1. Introduction

2. Definition of Terms
   2.1. Data types

3. General API Requirements and Conventions
   3.1. Base URL
   3.2. URL Encoding
   3.3. Responses
      3.3.1. Response Format
      3.3.2. JSON Response Schema: Common Fields
      3.3.3. HTTP Response Status Codes
      3.3.4. HTTP Response Headers
      3.3.5. Properties with unknown value
      3.3.6. Warnings
   3.4. Index Meta-Database
   3.5. Relationships

4. API endpoints
   4.1. Entry Listing Endpoints
      4.1.1. URL Query Parameters
      4.1.2. JSON Response Schema
   4.2. Single Entry Endpoints
      4.2.1. URL Query Parameters
      4.2.2. JSON Response Schema
   4.3. Info Endpoints
      4.3.1. Base URL Info Endpoint
      4.3.2. Entry Listing Info Endpoints
   4.4. Links Endpoint
      4.4.1. JSON Response Schema
      4.4.2. Parent and Child Objects
      4.4.3. Provider Objects
      4.4.4. Index Meta-Database Links Endpoint
   4.5. Custom Extension Endpoints

5. API Filtering Format Specification
   5.1. Lexical Tokens
   5.2. The Filter Language Syntax

6. Entry List
   6.1. Properties Used by Multiple Entry Types
      6.1.1. id
      6.1.2. type
      6.1.3. immutable_id
      6.1.4. last_modified
      6.1.5. database-provider-specific properties
   6.2. Structures Entries
      6.2.1. elements
      6.2.2. nelements
      6.2.3. elements_ratios
      6.2.4. chemical_formula_descriptive
1. Introduction

As researchers create independent materials databases, much can be gained from retrieving data from multiple databases. However, automating the retrieval of data is difficult if each database has a different application programming interface (API). This document specifies a standard API for retrieving data from materials databases. This API specification has been developed over a series of workshops entitled "Open Databases Integration for Materials Design", held at the Lorentz Center in Leiden, Netherlands and the CECAM headquarters in Lausanne, Switzerland.

The API specification described in this document builds on top of the JSON API v1.0 specification. In particular, the JSON API specification is assumed to apply wherever it is stricter than what is formulated in this document. Exceptions to this rule are stated explicitly (e.g. non-compliant responses are tolerated if a non-standard response format is explicitly requested).

2. Definition of Terms

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

- **Database provider**: A service that provides one or more databases with data desired to be made available using the OPTiMaDe API.
- **Database-provider-specific prefix**: Every database provider is designated a unique prefix. The prefix is used to separate the namespaces used by provider-specific extensions. These are defined in Appendix 1.
- **API implementation**: A realization of the OPTiMaDe API that a database provider uses to serve data from one or more databases.
- **Identifier**: Names that MUST start with a lowercase letter ([a-z]) or an underscore ("_") followed by any number of lowercase alphanumerics ([a-z0-9]) and underscores ("_").
- **Entry**: A single instance of a specific type of resource served by the API implementation. For example, structures entry is comprised by data that pertain to a single structure.
- **Entry type**: Entries are categorized into types, e.g., structures, calculations, references. Entry types MUST be named according to the rules for identifiers.
- **Entry property**: One data item which pertains to an entry, e.g., the chemical formula of a structure.
**Entry property name**: The name of an entry property. Entry property names MUST follow the rules for identifiers and MUST NOT have the same name as any of the entry types.

**Relationship**: Any entry can have one or more relationships with other entries. These are described in [3.5. Relationships]. Relationships describe links between entries rather than data that pertain to a single entry, and are thus regarded as distinct from the entry properties.

**Queryable property**: An entry property that can be referred to in the filtering of results. See section [5. API Filtering Format Specification] for more information on formulating filters on properties. The definitions of specific properties in [6. Entry List] states which ones MUST be queryable and which are RECOMMENDED.

**ID**: The ID entry property is a unique string referencing a specific entry in the database. The following constraints and conventions apply to IDs:
- Taken together, the ID and entry type MUST uniquely identify the entry.
- Reasonably short IDs are encouraged and SHOULD NOT be longer than 255 characters.
- IDs MAY change over time.

**Immutable ID**: A unique string that specifies a specific resource in a database. The string MUST NOT change over time.

**Response format**: The data format for the HTTP response, which can be selected using the `response_format` URL query parameter. For more info, see [3.3.1. Response Format].

**Field**: The key used in response formats that return data in associative-array-type data structures. This is particularly relevant for the default JSON-based response format. In this case, **field** refers to the name part of the name-value pairs of JSON objects.

### 2.1. Data types

An API implementation handles data types and their representations in three different contexts:

- In the HTTP URL query filter, see [5. API Filtering Format Specification].
- In the HTTP response. The default response format is JSON-based and thus uses JSON data types. However, other response formats may use different data types. For more info, see [3.3. Responses].
- The underlying database backend(s) from which the implementation serves data.

Hence, entry properties are described in this proposal using context-independent types that are assumed to have some form of representation in all contexts. They are as follows:

- **Basic types**: `string`, `integer`, `float`, `boolean`, `timestamp`.
- **list**: an ordered collection of items, where all items are of the same type, unless they are unknown. A list can be empty, i.e., contain no items.
- **dictionary**: an associative array of `keys` and `values`, where `keys` are pre-determined strings, i.e., for the same entry property, the `keys` remain the same among different entries whereas the `values` change. The `values` of a dictionary may be any basic type, list, dictionary, or unknown.

An entry property value that is not present in the database is **unknown**. This is equivalently expressed by the statement that the value of that entry property is `null`. For more information see [3.3.5. Properties with unknown value].

The definition of a property of an entry type specifies a type. The value of that property MUST either have a value of that type, or be unknown.

### 3. General API Requirements and Conventions

#### 3.1. Base URL

Each database provider will publish a base URL that serves the API. An example could be `http://example.com/optimade/`. Every URL component that follows the base URL MUST behave as standardized in this API specification.

The client MAY include a version number in the base URL, prefixed with the letter "v", where the version number indicates the
version of the API standard that the client requests. The format is either vMAJOR or vMAJOR.MINOR where MAJOR is the major version number, and MINOR is the minor version number of the standard being referenced. If the major version is 0, the minor version MUST also be included. The database provider MAY support further levels of versioning separated from the major and minor version by a decimal point, e.g., patch version on the format vMAJOR.MINOR.PATCH. However, the client MUST NOT assume levels beyond the minor version are supported.

If the client does not include a version number in the base URL, the request is for the latest version of this standard that the database provider implements. A query that includes a major and/or minor version is for the latest subversion of that major and/or minor version that the database provider implements.

A database provider MAY choose to only support a subset of possible versions. The client can find out which versions are supported using the `available_api_versions` field of the `attributes` field from a query to the base URL `info` endpoint (see section 4.3.1. Base URL Info Endpoint). The database provider SHOULD strive to implement the latest subversion of any major and minor version supported. Specifically, the latest version of this standard SHOULD be supported.

Examples of valid base URLs:

- http://example.com/optimade/
- http://example.com/optimade/v0.9/
- http://example.com/
- http://example.com/some/path/

Examples of invalid base URLs:

- http://example.com/optimade/v0/
- http://example.com/optimade/0.9/

### 3.2. URL Encoding

Clients SHOULD encode URLs according to RFC 3986. API implementations MUST decode URLs according to RFC 3986.

### 3.3. Responses

#### 3.3.1. Response Format

This document defines a JSON response format that complies with the JSON API v1.0 specification. All endpoints of an API implementation MUST be able to provide responses in the JSON format specified below and MUST respond in this format by default.

Each endpoint MAY support additional formats, and SHOULD declare these formats under `/info/entry_type` (see section 4.3.2. Entry Listing Info Endpoints). Clients can request these formats using the `response_format` URL query parameter. Specifying a `response_format` different from `json` (e.g. `response_format=xml`) allows the API to break conformance not only with the JSON response format specification, but also, e.g., in terms of how content negotiation is implemented.

Database-provider-specific `response_format` identifiers MUST include a database-provider-specific prefix as defined in Appendix 1.

#### 3.3.2. JSON Response Schema: Common Fields

In the JSON response format, property types translate as follows:

- **string**, **boolean**, **list** are represented by their similarly named counterparts in JSON.
- **integer**, **float** are represented as the JSON number type.
- **timestamp** uses a string representation of date and time as defined in RFC 3339 Internet Date/Time Format
- **dictionary** is represented by the JSON object type.
- **unknown** properties are represented by either omitting the property or by a JSON `null` value.
Every response SHOULD contain the following fields, and MUST contain at least one:

- **meta**: a JSON API meta member that contains JSON API meta objects of non-standard meta-information. It MUST be a dictionary with these fields:
  - **query**: information on the query that was requested. It MUST be a dictionary with these fields:
  - **representation**: a string with the part of the URL following the base URL.
  - **api_version**: a string containing the version of the API implementation.
  - **time_stamp**: a timestamp containing the date and time at which the query was executed.
  - **data_returned**: an integer containing the number of data objects returned for the query.
  - **more_data_available**: false if all data for this query has been returned, and true if not.
  - **provider**: information on the database provider of the implementation. It MUST be a dictionary with these fields:
    - **name**: a short name for the database provider.
    - **description**: a longer description of the database provider.
    - **prefix**: database-provider-specific prefix as found in Appendix 1.

Provider MAY include these fields:

- **homepage**: a JSON API links object, pointing to the homepage of the database provider, either directly as a string, or as a link object which can contain the following fields:
  - **href**: a string containing the homepage URL.
  - **meta**: a meta object containing non-standard meta-information about the database provider's homepage.

- **index_base_url**: a JSON API links object pointing to the base URL for the index meta-database of the provider as specified in Appendix 1, either directly as a string, or as a link object which can contain the following fields:
  - **href**: a string containing the base URL for the database provider's index meta-database.
  - **meta**: a meta object containing non-standard meta-information about this link.

If the index meta-database (see section 3.4. Index Meta-Database) is implemented by the provider, the index_base_url field MUST be included.

Meta MAY also include these fields:

- **data_available**: an integer containing the total number of data objects available in the database.
- **last_id**: a string containing the last ID returned.
- **response_message**: response string from the server.
- **implementation**: a dictionary describing the server implementation, containing the OPTIONAL fields:
  - **name**: name of the implementation.
  - **version**: version string of the current implementation.
  - **source_url**: URL of the implementation source, either downloadable archive or version control system.
  - **maintainer**: a dictionary providing details about the maintainer of the implementation, MUST contain the single field email with the maintainer's email address.
- **warnings**: a list of warning resource objects representing non-critical errors or warnings. A warning resource object is defined similarly to a JSON API error object, but MUST also include the field type, which MUST have the value "warning". The field detail MUST be present and SHOULD contain a non-critical message, e.g., reporting unrecognized search attributes or deprecated features. The field status, representing a HTTP response status code, MUST NOT be present for a warning resource object. This is an exclusive field for error resource objects.

Example:
For a deprecation warning

```
{
  "id": "dep_chemical_formula_01",
  "type": "warning",
  "code": "_exmpl_dep_chemical_formula",
  "title": "Deprecation Warning",
  "detail": "chemical_formula is deprecated, use instead chemical_formula_hill"
}
```

**Note:** ids MUST NOT be trusted to identify the exceptional situations (i.e., they are not error codes, use instead the field code for this. ids can only be trusted to be unique in the list of warning resource objects, i.e., together with the type. General OPTiMaDe warning codes are specified in 3.3.5. Warnings.

- Other OPTIONAL additional information global to the query that is not specified in this document, MUST start with a database-provider-specific prefix as defined in Appendix 1.

- Example:
  For a request made to http://example.com/optimade/v0.9/structures/?filter=a=1 AND b=2

```
{
  "meta": {
    "query": {
      "representation": "/structures/?filter=a=1 AND b=2",
    },
    "api_version": "v0.9",
    "time_stamp": "2007-04-05T14:30Z",
    "data_returned": 10,
    "data_available": 10,
    "more_data_available": false,
    "provider": {
      "name": "Example provider",
      "description": "Provider used for examples, not to be assigned to a real database",
      "prefix": "exmpl",
      "homepage": "http://example.com"
    },
    "implementation": {
      "name": "exmpl-optimade",
      "version": "0.1.0",
      "source_url": "http://git.example.com/exmpl-optimade",
      "maintainer": {
        "email": "admin@example.com"
      }
    }
  }
  // ...
}
```

- **data:** The schema of this value varies by endpoint, it can be either a single JSON API resource object or a list of JSON API resource objects. Every resource object needs the type and id fields, and its attributes (described in section 4. API Endpoints) need to be in a dictionary corresponding to the attributes field.

The response MAY also return resources related to the primary data in the field:

- **links:** JSON API links is MANDATORY for implementing pagination. (see section 4.1.1 URL Query Parameters page_*)
  Each field of a links object, i.e. a “link”, must be either
null
- a string representing a URI, or
- a dictionary ("link object") with fields
  - href: a string representing a URI
  - meta: (OPTIONAL) a meta object containing non-standard meta-information about the link

Example links objects:

- **base_url**: a links object representing the base URL of the implementation. Example:

  ```json
  {
    "links": {
      "base_url": {
        "href": "http://example.com/optimade/v0.9/",
        "meta": {
          "_exmpl_db_version": "3.2.1"
        }
      }
    // ...
  }
  // ...
  }
  ```

The following fields are REQUIRED for implementing pagination:

- **next**: represents a link to fetch the next set of results. When the current response is the last page of data, this field MUST be either omitted or null-valued.

The following fields are reserved for pagination. Their values are as with `next`, in the sense that they should be a "link". An implementation MAY offer these links:

- **prev**: the previous page of data. null or omitted when the current response is the first page of data.
- **last**: the last page of data.
- **first**: the first page of data.

- **included**: a list of JSON API resource objects related to the primary data contained in `data`. Responses that contain related resources under `included` are known as compound documents in the JSON API.

If there were errors in producing the response all other fields MAY be present, but the top-level `data` field MUST be skipped, and the following field MUST be present:

- **errors**: a list of JSON API error objects, where the field `detail` MUST be present. All other fields are OPTIONAL.

An example of a full response:
3.3.3. HTTP Response Status Codes

All HTTP response status codes MUST conform to [RFC 7231: HTTP Semantics](https://tools.ietf.org/html/rfc7231). The code registry is maintained by IANA and can be found [here](https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml).

See also the JSON API definitions of responses when fetching data, i.e., sending a GET request.

**Important**: If a client receives an unexpected 404 error when making a query to a base URL, and is aware of the index meta-database that belongs to the database provider (as described in 3.4. Index Meta-Database), the next course of action SHOULD be to fetch the resource objects under the `links` endpoint of the index meta-database and redirect the original query to the corresponding database ID that was originally queried, using the object’s `base_url` value.
3.3.4. HTTP Response Headers

There are relevant use-cases for allowing data served via OPTiMaDe to be accessed from in-browser JavaScript, e.g. to enable server-less data aggregation. For such use, many browsers need the server to include the header `Access-Control-Allow-Origin: *` in its responses, which indicates that in-browser JavaScript access is allowed from any site.

3.3.5. Properties with unknown value

Many databases allow specific data values to exist for some of the entries, whereas for others, no data value is present. This is referred to as the property having an `unknown` value, or equivalently, that the property value is `null`.

The text in this section describes how the API handles properties with the value `null`. The use of `null` values inside nested property values (such as, e.g., lists or dictionaries) are described in the definitions of those data structures elsewhere in the specification, see section 6. Entry List. For these properties, `null` MAY carry a special meaning.

REQUIRED properties with an unknown value MUST be returned in the response, unless explicitly left out (e.g., by using `response_fields`, see section 4.4.1. URL Query Parameters).

OPTIONAL properties with an unknown value MAY be returned in the response. If an OPTIONAL property is not returned in a full response (i.e., not using `response_fields`), the client MUST assume the property has an unknown value, i.e., `null`.

If a property is explicitly requested in a search query without value range filters, then all entries otherwise satisfying the query SHOULD be returned, including those with `null` values for this property. These properties MUST be set to `null` in the response.

Filters with `IS UNKNOWN` and `IS KNOWN` can be used to match entries with values that are, or are not, unknown for some property, respectively. This is discussed in section 5.2. The Filter Language Syntax.

3.3.6. Warnings

Non-critical exceptional situations occurring in the implementation SHOULD be reported to the referrer as warnings. Warnings MUST be expressed as a human-readable message, OPTIONALLY coupled with a warning code.

Warning codes starting with an alphanumeric character are reserved for general OPTiMaDe error codes (currently, none are specified). For implementation-specific warnings, they MUST be start with `_` and the database-provider-specific prefix as defined in Appendix 1.

3.4. Index Meta-Database

The main purpose of this "index" is to allow for automatic discoverability of all databases of a given provider. Thus, it acts as a meta-database for the database provider’s implementation(s).

The index meta-database MUST only provide the `info` and `links` endpoints, see sections 4.3. Info Endpoints and 4.4. Links Endpoint. It MUST not expose any entry listing endpoints (e.g., `structures`).

These endpoints do not need to be queryable, i.e., they MAY be provided as static JSON files. However, they MUST return the correct and updated information on all currently provided implementations.

The `index_base_url` field MUST be included in every response in the `provider` field under the top-level `meta` field (see section 3.3.2. JSON Response Schema: Common Fields).

The `is_index` field under `attributes`, as well as the `relationships` field, MUST be included in the `info` endpoint for the index meta-database (see section 4.3.1. Base URL Info Endpoint). The value for `is_index` MUST be `true`.

Note: A list of database providers acknowledged by the Open Databases Integration for Materials Design consortium can be found in Appendix 1. This list is also machine-readable, optimizing the automatic discoverability.

3.5. Relationships
The API implementation MAY describe many-to-many relationships between entries along with OPTIONAL human-readable descriptions that describe each relationship. These relationships can be to the same, or to different, entry types. Response formats have to encode these relationships in ways appropriate for each format.

In the default response format, relationships are encoded as JSON API Relationships, see 4.1.2. JSON API Response Schema.

For implementers: For database-specific response formats without a dedicated mechanism to indicate relationships, it is suggested that they are encoded alongside the entry properties. For each entry type, the relationships with entries of that type can then be encoded in a field with the name of the entry type, which are to contain a list of the IDs of the referenced entries alongside the respective human-readable description of the relationships. It is the intent that future versions of this standard uphold the viability of this encoding by not standardizing property names that overlap with the entry type names.

4. API Endpoints

The URL component that follows the base URL MUST represent one of the following endpoints:

- an "entry listing" endpoint
- a "single entry" endpoint
- an introspection info endpoint
- an 'entry listing' introspection info endpoint
- a links endpoint to discover related implementations
- a custom extensions endpoint prefix

These endpoints are documented below.

4.1. Entry Listing Endpoints

Entry listing endpoints return a list of resource objects representing entries of a specific type. For example, a list of structures, or a list of calculations.

Examples:

- http://example.com/optimade/v0.9/structures
- http://example.com/optimade/calculations

There MAY be multiple entry listing endpoints, depending on how many types of entries an implementation provides. Specific standard entry types are specified in section 6. Entry list. The API implementation MAY provide other entry types than the ones standardized in this specification, but such entry types MUST be prefixed by a database-provider-specific prefix.

4.1.1. URL Query Parameters

The client MAY provide a set of URL query parameters in order to alter the response and provide usage information. While these URL query parameters are OPTIONAL for clients, API implementations MUST accept and handle them. To adhere to the requirement on implementation-specific URL query parameters of JSON API v1.0, query parameters that are not standardized by that specification have been given names that consist of at least two words separated by an underscore (a LOW LINE character `_`).

Standard OPTIONAL URL query parameters standardized by the JSON API specification:

- **filter**: a filter string, in the format described below in section 5. API Filtering Format Specification.

- **page_limit**: sets a numerical limit on the number of entries returned. See JSON API 1.0. The API implementation MUST return no more than the number specified. It MAY return fewer. The database MAY have a maximum limit and not accept larger numbers (in which case an error code -- 403 Forbidden -- MUST be returned). The default limit value is up to the API implementation to decide.
Example: http://example.com/optimade/v0.9/structures?page_limit=100

- **page_{offset, page, cursor, above, below}** A server MUST implement pagination in the case of no user-specified *sort* parameter (via the "links" response field). A server MAY implement pagination in concert with *sort*. The following parameters, all prefixed by "page_", are RECOMMENDED for use with pagination. If an implementation chooses
  - *offset-based pagination*: using *page_offset* and *page_limit* is RECOMMENDED.
  - *cursor-based pagination*: using *page_cursor* and *page_limit* is RECOMMENDED.
  - *page-based pagination*: using *page_number* and *page_limit* is RECOMMENDED (*page_limit* is equivalent to *page"size").
  - *value-based pagination*: using *page_above* / *page_below* and *page_limit* is RECOMMENDED.

Examples (all OPTIONAL behavior a server MAY implement):
  - skip 50 structures and fetch up to 100: /structures?page_offset=50&page_limit=100
  - fetch page 2 of up to 50 structures per page: /structures?page_number=2&page_limit=50
  - fetch up to 100 structures above sort-field value 4000 (in this example, server chooses to fetch results sorted by increasing *id*, so *page_above* value refers to an *id* value): /structures?page_above=4000&page_limit=100

- **sort**: If supporting sortable queries, an implementation MUST use the *sort* query parameter with format as specified by JSON API 1.0.

An implementation MAY support multiple sort fields for a single query. If it does, it again MUST conform to the JSON API 1.0 spec.

If an implementation supports sorting for an entry listing endpoint, then the /info/<entries> endpoint MUST include, for each field name `<fieldname>` in its "data.properties.<fieldname>" response value, the key "sortable" with value true. This is in addition to each property description (and optional unit). An example is shown in section 4.4.2 Entry Listing Info Endpoints.

Standard OPTIONAL URL query parameters not in the JSON API specification:
  - **response_format**: the output format requested (see section 3.3.1 Response Format). Defaults to the format string 'json', which specifies the standard output format described in this specification. Example: http://example.com/optimade/v0.9/structures?response_format=xml
  - **email_address**: an email address of the user making the request. The email SHOULD be that of a person and not an automatic system. Example: http://example.com/optimade/v0.9/structures?email_address=user@example.com
  - **response_fields**: a comma-delimited set of fields to be provided in the output. If provided, only these fields MUST be returned and no others. Example: http://example.com/optimade/v0.9/structures?response_fields=id,url

Additional OPTIONAL URL query parameters not described above are not considered to be part of this standard, and are instead considered to be "custom URL query parameters". These custom URL query parameters MUST be of the format "<database-provider-specific prefix><url_query_parameter_name>". These names adhere to the requirements on implementation-specific query parameters of JSON API v1.0 since the database-provider-specific prefixes contain at least two underscores (a LOW LINE character `_`).

Example uses of custom URL query parameters include providing an access token for the request, to tell the database to increase verbosity in error output, or providing a database-specific extended searching format.

Examples:
  - http://example.com/optimade/v0.9/structures?_exmpl_key=A3242DSFJFEJE
  - http://example.com/optimade/v0.9/structures?_exmpl_warning_verbosity=10
Note: the specification presently makes no attempt to standardize access control mechanisms. There are security concerns with access control based on URL tokens, and the above example is not to be taken as a recommendation for such a mechanism.

### 4.1.2. JSON Response Schema

"Entry listing" endpoint response dictionaries MUST have a `data` key. The value of this key MUST be a list containing dictionaries that represent individual entries. In the default JSON response format every dictionary (resource object) MUST have the following fields:

- **`type`**: field containing the Entry type as defined in section 2. Term Definition
- **`id`**: field containing the ID of entry as defined in section 2. Term Definition. This can be the local database ID.
- **`attributes`**: a dictionary, containing key-value pairs representing the entry's properties, except for type and id.

Database-provider-specific properties need to include the database-provider-specific prefix (see Appendix 1).

OPTIONALLY it can also contains the following fields:

- **`links`**: a JSON API links object can OPTIONALLY contain the field
  - `self`: the entry's URL
- **`meta`**: a JSON API meta object that contains non-standard meta-information about the object
- **`relationships`**: a dictionary containing references to other entries according to the description in 3.5. Relationships encoded as JSON API Relationships. The OPTIONAL human-readable description of the relationship MAY be provided in the "description" field inside the "meta" dictionary.

Example:

```json
{
  "data": [
    {
      "type": "structures",
      "id": "example.db:structs:0001",
      "attributes": {
        "chemical_formula_descriptive": "Es2 O3",
        "url": "http://example.db/structs/0001",
        "immutable_id": "http://example.db/structs/0001@123",
        "last_modified": "2007-04-05T14:30Z"
      }
    },
    {
      "type": "structures",
      "id": "example.db:structs:1234",
      "attributes": {
        "chemical_formula_descriptive": "Es2",
        "url": "http://example.db/structs/1234",
        "immutable_id": "http://example.db/structs/1234@123",
        "last_modified": "2007-04-07T12:02Z"
      }
    }
  ]
}
```
4.2. Single Entry Endpoints

A client can request a specific entry by appending an URL-encoded ID component to the URL of an entry listing endpoint. This will return properties for the entry with that ID.

In the default JSON response format, the ID component MUST be the content of the `id` field.

Examples:

- http://example.com/optimade/v0.9/structures/exmpl%3Astruct_3232823
- http://example.com/optimade/v0.9/calculations/232132

4.2.1. URL Query Parameters

The client MAY provide a set of additional URL query parameters for this endpoint type. URL query parameters not recognized MUST be ignored. While the following URL query parameters are OPTIONAL for clients, API implementations MUST accept and handle them: `response_format`, `email_address`, `response_fields`. The meaning of these URL query parameters are as defined above in section 4.1.1. URL Query Parameters.

4.2.2. JSON Response Schema

The response for a 'single entry' endpoint is the same as for 'entry listing' endpoint responses, except that the value of the `data` field MUST have only one or zero entries. In the default JSON response format, this means the value of the `data` field MUST be a single response object or `null` if there is no response object to return.

Example:

```
{
    "data": {
        "type": "structures",
        "id": "example.db:structs:1234",
        "attributes": {
            "chemical_formula_descriptive": "Es2",
            "url": "http://example.db/structs/1234",
            "immutable_id": "http://example.db/structs/1234@123",
            "last_modified": "2007-04-07T12:02Z"
        }
    },
    "meta": {
        "query": {
            "representation": "/structures/example.db:structs:1234?"
        }
    }
}
```

4.3. Info Endpoints

Info endpoints provide introspective information, either about the API implementation itself, or about specific entry types.

There are two types of info endpoints:

1. the base URL (e.g., http://example.com/optimade/v0.9/info)
2. type-specific entry listing endpoints (e.g., http://example.com/optimade/v0.9/info/structures)
The types and output content of these info endpoints are described in more detail in the subsections below. Common for them all are that the `data` field SHOULD return only a single resource object. If no resource object is provided, the value of the `data` field MUST be `null`.

### 4.3.1. Base URL Info Endpoint

The Info endpoint on the base URL or directly after the version number (e.g., `http://example.com/optimade/v0.9/info`) returns information relating to the API implementation.

The single resource object's response dictionary MUST include the following fields:

- **type**: "info"
- **id**: "/"

- **attributes**: Dictionary containing the following fields:
  - **api_version**: Presently used version of the OPTiMaDe API.
  - **available_api_versions**: MUST be a list of dictionaries, each containing the fields:
    - **url**: a string specifying a base URL that MUST adhere to the rules in section 3.1. Base URL
    - **version**: a string containing the full version number of the API served at that base URL. The version number string MUST NOT be prefixed by, e.g., "v".
    - **formats**: List of available output formats.
    - **entry_types_by_format**: Available entry endpoints as a function of output formats.
    - **available_endpoints**: List of available endpoints (i.e., the string to be appended to the base URL).

  `attributes` MAY also include the following OPTIONAL fields:

  - **is_index**: if `true`, this is an index meta-database base URL (see section 3.4. Index Meta-Database).

  If this member is `not` provided, the client MUST assume this is `not` an index meta-database base URL (i.e., default: "is_index": false).

  If this is an index meta-database base URL (see section 3.4. Index Meta-Database), then the response dictionary MUST also include the field:

  - **relationships**: Dictionary that MAY contain a single JSON API relationships object
    - **default**: Reference to the child object under the links endpoint that the provider has chosen as their “default” OPTiMaDe API database. A client SHOULD present this database as the first choice when an end-user chooses this provider. This MUST include the field:
    - **data**: JSON API resource linkage. It MUST be either null or contain a single child identifier object with the fields:
      - **type**: child
      - **id**: ID of the provider's chosen default OPTiMaDe API database. MUST be equal to a valid child object's id under the links endpoint.

  Lastly, `is_index` MUST also be included in `attributes` and be `true`.

Example:
Example for an index meta-database:

```json
{
  "data": {
    "type": "info",
    "id": "/",
    "attributes": {
      "api_version": "v0.9",
      "available_api_versions": [
        {"url": "http://db.example.com/optimade/v0.9/", "version": "0.9.5"},
        {"url": "http://db.example.com/optimade/v1.0/", "version": "1.0.2"},
        {"url": "http://db.example.com/optimade/v0.9.2/", "version": "0.9.2"},
        {"url": "http://db.example.com/optimade/v0.9.5/", "version": "0.9.5"}
      ],
      "formats": [
        "json",
        "xml"
      ],
      "entry_types_by_format": {
        "json": [
          "structures",
          "calculations"
        ],
        "xml": [
          "structures"
        ]
      },
      "available_endpoints": [
        "structures",
        "calculations",
        "info",
        "links"
      ],
      "is_index": false
    }
  }
} // ...
```
4.3.2. Entry Listing Info Endpoints

Entry listing info endpoints are of the form "<base_url>/info/<entry_type>" (e.g., http://example.com/optimade/v0.9/info/structures). The response for these endpoints MUST include the following information in the data field:

- **description**: Description of the entry.
- **properties**: A dictionary describing queryable properties for this entry type, where each key is a property name. Each value is a dictionary, with the REQUIRED key **description** and OPTIONAL key **unit**.
- **formats**: List of output formats available for this type of entry.
- **output_fields_by_format**: Dictionary of available output fields for this entry type, where the keys are the values of the formats list and the values are the keys of the properties dictionary.

Example:
4.4. Links Endpoint

This endpoint exposes information on other OPTiMaDe API implementations that are linked to the current implementation. The endpoint MUST be provided at the path "<base_url>/links".

It may be considered an introspective endpoint, similar to the Info endpoint, but at a higher level: that is, Info endpoints provide information on the given implementation, while the Links endpoint provides information on the links between immediately related implementations (in particular, an array of none or a single "parent" object and none or more "child" objects, see section 4.5.2 Parent and Child Objects).

For Links endpoints, the API implementation MAY ignore any provided query parameters. Alternatively, it MAY handle the parameters specified in section 4.2.1. URL Query Parameters for single entry endpoints.

4.4.1. JSON Response Schema

The resource objects’ response dictionaries MUST include the following fields:

- **type**: MUST be either "parent", "child", or "provider". These objects are described in detail in sections 4.4.2. Parent and Child Objects and 4.4.3. Provider Objects.
- **id**: MUST be unique.
- **attributes**: Dictionary that MUST contain the following fields:
  - **name**: Human-readable name for the OPTiMaDe API implementation a client may provide in a list to an end-user.
  - **description**: Human-readable description for the OPTiMaDe API implementation a client may provide in a list to an end-user.
  - **base_url**: JSON API links object, pointing to the base URL for this implementation, either directly as a string, or as a links object, which can contain the following fields:
    - **href**: a string containing the OPTiMaDe base URL.
    - **meta**: a meta object containing non-standard meta-information about the implementation.
Example:

```json
{
  "data": [
    {
      "type": "parent",
      "id": "index",
      "attributes": {
        "name": "Index",
        "description": "Index for example's OPTiMaDe databases",
        "base_url": "http://example.com/optimade/index"
      }
    },
    {
      "type": "child",
      "id": "cat_zeo",
      "attributes": {
        "name": "Catalytic Zeolites",
        "description": "Zeolites for deNOx catalysis",
        "base_url": {
          "href": "http://example.com/optimade/denox/zeolites",
          "meta": {
            "_exmpl_catalyst_group": "denox"
          }
        }
      }
    },
    {
      "type": "child",
      "id": "frameworks",
      "attributes": {
        "name": "Zeolitic Frameworks",
        "description": "",
        "base_url": "http://example.com/optimade/zeo_frameworks"
      }
    },
    {
      "type": "provider",
      "id": "exmpl",
      "attributes": {
        "name": "Example provider",
        "description": "Provider used for examples, not to be assigned to a real database",
        "base_url": "http://example.com/optimade/index"
      }
    }
  ]
}
```

4.4.2. Parent and Child Objects

Resource objects that MAY be present under the Links endpoint.

Either none or a single "parent" object MAY be present as part of the data array. The "parent" object represents a "link" to
the OPTiMaDe implementation exactly one layer above the current implementation's layer.

Any number of "child" objects MAY be present as part of the data array. A "child" object represents a 'link' to an OPTiMaDe implementation exactly one layer below the current implementation's layer.

Note: The RECOMMENDED number of layers is two.

4.4.3. Provider Objects

"provider" objects are meant to indicate links to an "Index meta-database" hosted by database providers. The intention is to be able to auto-discover all providers of OPTiMaDe implementations.

A known list of providers can be found in Appendix 1.

Note: If a provider wishes to be added to "provider.json", please suggest a change to the OPTiMaDe main repository (make a pull request). A link to the main repository may be found at the OPTiMaDe homepage.

4.4.4. Index Meta-Database Links Endpoint

If the provider implements an "index meta-database" (see section 3.4 Index Meta-Database), it is RECOMMENDED to adopt a structure, where the index meta-database is the "parent" implementation of the provider's other OPTiMaDe databases.

This will make all OPTiMaDe databases and implementations by the provider discoverable as "child" objects under the Links endpoint of the "Index meta-database".

4.5. Custom Extension Endpoints

API implementations can provide custom endpoints under the Extensions endpoint. These endpoints should have the form "<base_url>/extensions/<custom paths>".

5. API Filtering Format Specification

An OPTiMaDe filter expression is passed in the parameter filter as an URL query parameter as specified by JSON API. The filter expression allows desired properties to be compared against search values; several such comparisons can be combined using the logical conjunctions AND, OR, NOT, and parentheses, with their usual semantics.

When provided as an URL query parameter, the contents of the filter parameter is URL-encoded by the client in the HTTP GET request, and then URL-decoded by the API implementation before any further parsing takes place. In particular, this means the client MUST escape special characters in string values as described below in the section "String values" before the URL encoding, and the API implementation MUST first URL-decode the filter parameter before reversing the escaping of string tokens.

Examples of syntactically correct query strings embedded in queries:

- http://example.org/optimade/v0.9/structures?filter=_exmpl_melting_point%3C300+AND+nelements=4+AND+elements="Si,O2"&response_format=xml

Or, fully URL encoded:

- http://example.org/optimade/v0.9/structures?
  filter=_exmpl_melting_point%3C300+AND+nelements%3D4+AND+elements%3D%22Si,O2%22&response_format=xml

5.1. Lexical Tokens

The following tokens are used in the filter query component:

- Property names: the first character MUST be a lowercase letter, the subsequent symbols MUST be composed of lowercase
letters or digits; the underscore ("_", ASCII 95 dec (0x5F)) is considered to be a lower-case letter when defining identifiers. The length of the identifiers is not limited, except that when passed as a URL query parameter the whole query SHOULD NOT be longer than the limits imposed by the URI specification. This definition is similar to one used in most widespread programming languages, except that OPTiMaDe limits allowed letter set to lowercase letters only. This allows to tell OPTiMaDe identifiers and operator keywords apart unambiguously without consulting a reserved word table and to encode this distinction concisely in the EBNF Filter Language grammar.

Examples of valid property names:

- band_gap
- cell_length_a
- cell_volume

Examples of incorrect property names:

- _kvak (starts with a number);
- "foo_bar" (contains space; contains quotes)
- "BadLuck" (contains upper-case letters)

Identifiers that start with an underscore are specific to a database provider, and MUST be on the format of a database-provider-specific prefix as defined in Appendix 1.

Examples:

- _exmpl_formula_sum (a property specific to that database)
- _exmpl_band_gap
- _exmpl_supercell
- _exmpl_trajectory
- _exmpl_workflow_id

- **Nested property names** A nested property name is composed of at least two identifiers separated by periods (\).

- **String values** MUST be enclosed in double quotes ("", ASCII symbol 92 dec, 0x5C hex). The quote and other special characters within the double quotes MUST be escaped using C/JSON/Perl/Python convention: a double quote which is a part of the value, not a delimiter, MUST be prepended with a backslash character (\", ASCII symbol), and the backslash character itself, when taken literally, MUST be preceded by another backslash. An example of the escaped string value is given below:

  - "A double quote character ("", ASCII symbol 92 dec) MUST be prepended by a backslash (\", ASCII symbol 92 dec) when it is a part of the value and not a delimiter; the backslash character \" itself MUST be preceded by another backslash, forming a double backslash: \\

(Note that at the end of the string value above the four final backslashes represent the two terminal backslashes in the value, and the final double quote is a terminator, it is not escaped).

String value tokens are also used to represent **timestamps** in form of the RFC 3339 Internet Date/Time Format

- **Numeric values** are represented as decimal integers or in scientific notation, using the usual programming language conventions. A regular expression giving the number syntax is given below as a POSIX Extended Regular Expression (ERE) or as a Perl-Compatible Regular Expression (PCRE):

  - PCRE: \[-+\]?(?:\d+(\.\d*)?|\.\d+)(?:[eE][\-+]?\d+)?

An implementation of the search filter MAY reject numbers that are outside the machine representation of the underlying hardware; in such case it MUST return the error 501 Not Implemented with an appropriate error message that indicates the cause of the error and an acceptable number range.
Examples of valid numbers:
- 12345, +12, -34, 1.2, .2E7, +1.0E-10, 6.03e23, .1E1, -1e1, 1.e-12, -1e-12, 1000000000.000000001, .1

Examples of invalid numbers (although they MAY contain correct numbers as substrings):
- 1.234D12, .e1, -.E1, +.E2, 1.23E++, +-123

Note: this number representation is more general than the number representation in JSON (for instance, 1. is a valid numeric value for the filtering language specified here, but it is not a valid float number in JSON, where one must write 1.0 instead).

While the filtering language supports tests for equality between properties of floating point type and decimal numbers given in the filter string, such comparisons come with the usual caveats for testing for equality of floating point numbers. Normally, a client cannot rely on the fact that a floating point number stored in a database takes on a representation that exactly matches the one obtained for a number given in the filtering string as a decimal number or as an integer. However, testing for equality to zero MUST be supported.

More examples of the number tokens and machine-readable definitions and tests can be found in the Materials-Consortia API Git repository (files integers.lst, not-numbers.lst, numbers.lst, and reals.lst).

Operator tokens
Operator tokens are represented by usual mathematical relation symbols or by case-sensitive keywords. Currently the following operators are supported: =, !=, <>, !=, <, > for tests of number, string (lexicographical) or timestamp (temporal) equality, inequality, less-than, more-than, less, and more relations; AND, OR, NOT for logical conjunctions, and a number of keyword operators discussed in the next section.

In future extensions, operator tokens that are words MUST contain only upper-case letters. This requirement guarantees that no operator token will ever clash with a property name.

5.2. The Filter Language Syntax

All filtering expressions MUST follow the EBNF grammar of Appendix 2 of this specification. The appendix contains a complete machine-readable EBNF, including the definition of the lexical tokens described above in 5.1. Lexical tokens. The EBNF is enclosed in special strings constructed as BEGIN and END, both followed by EBNF GRAMMAR Filter, to enable automatic extraction.

Basic boolean operations

The filter language supports conjunctions of comparisons using the boolean algebra operators "AND", "OR", and "NOT" and parentheses to group conjunctions. A comparison clause prefixed by NOT matches entries for which the comparison is false.

Examples:
- NOT ( chemical_formula_hill = "Al" AND chemical_formula_anonymous = "A" OR chemical_formula_anonymous = "H2O" AND NOT chemical_formula_hill = "Ti" )

Numeric and String comparisons

Comparisons involving Numeric and String properties can be expressed using the usual comparison operators: '<', '>', '<=', '>=', '='. Implementations MUST support comparisons on the form:

```
identifier <operator> constant
constant <operator> identifier
```

Where 'identifier' is a property name and 'constant' is either a numerical or string type constant. However, implementations MAY OPTIONALLY support comparisons with identifiers also on both sides, and comparisons with values on both sides, i.e., on the
forms:

identifier <operator> identifier
constant <operator> constant

Examples:

- `nelements > 3`
- `chemical_formula_hill = "H2O" AND chemical_formula_anonymous != "AB"`
- `_exmpl_activex <= +1e8 OR nelements >= 10 AND NOT (_exmpl_x != "Some string" OR NOT _exmpl_a = 7)`
- `_exmpl_spacegroup="P2"`
- `_exmpl_cell_volume<100.0`
- `_exmpl_bandgap > 5.0 AND _exmpl_molecular_weight < 350`
- `_exmpl_melting_point<300 AND nelements=4 AND elements="Si,O2"`
- `_exmpl_some_string_property = 42` (This is syntactically allowed without putting 42 in quotation marks, see the notes about comparisons of values of different types below.)
- `5 < _exmpl_a`
- `OPTIONAL: ((NOT (_exmpl_a>_exmpl_b)) AND _exmpl_x>0)`
- `OPTIONAL: 5 < 7`

Substring comparisons

In addition to the standard equality and inequality operators, matching of partial strings is provided by keyword operators:

- `identifier CONTAINS x`: Is true if the substring value `x` is found anywhere within the property.
- `identifier STARTS WITH x`: Is true if the property starts with the substring value `x`. The `WITH` keyword may be omitted.
- `identifier ENDS WITH x`: Is true if the property ends with the substring value `x`. The `WITH` keyword may be omitted.

OPTIONAL features:

- Support for `x` to be an identifier, rather than a string is OPTIONAL.

Examples:

- `chemical_formula_anonymous CONTAINS "C2" AND chemical_formula_anonymous STARTS WITH "A2"`
- `chemical_formula_anonymous STARTS "B2" AND chemical_formula_anonymous ENDS WITH "D2"`

Comparisons of list properties

In the following, `list` is a list-type property, and `values` is one or more `value` separated by commas (","), i.e., strings or numbers. An implementation MAY also support property names and nested property names in `values`.

The following constructs MUST be supported:

- `list HAS value`: matches if at least one element in `list` is equal to `value`. (If `list` has no duplicate elements, this implements the set operator IN.)
- `list HAS ALL values`: matches if, for each `value`, there is at least one element in `list` equal to that value. (If both `list` and `values` do not contain duplicate values, this implements the set operator >=.)
- `list HAS ANY values`: matches if at least one element in `list` is equal to at least one `value`. (This is equivalent to a number of HAS statements separated by OR.)
- `LENGTH list <operator> value`: applies the numeric comparison `<operator>` for the number of items in the list property.
The following construct MAY be supported:

- list HAS ONLY values: matches if all elements in list are equal to at least one value. (If both list and values do not contain duplicate values, this implements the \( \leq \) set operator.)

This construct is OPTIONAL as it may be difficult to realize in some underlying database implementations. However, if the desired search is over a property that can only take on a finite set of values (e.g., chemical elements) a client can formulate an equivalent search by inverting the list of values into inverse and express the filter as NOT list HAS inverse.

Furthermore, there is a set of OPTIONAL constructs that allows filters to be formulated over the values in correlated positions in multiple list properties. An implementation MAY support this syntax selectively only for specific properties. This type of filter is useful, for example, filtering on elements and correlated element counts available as two separate list properties.

- list1:list2:... HAS val1:val2:...
- list1:list2:... HAS ALL val1:val2:...
- list1:list2:... HAS ANY val1:val2:...
- list1:list2:... HAS ONLY val1:val2:...

Finally, all the above constructs that allow a value or lists of values on the right-hand side MAY allow <operator> value in each place a value can appear. In that case, a match requires that the <operator> comparison is fulfilled instead of equality. Strictly, the definitions of the HAS, HAS ALL, HAS ANY and HAS ONLY operators as written above apply, but with the word 'equal' replaced with the <operator> comparison.

For example:

- list HAS < 3: matches all entries for which list contains at least one element that is less than three.
- list HAS ALL < 3, > 3: matches only those entries for which list simultaneously contains at least one element less than three and one element greater than three.

An implementation MAY support combining the operator syntax with the syntax for correlated lists in particularly on a list correlated with itself. For example:

- list:list HAS >=2:<=5: matches all entries for which list contains at least one element that is between the values 2 and 5.

Further examples of various comparisons of list properties:

- elements HAS "H" AND elements HAS ALL "H","He","Ga","Ta" AND elements HAS ONLY "H","He","Ga","Ta" AND elements HAS ANY "H", "He", "Ga", "Ta"  
  OPTIONAL: elements HAS ONLY "H","He","Ga","Ta"
  OPTIONAL: elements:_exmpl_element_counts HAS "H":6 AND elements:_exmpl_element_counts HAS ALL "H":6,"He":7 AND elements:_exmpl_element_counts HAS ONLY "H":6 AND elements:_exmpl_element_counts HAS ANY "H":6,"He":7 
  OPTIONAL: _exmpl_element_counts HAS < 3 AND _exmpl_element_counts HAS ANY > 3, = 6, 4, != 8 (note: specifying the = operator after HAS ANY is redundant here, if no operator is given, the test is for equality.)
  OPTIONAL: _exmpl_element_counts:_exmpl_element_weights HAS ANY > 3:"He":>55.3 , = 6:"Ti":<37.6 , 8:"Ga":0

Nested property names

Everywhere in a filter string where a property name is accepted, the API implementation MAY accept nested property names as described in 5.1 Lexical tokens, consisting of identifiers separated by periods (\('.\)\). A filter on a nested property name consisting of two identifiers identifier1.identifier2 matches if either one of these points are true:

- identifier1 references a dictionary-type property that contains as an identifier identifier2 and the filter matches for the content of identifier2.
• identifier1 references a list of dictionaries that contain as an identifier identifier2 and the filter matches for a flat list containing only the contents of identifier2 for every dictionary in the list. E.g., if identifier1 is the list

\[\{"identifier2":42, "identifier3":36}, \{"identifier2":96, "identifier3":66}\], then identifier1.identifier2 is understood in the filter as the list [42, 96].

The API implementation MAY allow this notation to generalize to arbitrary depth. A nested property name that combines more than one list MUST, if accepted, be interpreted as a completely flattened list.

Relationships

As described in section 3.5. Relationships, it is possible for the API implementation to describe relationships between entries of the same, or different, entry types. The API implementation MAY support queries on relationships with an entry type <entry type> by using special nested property names:

- <entry type>.id references a list of IDs of relationships with entries of the type <entry type>.
- <entry type>.description references a correlated list of the human-readable descriptions of these relationships.

Hence, the filter language acts as, for every entry type, there is a property with that name which contains a list of dictionaries with two keys, id and description. For example: a client queries the structures endpoint with a filter that references calculations.id. For a specific structures entry, the nested property may behave as the list ["calc-id-43", "calc-id-96"] and would then, e.g., match the filter calculations.id HAS "calc-id-96". This means that the structures entry has a relationship with the calculations entry of that ID.

Note: formulating queries on relationships with entries that have specific property values is a multi-step process. For example, to find all structures with bibliographic references where one of the authors has the last name "Schmit" is performed by the following two steps:

- Query the references endpoint with a filter authors.lastname HAS "Schmit" and store the id values of the returned entries.
- Query the structures endpoint with a filter references.id HAS ANY <list-of-IDs>, where <list-of-IDs> are the IDs retrieved from the first query separated by commas.

(Note: the type of query discussed here corresponds to a "join"-type operation in a relational data model.)

Properties with unknown value

Properties may have an unknown value, see 3.3.5. Properties with unknown value.

Filters that match when the property is known, or unknown, respectively can be constructed using the following syntax:

<table>
<thead>
<tr>
<th>identifier IS KNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier IS UNKNOWN</td>
</tr>
</tbody>
</table>

Except for the above constructs, filters that use any form of comparison that involve properties of unknown values MUST NOT match. Hence, by definition, an identifier of value null never matches equality (=), inequality (<, <=, >, >=), or other comparison operators besides identifier IS UNKNOWN and NOT identifier IS KNOWN. In particular, a filter that compares two properties that are both null for equality or inequality does not match.

Examples:

- chemical_formula_hill IS KNOWN AND NOT chemical_formula_anonymous IS UNKNOWN

Precedence

The precedence (priority) of the operators MUST be as indicated in the list below:
1. Comparison and keyword operators ( < , <= , = , HAS , STARTS , etc.) -- highest priority;
2. NOT
3. AND
4. OR -- lowest priority.

Examples:

- NOT a > b OR c = 100 AND f = "C2 H6" : this is interpreted as (NOT (a > b)) OR ((c = 100) AND (f = "C2 H6")) when fully braced.
- a >= 0 AND NOT b < c OR c = 0 : this is interpreted as ((a >= 0) AND (NOT (b < c))) OR (c = 0) when fully braced.

Type handling and conversions in comparisons

The definitions of specific properties in this standard define their types. Similarly, for database-provider-specific properties, the database provider decides their types. In the syntactic constructs that can accommodate values of more than one type, types of all participating values are REQUIRED to match, with a single exception of timestamps (see below). Different types of values MUST be reported as 501 Not Implemented errors, meaning that type conversion is not implemented in the specification.

As the filter language syntax does not define a lexical token for timestamps, values of this type are expressed using string tokens in RFC 3339 Internet Date/Time Format. In a comparison with a timestamp property, a string token represents a timestamp value that would result from parsing the string according to RFC 3339 Internet Date/Time Format. Interpretation failures MUST be reported with error 400 Bad Request.

Optional filter features

Some features of the filtering language are marked OPTIONAL. An implementation that encounters an OPTIONAL feature that it does not support MUST respond with error 501 Not Implemented with an explanation of which OPTIONAL construct the error refers to.

6. Entry List

This section defines standard entry types and their properties.

6.1. Properties Used by Multiple Entry Types

6.1.1. id

- **Description**: An entry's ID as defined in section 2. Term Definition.
- **Type**: string.
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property with support for all mandatory filter operators.
  - See section 2. Term Definition.
- **Examples**:
  - "db/1234567"
  - "cod/2000000"
  - "cod/2000000@1234567"
  - "nomad/L1234567890"
  - "42"

6.1.2. type
6.1.3. immutable_id

- **Description**: The entry’s immutable ID (e.g., an UUID). This is important for databases having preferred IDs that point to “the latest version” of a record, but still offer access to older variants. This ID maps to the version-specific record, in case it changes in the future.
- **Type**: string.
- **Requirements/Conventions**:
  - Response: OPTIONAL in the response.
  - Query: If present, MUST be a queryable property with support for all mandatory filter operators.
- **Examples**:
  - "8bd3e750-b477-41a0-9b11-3a799f21b44f"
  - "fjeiwoj,54;@=%<>#32" (Strings that are not URL-safe are allowed.)

6.1.4. last_modified

- **Description**: Date and time representing when the entry was last modified.
- **Type**: timestamp.
- **Requirements/Conventions**:
  - Response: REQUIRED in the response unless explicitly excluded.
  - Query: MUST be a queryable property with support for all mandatory filter operators.
- **Example**:
  - As part of JSON response format: "2007-04-05T14:30Z" (i.e., encoded as an RFC 3339 Internet Date/Time Format string.)

6.1.5. database-provider-specific properties

- **Description**: Database providers are allowed to insert database-provider-specific entries in the output of both standard entry types and database-provider-specific entry types.
- **Type**: Decided by the API implementation.
- **Requirements/Conventions**:
  - Response: OPTIONAL in the response.
  - Query: Support for queries on these properties are OPTIONAL. If supported, only a subset of filter operators MAY be supported.
  - These MUST be prefixed by a database-provider-specific prefix as defined in Appendix 1.
- **Examples**: A few examples of valid database-provided-specific property names follows:
  - _exmpl_formula_sum
  - _exmpl_band_gap
  - _exmpl_supercell
  - _exmpl_trajectory
  - _exmpl_workflow_id

6.2. Structures Entries
"structures" entries (or objects) have the properties described above in section 6.1. Properties Used by Multiple Entry Types as well as the following properties:

6.2.1. elements

- **Description**: Names of the different elements present in the structure.
- **Type**: list of strings.
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property with support for all mandatory filter operators.
    - The strings are the chemical symbols, written as uppercase letter plus optional lowercase letters.
    - The order MUST be alphabetical.
- **Examples**:
  - ["Si"]
  - ["Al","O","Si"]
- **Query examples**:
  - A filter that matches all records of structures that contain Si, Al, and O, and possibly other elements: `elements HAS ALL "Si", "Al", "O"`.
  - To match structures with exactly these three elements, use `elements HAS ALL "Si", "Al", "O" AND LENGTH elements = 3`.

6.2.2. nelements

- **Description**: Number of different elements in the structure as an integer.
- **Type**: integer
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property with support for all mandatory filter operators.
- **Example**: 3
- **Querying**:
  - Note: queries on this property can equivalently be formulated using `LENGTH elements`.
  - A filter that matches structures that have exactly 4 elements: `nelements=4`.
  - A filter that matches structures that have between 2 and 7 elements: `nelements>=2 AND nelements<=7`.

6.2.3. elements_ratios

- **Description**: Relative proportions of different elements in the structure.
- **Type**: list of floats
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property with support for all mandatory filter operators.
    - Composed by the proportions of elements in the structure as a list of floating point numbers.
    - The sum of the numbers MUST be 1.0 (within floating point accuracy)
- **Examples**:
  - [1.0]
  - [0.3333333333333333, 0.2222222222222222, 0.44444444444444444]
- **Query examples**:
  - A filter that matches structures where approximately 1/3 of the atoms in the structure are the element Al is: `elements:elements_ratios HAS ALL "Al":>0.3333, "Al":<0.3334`.
6.2.4. chemical_formula_descriptive

- **Description**: The chemical formula for a structure as a string in a form chosen by the API implementation.
- **Type**: string
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property with support for all mandatory filter operators.
  - The chemical formula is given as a string consisting of properly capitalized element symbols followed by integers or decimal numbers, balanced parentheses, square, and curly brackets , , ] , { , commas, the + , - , : , and = symbols. The parentheses are allowed to be followed by a number. Spaces are allowed anywhere except within chemical symbols. The order of elements and any groupings indicated by parentheses or brackets are chosen freely by the API implementation.
  - The string SHOULD be arithmetically consistent with the element ratios in the chemical_formula_reduced property.
  - It is RECOMMENDED, but not mandatory, that symbols, parentheses and brackets, if used, are used with the meanings prescribed by IUPAC's Nomenclature of Organic Chemistry.

- **Examples**:
  - "(H2O)2 Na"
  - "NaCl"
  - "CaCO3"
  - "CCaO3"
  - 
    "(CH3)3N+ - [CH2]2-OH = Me3N+ - CH2 - CH2OH"

- **Query examples**:
  - Note: the free-form nature of this property is likely to make queries on it across different databases inconsistent.
  - A filter that matches an exactly given formula: `chemical_formula_descriptive="(H2O)2 Na"`.
  - A filter that does a partial match: `chemical_formula_descriptive CONTAINS "H2O"`.

6.2.5. chemical_formula_reduced

- **Description**: The reduced chemical formula for a structure as a string with element symbols and integer chemical proportion numbers. The proportion number MUST be omitted if it is 1.
- **Type**: string
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property. However, support for filters using partial string matching with this property is OPTIONAL (i.e., BEGINS WITH, ENDS WITH, and CONTAINS). Intricate querying on formula components are instead recommended to be formulated using set-type filter operators on the multi valued elements and elements_proportions properties.
  - Element names MUST have proper capitalization (e.g., "Si", not "SI" for "silicon").
  - Elements MUST be placed in alphabetical order, followed by their integer chemical proportion number.
  - For structures with no partial occupation, the chemical proportion numbers are the smallest integers for which the chemical proportion is exactly correct.
  - For structures with partial occupation, the chemical proportion numbers are integers that within reasonable approximation indicate the correct chemical proportions. The precise details of how to perform the rounding is chosen by the API implementation.
  - No spaces or separators are allowed.

- **Examples**:
  - "H2NaO"
  - "ClNa"
  - "CCaO3"

- **Query examples**:
  - A filter that matches an exactly given formula is `chemical_formula_reduced="H2NaO"`.
6.2.6. chemical_formula_hill

- **Description**: The chemical formula for a structure in Hill form with element symbols followed by integer chemical proportion numbers. The proportion number MUST be omitted if it is 1.
- **Type**: string
- **Requirements/Conventions**:
  - **Response**: OPTIONAL in the response.
  - **Query**: Support for queries on these properties are OPTIONAL. If supported, only a subset of filter operators MAY be supported.
  - The overall scale factor of the chemical proportions is chosen such that the resulting values are integers that indicate the most chemically relevant unit of which the system is composed. For example, if the structure is a repeating unit cell with four hydrogens and four oxygens that represents two hydroperoxide molecules, chemical_formula_hill is H2O2 (i.e., not HO, nor H4O4).
  - If the chemical insight needed to ascribe a Hill formula to the system is not present, the property MUST be handled as unset.
  - Element names MUST have proper capitalization (e.g., "Si", not "SI" for "silicon").
  - Elements MUST be placed in **Hill order**, followed by their integer chemical proportion number. Hill order means: if carbon is present, it is placed first, and if also present, hydrogen is placed second. After that, all other elements are ordered alphabetically. If carbon is not present, all elements are ordered alphabetically.
  - If the system has sites with partial occupation and the total occupations of each element do not all sum up to integers, then the Hill formula SHOULD be handled as unset.
  - No spaces or separators are allowed.
- **Examples**:
  - "H2O2"
- **Query examples**:
  - A filter that matches an exactly given formula is chemical_formula_hill="H2O2".

6.2.7. chemical_formula_anonymous

- **Description**: The anonymous formula is the chemical_formula_reduced, but where the elements are instead first ordered by their chemical proportion number, and then, in order left to right, replaced by anonymous symbols A, B, C, ..., Z, Aa, Ba, ..., Za, Ab, Bb, ... and so on.
- **Type**: string
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property. However, support for filters using partial string matching with this property is OPTIONAL (i.e., BEGINS WITH, ENDS WITH, and CONTAINS).
- **Examples**:
  - "A2B"
  - "A42B42C16D12E10F9G5"
- **Querying**:
  - A filter that matches an exactly given formula is chemical_formula_anonymous="A2B".

6.2.8. dimension_types

- **Description**: List of three integers. For each of the three directions indicated by the three lattice vectors (see property 6.2.9. lattice_vectors). This list indicates if the direction is periodic (value 1) or non-periodic (value 0). Note: the elements in this list each refer to the direction of the corresponding entry in 6.2.9. lattice_vectors and *not* the Cartesian x, y, z directions.
- **Type**: list of integers.
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
- **Query**: MUST be a queryable property. Support for equality comparison is REQUIRED, support for other comparison operators are OPTIONAL.
- MUST be a list of length 3.
- Each integer element MUST assume only the value 0 or 1.

**Examples**:
- For a molecule: [θ, θ, θ]
- For a wire along the direction specified by the third lattice vector: [θ, θ, 1]
- For a 2D surface/slab, periodic on the plane defined by the first and third lattice vectors: [1, θ, 1]
- For a bulk 3D system: [1, 1, 1]

**6.2.9. lattice_vectors**

- **Description**: The three lattice vectors in Cartesian coordinates, in ångström (Å).
- **Type**: list of list of floats.
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded, except when 6.2.8. **dimension_types** is equal to [θ, θ, θ] (in this case it is OPTIONAL).
  - **Query**: Support for queries on this property is OPTIONAL. If supported, filters MAY support only a subset of comparison operators.
  - MUST be a list of three vectors a, b, and c where each of the vectors MUST BE a list of the vector's coordinates along the x, y, and z Cartesian coordinates. (Therefore, the first index runs over the three lattice vectors and the second index runs over the x, y, z Cartesian coordinates).
  - For databases that do not define an absolute Cartesian system (e.g., only defining the length and angles between vectors), the first lattice vector SHOULD be set along x and the second on the xy plane.
  - This property MUST be an array of dimensions 3 times 3 regardless of the elements of 6.2.8. **dimension_types**. The vectors SHOULD by convention be chosen so the determinant of the lattice_vectors matrix is different from zero. The vectors in the non-periodic directions have no significance beyond fulfilling these requirements.

**Examples**:
- `[[4.0,0.0,0.0],[0.0,4.0,0.0],[0.0,1.0,4.0]]` represents a cell, where the first vector is (4, 0, 0), i.e., a vector aligned along the x axis of length 4 Å; the second vector is (0, 4, 0); and the third vector is (0, 1, 4).

**6.2.10. cartesian_site_positions**

- **Description**: Cartesian positions of each site. A site is an atom, a site potentially occupied by an atom, or a placeholder for a virtual mixture of atoms (e.g., in a virtual crystal approximation).
- **Type**: list of list of floats and/or unknown values
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: Support for queries on this property is OPTIONAL. If supported, filters MAY support only a subset of comparison operators.
  - It MUST be a list of length N times 3, where N is the number of sites in the structure.
  - An entry MAY have multiple sites at the same Cartesian position (for a relevant use of this, see e.g., the 6.2.14. **assemblies** property).
  - If a component of the position is unknown, the null value should be provided instead (see section 3.3.5. Properties with unknown value). Otherwise, it should be a float value, expressed in angstrom (Å). If at least one of the coordinates is unknown, the correct flag in the 6.2.15. **structure_features** list MUST be set.
  - **Notes**: (for implementers) While this is unrelated to this OPTiMaDe specification: If you decide to store internally the cartesian_site_positions as a float array, you might want to replace null values with NaN values. The latter being valid float numbers in the IEEE 754 standard in IEEE 754-1985 and in the updated version IEEE 754-2008.

**Examples**:
- `[[0,0,0],[0,0,2]]` indicates a structure with two sites, one sitting at the origin and one along the (positive) z axis, 2 Å away from the origin.
6.2.11. nsites

- **Description**: An integer specifying the length of the `cartesian_site_positions` property.
- **Type**: integer
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: MUST be a queryable property with support for all mandatory filter operators.
- **Examples**:
  - 42
- **Query examples**:
  - Match only structures with exactly 4 sites: `nsites=4`
  - Match structures that have between 2 and 7 sites: `nsites>=2 AND nsites<=7`

6.2.12. species_at_sites

- **Description**: Name of the species at each site (where values for sites are specified with the same order of the `cartesian_site_positions` property). The properties of the species are found in the 6.2.13. species property.
- **Type**: list of strings.
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: Support for queries on this property is OPTIONAL. If supported, filters MAY support only a subset of comparison operators.
  - Each species MUST have a unique name.
  - Each species name mentioned in the `species_at_sites` list MUST be described in the 6.2.13. species list (i.e. for each value in the `species_at_sites` list there MUST exist exactly one dictionary in the species list with the `name` attribute equal to the corresponding `species_at_sites` value).
  - Each site MUST be associated only to a single species.
  - **Note**: However, species can represent mixtures of atoms, and multiple species MAY be defined for the same chemical element. This latter case is useful when different atoms of the same type need to be grouped or distinguished, for instance in simulation codes to assign different initial spin states.
- **Examples**:
  - `"Ti","O2"` indicates that the first site is hosting a species labeled "Ti" and the second a species labeled "O2".

6.2.13. species

- **Description**: A list describing the species of the sites of this structure. Species can be pure chemical elements, or virtual-crystal atoms representing a statistical occupation of a given site by multiple chemical elements.
- **Type**: list of dictionary with keys:
  - `name`: string (REQUIRED)
  - `chemical_symbols`: list of strings (REQUIRED)
  - `concentration`: list of float (REQUIRED)
  - `mass`: float (OPTIONAL)
  - `original_name`: string (OPTIONAL).
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response unless explicitly excluded.
  - **Query**: Support for queries on this property is OPTIONAL. If supported, filters MAY support only a subset of comparison operators.
  - Each list member MUST be a dictionary with the following keys:
• **name**: REQUIRED; gives the name of the species; the name value MUST be unique in the species list.

• **chemical_symbols**: REQUIRED; MUST be a list of strings of all chemical elements composing this species.
  
  - It MUST be one of the following:
  - a valid chemical-element name, or
  - the special value "X" to represent a non-chemical element, or
  - the special value "vacancy" to represent that this site has a non-zero probability of having a vacancy (the respective probability is indicated in the concentration list, see below).
  
  - If any one entry in the species list has a chemical_symbols list that is longer than 1 element, the correct flag MUST be set in the list structure_features (see section 6.2.15. structure_features).

• **concentration**: REQUIRED; MUST be a list of floats, with same length as chemical_symbols. The numbers represent the relative concentration of the corresponding chemical symbol in this species. The numbers SHOULD sum to one. Cases in which the numbers do not sum to one typically fall only in the following two categories:
  
  - Numerical errors when representing float numbers in fixed precision, e.g. for two chemical symbols with concentrations 1/3 and 2/3, the concentration might look something like [0.33333333333, 0.66666666666]. If the client is aware that the sum is not one because of numerical precision, it can renormalize the values so that the sum is exactly one.
  - Experimental errors in the data present in the database. In this case, it is the responsibility of the client to decide how to process the data.

Note that concentrations are uncorrelated between different sites (even of the same species).

• **mass**: OPTIONAL. If present MUST be a float expressed in a.m.u.

• **original_name**: OPTIONAL. Can be any valid Unicode string, and SHOULD contain (if specified) the name of the species that is used internally in the source database.

Note: With regards to "source database", we refer to the immediate source being queried via the OPTiMaDe API implementation. The main use of this field is for source databases that use species names, containing characters that are not allowed (see description of the 6.2.12. species_at_sites list).

• For systems that have only species formed by a single chemical symbol, and that have at most one species per chemical symbol, SHOULD use the chemical symbol as species name (e.g. "Ti" for titanium, "O" for oxygen, etc.) However, note that this is OPTIONAL, and client implementations MUST NOT assume that the key corresponds to a chemical symbol, nor assume that if the species name is a valid chemical symbol, that it represents a species with that chemical symbol. This means that a species {"name": "C", "chemical_symbols": ["Ti"], "concentration": [1.0]} is valid and represents a titanium species (and not a carbon species).

• It is NOT RECOMMENDED that a structure includes species that do not have at least one corresponding site.

• **Examples**:

  "species": [ {"name": "Ti", "chemical_symbols": ["Ti"], "concentration": [1.0]} ]: any site with this species is occupied by a Ti atom.

  "species": [ {"name": "Ti", "chemical_symbols": ["Ti", "vacancy"], "concentration": [0.9, 0.1]} ]: any site with this species is occupied by a Ti atom with 90 % probability, and has a vacancy with 10 % probability.

  "species": [ {"name": "BaCa", "chemical_symbols": ["vacancy", "Ba", "Ca"], "concentration": [0.05, 0.45, 0.5], "mass": 88.5} ]: any site with this species is occupied by a Ba atom with 45 % probability, a Ca atom with 50 % probability, and by a vacancy with 5 % probability. The mass of this site is (on average) 88.5 a.m.u.

  "species": [ {"name": "C12", "chemical_symbols": ["C"], "concentration": [1.0], "mass": 12.0} ]: any site with this species is occupied by a carbon isotope with mass 12.

  "species": [ {"name": "C13", "chemical_symbols": ["C"], "concentration": [1.0], "mass": 13.0} ]: any site with this species is occupied by a carbon isotope with mass 13.
6.2.14. assemblies

- **Description**: A description of groups of sites that are statistically correlated.
- **Type**: list of dictionary with keys:
  - `sites_in_groups`: list of list of integers (REQUIRED)
  - `group_probabilities`: list of floats (REQUIRED)
- **Requirements/Conventions**:
  - **Response**: OPTIONAL in the response (SHOULD be absent if there are no partial occupancies).
  - **Query**: Support for queries on this property is OPTIONAL. If supported, filters MAY support only a subset of comparison operators.
  - If present, the correct flag MUST be set in the list `structure_features` (see section 6.2.15. `structure_features`).
  - Client implementations MUST check its presence (as its presence changes the interpretation of the structure).
  - If present, it MUST be a list of dictionaries, each of which represents an assembly and MUST have the following two keys:
    - `sites_in_groups`: Index of the sites (0-based) that belong to each group for each assembly.
      Example: `[[1], [2]]`: two groups, one with the second site, one with the third.
      Example: `[[1,2], [3]]`: one group with the second and third site, one with the fourth.
    - `group_probabilities`: Statistical probability of each group. It MUST have the same length as `sites_in_groups`. It SHOULD sum to one. See below for examples of how to specify the probability of the occurrence of a vacancy. The possible reasons for the values not to sum to one are the same as already specified above for the concentration of each species, see section 6.2.13. `species`.
      - If a site is not present in any group, it means that it is present with 100 % probability (as if no assembly was specified).
      - A site MUST NOT appear in more than one group.
- **Examples** (for each entry of the assemblies list):
  - `{"sites_in_groups": [[0], [1]], "group_probabilities: [0.3, 0.7]]`: the first site and the second site never occur at the same time in the unit cell. Statistically, 30 % of the times the first site is present, while 70 % of the times the second site is present.
  - `{"sites_in_groups": [[1,2], [3]], "group_probabilities: [0.3, 0.7]]`: the second and third site are either present together or not present; they form the first group of atoms for this assembly. The second group is formed by the fourth site. Sites of the first group (the second and the third) are never present at the same time as the fourth site. 30 % of times sites 1 and 2 are present (and site 3 is absent); 70 % of times site 3 is present (and sites 1 and 2 are absent).
- **Notes**:
  - Assemblies are essential to represent, for instance, the situation where an atom can statistically occupy two different positions (sites).
  - By defining groups, it is possible to represent, e.g., the case where a functional molecule (and not just one atom) is either present or absent (or the case where it is present in two conformations)
  - Considerations on virtual alloys and on vacancies:

    In the special case of a virtual alloy, these specifications allow two different, equivalent ways of specifying them. For instance, for a site at the origin with 30 % probability of being occupied by Si, 50 % probability of being occupied by Ge, and 20 % of being a vacancy, the following two representations are possible:
  - Using a single species:
Using multiple species and the assemblies:

```
{
  "cartesian_site_positions": [[0,0,0], [0,0,0], [0,0,0]],
  "species_at_sites": ["Si", "Ge", "vac"],
  "species": {
    "Si": { "chemical_symbols": ["Si"], "concentration": [1.0] },
    "Ge": { "chemical_symbols": ["Ge"], "concentration": [1.0] },
    "vac": { "chemical_symbols": ["vacancy"], "concentration": [1.0] }
  },
  "assemblies": [
    {
      "sites_in_groups": [ [0], [1], [2] ],
      "group_probabilities": [0.3, 0.5, 0.2]
    }
  ]
// ...
}
```

- It is up to the database provider to decide which representation to use, typically depending on the internal format in which the structure is stored. However, given a structure identified by a unique ID, the API implementation MUST always provide the same representation for it.

- The probabilities of occurrence of different assemblies are uncorrelated. So, for instance in the following case with two assemblies:

```
{
  "assemblies": [
    {
      "sites_in_groups": [ [0], [1] ],
      "group_probabilities": [0.2, 0.8],
    },
    {
      "sites_in_groups": [ [2], [3] ],
      "group_probabilities": [0.3, 0.7]
    }
  ]
}
```

Site 0 is present with a probability of 20 % and site 1 with a probability of 80 %. These two sites are correlated (either site 0 or 1 is present). Similarly, site 2 is present with a probability of 30 % and site 3 with a probability of 70 %. These two sites are correlated (either site 2 or 3 is present).
However, the presence or absence of sites 0 and 1 is not correlated with the presence or absence of sites 2 and 3 (in the specific example, the pair of sites (0, 2) can occur with 0.2*0.3 = 6 % probability; the pair (0, 3) with 0.2*0.7 = 14 % probability; the pair (1, 2) with 0.8*0.3 = 24 % probability; and the pair (1, 3) with 0.8*0.7 = 56 % probability).

6.2.15. structure_features

- **Description**: A list of strings that flag which special features are used by the structure.
- **Type**: list of strings
- **Requirements/Conventions**:
  - **Response**: REQUIRED in the response (SHOULD be absent if there are no partial occupancies).
  - **Query**: MUST be a queryable property. Filters on the list MUST support all mandatory HAS-type queries. Filter operators for comparisons on the string components MUST support equality, support for other comparison operators are OPTIONAL.
  - MUST be an empty list if no special features are used.
  - MUST be sorted alphabetically.
  - If a special feature listed below is used, the list MUST contain the corresponding string.
  - If a special feature listed below is not used, the list MUST NOT contain the corresponding string.
- **List of strings used to indicate special structure features**
  - disorder: This flag MUST be present if any one entry in the species list has a chemical_symbols list that is longer than 1 element.
  - unknown_positions: This flag MUST be present if at least one component of the cartesian_site_positions list of lists has value null.
  - assemblies: This flag MUST be present if the assemblies list is present.

- **Examples**: A structure having unknown positions and using assemblies:

  ```json
  ["assemblies", "unknown_positions"]
  ```

6.3. Calculations Entries

"calculations" entries have the properties described above in section 6.1. Properties Used by Multiple Entry Types

6.4. References Entries

"references" entries describe bibliographic references. The following properties are used to provide the bibliographic details: *address, annotate, booktitle, chapter, crossref, edition, howpublished, institution, journal, key, month, note, number, organization, pages, publisher, school, series, title, type, volume, year: Meanings of these properties match the BibTeX specification, values are strings; *authors and editors: lists of person objects which are dictionaries with the following keys: *name: Full name of the person, REQUIRED. *firstname, lastname: Parts of the person's name, OPTIONAL. *doi and url: values are strings.

- **Requirements/Conventions**:
  - **Response**: Every references entry MUST contain at least one of the properties.
  - **Query**: Support for queries on any of these properties is OPTIONAL. If supported, filters MAY support only a subset of comparison operators.

Example:
6.5. Database-Provider-Specific Entry Types

Names of database-provider-specific entry types MUST start with database-provider-specific namespace prefix as given in Appendix 1. Database-provider-specific entry types MUST have all properties described above in section 6.1. Properties Used by Multiple Entry Types.

- **Requirements/Conventions for properties in database-provider-specific entry types:**
  - **Response**: OPTIONAL in the response.
  - **Query**: Support for queries on these properties are OPTIONAL. If supported, only a subset of filter operators MAY be supported.

6.6. Relationships Used by Multiple Entry Types

In accordance with section 3.5. Relationships, all entry types MAY use relationships to describe relations to other entries.

6.6.1. References

The "references" relationship is used to provide bibliographic references for any of the entry types. It relates an entry with any number of "references" entries.

If the response format supports inclusion of entries of a different type in the response, then the response SHOULD include all references-type entries mentioned in the response.

For example, for the JSON response format, the top-level "included" field should be used as per the JSON API 1.0 specification:
6.6.2. Calculations

Relationships with calculations MAY be used to indicate provenance where a structure is either an input to or an output of
Note: We intend to implement in a future version of this API a standardized mechanism to differentiate these two cases, thus allowing databases a common way of exposing the full provenance tree with inputs and outputs between structures and calculations.

At the moment the database providers are suggested to extend their API the way they choose, always using their database-provider-specific prefix in non-standardized fields.

- Requirements/Conventions for database-provider-specific properties of calculations entries:
  - **Response**: OPTIONAL in the response.
  - **Query**: Support for queries on these properties are OPTIONAL. If supported, only a subset of filter operators MAY be supported.

Appendix 1: Database-Provider-Specific Namespace Prefixes

This standard refers to database-provider-specific prefixes. These are assigned and included in this standard in the file providers.json.

API implementations SHOULD NOT make up and use new prefixes not included in this standard, but SHOULD rather work to get such prefixes included in a future revision of this API specification.

**Example:**

Database-provider-specific prefix: "exmpl"

Use as a field name in a response: `_exmpl_custom_field`

The initial underscore indicates an identifier that is under a separate namespace under the ownership of that organization. Identifiers prefixed with underscores will not be used for standardized names.

The database-provider-specific prefixes currently assigned are listed in the providers.json file provided in the main repository. This file serves as a machine-readable list of OPTiMaDe providers.

The content of the providers.json file complies with the default JSON format specification for API responses. In particular, the resource objects under the top-level data field are defined to be valid resource objects for the Links endpoint, see section 4.4.3. Provider Objects.

Note: Any provider wishing to be added to providers.json is kindly asked to suggest a change to this repository (using a pull request).

Appendix 2: The Filter Language EBNF Grammar.

```
(* BEGIN EBNF GRAMMAR Filter *)
(* The top-level 'filter' rule: *)

Filter = [Spaces], Expression ;

(* Values *)

Constant = String | Number ;

Value = String | Number | Property ;

(* Note: support for Property in Value is OPTIONAL *)

ValueList = [ Operator ], Value, { Comma, [ Operator ], Value } ;

(* Support for Operator in ValueList is OPTIONAL *)

ValueZip = [ Operator ], Value, Colon, [ Operator ], Value, {Colon, [ Operator ], Value} ;
```
valueZip = [ Operator ], value, Colon, [ Operator ], value, {Colon, [ Operator ], value} ;

(* Support for Operator in ValueZip is OPTIONAL *)

ValueZipList = ValueZip, { Comma, ValueZip } ;

(* Expressions *)

Expression = ExpressionClause, [ OR, Expression ] ;

ExpressionClause = ExpressionPhrase, [ AND, ExpressionClause ] ;

ExpressionPhrase = [ NOT ], ( Comparison | PredicateComparison | OpeningBrace, Expression, ClosingBrace |

Comparison = ConstantFirstComparison |

PropertyFirstComparison ;

(* Note: support for ConstantFirstComparison is OPTIONAL *)

PropertyFirstComparison = Property, ( 

ValueOpRhs |

KnownOpRhs |

FuzzyStringOpRhs |

SetOpRhs |

SetZipOpRhs ) ;

(* Note: support for SetZipOpRhs in Comparison is OPTIONAL *)

ConstantFirstComparison = Constant, ValueOpRhs ;

PredicateComparison = LengthComparison ;

ValueOpRhs = Operator, Value ;

KnownOpRhs = IS, ( KNOWN | UNKNOWN ) ;

FuzzyStringOpRhs = CONTAINS, String | STARTS, [ WITH ], String | ENDS, [ WITH ], String ;

SetOpRhs = HAS, ( [ Operator ], Value | ALL, ValueList | ANY, ValueList | ONLY, ValueList ) ;

(* Note: support for ONLY in SetOpRhs is OPTIONAL *)

(* Note: support for [ Operator ] in SetOpRhs is OPTIONAL *)

SetZipOpRhs = PropertyZipAddon, HAS, ( ValueZip | ONLY, ValueZipList | ALL, ValueZipList | ANY, ValueZipList |

LengthComparison = LENGTH, Property, Operator, Value ;

PropertyZipAddon = Colon, Property, {Colon, Property} ;

(* Property *)

Property = Identifier, { Dot, Identifier } ;

(* TOKENS *)

(* Separators: *)

OpeningBrace = '(', [Spaces] ;

ClosingBrace = ')', [Spaces] ;

Dot = '.', [Spaces] ;

Comma = ',', [Spaces] ;
Note: when implementing a parser according this grammar, the implementers MAY choose to construct a lexer that ignores all whitespace (spaces, tabs, newlines, vertical tabulation and form feed characters, as described in the grammar 'Space' definition), and use such a lexer to recognize language elements that are described in the (* TOKENS *) section of the grammar. In that case, the [Spaces] element should probably be removed from the Filter = [Spaces], Expression definition as well, and the remaining grammar rules could then be used as a parser generator (like yacc, bison, antlr) input.

**Appendix 3. Regular Expressions for OPTiMaDe Filter Tokens.**

The string below contains Perl-Compatible Regular Expressions to recognize identifiers, number, and string values as specified in this specification.
The strings below contain Extended Regular Expressions (EREs) to recognize identifiers, number, and string values as specified in this specification.

```plaintext
#BEGIN ERE identifiers
[a-z_][a-z_0-9]*
#END ERE identifiers

#BEGIN ERE numbers
[-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+)\
#END ERE numbers

#BEGIN ERE strings
"([\^\"\\]|\.)*"
#END ERE strings
```