# Microsoft Graph REST API Guidelines

Table of Contents

[Microsoft Graph REST API Guidelines 1](#_Toc87203511)

[Introduction 1](#_Toc87203512)

[Design Approach 2](#_Toc87203513)

[Naming 3](#_Toc87203514)

[Uniform Resource Locators (URLs) 5](#_Toc87203515)

[Recommended Modeling Patterns 7](#_Toc87203516)

[Behavior Modeling 8](#_Toc87203517)

[Error Handling 9](#_Toc87203518)

[API contract and non-backward compatible changes 11](#_Toc87203519)

[Versioning and Deprecation 12](#_Toc87203520)

[Deprecation Process 12](#_Toc87203521)

[Common API Patterns 15](#_Toc87203522)

[Final thoughts 16](#_Toc87203523)

#### History

|  |  |
| --- | --- |
| Date | Notes |
| 2021-Sep-28 | Alignment with Azure style. |
| 2020-Oct-04 | Initial version in Wiki. |

## Introduction

When building a digital ecosystem providing APIs that are easy to discover, simple to use, fit to purpose, and consistent across your products can make the difference between success and failure.

This document offers guidance that Graph API developer teams MUST follow to ensure that customers have a great experience. A new API design should meet the following goals:

- Developer friendly via consistent naming, patterns, and web standards (HTTP, REST, JSON)

- Efficient and cost-effective.

- Work well with SDKs in many programming languages.

- Sustainable & versionable via clear API contracts.

The Microsoft Graph guidelines are an extension of the Microsoft REST API guidelines. Readers are assumed also be reading the Microsoft REST API guidelines and be familiar with them. Graph guidance is a superset of the Microsoft API guidelines and services should follow them except where this document outlines specific differences or exceptions to those guidelines.

This document borrows heavily from multiple public sources such as:

1. Microsoft Azure REST API Guidelines
2. Google Cloud Platform APIs
3. WSO2 Rest API Design Guidelines and others.

Technology and software is constantly changing and evolving, and as such, this is intended to be a living document. [Open an issue](https://github.com/microsoft/api-guidelines/issues/new/choose) to suggest a change or propose a new idea.

## 

## Design Approach

The design of your API is arguably the most important investment you will make in it. The design of your API is what creates the first impression for developers. Microsoft Graph APIs follow HTTP, REST, and JSON standards and described using ODATA conventions and CSDL for schema definition (see [Documentation · OData - the Best Way to REST](https://www.odata.org/documentation/)).

We promote API-first design approach where you begin by creating an interface or API for your service first. Subsequently you follow with the service implementation which relies on the specified interface. API -first approach is essential for agility, predictability, and reuse of your APIs as it promotes good understanding of your modeling domain, consistent interface contract, and understanding of how supporting service will evolve.

In general API design includes the following steps:

* Define your domain model
* Derive and name your API resources
* Determine required behavior
* Determine user roles and permissions
* Specify errors

To create a good API you need to start with understanding your use cases and supporting domain model. We describe domain models in terms of entities, their properties, and relationships and further refer to it as entity data model. There is no one-to-one correspondence between domain model elements and API resources as APIs usually support only customer-facing use cases.

After API resources are identified you need to name them and their properties so that the API will be discoverable and intuitive for developers, and consistent with other Graph resources.

When resources are defined it’s time to think about the behavior of your API and define required operations and actions.

At every step of your design you need to consider security, privacy and compliance as an intrinsic parts of your API implementation. And finally based on your API resources, their behavior, and anticipated exceptions you need to identify potential error scenarios with secure and descriptive messaging.

### Naming

Consistent naming is foundational for API usability. API resources are typically described by nouns. You need to consider that resources and property names appear in API URLs and payloads and should be descriptive and easy to understand. Therefore you should follow the rules in the table below:

|  |  |
| --- | --- |
| ✖ AVOID redundant words in names. | * Right: /places/{id}/**type** and /phones/{id}/**number** * Wrong: /places/{id}/***placeType*** and /phones/{id}/**phoneNumber** |
| ✖ AVOID using brand names in type or property names. | * Right: chat * Wrong: teamsChat |
| ✖ AVOID using acronyms or abbreviations unless they are broadly understood. | * Right: url or htmlSignature * Wrong: msodsUrl or dlp |
| ✔ DO use singular nouns for type names. | * Right: address * Wrong: addresses |
| ✔ DO use plural nouns for collections (for listing a type or collection properties). | * Right: addresses * Wrong: address |
| ✔ DO pluralize the noun even when followed by an adjective (a "postpositive"). | * Right: passersby or mothersInLaw * Wrong: notaryPublics or motherInLaws |
| ✔ DO name property as “email” | * Right: email * Wrong: mail |

#### Casing

|  |  |
| --- | --- |
| ✔ DO use lower camel case for all names and namespaces | * Right: automaticRepliesStatus. * Wrong: kebab-case or snake\_case. |
| ✔ DO case two-letter acronyms with the same case. | * Right: ioLimit or totalIOAmount * Wrong: iOLimit or totalIoAmount |
| ✔ DO case three+ letter acronyms the same as a normal word. | * Right: fidoKey or oauthUrl * Wrong: webHTML |
| ✖ DO NOT capitalize the word following a [prefix](https://www.thoughtco.com/common-prefixes-in-english-1692724) or words within a [compound word](http://www.learningdifferences.com/Main%20Page/Topics/Compound%20Word%20Lists/Compound_Word_%20Lists_complete.htm). | * Right: subcategory, geocoordinate or crosswalk * Wrong: metaData, semiCircle or airPlane |
| ✔ DO capitalize within hyphenated and open (spaced) compound words. | * Right: fiveYearOld, daughterInLaw or postOffice * Wrong: paperclip, changingroom or fullmoon |

#### Prefixes and Suffixes

|  |  |
| --- | --- |
| ✔ DO use namespaces | * Microsoft Graph model types can be declared within a [type namespaces](https://github.com/microsoft/api-guidelines/blob/graph/graph/type-namespaces) to reduce the need to prefix types with a qualifier to ensure uniqueness. |
| ✔ DO suffix date and time properties with | * Right: dueDate — an Edm.Date * Right: createdDateTime — an Edm.DateTimeOffset * Right: recurringMeetingTime — an Edm.TimeOfDay * Wrong: dueOn or startTime * Right: instead both above are an Edm.DateTimeOffset |
| ✔ DO use the Duration type for durations, but if using an int, append the units. | * Right: passwordValidityPeriod — an Edm.Duration * Right: passwordValidityPeriodInDays — an Edm.Int32 (NOTE use of Edm.Duration type is preferable) * Wrong: passwordValidityPeriod — an Edm.Int32 |
| ✖ DO NOT suffix property names with primitive type names unless the type is temporal. | * Right: isEnabled or amount * Wrong: enabledBool |
| ✔ DO prefix property names for properties concerning a different entity. | * Right: siteWebUrl on driveItem, or userId on auditActor * Wrong: webUrl on contact when its the companyWebUrl |
| ✔ DO prefix Boolean properties with is, unless this leads to awkward or unnatural sounding names for Boolean properties. | • Right: isEnabled or isResourceAccount  • Wrong: enabled or allowResourcAccount  • Right: allowNewTimeProposals or allowInvitesFrom — subjectively more natural than the examples below  • Wrong: isNewTimeProposalsAllowed or isInvitesFromAllowed — subjectively more awkward that the examples above |

### Uniform Resource Locators (URLs)

A Uniform Resource Locator (URL) is how developers access the resources of your API.

Navigation path to Graph resources generally broken into multiple segments:

**{scheme}://{host}/{version}/{category}/{resourcePath}[?{query}]** where

* **scheme and host segments** are always [https://graph.microsoft.com](https://graph.microsoft.com/v1.0/users);
* **version** can be V1.0 or beta;
* **category** segment is modeled as an entity set or a singleton representing logical top-level API category;
* **resourcePath segment** can address an entity, collection of entities, property or operation available for an entity. Structure of the resource path is covered in detail in the [OData Version 4.01. Part 2: URL Conventions](http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part2-url-conventions.html);
* **query string** must follow the OData standard for query representations and is covered in [Query](#_Query) section.

While HTTP defines no constraints on how different resources are related together, it does encourage the use of URL path segment hierarchies to convey a relationship. In Microsoft Graph lifetime relationships between resources supported by the notions of singletons, entitySets, entities, complex types and navigation properties.

#### Category

We define a **top-level API category** as a coherent area of API functionality which covers one or multiple high-level use cases defined from customer and enterprise perspectives and represents one of the following:

1. A core *user-centric concept* of the Graph

* For example: /users, /groups or /me

1. A Microsoft *product or service offerings* covering multiple use cases

* For example: /teamwork, /directory

1. A *feature*offering covering a single use case and *shared* across multiple Microsoft products

* For example: /search, /notifications, /subscriptions, /files

1. *Administrative configuration* functions for specific products. (Note: this is not final and may be adjusted based on the survey results)

* For example: /admin/exchange

1. Internal Microsoft requirements for publishing Privileged and Hidden APIs, routing, and load testing

* For example: /loadTestEntities

Top-level API categories are aligned with documentation, developer tools, and in general are relatively stable. If a new category needs to be created, it should follow supporting governance

#### Query

Microsoft Graph APIs should support basic query options in conformance with OData specifications and [Microsoft REST API Guidelines](https://github.com/microsoft/api-guidelines/blob/master/Guidelines.md#7102-error-condition-responses).

|  |
| --- |
| ✔ DO support $select, $top, $filter query options |
| ✔ DO support $filter with eq, ne operations on properties of entities in the requested entity set |
| ✔ may support $skip, $count |
| ✔ DO use batch request to avoid too long query options |
| ✔ DO use request body with the content-type text/plain for POST queries |
| ✔ DO use request body with the content-type |

Limitations of $query requests made to Microsoft Graph:

* Microsoft Graph only supports having all the query options completely in the request body or completely in the request url. Graph doesn't support query options present in both places.
* The parameters in $query should not span multiple workloads. Support for $query right now is limited to properties belonging to the same workload.

The query options part of an OData URL can be quite long, potentially exceeding the maximum length of URLs supported by components involved in transmitting or processing the request. One way to avoid this is wrapping the request in a batch request, which has the penalty of needing to construct a well-formed batch request body. An easier alternative for GET requests is to append /$query to the resource path of the URL, use the POST verb instead of GET, and pass the query options part of the URL in the request body as described in the chapter [OData Query Options](http://docs.oasis-open.org/odata/odata/v4.01/odata-v4.01-part2-url-conventions.html#sec_PassingQueryOptionsintheRequestBody).

#### Microsoft Graph rules for modeling resources:

|  |
| --- |
| ✔ DO verify that the primary id of an entity type is string |
| ✔ DO verify that the primary key must also be defined as a property. |
| ✔ DO verify that the primary key is composed of a single property and not multiple. |
| ✖ DO NOT add the property id to a complex type |
| **Serialization** |
| ✔ DO use an object as the root of all JSON payloads. |
| ✔ DO use a value property in the root object to return a collection. |
| ✔ DO include @odata.type annotations when the type is ambiguous. |

### Recommended Modeling Patterns

There are different approaches to design an API resource model in situations with multiple variants of common concept. Type Hierarchy, Facets, and Flat bag of properties are three most often used patterns in Microsoft Graph today:

* Type hierarchy is represented by one abstract base type with a few common properties and one sub-type for each variant [api-guidelines/adding-subtypes.md at graph · microsoft/api-guidelines (github.com)](https://github.com/microsoft/api-guidelines/blob/graph/graph/adding-subtypes.md)
* Facets are represented by a single entity type with common properties and one facet property (of complex type) per variant. The facet properties only have a value when the object represents that variant [api-guidelines/adding-subtypes.md at graph · microsoft/api-guidelines (github.com)](https://github.com/microsoft/api-guidelines/blob/graph/graph/adding-subtypes.md)
* Flat bag of properties is represented by one entity type with all the potential properties plus an additional property to distinguish the variants, often called type. The type property describes the variant and also defines properties that are required/meaningful for the variant given by the type property. [api-guidelines/adding-subtypes.md at graph · microsoft/api-guidelines (github.com)](https://github.com/microsoft/api-guidelines/blob/graph/graph/adding-subtypes.md)

The following table describes shows summary of main qualities for each pattern and will help to select a pattern preferred for your use case.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| API qualities  Patterns | Properties and behavior described in metadata | Suited for strongly typed languages | Simple query construction | Syntactical backward compatible |
| Type hierarchy | yes | yes | no | yes |
| Facets | ok | ok | yes | yes |
| Flat bag | no | no | yes | yes |

## Behavior Modeling

#### HTTP Operations

The HTTP operations dictate how your API behaves. The URL of an API, along with its request/response bodies, establishes the overall contract that developers have with your service. As an API provider, how you manage the overall request / response pattern should be one of the first implementation decisions you make.

|  |
| --- |
| ✔ DO use POST to create new entities in insertable entity sets |
| ✔ DO use PATCH to edit updatable resources |
| ✔ DO use DELETE to delete deletable resources |
| ✔ DO return a Location header with the edit URL or read URL of a created resource |

For a complete list of standard REST operations you can refer to the [Microsoft REST API Guidelines](https://github.com/microsoft/api-guidelines/blob/master/Guidelines.md#7102-error-condition-responses).

#### Microsoft Graph rules for modeling behavior:

|  |  |
| --- | --- |
| ✔ DO use GET …/{collection} and GET …/{collection}/{id} for listing and reading resources. | Error |
| ✔ DO use POST …/{collection} for creating resources. | Error |
| ✔ DO use PATCH …/{collection}/{id} for updating resources. | Error |
| ✖ AVOID using PUT …/{collection}/{id} for updating resources. | Warning |
| ✖ DO NOT use PATCH to replaces resources or PUT to partially update resources. | Error |
| ✖ AVOID patterns that require multiple round trips to complete a single logical action. | Warning |
| ✔ CONSIDER supporting return and omit-nulls preferences. | Warning |

### Error Handling

[Microsoft REST API Guidelines](https://github.com/microsoft/api-guidelines/blob/master/Guidelines.md" \l "7102-error-condition-responses) provide guidelines that Microsoft REST APIs should follow when returning error condition responses. However, the structure, form and content of the error response payloads is currently not enforced leading to undiscoverable and inconsistent error messages. You can improve API traceability and consistency by using recommended Graph error model:

{

  "error": {

    "code": "BadRequest",

    "message": "Unsupported functionality",

    "target": "query",

    "details": [

      {

       "code": "301",

       "target": "$search",

       "message": "$search query option not supported"

      }

    ],

    "innererror": {

"code": "301",

"message": "Cannot process the request because a required field is missing.",

      "stacktrace": [...],

    }

  }

}

The following examples demonstrate error modeling for common use cases:

* **Simple error**: A workload wants to report an error with top-level details only. The library allows the workload to create the error object and just specify the top-level error code, message and target (optional).

{

"error": {

"code": "badRequest",

"message": "Cannot process the request because it is malformed or incorrect.",

"target": "Service X (Optional)"

}

}

* **Detailed error**: A workload wants to report an error and provide service-specific details of the error via the innererror property of the error object. The code property in innererror is optional but highly recommended. It is intended to allow workloads to supply a service-specific error code to help differentiate errors that share the same top-level error code but reported for different reasons.

{

"error": {

"code": "badRequest",

"message": "Cannot process the request because it is malformed or incorrect.",

"innererror": {

"code": "requiredFieldOrParameterMissing",

"message": "Cannot process the request because a required field is missing.",

"stacktrace": "[StackTrace]"

}

}

}

* **Error with collection of related errors**: A workload wants to report an error together with a collection of related errors via the details collection property of the error object.

{

"error": {

"code": "forbidden",

"message": "Access to the resource is restricted.",

"details": [

{

"code": "unathorized",

"message": "You are not authorized to access the resource"

}

]

}

}

#### Microsoft Graph enforces the list of following error rules:

|  |  |
| --- | --- |
| ✔ DO return an error property with a child code property in all error responses. | Error |
| ✔ DO return a 403 Forbidden error when insufficient scopes are present on the auth token. | Error |
| ✔ CONSIDER returning a 404 Not found error if a 403 would result in information disclosure. | Warning |
| ✔ DO return a 429 Too many requests error when the caller has exceeded throttling limits. | Error |

For a complete mapping of error codes to HTTP statuses please refer to the [rfc7231 (ietf.org)](https://datatracker.ietf.org/doc/html/rfc7231#section-6).

### API contract and non-backward compatible changes

Microsoft Graph definition of breaking changes is based on the [Microsoft REST API Guidelines](https://github.com/microsoft/api-guidelines/blob/graph/Guidelines.md#123-definition-of-a-breaking-change).

In general, making changes to the API contract for existing elements is considered breaking. Adding new elements is allowed and not considered a breaking change.

Additional Microsoft Graph rules most often observed in practice are summarized in the table below:

|  |  |
| --- | --- |
| ✔ DO use **not-breaking** changes | * Addition of an annotation OpenType="true" * Addition of properties that are nullable or have a default value * Addition of a member to an evolvable enumeration * Removal, rename, or change to the type of an open extension * Removal, rename, or change to the type of an annotation * Introduction of paging to existing collections * Changes to error codes * Changes to the order of properties * Changes to the length or format of opaque strings, such as resource IDs |
| ✖ DO NOT use **breaking** changes | * Changes to the URL or fundamental request/response associated with a resource * Changing semantics of resource representation * Removal, rename, or change to the type of a declared property * Removal or rename of APIs or API parameters * Addition of a required request header * Addition of a EnumType members for non-extensible enumerations * Addition of a Nullable="false" properties to existing types * Addition of a Nullable="false" parameters to existing actions and functions * Adding attributes to existing nodes is considered breaking. * Adding annotations ags:IsHidden="true". |

For the full list of rules you can refer to [this section of the OData V4 spec](https://docs.oasis-open.org/odata/odata/v4.0/errata02/os/complete/part1-protocol/odata-v4.0-errata02-os-part1-protocol-complete.html#_Toc406398209).

## Versioning and Deprecation

When changes are imminent you need to support explicit versioning as it's critical that clients can count on services to be stable over time, and it's critical that services can add features and make changes. Microsoft Graph API follows the guidance described in the Model Versioning section in the [Microsoft REST API guidelines](https://github.com/Microsoft/api-guidelines/blob/master/Guidelines.md#12-versioning).

The following versions of the Microsoft Graph API are currently available:

1. API sets on the v1.0 endpoint (https://graph.microsoft.com/v1.0) are in general availability (GA) status.
2. API sets on the beta endpoint (https://graph.microsoft.com/beta) are in beta or private preview status.

In general API breaking changes are not allowed in the GA version of Microsoft Graph API. For beta API you can expect breaking changes and deprecation of APIs from time to time.

As new versions of the Microsoft Graph REST APIs and Microsoft Graph SDKs are released, earlier versions will be retired. Microsoft declares a version as deprecated at least 24 months in advance of retiring it. Similarly, for individual APIs that are generally available (GA), Microsoft declares an API as deprecated at least 24 months in advance of removing it from the GA version.

### Deprecation Process

If your API requires an introduction of breaking changes you must follow the deprecation process:

* After API review board approvals, add Revisions annotation to the API definition CSDL with the following terms:
  + Kind of change: Deprecated (vs "added" to track added properties/types)
  + Human readable description of the change: Used in changelog, documentation etc.
  + Version: Used to identify group of changes. Of the format "YYYY-MM/Category" where "YYYY-MM" is the month the deprecation is announced, and "Category" is the category under which the change is described in the ChangeLog
  + Date: Date when the element was marked as deprecated
  + RemovalDate: Date when the element may be removed

The annotation can be applied to a type, entity set, singleton, property, navigation property, function or action. If a type is marked as deprecated, it is not necessary to mark members of that type as deprecated, nor is it necessary to annotate any usage of that type in entity sets, singletons, properties, navigation properties, functions, or actions.

Example of property annotation:

<EntityType Name="outlookTask" BaseType="Microsoft.OutlookServices.outlookItem" ags:IsMaster="true" ags:WorkloadName="Task" ags:EnabledForPassthrough="true">

<Annotation Term="Org.OData.Core.V1.Revisions">

<Collection>

<Record>

<PropertyValue Property = "Date" Date="2020-08-20"/>

<PropertyValue Property = "Version" String="2020-08/Tasks\_And\_Plans"/>

<PropertyValue Property = "Kind" EnumMember="Org.OData.Core.V1.RevisionKind/Deprecated"/>

<PropertyValue Property = "Description" String="The Outlook tasks API is deprecated and will stop returning data on August 20, 2022. Please use the new To Do API."/>

<PropertyValue Property = "RemovalDate" Date="2022-08-20"/>

</Record>

</Collection>

</Annotation>

...

</EntityType>

 When the request URL contains a reference to a deprecated model element, the HTTP response includes a [Deprecation header](https://tools.ietf.org/html/draft-dalal-deprecation-header-02)  (with the date the element was marked as deprecated) and a Sunset header (with the date 2 years beyond the Deprecation date). Response also includes a link header pointing to the breaking changes page.

Deprecation header example:

*Deprecation: Thursday, 30 June 2022 11:59:59 GMT  
Sunset: Wed, 30 Mar 2022 23:59:59 GMT  
Link:*[*https://docs.microsoft.com/en-us/graph/changelog#2022-03-30\_name*](https://docs.microsoft.com/en-us/graph/changelog#2022-03-30_name)*; rel="deprecation"; type="text/html"; title="name",*[*https://docs.microsoft.com/en-us/graph/changelog#2020-06-30\_state*](https://docs.microsoft.com/en-us/graph/changelog#2020-06-30_state)*; rel="deprecation"; type="text/html"; title="state"*

Deprecation cadence:

* As an API developer you can mark individual API schema elements as deprecated on a quarterly basis, after going through a API review and approval process. Quarterly deprecation cadence will allow the services to evolve schemas over time, without waiting for a coordinated, monolithic endpoint change.
* Once marked as deprecated, the elements must continue to be supported for a minimum of 3 years before removal (or a minimum of 2 years if, based on telemetry, the element is no longer being used).
* Tools, documentation, SDKs, and other mechanisms are driven by this explicit deprecation to reach out to customers that may be affected by the changes.
* APIs in beta or preview versions can use the same mechanism but are not bound by the quarterly cadence or minimal support period before removal of deprecated elements.

## Common API Patterns

The guidelines in previous sections are intentionally high-level and provide a jump start for Graph API design. More detailed design guidance on REST APIs is published at the [Microsoft REST API Guidelines](https://github.com/microsoft/api-guidelines/) and Graph specific are outlined in the table below.

**API Patterns** are design documents providing best practices for MS Graph API development. They are to serve as the means by which API teams discuss and come to consensus on API guidance.

You can find references in the most common patterns in the table below:

|  |  |  |
| --- | --- | --- |
| Pattern | Description | Reference |
| Key Property |  | [Key Property](https://dev.azure.com/msazure/One/_wiki/wikis/Microsoft%20Graph%20Partners/103125/Design) |
| Entity Type |  |  |
| Complex Type |  |  |
| Shared Type |  |  |
| Type Hierarchy |  |  |
| Dictionary |  |  |
| Evolvable Enums |  |  |
| Type Namespace |  |  |
| Change Tracking |  |  |
| Long Running Operations |  |  |
| Delta Queries |  |  |

These patterns are provided as instruction for API desiners to help write simple, intuitive, and consistent APIs, and are used by API reviewers as a basis for review comments.

## Final thoughts

These guidelines describe the upfront design considerations, technology building blocks, and common patterns that teams encounter when building their Graph APIs.

The links below provide references to the foundational documentation on related topics:

* [Microsoft REST API Guidelines](https://github.com/microsoft/api-guidelines/)
* [OData Guidelines](http://www.odata.org/documentation/)
* [RESTful web API design](https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design)
* [Microsoft Graph Documentation](https://developer.microsoft.com/en-us/graph/docs/concepts/overview)
* [Microsoft Graph Explorer](https://aka.ms/ge)