/a1", "#/organizerCalendarAddress", and "#/participants/p1" use JSON Pointer syntax (prefixed with "#"), while the ResultReference path properties use a mix of JSON Path expressions (including the filter expression for selecting the chair participant) and simple paths.

3.3. Usage in /query FilterCondition

Example using JSON Path for array extraction (assuming server supports jsonPath capability):

```
[
  ["Mailbox/query", {
    "accountId": "a1",
    "filter": {
      "role": "inbox"
    }
  }, "c0"],
  ["Email/query", {
    "accountId": "a1",
    "filter": {
      "#inMailboxOtherThan": {
        "resultOf": "c0",
        "name": "Mailbox/query",
        "path": "$.ids[0]"
      },
      "from": "boss@example.com"
    },
    "sort": [{"property": "receivedAt", "isAscending": false}],
    "limit": 50
  }, "c1"]
]
```

In this example, the Email/query uses a dynamically resolved mailbox id from the previous Mailbox/query call. The JSON Path expression $.ids[0] extracts the first mailbox id from the query result. The server resolves this reference before executing the email query, effectively filtering emails that are not in the user’s inbox and are from a specific sender.

3.4. Extracting Multiple Values with JSON Path

Example demonstrating JSON Path for extracting multiple values:

```
[
  ["Email/get", {
    "accountId": "a1",
    "ids": ["template-email-id"],
    "properties": ["attachments"],
    "bodyProperties": ["blobId", "name", "type"]
  }, "c0"],
  ["Email/set", {
    "accountId": "a1",
    "create": {
      "new-email": {
        "mailboxIds": {
          "inbox-id": true
        },
        "subject": "Quarterly Reports",
        "from": [{"email": "sender@example.com"}],
        "to": [{"email": "recipient@example.com"}],
        "#attachments": {
          "resultOf": "c0",
          "name": "Email/get",
          "path": "$.list[0].attachments[?@.name &&
                @.name.toLowerCase().endsWith('.pdf')]"
        }
      }
    }
  }, "c1"]
]
```

In this example, the JSON Path expression `$.list[0].attachments[?@.name && @.name.toLowerCase().endsWith('.pdf')]` extracts all EmailBodyPart objects from the attachments array that have a "name" property ending with ".pdf" (case-insensitive). The result would be an array of EmailBodyPart objects (each containing blobId, name and type) that is then used to populate the attachments property of the new email, effectively copying only the PDF attachments from the template email.

4. Security Considerations

The security considerations described in [RFC8620] and [RFC9535] apply to this specification. This section discusses additional security implications introduced by allowing result references to be used in object properties and FilterCondition objects, and by supporting JSON Path expressions.

4.1. Denial of Service Considerations

Enhanced result references, particularly when combined with JSON Path support, introduce additional denial of service (DoS) vectors that implementations must address.

4.1.1. Computational Complexity of JSON Path

JSON Path expressions can be significantly more computationally expensive to evaluate than JSON Pointer references. Servers supporting JSON Path MUST implement appropriate safeguards against excessive resource consumption:

* Servers SHOULD enforce limits on the complexity of JSON Path expressions, such as maximum expression length, maximum nesting depth of selectors, or maximum number of filter predicates.

* Servers SHOULD enforce timeouts on JSON Path evaluation to prevent indefinitely long computations.

* Servers SHOULD limit the size of nodelists that can be produced by a single JSON Path expression. An expression that matches a very large number of nodes could consume excessive memory.

* Servers SHOULD consider the cumulative cost of evaluating multiple result references in a single request. An attacker might attempt to overwhelm the server by including many complex JSON Path expressions in a single request.

Servers that implement these limitations MUST return an "invalidResultReference" error when a limit is exceeded, with a description that indicates the nature of the limitation where possible without revealing sensitive operational details.

4.1.2. Resource Exhaustion Through Reference Chains

Enhanced result references enable longer and more complex chains of dependent method calls within a single request. While this is a design goal that improves efficiency, it also creates opportunities for resource exhaustion:

* An attacker might construct requests with deeply nested result reference dependencies that require the server to maintain large amounts of intermediate state.

* The expanded use of result references in /set operations means that a single method call might need to resolve many result references across complex nested object structures.

* Result references in FilterCondition objects might require repeated evaluation of the same reference multiple times during query execution.

Servers SHOULD enforce reasonable limits on the complexity of result reference usage, such as:

* Maximum number of result references per request

* Maximum number of result references per method call

* Maximum depth of property nesting when resolving result references in /set operations

* Maximum number of FilterCondition objects containing result references in a single query

These limits SHOULD be documented and consistent with other JMAP request limits defined in [RFC8620].

4.1.3. Algorithmic Complexity Attacks

Certain combinations of JSON Path expressions and data structures can exhibit pathological performance characteristics. For example:

* Recursive descent (..) combined with wildcard selectors on deeply nested structures can produce very large intermediate results.

* Filter expressions with complex predicates applied to large arrays can require substantial computation.

* JSON Path expressions that produce large nodelists, when used multiple times within a single request, can cause quadratic or worse time complexity.

Servers SHOULD implement query cost analysis and impose limits on expensive operations. When possible, servers SHOULD detect potentially problematic patterns before full evaluation and reject them proactively.

4.2. Data Integrity and Validation Concerns

The use of result references in /set operations creates additional considerations for data validation and integrity.

4.2.1. Type Confusion

Result references resolve to values of arbitrary JSON types. When these values are used to populate object properties or filter conditions, type mismatches can occur. While this specification requires servers to validate types after resolution, implementations must be careful to avoid type confusion vulnerabilities:

* Type validation MUST occur after result reference resolution but before the value is used in any security-sensitive operation.

* Servers MUST NOT make assumptions about the type of a resolved value based on the context. The value type must be explicitly checked.

* String-to-number or string-to-boolean coercions MUST NOT be performed automatically, as these can lead to unexpected behavior and potential security issues.

4.2.2. Injection Attacks

When result references are used to populate FilterCondition properties, there is a potential for filter injection attacks. Consider a scenario where a result reference is used to set a filter property that expects a simple string value, but the resolution produces a complex object or an array:

* Servers MUST validate that resolved values match the expected type and structure for the property or filter being populated.

* Servers MUST NOT attempt to serialize complex objects to strings or perform other automatic conversions that might enable injection attacks.

* When a resolved value is used in a filter that performs string matching or other operations, the server MUST ensure that the value is properly sanitized and does not alter the semantic meaning of the filter.

4.2.3. Circular Reference Prevention

While JMAP’s sequential processing model prevents direct circular references (a method cannot reference a method that comes after it), enhanced result references in nested object structures could potentially create complex data copying patterns that consume excessive memory or lead to unexpected behavior. Servers SHOULD implement safeguards against pathological data copying patterns.

4.3. Privacy Considerations

Enhanced result references can affect user privacy in several ways:

4.3.1. Data Leakage Across Contexts

The ability to copy data from one JMAP object to another using result references creates potential for unintended data leakage. For example:

* Private or sensitive data from one context might be inadvertently copied into a more widely shared context.

* Metadata or structural information might be leaked through the types and structures of resolved values.

Implementations SHOULD provide mechanisms for administrators and users to understand and control how data flows between different contexts through result references.

4.3.2. Audit and Logging

The use of result references can obscure the true source of data in audit logs. When an object is created or modified using result references, audit logs SHOULD record not just the final object state but also information about which data was derived from result references. This enables proper forensic analysis and compliance with data protection regulations.

4.4. Implementation Considerations

Implementers should be aware of several security-relevant implementation challenges:

4.4.1. Parser Security

JSON Path parsers are complex and may contain vulnerabilities. Servers implementing JSON Path support SHOULD:

* Use well-tested, maintained JSON Path libraries rather than custom implementations.

* Keep JSON Path libraries up to date with security patches.

* Consider sandboxing or isolating JSON Path evaluation from critical server components.

4.4.2. Cache Security

To improve performance, servers might cache the results of result reference resolution. Such caches MUST respect all security boundaries:

* Cached results MUST NOT be shared across different users or security contexts.

* Cache entries MUST be invalidated when relevant access control policies change.

* Cache implementations MUST be resistant to timing attacks that might reveal the presence or contents of cached data.

4.4.3. Backward Compatibility

Servers that support enhanced result references MUST continue to correctly handle requests from clients that do not use this extension. In particular:

* Servers MUST correctly reject enhanced result references from clients that have not included the urn:ietf:params:jmap:refplus capability in the "using" property of the request.

* The presence of properties with "#" prefixes in contexts where standard JMAP does not allow result references MUST be handled appropriately (typically as invalid property names).

5. IANA considerations

5.1. JMAP Capability Registration for "refplus"

IANA will register the "refplus" JMAP Capability as follows:

Capability Name: urn:ietf:params:jmap:refplus
Specification document: this document
Intended use: common
Change Controller: IETF
Security and privacy considerations: this document, Section 4

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/rfc/rfc6901>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/rfc/rfc8259>.

[RFC8620] Jenkins, N. and C. Newman, "The JSON Meta Application Protocol (JMAP)", RFC 8620, DOI 10.17487/RFC8620, July 2019, <https://www.rfc-editor.org/rfc/rfc8620>.

[RFC8621] Jenkins, N. and C. Newman, "The JSON Meta Application Protocol (JMAP) for Mail", RFC 8621, DOI 10.17487/RFC8621, August 2019, <https://www.rfc-editor.org/rfc/rfc8621>.

[RFC9535] Gssner, S., Ed., Normington, G., Ed., and C. Bormann, Ed., "JSONPath: Query Expressions for JSON", RFC 9535, DOI 10.17487/RFC9535, February 2024, <https://www.rfc-editor.org/rfc/rfc9535>.

6.2. Informative References

[I-D.ietf-jmap-calendars] Jenkins, N. and M. Douglass, "JSON Meta Application Protocol (JMAP) for Calendars", Work in Progress, Internet-Draft, draft-ietf-jmap-calendars-25, 8 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-jmap-calendars-25>.

[RFC9610] Jenkins, N., Ed., "JSON Meta Application Protocol (JMAP) for Contacts", RFC 9610, DOI 10.17487/RFC9610, December 2024, <https://www.rfc-editor.org/rfc/rfc9610>.

Appendix A. Changes

[[This section to be removed by RFC Editor]]

draft-degennaro-jmap-refplus-00

* Initial version

Author's Address

Mauro De Gennaro
Stalwart Labs LLC
1309 Coffeen Avenue, Suite 1200
Sheridan, WY 82801
United States of America
Email: mauro@stalw.art
URI: https://stalw.art