Fortellis.io API Design Guidelines
V6 December 1, 2018

The Fortellis Automotive Commerce Exchange platform provides Application Programming Interfaces (APIs) to be used as business solutions. The intent is to dovetail unified design with industry-standard principles. The result is a network of solutions to enhance business processes and customer experiences. We're publishing these guidelines to define acceptable standards of RESTful API design. If you have suggestions or are interested in contributing to our community, please contact us.

The Fortellis APIs rely on the [RESTful](https://en.wikipedia.org/wiki/Representational_state_transfer) architecture supporting diverse use cases following REST standards and conventions. Our calls are returned in [JSON](http://www.json.org/). These API Design Guidelines are best practices we recommend you follow when publishing APIs on the Fortellis Automotive Commerce Exchange platform. The terms REST and RESTful are interchangeable.

RESTful Architecture

Fortellis adheres to Roy Fielding's dissertation [section 5.2](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2) RESTful architecture resource definitions. All information is considered a resource. Below are resource definitions:

* A graphic, image or document is a resource.
* Resources include temporal services connected to data sets.
* Mapped values to empty sets are resources even before they're defined.
* Collections, concepts, maps and entities are resources.
* Values are resources when representing identifiers.
* REST resource identifier labels of the interaction between components are resources.

Fortellis is the authority over resource identifiers and naming conventions. We maintain the semantic authenticity of our system mapping to provide functional services to our users.

Representation

We rely on the structure of REST components to manage the action of resources. Our APIs obtain the current or future state of a resource and transfer that representation to system components using bytes and metadata.

Services

We define services as software products providing functionality to the APIs and their products. The Fortellis API services provide users access to our platform, APIs, and provider solutions. Services are described in two categories:

1. **Capability APIs**: These are public user APIs delivering product-based capabilities. Capability APIs are reusable and designed to manage the front-end user experience for internal and external users while protecting specific domain models.
2. **Experience-specific APIs**: These are product-based services solving business needs through specialization and built on Capability APIs. Experience-specific APIs manage user interaction with the Fortellis platform.

Capability

Our API architecture is a solution-driven network. To support our users, we've designed the Fortellis Automotive Commerce Exchange capability to resolve business needs through our APIs and platform.

Namespaces

Our APIs are designed using the namespace model to clarify isolated entities such as structured domains and application data as identifiers. Namespaces provide logic to the APIs maintaining rigid isolation of our APIs from the controllers.

Domain Model

The Fortellis domain model defines the data accessed by APIs. We also use domains to group APIs as functional collections within an automotive business service. Examples of our data domains are Vehicle Sales Quotes, Merchandisable Vehicles, Service Appointments, and Repair Orders.

Consistency

API service functions follow defined rules providing a consistent experience for users to learn how to interact with the Fortellis APIs. The architecture of our platform is built to provide consistent service standards, vocabulary, interaction styles, and granularity for seamless interoperability.

Service Design Principles

The following design principles outline best practices for REST architectural standards.

Loose Coupling

The Fortellis API services are loosely coupled from one another. Due to the relationship between users and their interactions with our APIs, we've developed the API platform by loosening the dependencies between API service products and members to maintain interoperability. Services are safeguarded from the risk of exposing API implementation functions and protect user interaction from updated version functionality. Domain services offer functionality independent of other Fortellis domain products.

Encapsulation

A specific domain service may access another domain, its data, and functionality when permission is explicitly requested and granted by the owner of the associated domain. The goal is to provide functional services while protecting the independence of all domains. Services provide defined boundaries restricting unauthorized data access to Fortellis domains.

Stability & Reusability

Robust API design provides stability to support user access for a defined length of time. New API versions must protect backward compatibility. Fortellis applications are designed to be reusable for business use cases in multiple contexts.

Ease of Use

We follow industry standards of composability to aid users with ease of use while learning our applications. Services are understandable, offer well-defined authentication processes, usage, pagination, error codes, and deployment.

Externalization

Services are easily externalizable for users to interact with multiple API domains and use cases while respecting the functionality of individual solutions. Authentication, authorization procedures, and rate-limiting standards follow the domain model use cases and binding protocols.

HTTP Methods

Fortellis APIs use standard HTTP verbs as methods (endpoints). These verbs correspond to the primary operations of CRUD (Create, Read, Update, and Delete). Below are the acceptable methods with examples:
Parameter	Description	Example
GET	Retrieves a resource. GET must not change the state of an underlying resource.	GET https://api.fortellis.io/service/v1/appointments

POST	Creates a resource, sub-resource or executes an operation of a resource.	POST https://api.fortellis.io/service/v1/appointments/12345/service-items
PATCH	Performs a partial update to a resource and sub-resource. This method is rarely used in the Fortellis APIs.	
DELETE	Permanently deletes an existing resource.	DELETE https://api.fortellis.io/v1/repair-orders/R67890/lines/2

REST Without PUT

We discourage the use of the HTTP verb PUT. The reason is that with state transitions PUT often overwrites existing information of domain events. Instead, we recommend the REST without PUT technique. For example, changes to a customer address is a POST to a new customer resource instead of a PUT to an existing one. The concept of REST without PUT means you're making a change to the resource as an iteration rather than new event. The benefit is a more streamlined process updating domain-relevant events.

CQRS

The Command and Query Interfaces (CQRS) are enhanced by REST without PUT by maintaining the consistency of resources. For example, an API that controls customers is built to view a customer resource with a GET Customer of its current state from the existing database. To make a change you POST a <code>CustomerChange</code> to that resource. You don't automatically get CQRS by using REST without PUT but instead, it's easier to make the change with a POST.

The problem with using a PUT is it allows clients too much flexibility to edit the internal domain schema. Allowing a client to edit an internal resource creates architectural risk. Granted, many developers may be tempted to use PUT for ease of command. However, Fortellis doesn't recommend the use of PUT.

Common HTTP Headers

The HTTP headers provide metadata information about the body defining the uniformity of the standard. HTTP header names aren't case sensitive. The API headers manage cross-cutting concerns and don't provide domain-specific data values. We recommend avoiding the use of custom headers.

Fortellis users should understand that in some instances an HTTP header can be dropped or changed without notification. Therefore, our APIs must not be reliant on the rigidity of HTTP headers. We follow [RFC 7231](https://tools.ietf.org/html/rfc7231), [RFC 7232](https://tools.ietf.org/html/rfc7232), the [IANA Header Registry](https://www.iana.org/assignments/message-headers/message-headers.xhtml) and the [Internet Engineering Task Force (IETF) guidelines](http://tools.ietf.org/html/rfc7231#page-33).

In some instances HTTP headers can be dropped or changed without notification. Therefore, our APIs must not rely on the rigidity of HTTP headers.

Request Headers

The following HTTP request headers should be used consistently by the API's. Most of the headers specified follow [IETF 7231](http://tools.ietf.org/html/rfc7231), [IETF 7232](http://tools.ietf.org/html/rfc7232) and the [IANA Header Registry](https://www.iana.org/assignments/message-headers/message-headers.xhtml).

Accept

The `Accept` header indicates which MIME types the client is capable of understanding. During content negotiation, the server should select which of the MIME types for the response. That choice is communicated back to the client using the `Content-Type` response. All services must support the `Accept` header.

```
Accept: <MIME-type>/<MIME-subtype>
Accept: <MIME-type>/*
Accept: */*
```

Accept-Charset

The `Accept-Charset` indicates which character sets the requesting client is capable of understanding. Services must support `Accept-Charset` & [UTF-8](https://en.wikipedia.org/wiki/UTF-8).

```
Accept-Charset: <charset_a>, <charset_b>,...
```

Accept-Encoding

The `Accept-Encoding` indicates which encoding schemes the requesting client is capable of understanding. Services should support compressed `Accept-Encoding` when large responses are generated and support [gzip](https://en.wikipedia.org/wiki/Gzip) and [deflate](https://en.wikipedia.org/wiki/DEFLATE) when offering compressed payloads.

```
Accept-Encoding: <encoding>
```

Accept-Language

The `Accept-Language` indicates which local language the client is capable of understanding. All services that return language text must support the `Accept-Language` header.

```
Accept-Language: <language>
Accept-Language: <locale>
Accept-Language: <language>-<locale>
```

Authorization

The `Authorization` header specifies the credentials of the client to authenticate with a service. All services must support the `Authorization` header.

```
Authorization: <type> <credentials>
```

If-Match

The `If-Match` header specifies a conditional request to only return the resource when it does match one of the included `ETag` values. All services must support the `If-Match` header provided they support the `ETag`response header.

```
If-Match: <etag_value>
If-Match: <etag_value>, <etag_value>, ...
```

If-None-Match

The `If-None-Match` specifies a conditional request to only return the resource when it doesn't match one of the included `ETag` values. All services must support the `If-None-Match` header provided they support the `ETag`response header.

```
If-None-Match: "<etag_value>"
If-None-Match: "<etag_value>", "<etag_value>", …
If-None-Match: *
```

Prefer
For query responses: when `return=minimal` services must return only the required properties of returned resources as per their defined schema. When `return=representation`, services must return all properties of returned resources as per their defined schema.

For create and update: when `return=minimal` services must return only the `LinkDescriptionObject` of the created/updated resource in the response. When `return=representation`services must return a complete representation of the resource in the response.

Services should implement the `Prefer` header when returning the full representation of a resource would impose performance problems due to bandwidth or latency issues.

```
Prefer: return=minimal
Prefer: return=representation
```

Request-Id

The `Request-Id` specifies the unique identifier of the originating request. This is used to support request tracing and enforce [idempotency](https://en.wikipedia.org/wiki/Idempotence) of create and update requests. Services should implement `Request-Id`.

```
Request-Id: <unique_identifier>
```

Subscription-Id

The `Subscription-Id` specifies the Fortellis Marketplace customer subscription request. All services must support the `Subscription-Id` header.

```
Subscription-Id: <identifier>
```

Response Headers

The following HTTP response headers should be used consistently by the API's. Most of the headers specified follow [IETF 7231](http://tools.ietf.org/html/rfc7231), [IETF 7232](http://tools.ietf.org/html/rfc7232), and the [IANA Header Registry](https://www.iana.org/assignments/message-headers/message-headers.xhtml).

Content-Encoding

The `Subscription-Id` specifies the Fortellis Marketplace customer subscription request. All services must support the `Subscription-Id` header.

```
Content-Encoding: <encoding>
```

Content-Language

The `Content-Language` specifies the local language of returned content. All services that return text content must include the `Content-Language` header in their responses.

```
Content-Language: <language>-<locale>
```

Content-Type

The `Content-Type` specifies the MIME Type of the response. All services must return `Content-Type`.

Content-Language: <language>-<locale>

ETag

The `ETag` provides a unique identifier for a specific version of a resource. This allows downstream service applications to cache resources efficiently. When a resource changes, a new `ETag` value should be generated and included in future responses. All services must include the `ETag` header with a representation of a resource in a response.

ETag: <unique-identifier>

Preference-Applied

The `Preference-Applied` indicates the applied preferences requested using the `Prefer` request header. All services that accept the Prefer request header MUST include the `Preference-Applied` header in their response.

Preference-Applied: <applied-preferences>

Request-Id

The `Request-Id` specifies the unique identifier of the originating request. All services must include the `Request-Id` header with a representation of a resource in a response.

ETag: <unique-identifier>

Response Body - Single Resource

When responding to a query that returns a singular resource, the service must include the `links` property in the response body. `The links` property is an array of the `LinkDescriptionObject` containing hypermedia links describing possible actions given the query response. It must include the `self` relations.</td>

For example, a query response for a single resource `Dog` would be represented in the following response:
<pre><code>
200 OK
{
id: '1',
name: 'Fido',
breed: 'Labrador Retriever',
color: 'Yellow',

```
        links: [
                {
                        href: 'fortellis.io/v1/dogs/1',
                        method: 'GET',
                        rel: 'self',
                        title: 'Fido the dog'
                }
        ]
   }
</pre></code>
### Response Body - Multiple Resources

When responding to a query that returns multiple resources, the service must respond with the following required schema:

 - `items` is an array of the `{ResourceName}` that match the query.
 - `links` is an array of the `LinkDescriptionObject` hypermedia links that describe possible actions given the query response. This must include `self`, `first`, `last`, `next` and `prev` relations.
 - `totalItems` is the number of resources that match the query. Don't assume this value is constant across queries.
 - `totalPages` is the total number of pages which can be returned. Don't assume this value is constant across queries.

For example, a query response for multiple `Dog` resources is represented below:
<pre><code>
   200 OK
   {
        totalItems: 12
        totalPages:     4
        dogs: [
                {
                        id: '1',
                        name: 'Fido',
                        breed: 'Labrador Retriever',
                        color: 'Yellow',
                        links: [
                                {
                                        href: 'fortellis.io/v1/dogs/1',
                                        rel: 'self'
                                }
                        ]
                },
```

```
        {
                id: "2",
                name: 'Bowser',
                breed: 'Great Dane',
                color: 'Brown',
                links: [
                        {
                                href: 'fortellis.io/v1/dogs/2',
                                rel: 'self'
                        }
                ]
        },
        {
                id: '3',
                name: 'Skip',
                breed: 'Corgi',
                color: 'Sable',
                links: [
                        {
                                href: 'fortellis.io/v1/dogs/3',
                                rel: 'self'
                        }
                ]
        }
],
links: [
        {
                href: 'fortellis.io/v1/dogs/1234&page=1;pageSize=3',
                method: 'GET',
                rel: 'self',
                title: 'current page'
        },
        {
                href: 'fortellis.io/v1/dogs/1234&page=1;pageSize=3',
                method: 'GET',
                rel: 'first',
                title: 'first page'
        },
        {
                href: 'fortellis.io/v1/dogs/1234&page=4;pageSize=3',
                method: 'GET',
                rel: 'last',
                title: 'last page'
```

```
                },
                {
                        href: 'fortellis.io/v1/dogs/1234&page=2;pageSize=3',
                        method: 'GET',
                        rel: 'next',
                        title: 'next page'
                },
                {
                        href: 'fortellis.io/v1/dogs/1234&page=1;pageSize=3',
                        method: 'GET',
                        rel: 'prev',
                        title: 'previous page'
                },
        ]
  }
</pre></code>
```

HTTP Header Propagation

Request headers transmit relevant custom API headers as well as HTTP header requests to downstream services.

HTTP Response Status Codes

Fortellis uses service status codes to define response errors and their descriptions. For example, a bad request returns an error code of 400\. The Fortellis API servers issue standard status code conventions and descriptions. Refer to [RFC 7231](https://tools.ietf.org/html/rfc7231#section-6) for details on industry-standard response status codes.

Hypermedia

The following standards outline the Fortellis use of hypermedia.

HATEOAS

We subscribe to [Roy Fielding's definition](https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) of HATEOAS as Hypermedia As The Engine Of Application State. The Fortellis API services are accessible through the use of hypermedia links. The APIs are hypermedia-compliant and permit standard requests such as DELETE, PATCH, and POST. Examples are provided below.

A client makes a POST request creating a new user:

POST https://api.fortellis.io/crm/v1/customer/users

```
{
"givenName": "James",
"surname" : "Greenwood",
...

}
```

The API creates a new user from the input and returns the following links to the client in the response:

```
{
HTTP/1.1 201 CREATED
Content-Type: application/json
...

"links": [
{
"href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
"rel": "self",

},
{
"href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
"rel": "delete",
"method": "DELETE"
},
{
"href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
"rel": "replace",
"method": "PUT"
},
{
"href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
"rel": "edit",
"method": "PATCH"
}
]
}
```

A Fortellis API user can store these links in their database for later use or for an admin to delete one of the users. The API user makes a GET request to the same fixed URI '/users' in the example below:

```
GET https://api.fortellis.io/crm/v1/customer/users
```

The API returns all of the users in the system with respective `self` links in the following response:


```
{
  "totalItems": "166",
  "totalPages": "83",
  "users": [
  {
  "givenName": "James",
  "surname": "Greenwood",
  ...
  "links": [
  {
  "href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
  "rel": "self"
  }
  ]
  },
  {
  "givenName": "David",
  "surname": "Brown",
  ...
  "links": [
  {
  "href": "https://api.fortellis.io/crm/v1/customer/users/ALT-MDFSKFGIFJ86DSF",
  "rel": "self"
  }
  },
  ...
  }
```


To delete the user, the client retrieves the URI of the link relation type `delete` from the database and performs a delete operation on the URI.

Request:

```
GET https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI
```

Response:

```
HTTP/1.1 200 OK
  Content-Type: application/json
  {
  "givenName": "James",
  "surname": "Greenwood",
  ...
  "links": [
  {
  "href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
  "rel": "self",

  },
  {
  "href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
  "rel": "delete",
  "method": "DELETE"
  },
  {
  "href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
  "rel": "replace",
  "method": "PUT"
  },
  {
  "href": "https://api.fortellis.iocrm/v1/customer/users/ALT-JFWXHGUV7VI",
  "rel": "edit",
  "method": "PATCH"
  }

  }
```

Naming Conventions

Fortellis URIs follow the [RFC 3986](http://tools.ietf.org/html/rfc3986) naming specification. Below are examples and a table of naming convention standards:

```
  [scheme"://"][host[':'port]]'/'domain ('/'sub-domain) '/v'major-version '/'namespace '/'resource ('/'resource)* '?' query

        https://api.fortellis.com/sales/deal-creation/v1/quoting
        \___/   \_________________/\_________/\_________/\__/
```

```
          |          |          |          |       |
        scheme     authority       path       query   fragment
```

end-point

```
   [scheme "://"][ host [':' port]]
   api.fortellis.io/v1/appointments
```

domain

```
   fortellis.io
```

expressions

```
   [scheme "://"][ host [':' port]]
   "{" [ operator ] variable-set "}"
```

literals

```
   ( - )
   2019-10-11T09:30:00+00:00
```

name

```
   Alpha (Alpha | Digit | '_')*
   accountId
```

query

```
   name '=' value ('&' name = value)*
   name: "name.firstName"
```

resource

```
   resource-name ['/' resource-id]
   name: "name.firstName"
```

resource-id

```
   value
   operationId: queryDealershipSettings
```

resource-path

"/v" version '/' namespace-name '/' resource ('/' resource)
api.fortellis.io/v1/appointments

scheme

http or https
https://api.fortellis.io

sub-domain

URL pattern
api.example.io

URI

[end-point] '/' resource-path ['?'query]
api.fortellis.io/sales/deal-creation

value

URI-percent encoded value
percent sign %

</table>

Usage Rules

* Asterisks * specify zero or occurrences.
* Brackets [ ] define optional fields.
* Double quotes** " "** denote strings.
* Parenthesis ( ) group resources.
* Single quotes ' **'** identify special characters.

Resource Names

Resource names should be lowercase and use only alphanumeric characters and hyphens (-) used as word separators in URI path literals. It's important to note this is the only instance hyphens are used as word separators. For other uses, the underscore character (_) is used.

Query Parameter Names

Query parameter values are percent-encoded. Query parameters start with a letter and must be lower case. Lower case alpha characters, digits and underscores (_) are permitted.

Query Parameter Headers

Some query parameter headers for cross-domain calls may not be supported. The criteria when to accept headers as parameters are below:

1. Custom headers are accepted as parameters.
2. Required headers are accepted as parameters.
3. Required headers such as the authorization header may not be appropriate as a parameter.
4. The Accept header is the exception to the rule. Commonly, it's best to use simple names.

Field Names

The Fortellis APIs follow the [JSON standard](http://json.org/) to handle field names. The values can be numbers, arrays, objects, booleans, and strings. Below are sample field names and their descriptions:

1. Key names are camelCase words separated by underscores (_) such as bookingSessionId or foo_barBaz.
2. For boolean type keys, do not use prefixes like `is` or `has` (use closedRO, not ClosedRO).
3. Array field representations are named using plural nouns (for example, priceFormulas).
4. The suffix 'Id' is used to identify the main resource of a domain (for example, vehicleId or customerId).
5. The suffix 'Code' is used to indicate the field is a lookup code in the system (for example, titleCode or modelCode).

Enum Names

Enum names use only uppercase alphanumeric characters and an underscore (_). Two examples are below:

'FIELD_14'

NOT_EQUAL'

Link Relation Names

Link relation types are represented by `rel` and must be created using lowercase characters. See the following example:

```
"links": [
{
"href": "href":
"https://api.fortellis.io/crm/v1/customer/partner-referrals/ALT-JFWXHGUV7VI/activate",
"rel": "activate",
"method": "POST"

}
]
```

File Names

The JSON Schema types are located in separate files. They're referenced using the `$ref` syntax (for example:`"$ref":"object.json"`) and follow underscore naming syntax such as `transaction_history.json`.

Casing

The HTTP headers are in camelCase and hyphenated (-) syntax. Payload properties should be camelCase; for example, `vehicleSpecId`. Domains and subdomains in the URI should be list case such as `merchandisable-vehicles`.

JSON Primitive Types

The [RFC draft](https://tools.ietf.org/html/draft-zyp-json-schema-04) is the JSON Schema standard Fortellis uses to define all of the fields in our APIs. Below are specific uses of the JSON primitive types.

Strings

The use of strings is defined with `minLength` and `maxLength`. The reason strings should contain a maxLength is to maintain database columns for backward compatibility.

The one caveat when not to use the `maxLength` string is with an undefined string length from upstream resources. Use good judgement when naming your strings. Fewer characters is a

recommended. Your strings should also use consistent pattern properties when defining enumerated values and numbers.

Enumeration

Avoid creating new values to an enum in your APIs which provide a service response as it may conflict with backward compatibility. Often, it's due to a client rejecting a response as it tries to return values from a previous version.

Therefore, avoid new enum values at all costs and adhere to the following recommendations using an enum with the JSON type string:

1. Only use an enum when values will remain fixed and never change.
2. Instead of using the keyword enum in arrays, use a string type and include acceptable values in your documentation.
3. When using a string type expressing enum values, use strict naming conventions by using a pattern field.
4. For pre-existing database columns, set the maxLength to 255 and minLength to 1 to prevent clients sending empty string values.

Below is a JSON example which enforces naming conventions, length constraints and pattern field:

```
  {
    "type": "string",
    "minLength": 1,
    "maxLength": 255,
    "pattern": "^[0-9A-Z_]+$",
    "description": "Field description."
  }
```

Number Types

JSON defines number types as fixed-point values for numbers and integers. These types are unbounded except if the schema requires minimum and maximum values. To maintain compatibility, we recommend the following conventions:

* Use a string to define decimal values.
* Represent integer types with minimum and maximum values.
* Define your integer type values as 32-bit integers (between ((2^31) - 1) and -(2^31).
* Avoid JSON Schema number types since some languages convert number types as fixed-point or floating-point values.

Example

```
{
   "type": "integer",
   "minimum": 0,
   "maximum": 2147483647
}
```

When using a string type to represent a number, use `minLength` and `maxLength` and constrain the definition of the string by using number patterns. The example below uses positive integers and zero with a maxLength of 6:

```
{
   "type": "string",
   "pattern": "^[0-9]+$",
   "minLength": 1,
   "maxLength": 6
}
```

The following is a representation of fixed-point decimal values (positive or negative) and a maxLength of 32:

```
{
   "type": "string",
   "pattern": "^(-?[0-9]+|-?([0-9]+)?[.][0-9]+)$"
   "maxLength": 32,
   "minLength": 1,
}
```

Array

A JSON array is unbounded. Most programming languages require a maximum limit of the size of arrays. We recommend APIs maintain cross-compatibility for all languages.

Your `maxItems` should always be fully defined. However, `maxItems` should be constrained to a 16-bit signed integer. Design your APIs for scale rather than merely for the scope of your current build. `MinItems` must be defined with values of 0 or 1.

Null

A JSON property can only be null if it's defined by the schema and represented by the type keyword {"type": "null"}. Avoid using composition keywords.

For example, `anyOf` and `oneOf` permit multiple types and return invalid data. Keep in mind that a missing JSON property is undefined (excluding null). We strongly recommend you avoid JSON null to maintain cross-language compatibility.

Common Types

Below are the Fortellis common types:

Type	Recommended Use	Notes
% interest rate	`percentage.json`	Represents a fixed decimal point. Example: 16.99% must be returned by the API as 16.99.
Address	`address_portable.json`	Offers backward compatibility, supports [i18n](https://tools.ietf.org/html/draft-zyp-json-schema-03) and W3 HTML5.1's autofill fields.
Block name	`block_name`	The name of the residence block.
City	`city`	The name of the city.
County	`county`	The geographic region of the city.
Country code	`country_code`	Your API must use the [ISO 3166-1](https://www.iso.org/iso-3166-country-codes.html) two-letter country code standard.
Currency code	`currency_code`	Use three letter currency codes as defined in [ISO 4217](https://www.currency-iso.org/en/home.html).
Date no time	`date_no_time.json`	Represents full date. Refer to [RFC 3339](https://tools.ietf.org/html/rfc3339) for details.
Date time	`date_time.json`	The name of the city.
Date year month	`date_year_month.json`	Defines year and month such as 2018-04.
Door number	`door_number`	Expresses date and time. Refer to [RFC 3339](https://tools.ietf.org/html/rfc3339) for details.
Email address	`local-part@domain.xx`	The user email address. Refer to [RFC 5322](https://tools.ietf.org/html/rfc5322#section-3.4.1).
Floor number	`floor_number`	The apartment or condominium floor number.
Geographical location		Refer to [RFC 4119](https://tools.ietf.org/html/rfc4119).
House number	`house_number`	The number of residence.
Internationalization		Represents the country, currency & language.
Language	`language.json`	Use [BCP-47](https://tools.ietf.org/html/bcp47) language tag.
Phone number	`+6189761234`	Apply country code with preceding +. Refer to [RFC 2916](https://tools.ietf.org/html/rfc2916).
Postal code	`postal_code`	The five-digit postal code of the city.
Postal code	`postal_code`	The five-digit postal code of the city.
Province	`province`	The name of the province or state.
Street name	`street_name`	The name of the street.
Suburb	`suburb`	The region of the city.
Time no date	`time_nodate.json`	Expresses the time. Refer to [RFC 3339](https://tools.ietf.org/html/rfc3339) for details.

| Time zone | `time_zone.json` | Indicates the region time zone. All APIs must use [UTC](https://en.wikipedia.org/wiki/Coordinated_Universal_Time). Refer to [RFC 3339](https://tools.ietf.org/html/rfc3339) for details. |
| Units of measurement | | Refer to [RFC 2916.](https://tools.ietf.org/html/rfc2916) |

Link Description Object

Fortellis uses the Link Description Object (LDO) to describe actions relative to a given resource as a hypermedia link. We use the [IETF 5988](https://tools.ietf.org/html/rfc5988) standard. The LDO properties interacting with our APIs are described below:

href

The URL of the linked resource (required). The incoming value of the host header should be used for the host field. For example: api.fortellis.io.

method

The method property (optional) defines the HTTP verb and is required to create a request to target a resource link. In cases when the method property is omitted, a `GET` value is used.</td>

rel

The Link Relation Type describes how the current context (source) is related to the target resource and is required. Read more about the `rel` [property](https://www.iana.org/assignments/link-relations/link-relations.xhtml).</td>

title

The title property is a human readable name of the link (optional).</td>

Below is an example of a `Link Description Object` that refers to a single resource:

```
{
   href: 'fortellis.io/v1/dogs/1234'
   type: 'GET'
   rel: 'self'
   title: 'Fido the dog'
}
```

Link Relation Types

A Link Relation Type (LRT) is an identifier for a resource link. The Fortellis APIs provide LRTs which are unambiguous and clearly describe the semantics of a link and its representation.

We base our API Design Guidelines on the [RFC 5988](https://tools.ietf.org/html/rfc5988#section-4) model and the [IANA](https://www.iana.org/assignments/link-relations/link-relations.xhtml) standard. Controller style operations must be passed with the 'action' name for the link relation type. For example, 'activate' or 'cancel'. The following table describes common LRTs and their descriptions:

LRT	Description
collection	A collection of resources. For example: /common/v1/users.
create	A link used to create a resource.
delete	Deletes a link resource used as an extended operation.
edit	Edits or makes a partial update to a link. It's used to represent the `PATCH` operation link.
first	The first page list result.
item	The target Internationalized Resource Identifier (IRI) pointing to a resource belonging to a collection represented by the context IRI.
last	The last page list result when `total_required` is include in the query parameter.
latest-version	Targets the latest version of a resource.
next	The next page list result.
previous	The previous page list result.
replace	Updates or replaces a link.
search	Searches a link's resources.
start	Identifies the first resource within a resource collection.
self	The link's identifier pointing to a resource.
up	A parent resource based on the architectural hierarchy.

Avoid using HTTP link and location headers for LDOs. Instead, we advise returning LDOs in the HTTP response body. The reason is that the HTTP header is a point-to-point connection between the client and service.

Some responses may require interaction with other service layers and won't relay header data to these services. For more details on when to avoid using HTTP headers for LDOs refer to the [JSON Hyper-Schema Release Notes (draft 4 section)](http://json-schema.org/draft-07/json-hyper-schema-release-notes.html).

Error Handling

We follow standard HTTP specification error codes and descriptions. For example, codes returned with values in the 400s are client-side errors.

Codes in the 500s return server-side errors. Sometimes users may need additional guidance with error code definitions and causes. Document your error codes with clarity so users can understand error responses. When building your APIs, they must return JSON error codes adhering to the `error.json` schema.

Error Schema

Good API design must include the values below:

Value	Description
debug_id	A server-side error defined by a unique identifier.
details	A summary of the field error, value, the reason and the location of the error (for example, request, path or body).
links	HATEOAS links to help documentation about the error and how to resolve.
message	A message defining the error and how to fix it. We recommend you build an error catalog for users to refer to when searching for answers.
name	The name of the error. Be smart by integrating an `error_spec.json#name` catalog into your API so errors can be retrieved and delivered to users.

Sample Errors

The following is a sample VALIDATION_ERROR in one field. A `400 Bad Request` HTTP status code is returned:


```
{
"name":"VALIDATION_ERROR",
"details":[
{
"field":"customerId",
"issue":"Required field is missing",
"location":"body"
}
],
"debugId":"d1e6bbbc4b30e493",
"message":"customerId was not provided",
"information_link":"http://developer.fortellis.io/errors/validation-error"
}
```

Below is a `VALIDATION_ERROR` in two fields. The `details` variable is an array which lists all instances in the error:

```
{
"name": "VALIDATION_ERROR",
"details": [
{
"field":"customerId",
"issue":"customerId is required",
"location":"body"
},
{
"field": "vehicleId",
"issue": "vehicleId is required",
"location": "body"
}
],
"debugId": "53a75d4c6c3249e2",
"message": "Invalid data provided",
"information_link": "http://developer.fortellis.io/errors/validation-error"
}
```

When a request requires interaction with the Fortellis API the HTTP status code of `422 Unprocessable Entity` is returned in the following example:

```
{
"name": "BALANCE_ERROR",
"debug_id": "123456789",
"message": "The account balance is too low. Add balance to your account to proceed.",
"information_link": "http://developer.fortellis.io/errors/validation-error"
}
```

Filtering

Use field filtering as a query parameter by entering name values to the top-level attribute of a domain resource. Below is an example filtering a payment calculation by Ids:

```
GET /payments/paymentsCalculationId

{
"paymentsCalculationId": [

{
```

"5e7c55bf-8fd5-4872-8cff-e5043b0dc348": "5e7c55bf-8fd5-4872-8cff-e5043b0dc360",
},
]
}

When filtering multiple values, the only objects returned are ones that meet the filter criteria.

Sorting

Sorting keys are used to sort a set of query string parameters. The sort direction value for ascend is `asc`. The descend value is `desc`. The default sort direction is managed by the server specification. In the table below are acceptable sorting values:

- `sort=key1,key2`: key1 is the first value; key2 is used as the second key.
- `sort=key1:asc,key2:desc`: sort=key1:asc is the ascending direction; key2:desc is descending.</td>

Pagination

Fortellis recommends using page/pageSize pagination queries since it integrates well with Apps using SQL databases. There's two query parameters that must be included when querying a collection:

page
* A non-negative and non-zero integer that indicates which page of the query results should be returned.
* It must be optional.
* It must have a default value of 1.
* If a client provides an invalid value (e.g. negative), the response must be a `400 Bad Request`.
* If the page value exceeds the total number of possible pages given the number of query results, the resource server must respond with a `200 OK` status code and an empty result set.

pageSize
* A non-negative, non-zero integer of the maximum number of results in the response.
* It must be optional.
* It must have a default value.
* If a client provides an invalid value (e.g. negative), then the response MUST be a `400 Bad Request` status code.

HTTP Status Codes

We rely on REST service status codes to define errors and their descriptions. For example, a bad request returns an error code of 400\. The Fortellis API servers issue standard status code conventions and descriptions. Refer to [RFC 7231](http://tools.ietf.org/html/rfc7231#section-6) for more guidance on HTTP status codes.

Links Array

JSON schemas use the links array property to identify their associated Link Description Objects. The links array must be included in the API resource schema URI template. The links array URI template should be declared outside of the properties keyword to support code generator setter/getter methods of the links array generated object resource.

The following example shows a links array schema:

```
{
"type": "object",
"$schema": "http://json-schema.org/draft-04/hyper-schema#",
"description": "A sample resource representing a customer name.",
"properties": {
"id": {
"type": "string",
"description": "Unique ID to identify a customer."
},
"firstName": {
"type": "string",
"description": "Customer's first name."
},
"lastName": {
"type": "string",
"description": "Customer's last name."
},
"links": {
"type": "array",
"items": {
"$ref": "http://json-schema.org/draft-04/hyper-schema#definitions/linkDescription"
}
}
},
"links": [
{
"href": "https://api.fortellis.io/crm/v1/customer/users/{id}",
"rel": "self"
},
```

```
{
"href": "https://api.fortellis.io/crm/v1/customer/users/{id}",
"rel": "delete",
"method": "DELETE"
},
{
"href": "https://api.fortellis.io/crm/v1/customer/users/{id}",
"rel": "replace",
"method": "PUT"
},
{
"href": "https://api.fortellis.io/crm/v1/customer/users/{id}",
"rel": "edit",
"method": "PATCH"
}
]
}
```

The following example shows a response that is compliant with the above schema:

```
{
"id": "ALT-JFWXHGUV7VI",
"firstName": "John",
"lastName": "Doe",
"links": [
{
"href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
"rel": "self"
},
{
"href": "https://api.fortellis.io/crm/v1/customer/users/ALT-JFWXHGUV7VI",
"rel": "edit",
"method": "PATCH"
}
]
}
```

URI

The example below shows the resource URI format used by the Fortellis API endpoints:

```
https://api.fortellis.io/service/v1/appointments/75d566bbba1b
```

The following table defines sample URI paths:

Path	Description
75d566bbba1b	The resource ID.
appointments	The domain resource.
service	The namespace resource.
v1	The API version.

Sub-Resources

The sub-resource name defines the relationship between itself and its parent resource. If cardinality is 1:1 then no additional information is required. Only two levels of sub-resources are supported. See the examples below:

The following `GET` returns all of the associated items of appointment 75d566bbba1b.

```
GET https://api.fortellis.io/service/v1/appointments/75d566bbba1b/items
```

The following `GET` returns all of the associated parts of the repair order and details (part ID is not required):

```
GET https://api.fortellis.io/service/v1/repair-orders/ABCD1234/parts
```

When part IDs are required the following path is used:

```
/service/v1/parts/ABCD1234
```

Resource Identifiers

The Fortellis APIs conform to the following conventions:

1. Resource identifiers are owned by a resource domain.
2. Database sequence numbers are not permitted to be used as a resource identifier.
3. We follow the [RFC 4122](https://tools.ietf.org/html/rfc4122) standards for universal unique identifiers.
4. Sub-resource IDs are scoped within the parent resource to support security and data access.
5. Enumeration values (including string representations) are used as sub-resource IDs.
6. No two resource identifiers may be next to one another in a single resource path.
7. Resource identifiers use Resource Identifier Characters or ASCII characters.
8. [UTF-8](https://tools.ietf.org/html/rfc3629) characters are not permitted in resources identifiers unless they're encoded.

9. Query parameters and resource identifiers must use the URI-percent-encoding [RFC 3986](http://tools.ietf.org/html/rfc3986) standard for syntax (except URI unreserved).

Query Parameters

The Fortellis APIs follow the [RFC 3986](http://tools.ietf.org/html/rfc3986) query parameter standards. Below are summary points of these conventions:

1. Query parameters are used to restrict the resource collection and to search or filter specific criteria.
2. At a minimum, the following query parameters should be used consistently where applicable for resource collection queries: `page` or `pageSize`.
3. Pagination parameters use the pagination syntax.
4. Default sort order is undefined and non-deterministic. Explicit query parameter sort orders use the following syntax: `{fieldName}|{asc|desc}`.
5. Single-source query parameters are not supported.
6. When the need arises for highly cacheable query parameters, avoid using `POST` in your request body. `GET` is preferred.
7. Query parameters are not ideal when making `POST` operations.