OPTIMADE API specification v0.9.5

1. Introduction

As researchers create independent materials databases, much can be gained from retrieving data from multiple databases. However, the retrieval process is difficult if each database has a different API. This document defines a standard API for retrieving data from materials databases. This API was developed by the participants of the workshop "Open Databases Integration for Materials Design", held at the Lorentz Center in Leiden, Netherlands from 2016-10-24 to 2016-10-28.

It is the intent that the API in the present document adheres to the JSON API specification (with the exception that non-conformant responses can be generated if an API user specifically requests a non-standard response format.) In cases where specific restrictions are given in the JSON API specification (e.g., on format of Member Names, on return codes) that are stricter than what is formulated in this document, they are expected to be upheld by API implementations unless otherwise noted in this document.

The full present version number of the specification is shown as part of the top headline of this document.

2. Term definition

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 a database of materials information.
- Database-specific prefix: This specification defines a set of database-specific prefixes in its Appendix.
- Entry: a type of resource over which a query can be formulated using the API (structure, calculations).
- Property: anything that can be in the filtering of results.
- Field: an entity that can be requested as partial output from the API.
- ID: a unique identifier that specifies a specific resource in a database, that does not necessarily need to be immutable. It MUST NOT be a reserved word.
- Immutable ID: a unique identifier that specifies a specific resource in a database that MUST be immutable.
- Reserved words: the list of reserved words in this standard is: info.
- Property Types :
  - **string**, **integer**, **float**, **boolean**, **null value**: base data types as defined in more detail in section '5.1. Lexical tokens'.
  - **list**, **dictionary**: with the meaning they have in the JSON data-interchange format, i.e., respectively, an ordered list of elements (where each element can have a different type), or a

hash table (with the limitation that the hash key MUST be a string).

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 '4.4.1. 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

API responses MUST be returned in the format specified in the request. If no specific response format is specified in the request by use of the **response_format** URL query parameter (see below), the default response format is JSON API, which is compliant to the response format described by the JSON API specification. All endpoints MUST support at least the JSON API format. Each endpoint MAY OPTIONALLY support multiple formats, and declare these formats in their 'info' endpoints.

An API implementation MAY return other formats than as specified here. These can be implemented and documented however the implementor chooses. However, they MUST be prefixed by a database-specific prefix as defined in the Appendix.

Specifying a **response_format** URL query parameter that selects a response format different from JSON API allows the API specification to break conformance with the JSON API specification. Not only in response format, but also in, e.g., how content negotiation is implemented.

3.3.2. JSON API response schema: common fields

Every response MUST contain the following fields:

- **links**: a JSON API links object. Where,
  - **next**: is an URI that represents a suggested way to fetch the next set of results if not all were returned, either directly as string, or as link object which can contain the following members:
    - `href`: a string containing the link's URL.
    - `meta`: a meta object containing non-standard meta-information about the link.
  - The **next** field MUST be null or omitted if there is no additional data, or if there is no way to fetch additional data.
- The links object can OPTIONALLY contain the field
  - **base_url**: the URL of the api endpoint`"resource": {`

    ```
    "base_url": "http://example.com/optimade/v0.9/",
    "_example.com_db_version": "3.2.1"
    },
    ```
- **meta**:
  - **query**: information on the query that was requested.
    - It MUST be a dictionary with these items:
      - **representation**: a string with the part of the URL that follows the base URL.
  - **api_version**: a string containing the version of the API .
  - **time_stamp**: a string containing the date and time at which the query was executed, in ISO 8601 format. Times MUST be timezone-aware (i.e. MUST NOT be local times), in one of the formats allowed by ISO 8601 (i.e. either be in UTC, and then end with a Z, or indicate explicitly the offset).

    - **data_returned**: an integer containing the number of data returned for the query.
    - **more_data_available**: 'False' if all data for this query has been returned, and 'True' if not.
    - The dictionary MAY also include these OPTIONAL items:
        - **data_available**: an integer containing the total number of data available in the database.
        - **last_id**: a string containing the last ID returned.
        - **response_message**: OPTIONAL response string from the server.
        - Other OPTIONAL additional information *global to the query* that is not specified in this document, MUST start with a database-specific prefix as defined in the Appendix.
    - 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"
  }
  ...
  <additional response items>
}
```

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

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

- **include**: a list of JSON API Resource Objects related to the primary data contained in `data` that are included in this document.

If there were errors in producing the response all other fields can be skipped, and the following field MUST be present

- **errors**: a list of JSON API Error Objects

An example of a full response:

```
{
  "links": {
    "next": null,
    "base_url": {
      "href": "http://example.com/optimade/v0.9/",
      "meta": {
         "_example.com_db_version": "3.2.1"
      }
    }
  },
  "resource": {
  },
  "data": [...],
  "meta": {
    "query": {
      "represeation": "/structures?a=1&b=2"
    }
    "api_version": "v0.9",
    "time_stamp": "2007-04-05T14:30Z",
    "data_returned": "10",
    "data_available": "10",
    "last_id": "xy10",
    "more_data_available": "False"
  },
  "response_message": "OK",
  <OPTIONAL DB-specific meta_data, global to the query>
}
```

3.3.4. HTTP response status codes

Code	Message
200	OK (Successful operation)
400	Bad request (e.g., mistyped URL)
401	User does not have permission
403	Forbidden (the request was understood but not authorized)
404	Not found (e.g., database not found)
408	Request timeout because it is taking too long to process the query
410	The database has been moved
413	The response is too large
414	The request URI contains more than 2048 characters
418	Asked for a non-existent keyword
422	Database returned (200) but the translator failed

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

- a general filtering 'all' endpoint that can search all entry types
- an introspection 'info' endpoint
- a custom 'extensions' endpoint prefix

These endpoints are documented below.

4.1. Entry listing endpoints

Entry listing endpoints return a list of documents 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/v0.9/calculations

There MAY be multiple entry listing endpoints, depending on how many types of entries a database provides. Specific standard entry types are specified in a later section of this specification. 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-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 implementers MUST accept and handle them. To adhere to the requirement on implementation-specific URL query parameters of JSON API, 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'.

Standard OPTIONAL URL query parameters not in the JSON API specification:

- **response_format**: specifies which output format is requested. Specifically, the format string 'jsonapi' specifies the standard output format documented in this specification as the JSON API response format.
  Example: http://example.com/optimade/v0.9/structures/?response_format=xml
- **email_address**: specifies 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_limit**: sets a numerical limit on the number of entries returned. The API implementation MUST return no more than the number specified. It MAY return less. The database MAY have a maximum limit and not accept larger numbers (in which case an error code MUST be returned).

The default limit value is up to the API implementation to decide.
Example: http://example.com/optimade/v0.9/structures/?response_limit=100
- **response_fields**: specify 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-specific prefix><url_query_parameter_name>". These names adhere to the requirements on implementation-specific query parameters of JSON API since the database-specific prefixes contain an underscore (a LOW LINE character '_').

Example uses of custom URL query parameters include providing an access token for the request or 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
- http://example.com/optimade/v0.9/structures/?_exmpl_filter="elements all in [Al, Si, Ga]"

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 API 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 JSON API format every dictionary (Resource Object) needs the following fields

- **type**: field containing the type of the entry
- **id**: a string which together with the type uniquely identifies the object, this can be the local database ID
- **attributes**: a dictionary, containing key-value pairs representing the entries properties, and the following fields:
  - **local_id**: the entry's local database ID
  - **last_modified**: an ISO 8601 representing the entry's last modification time
  - **immutable_id**: an OPTIONAL field containing the entry's immutable ID db specific properties need to be prefixed by the db specific prefix

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 meta object that contains non-standard meta-information about the object
- **relationships**: a dictionary containing references to other Resource Objects as defined in JSON API Relationships Object

Example:

```
{
  ... <other response items> ...
  "data": [
    {
      "type": "structure",
      "id": "example.db:structs:0001",
      "attributes": {
        "formula": "Es2 O3",
        "local_id": "example.db:structs:0001",
        "url": "http://example.db/structs/0001",
        "immutable_id": "http://example.db/structs/0001@123",
        "last_modified": "2007-04-05T14:30Z"
      }
    },
    ...
    {
      "type": "structure",
      "attributes": {
        "formula": "Es2"
        "local_id": "example.db:structs:1234",
        "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 ID component to the URL of an entry listing endpoint. This will return properties for the entry with that ID.

If using the JSON API format, the ID component MUST be the content of the id field.

Note that entries cannot have an ID of 'info', as this would collide with the 'Info' endpoint (described in section '4.4. Info endpoints') for a given entry type.

Examples:

- http://example.com/optimade/v0.9/structures/exmpl:struct/3232823
- http://example.com/optimade/v0.9/calculations/exmpl:calc/232132

4.2.1. URL Query Parameters

The client MAY provide a set of additional URL query parameters also for this endpoint type. URL query parameters not recognized MUST be ignored. While the following URL query parameters are OPTIONAL for clients, API endpoints 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. 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" key MUST have only one entry.

4.3. General entry listing 'All' endpoint

The 'general entry listing endpoint' returns a list of entries representing all entries a database provides, regardless of type. This endpoint MUST be provided at the path "<base_url>/all" . The purpose of this endpoint is to allow more general searches across entry types. The general entry listing endpoint MUST accept both GET and a POST-type requests, with provided POST-type URL query parameters overriding duplicate URL query parameters provided as GET URL query parameters.

4.3.1. URL Query Parameters

The following URL query parameters MUST be recognized and handled: **filter**, **response_fields**, **response_format**, **response_limit**, **email_address**. The meaning of these URL query parameters are as defined above in section '4.1.1. URL Query Parameters'. Furthermore, custom OPTIONAL URL query parameters, also described above, are also allowed.

Example: http://example.com/optimade/v0.9/all/?response_fields=id,url&response_format=jsonapi

4.3.2. Response schema

The response for a general entry 'all' endpoint is the same as for 'entry listing' endpoint responses.

4.4. Info endpoints

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

Info endpoints are constructed by appending "**info**" to any of:

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/structures/info/)
3. the general entry listing endpoint (e.g., http://example.com/optimade/v0.9/all/info/)

These types and output content of these info endpoints are described in more detail in the

subsections below.

4.4.1. 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 response dictionary MUST include the following fields

- **type**: MUST be "info"
- **id**: "/"
- **attributes**: a dictionary containing the following fields:
  - **api_version**: Presently used version of the API.
  - **available_api_versions**: a dictionary where values are the base URLs for the versions of the API that are supported, and the keys are strings giving the full version number provided by that base URL. Provided base urls MUST adhere to the rules in section '3.1. Base URL'.
  - **formats**: a list of available output formats.
  - **entry_types_by_format**: Available entry endpoints as a function of output formats.

Example:

```
{
  ... <other response items> ...
  "data": [
    {
      "type": "info",
      "id": "/",
      "attributes": {
        "api_version": "v0.9",
        "available_api_versions": {
          "0.9.5": "http://db.example.com/optimade/v0.9/",
          "0.9.2": "http://db.example.com/optimade/v0.9.2/",
          "1.0.2": "http://db.example.com/optimade/v1.0/"
        },
        "formats": [
          "json",
          "xml"
        ],
        "entry_types_by_format": {
          "json": [
            "structure",
            "calculation"
          ],
          "xml": [
            "structure"
          ]
        },
        "available_endpoints": [
          "entry",
          "all",
          "info"
        ]
```

```
        }
      }
    ]
  }
```

4.4.2. Entry listing info endpoints

Entry listing info endpoints are of the form "<base_url>/<entry_type>/info/" (e.g., http://example.com/optimade/v0.9/structures/info/). 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 ID, each value is a description of the property, along with the units.
- **formats**: Output formats available for this type of entry.
- **output_fields_by_format**: Available output fields for this entry type as function of output format.

Example:

```
{
  ... <other response items> ...
  "data": [
    {
      "description": "a structure",
      "properties": {
        "nelements": {
          "description": "number of elements"
        },
        "unit": "MPa",
        ... <other property descriptions>
      },
      "formats": ["json", "xml"],
      "output_fields_by_format": {
        "json": ["nelements", ... ],
        "xml": ["nelements"]
      }
    }
  ]
}
```

4.5. Custom extension endpoints

API implementors can provide custom endpoints, in this form "<base_url>/extensions/<custom paths>".

5. API Filtering Format Specification

A filter language will be used in the 'filter=' component of the query string. The 'filter=' component

can be used to select a subset of records to be returned for a specific query. The value of the 'filter=' component MUST follow the filter language syntax described below.

The filter language MUST support at least the following features:

- flat filters with one level of "AND" and "OR" conjunctions.

5.1. Lexical tokens

The following tokens are used in the filter query component:

- **Property names** (see section '6. Entry list'): are to follow the identifier syntax of programming languages -- the first character MUST be a letter, the subsequent symbols MUST be alphanumeric; the underscore ("_", ASCII 95 dec (0x5F)) is considered to be a letter. Identifiers are case-sensitive. The length of the identifiers is not limited, except that the whole query SHOULD NOT be longer than the limits imposed by the URI specification.Examples of valid property names:
  - band_gap
  - cell_length_a
  - cell_volume
- Examples of incorrect property names:
  - 0_kvak (starts with a number);
  - "foo bar" (contains space; contains quotes)
- Identifiers that start with an underscore are specific to a database, and MUST be on the format of a database specific prefix as defined in the Appendix.Examples:
  - _exmpl_formula_sum (a property specific to that database)
  - _exmpl_band_gap
  - _exmpl_supercell
  - _exmpl_trajectory
  - _exmpl_workflow_id

- **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).
- **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):
  - ERE: [-+]?([0-9]+(\.[0-9]*)?|\.[0-9]+)([eE][-+]?[0-9]+)?
  - 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 an appropriate error message, indicating the cause of the error and an acceptable number range.
- Examples of valid numbers:
  - 12345, +12, -34, 1.2, .2E7, -.2E+7, +10.01E-10, 6.03e23, .1E1, -.1e1, 1.e-12, -.1e-12, 1000000000.E1000000000
- **A comma-separated list of** *incorrect* number examples (although they MAY contain correct numbers as substrings):
  - 1.234D12, .e1 , -.E1 , +.E2, 1.23E+++, +-123

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** are represented by usual mathematical relation symbols or by case-sensitive keywords. Currently the following operators are supported: =, !=, <=, >=, <, > for tests of number or string (lexicographical) equality, inequality, less-than, more-than, less, and more relations; AND, OR, NOT for logical conjunctions. The mathematical relations have higher priority than logical relations; relation NOT has higher priority than AND; relation AND has higher priority than OR. Thus, the expression 'a >= 0 AND NOT b < c OR c = 0' is interpreted as '((a >= 0) AND (NOT (b < c))) OR (c = 0)' if the expression was fully braced.
- The current API supports only one level of braces (no nested braces) : the expression 'a > b AND (a > 0 OR b > 0)' MUST be supported to allow changing of the priority of the logical operations.

5.2. The filter language syntax

Filtering expressions MUST follow the following EBNF grammar (fig 1.). Briefly, the filtering expression consists of the prefixing keyword 'filter=' that distinguishes it from other query string components that is followed by a Boolean expression on the search criteria. In the expression, desired properties are compared against search values; several such comparisons can be combined using AND, OR and NOT logical conjunctions with their usual semantics. Examples of the comparisons and logical expressions:

- spacegroup="P2"
- _exmpl_cell_volume<100.0
- _exmpl_bandgap > 5.0 AND _exmpl_molecular_weight < 350

The precedence (priority) of the operators MUST be as indicated in the list below:

1. The comparison operators ('<', '<=', '=', etc.) -- highest priority;
2. NOT
3. AND

4. OR -- lowest priority.

Thus, the expression 'NOT a > b OR c = 100 AND f = "C2 H6"' is interpreted as

'(NOT (a > b)) OR ((c = 100) AND (f = "C2 H6"))' when fully braced.

Fig. 1 The top-level rules of the Filter language grammar.

```
(* The top-level 'filter' rule: *)
Filter = Keyword, Expression;
(* Keywords *)
Keyword = "filter=" ;
(* Values *)
Value = Identifier | Number | String ;
(* The white-space: *)
Space = ' ' | '\t' ;
Spaces = Space, { Space } ;
(* Boolean relations: *)
AND = "AND" ; (* a short-hand for: AND = 'A', 'N', 'D' *)
NOT = "NOT" ;
OR = "OR" ;
(* Expressions *)
Expression = Term, [Spaces], [ OR, [Spaces], Expression ] ;
Term = Atom, [Spaces], [ AND, [Spaces], Term ] ;
Atom = [ NOT, [Spaces] ], ( Comparison |
           '(', [Spaces], AndComparison,
                [Spaces], { OR,
                [Spaces], AndComparison, [Spaces] }, ')' );
AndComparison = [ NOT, [Spaces] ], Comparison,
                [Spaces], { AND,
                [Spaces], [ NOT, [Spaces] ], Comparison, [Spaces] } ;
(* Comparison operator tokens: *)
Operator = '<', [ '=' ] | '>', [ '=' ] | '=' | '!', '=' ;
Comparison = Value, [Spaces], Operator, [Spaces], Value ;
```

The structure of tokens 'Identifier', 'Number', 'String' and 'Operator' are described above in section '5.1. Lexical tokens' and omitted here for brevity; a full length machine readable version of the grammar, including the definition of the lexical tokens, is available in the Materials-Consortia API Git repository (file grammars/flat-filters.ebnf).

Since the usual arithmetic expression syntax used for specifying filter expressions can contain characters that are not URL transparent, they MUST be URL-encoded before their use in a GET query. The specified order of escaping and encoding of the Filter language statements is the following:

1. special characters in string values MUST be escaped first as described above in the section "String values";
2. the resulting expression MUST be URL-encoded.

The extraction flow is obviously the opposite -- first the Filter string MUST be first URL-decoded, and after that string tokens MUST be unescaped to get values for comparisons.

When comparisons are performed, comparison operators '<', '<=', '=', '!=' and so on are interpreted either as numeric value comparisons or as string comparisons, depending on the type of the search parameter. The literal string value MUST be converted to the type of the search parameter (e.g., 'x > "0.0"' where x is a coordinate MUST treat this expression as numeric filter 'x > 0', and 's = 0' search against text parameter 's' MUST perform string comparison as in 's = "0"'). Strings are converted to numbers using the token syntax specified in section '5.1. Lexical tokens', p. "Numeric values"; numbers SHOULD be converted to strings using the libc '%g' format. In all cases the application MUST return a warning when conversion was performed and specify the actual search values that were used.

For comparisons of two parameters (e.g. 'x > y') or two constants (1 = 1) both compared values MUST be of the same type (i.e. both MUST be either numeric or string values); implementation MUST return error code if this is not the case.

Examples of syntactically correct filter strings:

- filter=_exmpl_melting_point<300 AND nelements=4 AND elements="Si,O2"

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%2CO2%22&response_format=xml

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.
- Requirements/Conventions:
  - IDs MUST be URL-safe; in particular, they MUST NOT contain commas.
  - Reasonably short IDs are encouraged and SHOULD NOT be longer than 255 characters.
- Examples:
  - db/1234567
  - cod/2000000
  - cod/2000000@1234567
  - nomad/L1234567890

6.1.2. modification_date

- Description: A date representing when the entry was last modified.
- Requirements/Conventions: ISO 8601 format
- Example:
  - 2007-04-05T14:30Z
- Querying: Date-time queries are permitted (RFC 3339).

6.1.3. database-specific properties

- Description: Databases are allowed to insert database-specific entries in the output of both standard entry types, and database-specific entry types.
- Requirements/Conventions: these MUST be prefixed by a database-specific prefix as defined in the Appendix.
- Examples:
  - _exmpl_formula_sum
  - _exmpl_band_gap,
  - _exmpl_supercell
  - _exmpl_trajectory
  - _exmpl_workflow_id

6.2. 'Structure' Entries

Structure entries have the properties described above in "Properties Used by Multiple Entry Types", as well as the following properties:

6.2.1. elements

- Description: names of elements found in the structure.
- Requirements/Conventions:
- Examples:
  - "Si"
  - "Si,Al,O"
- Querying: the conjunction means "AND", i.e., all records pertaining to materials containing Si, Al **and** O, and possibly other elements, MUST be returned; use 'nelements=3' to specify **exactly** 3 elements; (element="Si,Al,O" means you want structures with at least the 3 elements, and it MUST contain Si, Al AND O).

6.2.2. nelements

- Description: The number of elements found in a structure.
- Requirements/Conventions: an integer
- Example: 1
- Querying: Use numerical operators, as defined in the filtering section above.
  - Examples:
    - return only entities that have exactly 4 elements: " nelements=4"

        - query for structures that have between 2 and 7 elements: "nelements>=2 AND nelements<=7"

6.2.3. chemical_formula

- Description: The chemical formula for a structure.
- Requirements/Conventions:
    - The formula MUST be **reduced**.
    - Element names MUST be with proper capitalization (Si, not SI for "silicon").
    - The order in which elements are specified SHOULD NOT be significant (e.g., "O2Si" is equivalent to "SiO2").
    - No spaces or separators are allowed.

6.2.4. formula_prototype

- Description: The formula prototype obtained by sorting elements by the occurrence number in the **reduced** chemical formula and replace them with subsequent alphabet letters A, B, C and so on.

6.3. 'Calculation' entries

Calculation entries have the properties described above in "Properties Used by Multiple Entry Types".

Appendix: Database-specific namespace prefixes

This standard refers to database-specific prefixes. These are assigned and included in this standard. The presently assigned prefixes are:

- _exmpl_: used for examples, not to be assigned to a real database
- _aflow_: aflow.org
- _cam_: Cambridge databases
- _cod_: crystallography open database
- _mcloud_: materialscloud.org
- _mp_: materialsproject.org
- _nmd_: nomad laboratory
- _omdb_: open materials database
- _oqmd_: open quantum materials database
- _pcod_: predicted crystallography open database
- _tcod_: theoretical crystallography open database

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.

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

standardized names.