(Meta)

This document exists as a gDoc for ease of collaboration, but, after it is no longer a draft, it will be relocated to a more permanent, and more easily-discoverable, home. This document may be incorporated into the API Cookbook or the Common Experience: Content Strategy documentation; this is still TBD.

About

This is an opinionated style guide for Cox Automotive API documentation, covering both what to document and the format in which to document it. The aim of this style guide is to facilitate the creation of high quality and consistent documentation for our APIs. We believe writers benefit from strict direction and standards, so this style guide includes no "should" statements, only "must" statements. As an example:

To be included in the Cox Automotive Developer Portal, your API documentation must comply with all standards and guidelines in this style guide.

This style guide draws from several sources. The most notable ones are:

- "The Definitive Guide to Creating API Documentation" (PDF) by Ed Marshall;
- "The Best API Documentation" by Brad Fults; and
- Manheim's API Documentation.

About Style Guide Revisions

While authoritative, this is a "living" style guide, and the authors welcome contributions and corrections. Please help make this style guide better.

Once your API documentation is included in the Cox Automotive Developer Portal, every effort must be made to maintain compliance with this style guide. The Retail API Platform team will work with API producers (developers and writers) to alert them to style guide changes.

The Style Guide Cascade

In brief:

1. Start here.
2. Then, see https://common-experience.firebaseapp.com/product-content/voice-and-tone
3. Then, see https://docs.microsoft.com/en-us/style-guide/welcome/
4. Then, see https://ahdictionary.com/

These guidelines address concerns specific to API documentation, but they are not a comprehensive guide to writing. When a question arises that is not answered here, see the Common Experience: Product Content. From there, follow *their* style guide cascade: if the question is not answered there, see the Microsoft Writing Style Guide. If the question is not answered there, see the American Heritage Dictionary.

While this guide supersedes all others below it (and Common Experience: Content Strategy supersedes those below it, and so on), there may still be times where inconsistency or ambiguity in the guidelines make API documentation difficult. In these (hopefully rare) cases, please work with the Retail API Platform team and Common Experience: Content Strategy team to improve the guidelines.

Scope

The content on the Cox Automotive Developer Portal primarily consists of

- Tutorials and Articles: step-by-step instructions, success stories, best practices, and other documentation which, while informative, does not specifically describe the surface area of our APIs.
- API Directory: highly-structured documentation which organizes and specifically describes the surface area of our APIs.

The content in the API Directory primarily consists of

- A Reference section: functions that are common across all of our APIs, e.g., authentication, pagination, Service Level Objectives (SLOs).
- The API Catalog: endpoint-operation documentation organized by resources and grouped by categories with related content, e.g., code examples, resource object models.

Except where explicitly noted, this style guide refers to the documentation within the API Catalog.

Implementation Details

All APIs must be documented in compliance with the OpenAPI Specification (OAS). That is, the API documentation must be comprised of valid OpenAPI documents.

OpenAPI documents can be imported into the API Directory section of our Readme.io-powered Developer Portal.

In the event that OpenAPI documents are not organized by contributing writers in the way they need to be presented in the Developer Portal, the Retail API Platform team performs the necessary transformations. That is, the OpenAPI documents may be split or combined to create the desired API Directory information architecture (Developer Portal navigation).

Note that, while the OpenAPI documents directly inform the API documentation, the OpenAPI documents themselves are not promoted on the Developer Portal. (However, as we learn more about API consumer needs, we anticipate this may change in the future.)

JSON or YAML

While all examples and snippets in this style guide are JSON, it does not matter whether API producers use JSON or YAML. JSON-to-YAML conversion tools may make the examples and snippets in this guide more useful to some API producers.

Formatting

Within the OpenAPI documents, use Markdown for text formatting. OAS stipulates supporting CommonMark 0.27, but Readme.io supports GitHub Flavored Markdown, extending the Markdown the Developer Portal supports beyond CommonMark.

Guidelines

Be consistent. One of the main goals of this style guide is to facilitate consistency. Do not knowingly deviate from guidelines and precedent, but do not silently comply with what you believe to be inferior. If you believe you have a better solution than what has been stated in this style guide (and possibly widely implemented), please help make this style guide better.

Be terse and factual. Most well-established practices for technical writing apply to documenting APIs, i.e., write clearly in plain English; be factual. Additionally, write very tersely; consumers do not like flowery prose. (This is based largely on Best Practice 2 from "The Definitive Guide to Creating API Documentation" (PDF)).

Question/challenge adjectives and adverbs. Related to “be terse and factual” above, API documentation is not the place to cite product benefits. Consumers will lose confidence and trust in your documentation if you use unnecessary adjectives and adverbs.

- Bad: GET /menageries/id/{menageriesId} - Speedily retrieves one of your intuitively-organized Pet Store mViews™ so you can manage your menagerie with ease.
- Good: GET /menageries/id/{menageriesId} - Retrieves a single menagerie by identifier.

If it’s hard to document, the problem might be the API itself. Consider your API documentation as a contract between your services and the consumers. If it is difficult to explain to consumers how to use your service, the problem is less likely that they don’t understand you…and more likely that you don’t understand them. In other words, just because you built the service right doesn’t mean you built the right service! Work with Product and UX team members to better understand your consumers’ needs and then change your API to address those needs. Consider contract-first development strategies, i.e., start with the documentation.

Avoid circular definitions and tautological statements. This is best illustrated by example:

- Bad:
  Get a pet - GET /pets/id/{petId}

  Gets a pet.

  Path Params

  petId* - string - The required petId.

  Query Params

  version - integer - The optional version.

- Good:
  Retrieve a pet - GET /pets/id/{petId}

  Retrieves a pet resource by id. An optional version parameter can be set to download a previous version.

  Path Params

  petId* - string - The pet identifier. Format is 32-character UUID.

  Query Params

version - integer - The version to download. Defaults to the current version.

See also Tautology Club.

Write URLs in lowercase, e.g., /pets. The only exceptions to this are

- Path parameters may use camelCase, e.g., /pets/id/{petId}
- Enumerated values are written in uppercase, e.g., /pets?type=DOG
- Code examples may contain case-sensitive parameter values, e.g., /pets?eq(breed,Persian)

Do not use an ambiguous "id." Use camelCase to indicate particular identifiers, e.g., petId.

Use the recipes and snippets. Below are recipes which address common needs and OpenAPI document snippets for common scenarios.

Recipes

I want to document a category, e.g., Pet Store Services.

When we have more than ten endpoints documented on the Cox Automotive Developer Portal, then endpoints must be categorized. Valid categories are:

- Accounting Services
- Collateral Management Services
- Core Services
- Customer Services
- Digital Document Services
- Finance and Insurance Services
- Inventory Services
- Media Services
- Registration and Titling Services
- Service and Parts Services

Every category must be titled with the services (e.g., Pet Store Services) and described with a category overview.

The category overview answers the following questions for the reader:

- What is this category?
- In which circumstances should you use the APIs within this category?

- What do the APIs within this category help you achieve?
- What do the APIs within this category provide?

Additionally, there must be a way for consumers to contact us about the category and the endpoints/operations within it.

OpenAPI document

- Use the info.title field to title the category with the services, e.g., Pet Store Services.
- Use the info.description field to describe the category with the category overview.
- Use the info.contact to indicate how API consumers contact us.

The category overview consists of

- A paragraph of short sentences (strive for no more than five) which help the consumer understand which resources are managed by endpoints in this category;
- A list of the resources with brief descriptions; and
- Other notes, e.g., links to tutorials, if available.

The Cox Automotive Developer Portal will generate content based on info.contact immediately below the category overview.

Then, document each endpoint within the category.

Example

```
"info": {
    "title": "Pet Store Services",
        "description": "The **Pet Store Services APIs** allow pet
store employees and other API consumers to manage their pets and
owners, through an API for each of the following key data
objects:\n\n- pets, which are sellable animals;\n- pet stores, which
are locations where a pet may be sold to an owner; and\n- owners,
which are animal buyers.\n\nSee [Building pet listings to post on
Craigslist](#some-article-url).",
    "contact": {
        "name": "API Support",
        "url": "https://coxauto.service-now.com/"
    },
    …
}
```

I want to document endpoints in a consistent order.

Endpoints are ordered alphabetically.

For example, the endpoints under Pets might include /pets, /pets/id/{petId}, /pets/id/{petId}/sell, and /pets/mine, with various operations on each, and the documentation would appear in the following order:

Pets

GET /pets
POST /pets

GET /pets/id/{petId}
PUT /pets/id/{petId}
DELETE /pets/id/{petId}
PATCH /pets/id/{petId}

POST /pets/id/{petId}/sell

GET /pets/mine

I want to document an endpoint, e.g., /pets.

Every endpoint must be titled with the managed resource and described with an endpoint overview.

The endpoint overview answers the following questions for the reader:

- What is this API endpoint?
- In which circumstances should you use this API endpoint?
- What does this API endpoint help you achieve?
- What does this API endpoint provide?
- What is the shape of the objects used by this API endpoint?

OpenAPI document

- Use the tags field of each operation within the path item object to title the endpoint with the resource (sentence case, pluralized) which it manages, e.g., all operations on the /pets endpoint will be be tagged with “Pets”; all operations on the /pet-stores endpoint will be tagged with “Pet stores”.

- In the tags section, use the tag's description field to describe the endpoint with the endpoint overview.
- Include camelCased path parameters in curly braces, e.g., "/pets/id/{petId}" in the operation key.

The endpoint overview consists of

- A paragraph of short sentences (strive for no more than five) which help the consumer understand in which circumstances to use this endpoint; and
- Documentation for each resource object model associated with this endpoint (typically one).

The documentation for each resource object model consists of

- A table describing the fields of the resource object model; and
- An example of the resource object model (an example response).

The table must have column headings of "name", "type", and "description" and have a title "The [resource] object" styled as an h3 (###).

The example response must be styled as a code snippet (```…```) and have a title "Example response" styled as an h4 (####).

Then, document each operation which can be performed on the endpoint.

Example

```
"paths": {
    "/pets": {
        "get": {
            "tags": [
                "Pets"
            ],
            …
        },
        "post": {
            "tags": [
                "Pets"
            ],
            …
        }
    }
},
```

```
    "tags": [
        {
            "name": "Pets",
            "description": "The Pets API allows users to manage pets,
which are animals that a pet store can sell or has sold. Pets are
created when an animal is received by a pet store.\n\n### The pet
object\n\nname | type | description\n------|------|------------\nhref
| URL | Link to this object.\ncreatedOn | timestamp | Time at which
the object was created.\nname | string | The pet’s name. This will be
printed on name tags.\ntype | string | One of `dog`, `cat`, or
`fish`.\ncolor | string | The pet’s primary color.\ngender | string |
One of `male` or `female`.\nbreed | string | The pet’s breed.\nowner
| URL | Link to the pet’s current owner.\n\n#### Example
response\n\n```js\n{\n  \"href\":
\"https://api.coxautoservices.com/pets/id/123e4567-e89b-12d3-a456-426
655440000\",\n  \"created\": 1528992903,\n  \"name\": \"Fluffy\",\n
\"type\": \"DOG\",\n  \"color\": \"Black\",\n  \"gender\":
\"FEMALE\",\n  \"breed\": \"mixed\",\n  \"owner\":
\"https://api.coxautoservices.com/owners/id/00112233-4455-6677-8899-a
abbccddeeff\"\n}\n```"
        }
    ]
}
```

TODO: I want to document resource object models which are used by endpoints with disparate tags.

Is this a genuine need? I anticipate resource object models being shared across different endpoints, e.g., `/pets` and `/pets/id/{petId}`, but they would both be organized, via tags, under "Pets"—and, per I want to document an endpoint, e.g., /pets, the endpoint overview goes with the tag. So, really, I'm wondering if there are resources that are equally fundamental to different tags (which group similar endpoints)—and does resource object model documentation for those resources thus belong in the category overview? What about resource object models which are used by endpoints in different categories?

I want to document operations in a consistent order.

For every unique endpoint, operations are documented in the following order:

- GET
- PUT
- POST

- DELETE
- OPTIONS
- HEAD
- PATCH
- TRACE

For example, the endpoints under Pets might include /pets, /pets/id/{petId}, /pets/id/{petId}/sell, and /pets/mine, with various operations on each, and the documentation would appear in the following order:

Pets

GET /pets
POST /pets

GET /pets/id/{petId}
PUT /pets/id/{petId}
DELETE /pets/id/{petId}
PATCH /pets/id/{petId}

POST /pets/id/{petId}/sell

GET /pets/mine

I want to document an operation, e.g., GET /pets.

Every operation must be titled with the equivalent brief, plain English imperative statement and described with an operation overview.

Operation → brief, plain English imperative statement
- GET (list of items) operation → List all [resources]
- GET (single item) operation → Retrieve a/an [resource]
- PUT (update) operation → Update a/an [resource]
- PUT (replace) operation → Replace a/an [resource]
- POST (create) operation → Create a/an [resource]
- POST (other?) operation → ???
- DELETE operation → Delete a/an [resource]
- OPTIONS operation → ???
- HEAD operation → ???
- PATCH (update) operation → Update a/an [resource]
- TRACE operation → ???

Additionally, every operation must indicate the required OAuth 2.0 scopes a consumer needs to perform the operation.

Additionally, every operation must have an operationId. (The Retail API Platform team uses operationIds to transform OpenAPI documents into API documentation.)

Additionally, the first tag on each operation must indicate the resource, e.g., Pets. (The Retail API Platform team uses the first tag on each endpoint to create the navigation.)

OpenAPI document

- Use the tags field of the operation to indicate the resource (sentence case, pluralized) which the parent path item object manages, e.g., an operation within the /pets endpoint will be be tagged with “Pets”; an operation within the /pet-stores endpoint will be tagged with “Pet stores”. See I want to document an endpoint, e.g., /pets for more about tags.
- Use the summary field of the operation to title the operation.
- Use the description field of the operation to describe the operation with the operation overview.
- Use the security field of the operation to list the OAuth 2.0 scopes.
- Use the operationId field of the operation to uniquely identify the operation.

The operation overview consists of

- A paragraph of short sentences (strive for no more than five) which help the consumer understand what this operation does, how parameters affect what it does, deviations from standard SLOs, and any caveats/gotchas to performing this operation.

The Cox Automotive Developer Portal will generate content based on the security field immediately below the operation overview.

Then, document each parameter, the responses, examples, and errors.

Additional Notes for Specific Operations

Expanding on OAS description of requestBody, GET, DELETE, OPTIONS, HEAD, and TRACE operations must not have body parameters.

Example

```
"/pets": {
    …
    "get": {
        "operationId": "getPets",
        …
```

```
    }
}
```

I want to document parameters in a consistent order.

Put required parameters before optional parameters. After that, consider what order makes the most sense to a consumer. When in doubt (if any of several parameters are just as likely to be relevant to a consumer), alphabetize.

I want to document a non-body parameter.

As of OAS 3.0.1, body parameters are documented with different fields than non-body parameters. Do not use `"in": "body"`.

Every non-body parameter, i.e., a header, path, query, or cookie parameter, must be named, typed, and described.

OpenAPI document

- Use the name field of the parameter to name the parameter.
- Use the schema field of the parameter to indicate the parameter type.
- Use the description field of the parameter to describe the parameter.
- Use the in field of the parameter to indicate how to group it.
- In order to present grouped parameters in a consistent order in the API documentation, they must appear in the following order: "header", "path", "query", "cookie".
- Use the required field to indicate whether the parameter is required.

The description consists of

- A declarative sentence fragment which restates the name and spells out any abbreviations or acronyms, e.g., if the name is “petId”, the first sentence is “The pet identifier”; if the name is “serviceDate”, the first sentence is “The date of when the vehicle was last serviced.”
- Any format notes, e.g., “Format is `bytes={startRange}-{endRange}`.”
- Any default values, e.g., “Defaults to the current version.”
- Any additional notes.
- Any links, e.g., “See [HTTP Header Linking: 3. The Link Header Field](https://www.greenbytes.de/tech/webdav/draft-nottingham-http-link-header-00.html#rfc.section.3).”

The Cox Automotive Developer Portal will generate content based on the name, schema, and required field in-line with the description.

The description does not include the type (it would be redundant with the content generated from the schema field) nor whether the field is required or optional (it would be redundant with the content generated from the required field).

When there are no params in a group, e.g., no path params, the Retail API Platform team will ensure the documentation explicitly indicates this. (This is based largely on the note in Best Practice 1 from "The Definitive Guide to Creating API Documentation" (PDF)).

Additional Notes for Specific Parameters

Path Params

Path parameters are always required.

Example

```
{
    "name": "limit",
    "in": "query",
    "description": "The page indicator. Format is
`limit={start},{count}`. Defaults to `1,25` (the first 25 items).
Alternative format: `limit({start},{count})`. See
[Pagination](#TBD).",
    "schema": {
        "type": "string"
    }
}
```

I want to document the body parameter (request body).

As of OAS 3.0.1, body parameters are documented with different fields than non-body parameters. Do not use `"in": "body"`.

When the request body is not an object

The request body must be described. Note that a non-object cannot be required; if the request body needs to be required, wrap it in an object.

OpenAPI document

- Set the content field to an object with a key of "application/json" or something more specific, e.g.,"application/vnd.coxauto.v1+json". (Per the API Cookbook, the data exchange must be JSON.)

- Use the content["application/json"].description field of the request body object to describe the request body.
- Use the content["application/json"].schema.type field of the request body to indicate the request body type.

The request body description consists of

- A declarative sentence fragment which spells out any abbreviations or acronyms, e.g., if the request body were instead a path or query param with a name of “petId”, the first sentence is “The pet identifier”; if the name would be “serviceDate”, the first sentence is “The date of when the vehicle was last serviced.”
- Any format notes, e.g., “Format is `bytes={startRange}-{endRange}`.”
- Any default values, e.g., “Defaults to the current version.”
- Any additional notes.
- Any links, e.g., “See [HTTP Header Linking: 3. The Link Header Field](https://www.greenbytes.de/tech/webdav/draft-nottingham-http-link-header-00.html#rfc.section.3).”

The Cox Automotive Developer Portal will generate content based on the schema in-line with the description.

Example

```
"requestBody": {
    "content": {
        "application/vnd.coxauto.v1+json": {
            "schema": {
                "description": "The pet type to create. Any lowercase
characters will be converted to uppercase.",
                "type": "string"
            }
        }
    }
}
```

When the request body is an object

The request body must be named. Every field on that object must be named, typed, and described. Include at least one example of the request body.

OpenAPI document

- Set the content field to an object with a key of "application/json" or something more specific, e.g.,"application/vnd.coxauto.v1+json". (Per the API Cookbook, the data exchange must be JSON.)

- Use the content["application/json"].description field of the request body object to name the request body.
- Set the content["application/json"].schema.type field of the request body to "object".
- Use the content["application/json"].schema.properties field of the request body to name, type, and describe the fields on the object. (See below for more details.)
- Use the content["application/json"].schema.required field of the request body to indicate which fields are required.
- Use the content["application/json"].examples field to provide an example.

When the field of the request body object is not an object

These guidelines largely align with the guidelines for non-body parameters.

Every field must be named, typed, and described.

OpenAPI document

- Use the key of the properties object to name the field.
- Use the type field of the property to indicate the field type.
- Use the description field of the property to describe the field.

The description consists of

- A declarative sentence fragment which restates the name and spells out any abbreviations or acronyms, e.g., if the name is “petId”, the first sentence is “The pet identifier”; if the name is “serviceDate”, the first sentence is “The date of when the vehicle was last serviced.”
- Any format notes, e.g., “Format is `bytes={startRange}-{endRange}`.”
- Any default values, e.g., “Defaults to the current version.”
- Any additional notes.
- Any links, e.g., “See [HTTP Header Linking: 3. The Link Header Field](https://www.greenbytes.de/tech/webdav/draft-nottingham-http-link-header-00.html#rfc.section.3).”

The Cox Automotive Developer Portal will generate content based on the name, type, and required field (which is set outside the properties) in-line with the description.

The description does not include the type (it would be redundant with the content generated from the type field) nor whether the field is required or optional (it would be redundant with the content generated from the required field).

When the field of the request body object is an object (and deeper levels of nested objects)

These guidelines largely align with the guidelines for when the request body is an object.

The field must be named and described. Every field on that object must be named, typed, and described.

OpenAPI document

- Use the key of the properties object (of the parent field) to name the field.
- Set the type field of the property to "object".
- Use the description field of the property to describe the field.
- Use the properties field of the field to name, type, and describe the fields on the object. (See above for more details.)
- Use the required field of the parent field to indicate which fields are required.

The description consists of

- A declarative sentence fragment which restates the name and spells out any abbreviations or acronyms, e.g., if the name is “petId”, the first sentence is “The pet identifier”; if the name is “serviceDate”, the first sentence is “The date of when the vehicle was last serviced.”
- Any additional notes.

The description does not include the type (it will be obvious it is an object) nor whether the field is required or optional (object fields themselves are constructs for fields which may be required or optional).

Example

This example is contrived/complicated to showcase the various guidelines above and is *not* an endorsement of circular definitions.

```
"requestBody": {
    "content": {
        "application/vnd.coxauto.v1+json": {
            "schema": {
                "type": "object",
                "description": "nameOfRequestBodyObject",
                "properties": {
                    "field1": {
                        "type": "string",
                        "description": "Description of first field
(which happens to be required)."
                    },
                    "field2": {
                        "type": "string",
```

```
                        "description": "Description of second field
(which happens to be optional)."
                    },
                    "field3": {
                        "type": "object",
                        "description": "Description of third field
(which happens to be a nested object).",
                        "properties": {
                            "field4": {
                                "type": "string",
                                "description": "Description of fourth
field (which happens to be required)."
                            },
                            "field5": {
                                "type": "string",
                                "description": "Description of fifth
field (which happens to be optional)."
                            }
                        },
                        "required": [
                            "field4"
                        ]
                    }
                },
                "required": [
                    "field1"
                ]
            },
            "examples": …
        }
    }
}
```

Regardless of whether the request body is an object or not

When there is no request body, the Retail API Platform team will ensure the documentation explicitly indicates this. (This is based largely on the note in Best Practice 1 from "The Definitive Guide to Creating API Documentation" (PDF)).

I want to document examples for individual parameters.

Every example must be titled.

OpenAPI Document

- Use the summary field to title the example.
- Use the description field to provide any additional notes.
- Use the value field to capture the literal example.

The title is a unique (if there are multiple examples) imperative statement.

The value is what the parameter is set to and does not need to be prefixed with “parameter=” nor “object: ”.

The Cox Automotive Developer Portal will prefix a single example with “Example: “ or insert an “Examples” heading above a list of examples.

Example

```
"examples": {
    "secondPage": {
        "summary": "Retrieve second page of results",
        "description": "Alternative format: `limit(26,25)`.",
        "value": "26,25"
    },
    "thirdPage": {
        "summary": "Retrieve third page of results",
        "description": "Alternative format: `limit(51,25)`.",
        "value": "51,25"
    }
}
```

TODO: I want to document how parameters are used in conjunction.

Is this a genuine need? I think this would only apply to path params, query params, or a combination of two kinds of params, e.g., path+body, path+query, or query+body. Should we provide a way to document a “full” example for an endpoint’s operation?

I want to document expected responses in a consistent order.

Order by HTTP status code, e.g., 200 before 404.

I want to document expected responses.

Several expected responses are documented in the centralized Reference section above API-specific documentation. Document expected responses when the API responds with an HTTP status code not covered in the Reference section or when you can provide more specific details.

Every expected response must be described.

HTTP status code (to particular operation) → description template

- 200 response to GET (list of items) operation → OK: The requested items are in the response body. When no [resources] meet the filter criteria, the items array in the response body is empty.
- 200 response to GET (single item) operation → OK: The requested [resource] is in the response body.
- 201 response to POST (create) operation → Created: The [resource] was created. Retrieve it by the URL in the Location header.
- 202 response to POST (update) operation → Accepted: The [resource] was queued to be updated. Retrieve the latest status by the URL in the Location header.
- 202 response to DELETE operation → Accepted: The [resource] was queued for deletion. Query for updates by the URL in the Location header.
- 202 response to POST (request bulk data) operation → Accepted: The bulk data request was queued. When the data is ready, retrieve it by the URL in the Location header.
- 204 response to POST (to start async process) operation → No Content: The request to [verb] the [resource] was queued.
    - Example: No Content: The request to sign the types with the eSignature was queued.
- 204 response to PUT (to replace) operation → No Content: The [resource] was replaced.
- 204 response to PATCH (to update) operation → No Content: The [resource] was updated.
- 204 response to DELETE (single item) operation → No Content: The [resource] was deleted.
- 400 response to POST (to create) operation → Bad Request: The [resource] was not created. The JSON in the request payload was malformed. Do not repeat the request without modifications. See response body.
- 400 response to PUT (to update) operation → Bad Request: The [resource] was not updated. The JSON in the request payload was malformed. Do not repeat the request without modifications. See response body.
- 401 response to ANY operation → Unauthenticated: The Authorization header on the request is expired or invalid. See the WWW-Authenticate header.
- 403 response to POST (to create) operation → Forbidden: The user was not authorized to create the [resource]. The [specific] scope is required.

- 403 response to PATCH (to update) operation → Forbidden: The user was not authorized to update the [resource]. The [specific] scope is required.
- 403 response to PUT (to replace) operation → Forbidden: The user was not authorized to replace the [resource]. The [specific] scope is required.
- 403 response to DELETE operation → Forbidden: The user was not authorized to delete the [resource]. The [specific] scope is required.
- 403 response to GET (list of items) operation → Forbidden: The user was not authorized to retrieve the items. The [specific] scope is required.
- 403 response to GET (single item) operation → Forbidden: The user was not authorized to retrieve the [resource]. The [specific] scope is required.
- 404 response to GET (request bulk data, following 202) operation → Not Found: The requested bulk data is not yet available (or no longer available). Try again later.
- 404 response to ANY operation → Not Found: The requested [resource] was not found.
- 405 response to !GET (when only GET is allowed) operation → Method Not Allowed: This URL does not support that method. Make a GET request instead.
- 405 response to !POST (when only POST is allowed) operation → Method Not Allowed: This URL does not support that method. Make a POST request instead.
- 405 response to !SOME (when only SOME are allowed) operation → Method Not Allowed: This URL does not support that method. Make a […, …, or …] request instead.
- 405 response to ANY (when the path params indicate a URI which does not support ANY) operation → Method Not Allowed: This URL does not support that method. Check the [`specific path param`] param.
- 406 response to ANY operation → Not Acceptable: The server could not generate content acceptable to the Accept headers on the request.
- 408 response to ANY operation → Request Timeout: The client canceled the request or sent an incomplete request. Try again later.
- 409 response to POST (to create) operation → Conflict: The [resource] could not be created because it already exists. Retrieve it by the URL in the Location header.
- 409 response to POST (to create [resource] with unique constraint on [field]) operation → Conflict: The [resource] could not be created because a [resource] with [field] [field value] already exists. Retrieve it by the URL in the Location header.
- 409 response to PATCH (to update a [resource] in a way that violates the unique constraint on [field]) operation → Conflict: The [resource] could not be updated because a [resource] with [field] [field value] already exists. Retrieve it by the URL in the Location header.
- 409 response to PATCH (to [verb] a [resource] in a way that violates a constraint) operation → Conflict: The [resource] could not be [verbed] because [constraint].
    - Example: Conflict: The bid could not be accepted because an accepted bid already exists.
- 412 response to PUT (to update) operation → Precondition Failed: The [resource] could not be updated because it had been more recently modified. See ETag and Last-Modified headers. Include valid [If-Match] header on the request.

- 415 response to POST (to create) operation → Unsupported Media Type: The [resource] could not be created because the Content-Type header on the request was incorrect.
- 415 response to PATCH (to update) operation → Unsupported Media Type: The [resource] could not be updated because the Content-Type header on the request was incorrect.
- 417 response to PATCH (to execute one or more tests and replace one or more resources) operation → Expectation Failed: None of the [resources] have been replaced. One or more of the requested tests failed.
- 417 response to PATCH (to act on [resource] in a way that violates business logic) operation → Expectation Failed: The [resource] could not be [acted upon] because [business logic].
    - Example: Expectation Failed: The bid could not be accepted because the vehicle is ineligible.
    - Example: Expectation Failed: The bid could not be accepted because the offer expired.
- 417 response to DELETE (in a way that violates hierarchy constraints) operation → Expectation Failed: The requested [resource] could not be deleted because it had dependent [resources] which are not marked deleted.
- 418 response to ANY operation → I'm a teapot: The server was unable to brew coffee because it is a teapot.
- 420 response to ANY operation → Enhance Your Calm: Slow down your API calls if at all possible. [See Retry-After header.]
- 422 response to GET (list of items) operation → Unprocessable Entity: The [resources] could not be retrieved because of a validation error. Check the parameters and see [CarQL](#). Do not repeat the request without modifications. See response body.
- 422 response to GET (single item) operation → Unprocessable Entity: The [resource] could not be retrieved because of a validation error. Do not repeat the request without modifications. See response body.
- 422 response to POST (to create) operation → Unprocessable Entity: The [resource] was not created. The request payload was missing required fields or contained invalid values. Do not repeat the request without modifications. See response body.
- 422 response to PUT, POST (to update) operation → Unprocessable Entity: One or more fields could not be updated. The request payload was missing required fields or contained invalid values. Do not repeat the request without modifications. See response body.
- 422 response to OTHER operation → Unprocessable Entity: [Clarify what did or did not happen.] [Answer what was bad about it as specifically as possible. If more than one circumstance could return a 422, list the possible circumstances and provide multiple examples.] Do not repeat the request without modifications. See response body.
- 500 response to ANY operation → Internal Server Error: The server encountered an unexpected error. If this persists, contact support.
- 503 response to ANY operation → Service Unavailable: The server temporarily timed out. Try again later. [See Retry-After header.] If this persists, contact support.

HTTP status codes 301 and 304 will be documented in the Reference section; it is not necessary to document them on a per-call basis.

When documenting expected responses to other or more specific status code/operation scenarios, follow the guidelines below (and suggest new entries to the description template list above):

- The description starts with the code's meaning in Title Case, per https://en.wikipedia.org/wiki/List_of_HTTP_status_codes, followed by a colon.
    - Note that we intentionally replace 401 status code "Unauthorized" with "Unauthenticated" to disambiguate it from 403 status code "Forbidden."
- When possible, be specific. Prefer actual resource, e.g., "pet," over generic "resource."
- Use simple past tense (was/were) to describe the request or parts of the request.
- Use simple past tense (was/were) to describe what happened on the server before (perhaps to warrant) this response being sent and read.
- Use simple present tense (is/are) to describe the response.
- Use imperative statements to describe suggested next actions.
- Use "header" to refer to response headers. If you need to refer to a request header, clarify with "on the request," e.g., see description template for 406 response above.

Additionally, any headers must be described. Include at least one example of the response headers and response body.

OpenAPI Document

- Use the key of the responses object to indicate the HTTP status code.
- Use the description field to describe the expected response.
- Use the headers field to document the headers:
    - Use the key of the headers object to indicate the header name.
    - Use the description field to describe the header.
    - Use the examples field to provide examples.
- Use the content field to document the response body:
    - Set the content field to an object with a key of "application/json" or something more specific, e.g.,"application/vnd.coxauto.v1+json". (Per the API Cookbook, the data exchange must be JSON.)
    - Set the content["application/json"].schema.type field of the request body to "object".
    - Use the content["application/json"].schema.properties field of the request body to name and type the fields on the object. (This may nest.)
    - Use the content["application/json"].examples field to provide an example.

The Cox Automotive Developer Portal will flag 4XXs and 5XXs as errors.

Example

See Snippets for examples.

I want to document non-standard Service Level Objectives (SLOs).

Document deviations from standard SLOs on a per-operation basis. See I want to document an operation, e.g., GET /pets.

I want to write about our philosophy, roadmap, or other things which are not easily addressed by the recipes above.

Share your content idea with the Retail API Platform team so we can find the best place to showcase your work and help you coordinate with others who may be working on similar topics. Then, write your documentation in a GitHub Enterprise repository. The Retail API Platform team will help you set up your repository so the content will sync to the Cox Automotive Developer Portal, e.g., when changes are committed to the master branch.

Some topics may be considered for inclusion in the Reference section of the API Directory.

I want to inform consumers about backwards-compatible changes.

Backwards-compatible changes do not require bumping the API version or any other special callout. Write an article to make consumers aware of the new capabilities you have added.

I want to inform consumers about backwards-incompatible changes.

Backwards-incompatible changes require releasing a new version of the API. Additionally, the change must be noted in the Cox Automotive Developer Portal's changelog.

I want an operation to be hidden from the Cox Automotive Developer Portal.

Currently, the Retail API Platform team assumes all content in the OpenAPI documents are intended for publication. If you want an operation to be hidden from the Cox Automotive Developer Portal, do not include it in the OpenAPI document you provide to the team.

I want an operation to only appear on the Cox Automotive Developer Portal for authenticated Cox Automotive team members.

At this time (2018-Q4), the Cox Automotive Developer Portal is only available to authenticated Cox Automotive team members. When we allow other consumers access, e.g., third-pary developers, we will evaluate the need to create per-audience documentation and share our strategy via this recipe.

Example API documentation

Pet Store Services

Pet Store Services - Overview

The **Pet Store Services APIs** allow pet store employees and other API consumers to manage their pets and owners, through an API for each of the following key data objects:

- pets, which are sellable animals;
- pet stores, which are locations where a pet may be sold to an owner; and
- owners, which are animal buyers.

See Building pet listings to post on Craigslist.

Contact us at API Support with questions and comments about Pet Store Services.

Pets

The Pets API allows users to manage pets, which are animals that a pet store can sell or has sold. Pets are created when an animal is received by a pet store.

The pet object

name	type	description
href	URL	Link to this object.
createdOn	ISO 8601 string	Date and time at which the object was created.

name	string	The pet's name. This will be printed on name tags.
type	string	One of the values returned by List all pet types.
gender	string	One of `MALE` or `FEMALE`.
breed	string	The pet's breed.
color	string	The pet's primary color.
owner	URL	Link to the pet's current owner.

Example response

```
{
  "href":
"https://api.coxautoservices.com/pets/id/123e4567-e89b-12d3-a456-4266
55440000",
  "createdOn": "2013-07-30T12:20:08Z",
  "name": "Fluffy",
  "type": "DOG",
  "gender": "FEMALE",
  "breed": "mixed",
  "color": "Black",
  "owner":
"https://api.coxautoservices.com/owners/id/00112233-4455-6677-8899-aa
bbccddeeff"
}
```

List all pets

GET https://api.coxautoservices.com/pets

Retrieves pet resources, optionally filtered and paginated by CarQL path parameters. The pet resources are returned in sorted order, with the most recently created pets appearing first.

This operation requires the following scope(s):

- read:pets

TODO: Headers

…

Path Params

No path params.

Query Params

type [string]	Filter by pet type. Format is `type={CarQL Operators}`. Defaults to no filter. See CarQL. **Example: Retrieve only cats and dogs** `type=in=(CAT,DOG)` Alternative format: `in(type,(CAT,DOG))`.
gender [string]	Filter by gender. Format is `gender={CarQL Operators}`. Defaults to no filter. See CarQL. **Example: Retrieve only female pets** `gender=FEMALE` Alternative format: `eq(gender,FEMALE)`.
breed [string]	Filter by breed. Format is `breed={CarQL Operators}`. Defaults to no filter. See CarQL. **Example: Retrieve only pets of of Persian breed** `breed=Persian` Alternative format: `eq(breed,Persian)`.
limit [string]	The page indicator. Format is `limit={start},{count}`. Defaults to `1,25` (the first 25 items). Alternative format: `limit({start},{count})`. See Pagination.

	Examples:
	
	Retrieve second page of results
	
	`limit=26,25`
	
	Alternative format: `limit(26,25)`.
	
	Retrieve third page of results
	
	`limit=51,25`
	
	Alternative format: `limit(51,25)`.

sort [string]	The sort indicator. Format is `sort={field1},{-field2}`. Defaults to `-createdOn` (the most recently created pets appearing first). Alternative format: `sort({field1},{-field2})`. See Sorting.
	
	Examples
	
	Sort results by gender ascending (FEMALE before MALE)
	
	`sort=gender`
	
	Alternative format: `sort(gender)`.
	
	Sort results by type descending (FISH before DOG), then breed ascending (Mixed before Persian)
	
	`sort=-type,breed`
	
	Alternative format: `sort(-type,breed)`.

Body Params

No body params.

TODO: Responses

…

TODO: Create a pet

POST https://api.coxautoservices.com/pets

Creates a new pet resource.

This operation requires the following scope(s):

- write:pets

TODO: Headers

…

Path Params

No path params.

Query Params

No query params.

Body Params

pet

name* [string]	The pet's name. This will be printed on name tags.
type* [string]	The pet's type. Format is uppercase. This must be one of the values returned by List all pet types.
gender* [string]	The pet's gender. Must be one of `MALE` or `FEMALE`.
breed [string]	The pet's breed.
color [string]	The pet's primary color.

Example: Create a Russian Blue cat

```
{
    "name": "Alexander",
    "type": "CAT",
    "gender": "MALE",
    "breed": "Russian Blue",
```

```
    "color": "gray"
}
```

TODO: Responses

…

Pet types

The Pet types API allows users to retrieve a list of possible pet types, which can be used for the `type` argument in the Pets API. Additionally, the value of the `type` property in the pet resource object will be one of these values. Pet types are centrally managed by Cox Automotive.

List all pet types

GET https://api.coxautoservices.com/pet-types

Retrieves pet types. The pet types are returned in alphabetical order.

This operation requires the following scope(s):

- read:pets

TODO: Headers

…

Path Params

No path params.

Query Params

No query params.

Body Params

No body params.

TODO: Responses

…

TODO: Example OpenAPI Document

```
{
    "openapi": "3.0.1",
    "info": {
```

```
        "title": "Pet Store Services",
        "description": "The **Pet Store Services APIs** allow pet
store employees and other API consumers to manage their pets and
owners, through an API for each of the following key data
objects:\n\n- pets, which are sellable animals;\n- pet stores, which
are locations where a pet may be sold to an owner; and\n- owners,
which are animal buyers.\n\nSee [Building pet listings to post on
Craigslist](#some-article-url).",
        "contact": {
            "name": "API Support",
            "url": "https://coxauto.service-now.com/"
        },
        "version": "1.0.0"
    },
    "servers": [
        {
            "url": "https://api.coxautoservices.com/",
            "description": "production"
        }
    ],
    "paths": {
        "/pets": {
            "get": {
                "tags": [
                    "Pets"
                ],
                "summary": "List all pets",
                "description": "Retrieves pet resources, optionally
filtered and paginated by CarQL path parameters. The pet resources
are returned in sorted order, with the most recently created pets
appearing first.",
                "operationId": "getPets",
                "parameters": [
                    {
                        "name": "type",
                        "in": "query",
                        "description": "Filter by pet type. Format is
`type={CarQL Operators}`. Defaults to no filter. See [CarQL](#TBD).",
                        "schema": {
                            "type": "string"
                        },
                        "examples": {
                            "catOrDog" : {
```

```
                                "summary": "Retrieve only cats and
dogs",
                                "description": "Alternative format:
`in(type,(cat,dog))`.",
                                "value": "in=(cat,dog)"
                            }
                        }
                    },
                    {
                        "name": "gender",
                        "in": "query",
                        "description": "Filter by gender. Format is
`gender={CarQL Operators}`. Defaults to no filter. See
[CarQL](#TBD).",
                        "schema": {
                            "type": "string"
                        },
                        "examples": {
                            "femaleOnly": {
                                "summary": "Retrieve only female
pets",
                                "description": "Alternative format:
`eq(gender,female)`.",
                                "value": "female"
                            }
                        }
                    },
                    {
                        "$ref": "#/components/parameters/limit-25"
                    }
                ],
                "responses": {
                    "200": {
                        "description": "…"
                    }
                },
                "security": [
                    {
                        "coreDomain": [
                            "read:pets"
                        ]
                    }
                ]
```

```
            },
            "post": {
                "tags": [
                    "Pets"
                ],
                "summary": "Create a pet",
                "description": "Create a new pet resource.",
                "operationId": "postPets",
                "responses": {
                    "201": {
                        "description": "…"
                    }
                },
                "security": [
                    {
                        "coreDomain": [
                            "write:pets"
                        ]
                    }
                ]
            }
        }
    },
    "components": {
        "parameters": {
            "limit-25": {
                "name": "limit",
                "in": "query",
                "description": "The page indicator. Format is
`limit={start},{count}`. Defaults to `1,25` (the first 25 items).
Alternative format: `limit({start},{count})`. See
[Pagination](#TBD).",
                "schema": {
                    "type": "string"
                },
                "examples": {
                    "secondPage": {
                        "summary": "Retrieve second page of results",
                        "description": "Alternative format:
`limit(26,25)`.",
                        "value": "26,25"
                    },
                    "thirdPage": {
```

```
                        "summary": "Retrieve third page of results",
                        "description": "Alternative format:
`limit(51,25)`.",
                        "value": "51,25"
                    }
                }
            }
        },
        "securitySchemes": {
            "coreDomain": {
                "type": "oauth2",
                "flows": {
                    "implicit": {
                        "authorizationUrl":
"https://auth.coxautoinc.com/authorize",
                        "scopes": {
                            "petstore.pets.read": "read pets",
                            "petstore.pets.write": "modify pets"
                        }
                    },
                    "clientCredentials": {
                        "tokenUrl":
"https://auth.coxautoinc.com/token",
                        "scopes": {
                            "petstore.pets.read": "read pets",
                            "petstore.pets.write": "modify pets"
                        }
                    }
                }
            }
        }
    },
    "tags": [
        {
            "name": "Pets",
            "description": "The Pets API allows users to manage pets,
which are animals that a pet store can sell or has sold. Pets are
created when an animal is received by a pet store.\n\n### The pet
object\n\nname | type | description\n------|------|------------\nhref
| URL | Link to this object.\ncreated | timestamp | Time at which the
object was created. Measured in seconds since the Unix epoch.\nname |
string | The pet’s name. This will be printed on name tags.\ntype |
string | One of `dog`, `cat`, or `fish`.\ncolor | string | The pet’s
```

```
primary color.\ngender | string | One of `male` or `female`.\nbreed |
string | The pet’s breed.\nowner | URL | Link to the pet’s current
owner.\n\n#### Example response\n\n```js\n{\n  \"href\":
\"https://api.coxautoservices.com/pets/id/123e4567-e89b-12d3-a456-426
655440000\",\n  \"created\": 1528992903,\n  \"name\": \"Fluffy\",\n
\"type\": \"dog\",\n  \"color\": \"Black\",\n  \"gender\":
\"female\",\n  \"breed\": \"mixed\",\n  \"owner\":
\"https://api.coxautoservices.com/owners/id/00112233-4455-6677-8899-a
abbccddeeff\"\n}\n```"
        }
    ]
}
```

To understand how OAS fields are used by Readme.io, see the following Knowledge Base documentation:

- Swagger/OpenAPI Import
- Swagger Specification
- Categories, Pages and Subpages
- Swagger Extensions
- Supported Features/FAQs

Additionally, the Retail API Platform team creates Overviews for each API by combining info.description and components.schemas.

Snippets

The following reference objects can be included to expedite documentation and ensure consistency.

TODO: Header Parameters: X-CoxAuto-AcceptFallback

…

TODO: Header Parameters: Correlation Id

…

TODO: Header Parameters: Content-Type

…

TODO: Query Parameters: CarQL Version

…

Query Parameters: CarQL Pagination

```
"limit-25": {
   "name": "limit",
   "in": "query",
   "description": "The page indicator. Format is `limit={start},{count}`. Defaults to `1,25` (the first 25 items). Alternative format: `limit({start},{count})`. See [Pagination](#TBD).",
   "schema": {
      "type": "string"
   },
   "examples": {
      "secondPage": {
         "summary": "Retrieve second page of results",
         "description": "Alternative format: `limit(26,25)`.",
         "value": "26,25"
      },
      "thirdPage": {
         "summary": "Retrieve third page of results",
         "description": "Alternative format: `limit(51,25)`.",
         "value": "51,25"
      }
   }
},
```

Security Schemes: Core Domain OAuth2

```
"coreDomain": {
    "type": "oauth2",
    "flows": {
        "implicit": {
            "authorizationUrl":
"https://auth.coxautoinc.com/authorize",
            "scopes": {
                …
            }
        },
        "clientCredentials": {
            "tokenUrl": "https://auth.coxautoinc.com/token",
```

```
            "scopes": {
                …
            }
        }
    }
}
```

TODO: Responses

See I want to document expected responses for guidance on describing headers.

2XX Responses

200 - response to GET (by id) operation

Replace `[resource]` with the actual resource, e.g., `pet`, describe actual headers, and provide actual header examples and body examples.

```
"200": {
    "description": "OK: The requested [resource] is in the response
body."
    "headers": {
        "X-Some-Header": {
            "description": "description goes here",
            "schema": {
                "type": "integer"
            },
            "examples": {
                "example1": {
                    "summary": "summary of example 1",
                    "description": "description of example 1",
                    "value": 1
                }
            }
        }
    },
    "content": {
        "application/json": {
            "schema": {
                "type": "object",
                "properties": {
                    "field1": {
                        "type": "string"
```

```
                    },
                    "field2": {
                        "type": "string"
                    }
                }
            },
            "examples": {
                "example1": {
                    "summary": "summary of example 1",
                    "description": "description of example 1",
                    "value": {
                        "field1": "foo",
                        "field2": "bar"
                    }
                }
            }
        }
    }
}
```

202 - response to POST (request bulk data) operation

Replace `[path-to-date]` with the actual path.

```
"202": {
    "description": "Accepted: The bulk data request was queued. When
the data is ready, retrieve it by the URL in the Location header."
    "headers": {
        "Location": {
            "description": "URL where data can be retrieved.",
            "schema": {
                "type": "string"
            },
            "examples": {
                "example1": {
                    "value":
"https://api.coxautoservices.com/[path-to-data]"
                }
            }
        }
    }
}
```

4XX Responses

403 - response to POST (to create) operation

Replace `[resource]` with the actual resource, e.g., `pet`, and replace `[specific]` with actual scope, e.g., `write:pets`.

```
"403": {
    "description": "Forbidden: The user was not authorized to create
the [resource]. The [specific] scope is required.",
    "content": {
        "application/json": {
            "schema": {
                "type": "object",
                "properties": {
                    "errors": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "code": {
                                    "type": "string",
                                    "description": "A unique message
code that facilitates translation lookups."
                                },
                                "message": {
                                    "type": "string",
                                    "description": "A description of
the error"
                                }
                            }
                        }
                    }
                }
            },
            "examples": {
                "example1": {
                    "value": {
                        "errors": [
                            {
                                "code": "insufficient_privileges",
```

```
                                "message": "You are not authorized to
perform this action."
                            }
                        ]
                    }
                }
            }
        }
    }
}
```

404 - response to ANY operation

Replace `[resource]` with the actual resource, e.g., `pet`, replace `[message-code]` with actual, namespaced message code, e.g., `petstore.pets.not_found`.

```
"404": {
    "description": "Not Found: The requested [resource] was not
found.",
    "content": {
        "application/json": {
            "schema": {
                "type": "object",
                "properties": {
                    "errors": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "code": {
                                    "type": "string",
                                    "description": "A unique message
code that facilitates translation lookups."
                                },
                                "message": {
                                    "type": "string",
                                    "description": "A description of
the error"
                                }
                            }
                        }
                    }
```

```
                }
            },
            "examples": {
                "example1": {
                    "value": {
                        "errors": [
                            {
                                "code": "[message-code]",
                                "message": "The [resource] was not
found."
                            }
                        ]
                    }
                }
            }
        }
    }
}
```

TODO: 406 - response to ANY operation

```
{
  "errors": [
    {
      "code": "unknown_accept",
      "message": "The Accept header on the request is invalid."
    }
  ]
}
```

TODO: 409 - response to POST (to create) operation

```
{
  "errors": [
    {
      "code": "…",
      "message": "The [resource] already exists."
    }
  ]
}
```

TODO: 409 - response to POST (to create [resource] with unique constraint on [field]) operation

```
{
```

```
   "errors": [
      {
         "code": "…",
         "message": "A [resource] with [field] [field value] already exists."
      }
   ]
}
```

TODO: 415 - response to POST, PUT, PATCH operation

```
{
   "errors": [
      {
         "code": "unknown_content_type",
         "message": "The Content-Type header on the request is invalid."
      }
   ]
}
```

TODO: 422 - response to OTHER operation (with multiple errors)

```
{
   "errors": [
      {
         "code": "…",
         "message": "Cannot update type field for unit."
         "properties": [
            {
               "property": "type"
            }
         ]
      },
      {
         "code": "gfb.missing_required_property",
         "message": "A required field was missing",
         "properties": [
            {
               "property": "vehicleInformation.description.modelYear"
            }
         ]
      },
      {
         "code": "…",
         "message": "The value for age was not an integer."
```

```
        {
          "property": "age"
        }
      }
  ]
}
```

Grammar and Punctuation

When to use sentence fragments

When providing an Endpoint-Operation description, write the first sentence as “This method …” Then, remove “This method” and capitalize the first word. Subsequent sentences are full sentences. E.g.,

> Returns a list of transactions that have contributed to the Stripe account balance (e.g., charges, transfers, and so forth). The transactions are returned in sorted order, with the most recent transactions appearing first.

- TODO: Capitalization rules
    - id? ID?
    - When in copy, write out “identifier.” Do not call it “unique identifier”; unique is implied. Always use lowercase. When describing a variable or parameter, prefix it and camelCase it such that the “i” is uppercase, e.g., petId.
- TODO: Acceptable abbreviations and acronyms
    - REST, HTTP, HTTPS, id, params, CarQL, MSRP
- TODO: How to indicate variables
    - Uppercase? In brackets?
    - In URLs, path params should be camelCase and surrounded by curly braces. Outside of URLs, e.g., when documenting a single path parameter, omit the curly braces.
    - Curly braces should surround variables when a reader needs to “mentally interpolate,” i.e., distinguish them from surrounding code content. When variables are in isolation or in a body of text, the code formatting will be sufficient to offset them.
    - `this {variable} requires curly braces`, but this `variable` does not.

Nomenclature

“Endpoint” versus “Resource”

- Per the Cox Automotive API Cookbook, an “endpoint” is a URL that our consumers call. A “resource” is a modeled object upon which an endpoint facilitates operations, e.g., updating a user resource by calling the /users endpoint.

“Params” versus “Arguments”

- Per Stackoverflow, A parameter is a variable in a method definition. When a method is called, the arguments are the data you pass into the method's parameters.
- In general, use “params” when documenting endpoints.
- If describing the data passed when calling endpoints, e.g., providing descriptions for examples, use “arguments.”

Prefer “Retrieves” over “Queries” or “Fetches”

- The RFC 7231, section 4.3.1: GET itself uses the word “retrieve.”
- “Queries” is ambiguous for consumers who are familiar with query languages, e.g., SQL, which regard update statements as well as select statements as “queries.”
- “Fetch” is a loaded term for consumers who are familiar with the Fetch standard.

“List” versus “Retrieve”

- If a GET method returns an array or list, use “list” (or, more likely, “list all [resources]”). This hold true even if the list may be empty or contain a single item.
- If a GET method returns a single item, not wrapped in an array or list, use “retrieve” (or, more likely “retrieve a/an [resource]”

TODO: “Search” versus “Filter”

- …

TODO: “Create” versus “Add”

- …

TODO: Prefer “Update” or “Replace” over “Modify”

- …

Changelog

- 201809XX:
    - Published initial version

Colophon

The following individuals have contributed to the creation of this style guide:

- Adam Hutton
- Vinod Chirayath
- Paul O’Fallon
- Jennifer Heater