

# Open Geospatial Consortium

Submission Date: 2022-04-15

Approval Date: <yyyy-mm-dd>

Publication Date: <yyyy-mm-dd>

External identifier of this OGC® document: <[http://www.opengis.net/doc/\[doc-type\]/standard/m.n](http://www.opengis.net/doc/[doc-type]/standard/m.n)>

Internal reference number of this OGC® document: 21-050r1

Version: 2.0

Category: OGC® Community Standard

Editor: Zarr Developers

## Zarr Storage Specification 2.0 Community Standard

### Copyright notice

Copyright © 2022 Open Geospatial Consortium, Zarr Developers  
To obtain additional rights of use, visit <http://www.opengeospatial.org/legal/>.

### Warning

This document is not an OGC Standard. This document is distributed for review and comment. This document is subject to change without notice and may not be referred to as an OGC Standard.

Recipients of this document are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: OGC® Community Standard  
Document subtype:  
Document stage: Draft  
Document language: English

## License Agreement

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD.

THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER'S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR's sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to indicate compliance with any LICENSOR standards or specifications. This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

# Preface

## 1. Introduction to Zarr

Multidimensional array data (a.k.a. N-dimensional arrays, ND-arrays, “tensors”) is ubiquitous in scientific research and engineering. Zarr is an open-source specification for the storage of ND-arrays and associated metadata. Zarr stores metadata using .json text files and array data as [optionally] compressed binary chunks. The Zarr specification details the contents and layout of these elements. Zarr can store data into any storage system that can be described as a key/value store. In a standard filesystem, the keys are filenames within a directory hierarchy, and the values are the file contents. In a cloud object store (e.g., Amazon S3), the keys are the object IDs and the values are the object data. This flexibility allows implementations to experiment with novel storage technologies while maintaining a uniform API for downstream libraries and users.

Zarr arose in genomics research in 2016. It was created by Alistair Miles of Oxford as a library optimized for massively parallel array analytics. It has since grown into a community project with a range of developers and users from fields such as genomics, bioimaging, astronomy, physics, quantitative finance, oceanography, atmospheric science, climate science, and geospatial imaging. Because it can represent very large array datasets in a simple, scalable way, and is compatible with cloud object storage, Zarr is an ideal format for analysis-ready geospatial data in the cloud. A prominent example is the [Google Cloud CMIP6 Public Dataset](#), which currently comprises over 800 TB of Zarr data. While Zarr is not inherently a geospatial-specific format, because of its rapid growth and adoption in geospatial and related fields, we are proposing it as an OGC community standard.

The Zarr project is open source and practices open development on GitHub. The project is governed by a [steering council](#). [NumFocus](#) has been the fiscal and legal sponsor of Zarr since 2019. In 2020, Zarr received a Chan-Zuckerberg [Essential Open Source Software](#) grant.

## 2. The Zarr V2 Specification

This community standard refers to the Zarr V2 Specification. The Zarr V2 Specification is hosted on the Zarr website at <https://zarr.readthedocs.io/en/stable/spec/v2.html>. **The Zarr V2 Specification is the OGC Community Standard.** Everything that follows is a non-normative, informal description of Zarr usage written for the benefit of the geospatial community.

Work is underway on a Zarr V3 specification, which will include a mechanism for optional spec extensions. A new OGC standard will be filed for Zarr V3 in the future.

### 3. Zarr Storage Systems

The Zarr V2 Specification is agnostic regarding the underlying storage system. The specification states:

*“A Zarr array can be stored in any storage system that provides a key/value interface, where a key is an ASCII string and a value is an arbitrary sequence of bytes, and the supported operations are read (get the sequence of bytes associated with a given key), write (set the sequence of bytes associated with a given key) and delete (remove a key/value pair).”*

For the purposes of this OGC community standard, it is helpful to enumerate some of the most common storage systems (“Zarr Stores”) that are supported by most Zarr implementations.

- **Directory Store** - A directory within a filesystem can be a Zarr Store. The Zarr keys are the paths relative to the directory root. Chunk keys are separated with the “.” character, such that the chunks for each array all live within a single directory.
- **Nested Directory Store** - Same as Directory Store, but the chunk keys are separated with a “/”, leading to a nesting of chunk keys for each array.
- **Zipfile Store** - A Directory Store that has been zipped into a .ZIP file (<https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT>). This allows a Zarr Array or Zarr Group to be stored in a single logical file.
- **Object Store** - Cloud object storage (e.g., Amazon S3, Google Cloud Storage, Azure Blob Storage, etc.) can act as a Zarr Store. The store location is specified as a bucket and key prefix. The Zarr keys are paths relative to the prefix.
- **HTTP Store** - An HTTP[S] URL can be the root of a Zarr Store. The Zarr keys are the paths relative to the root URL. This type of store is most commonly used in a read-only configuration, e.g., to serve public data over the internet. The ability to access Zarr over vanilla HTTP means that, rather than just a file format, Zarr can become a generic protocol for remote data access, with the underlying data served either statically (from files) or dynamically (generated on demand). This has been demonstrated by the [Xpublish](#) project.

Many other stores have been implemented (e.g., DBM, MongoDB, Redis). The ability to easily experiment with new storage systems is a key part of the Zarr design.

### 4. Geospatial Data in Zarr

Zarr is not inherently a geospatial format, and thus the Zarr V2 Specification makes no reference to any geospatial concepts. However, several common practices have emerged regarding geospatial data. The forthcoming [GeoZarr](#) specification will formalize these

best practices into a more explicit OGC standard. For the purposes of this community standard, here we aim to simply document existing practices.

## 4.1 Named Dimensions

In the Zarr V2 Specification, arrays have a shape, encoded as a tuple of integers, e.g., (100, 200, 300). The dimensionality of the array is determined by the length of the tuple. These dimensions do not have names.

An ad-hoc convention for named dimension emerged from the Xarray python library (<http://xarray.pydata.org/en/stable/internals/zarr-encoding-spec.html>). The names of dimensions of a Zarr Array are encoded within the Array’s custom attributes (“`.zattrs`”) under the “`_ARRAY_DIMENSIONS`” key. The value of this key is a tuple of strings, e.g., (“`time`”, “`lat`”, “`lon`”). Downstream applications which can leverage named dimensions can parse this information in order to define dimension names for each Array. This approach to named dimensions is optional and fully compatible with the Zarr V2 Specification.

*We recommend geospatial data stored in Zarr to use this named dimension convention to identify dimension names.*

## 4.2 NetCDF / CF Data Model

Because Zarr is a generic container for multidimensional arrays, similar to HDF5, together with 4.1 (Named Dimensions) Zarr can be used to store the [NetCDF data model](#). The semantic mapping from the NetCDF Data Model to Zarr Data Model is as follows

NetCDF Data Model	Zarr V2 Data Model
File	Store
Group	Group
Variable	Array
Attribute	User Attribute
Dimension	Not supported natively. (See 4.1)

Note that this mapping excludes the NetCDF concept of “user-defined types”, which are not supported by Zarr.

The [Climate / Forecast \(CF\) Conventions](#) sit on top of the NetCDF data model, defining required and recommended metadata to be placed in the *Attribute* fields. Software which

understands these conventions and can read both formats can thus interoperate with both NetCDF and properly encoded Zarr dataset using the same data model. The python [Xarray](#) package is a primary example.

### 4.3 NCZarr

Beginning with NetCDF-C version 4.8.0, Unidata introduced [experimental Zarr support](#) into the NetCDF-C library. This was accomplished via creating a new specification - [NCZarr](#) - which is “similar to, but not identical with the Zarr Version 2 Specification.” Specifically, NCZarr adds two additional metadata files (“`.nczarray`” and “`.nczattr`”), which are not part of the Zarr V2 Spec. Since NCZarr stores are not fully compatible and interoperable with Zarr V2, *this community standard excludes NCZarr*. Work is ongoing to reconcile NCZarr and the architectural reasons that motivated its development with the forthcoming Zarr V3 Specification.

Fortunately, the NetCDF-C library also supports reading / writing of data using the simpler Named Dimension convention described in 4.1.

### 4.4 Coordinate Reference Systems

Since much existing Zarr geospatial data is from global and regional climate / weather models, rectangular or curvilinear lat / lon coordinates are common, without explicit reference to a CRS.

For geospatial raster datasets stored in Zarr, there are currently two different approaches being used to explicitly encode CRS information in Zarr

#### 4.4.1 CF Conventions Approach

Leveraging the NetCDF / CF compatibility described in 4.2, it is possible to store Coordinate Reference System CRS information about arrays inside Zarr groups. The relevant CF conventions regarding CRS are described in [Section 5.6](#) of the CF Conventions document, specifically, the “`grid_mapping`” variable and associated “`crs_wkt`” attribute. This is the approach is endorsed by the forthcoming GeoZarr spec.

#### 4.4.2 GDAL Approach

In implementing the Zarr driven for GDAL, the developers took a different approach: CRS is stored as a nested JSON data structure in the “`_CRS`” attribute on a Zarr array:

*“GDAL uses a `_CRS` attribute that is a dictionary that may contain one or several of the following keys: `url` (using a OGC CRS URL), `wkt` (WKT:2019 used by default on writing, WKT1 also supported on reading.), `projjson`. On reading, it will use `url` by default, if not found will fallback to `wkt` and then `projjson`.”*

The forthcoming GeoZarr Spec should seek to reconcile these discrepancies in CRS encoding in Zarr.

## 4.5 Overviews / Pyramids / Multiscale Data

A common need in geospatial raster data is to store the same image at multiple different resolutions. The Zarr V2 specification does not explicitly address this scenario; however, given the flexibility of the format, it can easily be accommodated. A convention called [Multiscale Arrays 0.1](#) was defined by the bioimaging community. This convention specifies how multiscale arrays can be stored, described, and discovered within standard Zarr Groups. Carbonplan leveraged this convention to develop a new toolkit for data-driven maps, which includes a [Javascript front end](#) and a [python utility](#) for generating image pyramids. It is recommended that new applications that require multiscale data attempt to follow this existing convention when feasible.

The forthcoming Zarr V3 specification will likely address multiscale explicitly via an extension mechanism.

## 5. Relationship of Zarr to other OGC standards

Zarr is a new, standalone standard, without dependencies on existing OGC standards. However, because of its generic nature, other OGC standards could easily be implemented on top of Zarr. For example:

- Zarr may be a container format for Coverage data made available to a Web Coverage Service, similar to the OGC CF-netCDF 3.0 encoding using GML Coverage Application Schema (OGC 14-100r2);
- The NetCDF library supports Zarr as a storage container since version 4.8.0; Unidata developed a standard called [NCZarr](#) which specifies how to encode the NetCDF data model into Zarr; and
- WKT representation of coordinate reference systems can be placed in Zarr metadata.

## 6. Alignment of Zarr with the OGC Standards Baseline

As a generic storage format for multidimensional array data + metadata, Zarr will play a similar role to HDF5 within the OGC ecosystem. Many different types of data and metadata can be stored using Zarr, targeting a wide range of storage technologies (filesystems, databases, cloud object store). We anticipate Zarr to be particularly popular for cloud-native data storage and processing. Also, because Zarr can store arbitrarily large arrays, without constraints related to individual file sizes, it will likely prove useful for creating homogeneous analysis-ready datasets from many individual granules

Zarr has already been adopted by several OGC communities as a format for cloud-optimized, analysis-ready geospatial data. Examples at the time of publication include the following.

- Climate Science: The [CMIP6 Google Cloud Public Dataset](#)
- Oceanography: The [ECCOV4r3 Ocean State Estimate](#)
- Atmospheric Science: [Global cloud-resolving aquaplanet simulations with the System for Atmospheric Modeling](#)



# Zarr storage specification version 2

## Zarr storage specification version 2

This document provides a technical specification of the protocol and format used for storing Zarr arrays. The key words “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](#).

### Status

This specification is the latest version. See [Specifications](#) for previous versions.

### Storage

A Zarr array can be stored in any storage system that provides a key/value interface, where a key is an ASCII string and a value is an arbitrary sequence of bytes, and the supported operations are read (get the sequence of bytes associated with a given key), write (set the sequence of bytes associated with a given key) and delete (remove a key/value pair).

For example, a directory in a file system can provide this interface, where keys are file names, values are file contents, and files can be read, written or deleted via the operating system. Equally, an S3 bucket can provide this interface, where keys are resource names, values are resource contents, and resources can be read, written or deleted via HTTP.

Below an “array store” refers to any system implementing this interface.

### Arrays

#### Metadata

Each array requires essential configuration metadata to be stored, enabling correct interpretation of the stored data. This metadata is encoded using JSON and stored as the value of the “.zarray” key within an array store.

The metadata resource is a JSON object. The following keys **MUST** be present within the object:

**zarr\_format**

An integer defining the version of the storage specification to which the array store adheres.

### **shape**

A list of integers defining the length of each dimension of the array.

### **chunks**

A list of integers defining the length of each dimension of a chunk of the array. Note that all chunks within a Zarr array have the same shape.

### **dtype**

A string or list defining a valid data type for the array. See also the subsection below on data type encoding.

### **compressor**

A JSON object identifying the primary compression codec and providing configuration parameters, or `null` if no compressor is to be used. The object MUST contain an `"id"` key identifying the codec to be used.

### **fill\_value**

A scalar value providing the default value to use for uninitialized portions of the array, or `null` if no fill\_value is to be used.

### **order**

Either "C" or "F", defining the layout of bytes within each chunk of the array. "C" means row-major order, i.e., the last dimension varies fastest; "F" means column-major order, i.e., the first dimension varies fastest.

### **filters**

A list of JSON objects providing codec configurations, or `null` if no filters are to be applied. Each codec configuration object MUST contain a `"id"` key identifying the codec to be used.

Other keys MUST NOT be present within the metadata object.

For example, the JSON object below defines a 2-dimensional array of 64-bit little-endian floating point numbers with 10000 rows and 10000 columns, divided into chunks of 1000 rows and 1000 columns (so there will be 100 chunks in total arranged in a 10 by 10 grid). Within each chunk the data are laid out in C contiguous order. Each chunk is encoded using a delta filter and compressed using the Blosc compression library prior to storage:

```

{
  "chunks": [
    1000,
    1000
  ],
  "compressor": {
    "id": "blosc",
    "cname": "lz4",
    "clevel": 5,
    "shuffle": 1
  },
  "dtype": "<f8",
  "fill_value": "NaN",
  "filters": [
    {"id": "delta", "dtype": "<f8", "astype": "<f4"}
  ],
  "order": "C",
  "shape": [
    10000,
    10000
  ],
  "zarr_format": 2
}

```

## Data type encoding

Simple data types are encoded within the array metadata as a string, following the [NumPy array protocol type string \(typestr\) format](#). The format consists of 3 parts:

- One character describing the byteorder of the data ("**<**": little-endian; "**>**": big-endian; "**|**": not-relevant)
- One character code giving the basic type of the array ("**b**": Boolean (integer type where all values are only True or False); "**i**": integer; "**u**": unsigned integer; "**f**": floating point; "**c**": complex floating point; "**m**": timedelta; "**M**": datetime; "**S**": string (fixed-length sequence of char); "**U**": unicode (fixed-length sequence of Py\_UNICODE); "**V**": other (void \* - each item is a fixed-size chunk of memory))
- An integer specifying the number of bytes the type uses.

The byte order **MUST** be specified. E.g., "**<f8**", "**>i4**", "**|b1**" and "**|S12**" are valid data type encodings.

For datetime64 ("M") and timedelta64 ("m") data types, these **MUST** also include the units within square brackets. A list of valid units and their definitions are given in the [NumPy documentation on Datetimes and Timedeltas](#). For example, "**<M8[ns]**" specifies a datetime64 data type with nanosecond time units.

Structured data types (i.e., with multiple named fields) are encoded as a list of lists, following [NumPy array protocol type descriptions \(descr\)](#). Each sub-list has the form `[fieldname, datatype, shape]` where `shape` is optional. `fieldname` is a string, `datatype` is a string specifying a simple data type (see above), and `shape` is a list of integers specifying subarray shape. For example, the JSON list below defines a data type composed of three single-byte unsigned integer fields named “r”, “g” and “b”:

```
[["r", "|u1"], ["g", "|u1"], ["b", "|u1"]]
```

For example, the JSON list below defines a data type composed of three fields named “x”, “y” and “z”, where “x” and “y” each contain 32-bit floats, and each item in “z” is a 2 by 2 array of floats:

```
[["x", "<f4"], ["y", "<f4"], ["z", "<f4", [2, 2]]]
```

Structured data types may also be nested, e.g., the following JSON list defines a data type with two fields “foo” and “bar”, where “bar” has two sub-fields “baz” and “qux”:

```
[["foo", "<f4"], ["bar", [{"baz", "<f4"}, {"qux", "<i4"}]]]
```

## Fill value encoding

For simple floating point data types, the following table MUST be used to encode values of the “fill\_value” field:

Value	JSON encoding
Not a Number	"NaN"
Positive Infinity	"Infinity"
Negative Infinity	"-Infinity"

If an array has a fixed length byte string data type (e.g., `"|S12"`), or a structured data type, and if the fill value is not null, then the fill value MUST be encoded as an ASCII string using the standard Base64 alphabet.

## Chunks

Each chunk of the array is compressed by passing the raw bytes for the chunk through the primary compression library to obtain a new sequence of bytes comprising the compressed chunk data. No header is added to the compressed bytes or any other modification made. The internal structure of the compressed bytes will depend on which primary compressor was used. For example, the [Blosc compressor](#) produces a sequence of bytes that begins with a 16-byte header followed by compressed data.

The compressed sequence of bytes for each chunk is stored under a key formed from the index of the chunk within the grid of chunks representing the array. To form a string key for a chunk, the indices are converted to strings and concatenated with the period character (“.”) separating each index. For example, given an array with shape (10000, 10000) and chunk shape (1000, 1000) there will be 100 chunks laid out in a 10 by 10 grid. The chunk with indices (0, 0) provides data for rows 0-1000 and columns 0-1000 and is stored under the key “0.0”; the chunk with indices (2, 4) provides data for rows 2000-3000 and columns 4000-5000 and is stored under the key “2.4”; etc.

There is no need for all chunks to be present within an array store. If a chunk is not present then it is considered to be in an uninitialized state. An uninitialized chunk **MUST** be treated as if it was uniformly filled with the value of the “fill\_value” field in the array metadata. If the “fill\_value” field is `null` then the contents of the chunk are undefined.

Note that all chunks in an array have the same shape. If the length of any array dimension is not exactly divisible by the length of the corresponding chunk dimension then some chunks will overhang the edge of the array. The contents of any chunk region falling outside the array are undefined.

## Filters

Optionally a sequence of one or more filters can be used to transform chunk data prior to compression. When storing data, filters are applied in the order specified in array metadata to encode data, then the encoded data are passed to the primary compressor. When retrieving data, stored chunk data are decompressed by the primary compressor then decoded using filters in the reverse order.

## Hierarchies

### Logical storage paths

Multiple arrays can be stored in the same array store by associating each array with a different logical path. A logical path is simply an ASCII string. The logical path is used to form a prefix for keys used by the array. For example, if an array is stored at logical path “foo/bar” then the array

metadata will be stored under the key “foo/bar/.zarray”, the user-defined attributes will be stored under the key “foo/bar/.zattrs”, and the chunks will be stored under keys like “foo/bar/0.0”, “foo/bar/0.1”, etc.

To ensure consistent behaviour across different storage systems, logical paths **MUST** be normalized as follows:

- Replace all backward slash characters (“\”) with forward slash characters (“/”)
- Strip any leading “/” characters
- Strip any trailing “/” characters
- Collapse any sequence of more than one “/” character into a single “/” character

The key prefix is then obtained by appending a single “/” character to the normalized logical path.

After normalization, if splitting a logical path by the “/” character results in any path segment equal to the string “.” or the string “..” then an error **MUST** be raised.

N.B., how the underlying array store processes requests to store values under keys containing the “/” character is entirely up to the store implementation and is not constrained by this specification. E.g., an array store could simply treat all keys as opaque ASCII strings; equally, an array store could map logical paths onto some kind of hierarchical storage (e.g., directories on a file system).

## Groups

Arrays can be organized into groups which can also contain other groups. A group is created by storing group metadata under the “.zgroup” key under some logical path. E.g., a group exists at the root of an array store if the “.zgroup” key exists in the store, and a group exists at logical path “foo/bar” if the “foo/bar/.zgroup” key exists in the store.

If the user requests a group to be created under some logical path, then groups **MUST** also be created at all ancestor paths. E.g., if the user requests group creation at path “foo/bar” then groups **MUST** be created at path “foo” and the root of the store, if they don’t already exist.

If the user requests an array to be created under some logical path, then groups **MUST** also be created at all ancestor paths. E.g., if the user requests array creation at path “foo/bar/baz” then groups must be created at path “foo/bar”, path “foo”, and the root of the store, if they don’t already exist.

The group metadata resource is a JSON object. The following keys **MUST** be present within the object:

## `zarr_format`

An integer defining the version of the storage specification to which the array store adheres.

Other keys **MUST NOT** be present within the metadata object.

The members of a group are arrays and groups stored under logical paths that are direct children of the parent group's logical path. E.g., if groups exist under the logical paths "foo" and "foo/bar" and an array exists at logical path "foo/baz" then the members of the group at path "foo" are the group at path "foo/bar" and the array at path "foo/baz".

## Attributes

An array or group can be associated with custom attributes, which are simple key/value items with application-specific meaning. Custom attributes are encoded as a JSON object and stored under the ".zattrs" key within an array store. The ".zattrs" key does not have to be present, and if it is absent the attributes should be treated as empty.

For example, the JSON object below encodes three attributes named "foo", "bar" and "baz":

```
{
  "foo": 42,
  "bar": "apples",
  "baz": [1, 2, 3, 4]
}
```

## Examples

### Storing a single array

Below is an example of storing a Zarr array, using a directory on the local file system as storage.

Create an array:

```
>>> import zarr
>>> store = zarr.DirectoryStore('data/example.zarr')
>>> a = zarr.create(shape=(20, 20), chunks=(10, 10), dtype='i4',
...                 fill_value=42, compressor=zarr.Zlib(level=1),
...                 store=store, overwrite=True)
```

No chunks are initialized yet, so only the ".zarray" and ".zattrs" keys have been set in the store:



```
>>> import os
>>> sorted(os.listdir('data/example.zarr'))
['.zarray']
```

Inspect the array metadata:

```
>>> print(open('data/example.zarr/.zarray').read())
{
  "chunks": [
    10,
    10
  ],
  "compressor": {
    "id": "zlib",
    "level": 1
  },
  "dtype": "<i4",
  "fill_value": 42,
  "filters": null,
  "order": "C",
  "shape": [
    20,
    20
  ],
  "zarr_format": 2
}
```

Chunks are initialized on demand. E.g., set some data:

```
>>> a[0:10, 0:10] = 1
>>> sorted(os.listdir('data/example.zarr'))
['.zarray', '0.0']
```

Set some more data:

```
>>> a[0:10, 10:20] = 2
>>> a[10:20, :] = 3
>>> sorted(os.listdir('data/example.zarr'))
['.zarray', '0.0', '0.1', '1.0', '1.1']
```

Manually decompress a single chunk for illustration:



The metadata resource for the root group has been created:

```
>>> import os
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup']
```

Inspect the group metadata:

```
>>> print(open('data/group.zarr/.zgroup').read())
{
  "zarr_format": 2
}
```

Create a sub-group:

```
>>> sub_grp = root_grp.create_group('foo')
```

What has been stored:

```
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup', 'foo']
>>> sorted(os.listdir('data/group.zarr/foo'))
['.zgroup']
```

Create an array within the sub-group:

```
>>> a = sub_grp.create_dataset('bar', shape=(20, 20), chunks=(10, 10))
>>> a[:] = 42
```

Set a custom attributes:

```
>>> a.attrs['comment'] = 'answer to life, the universe and everything'
```

What has been stored:

```
>>> sorted(os.listdir('data/group.zarr'))
['.zgroup', 'foo']
>>> sorted(os.listdir('data/group.zarr/foo'))
['.zgroup', 'bar']
>>> sorted(os.listdir('data/group.zarr/foo/bar'))
['.zarray', '.zattrs', '0.0', '0.1', '1.0', '1.1']
```

Here is the same example using a Zip file as storage:

```
>>> store = zarr.ZipStore('data/group.zip', mode='w')
>>> root_grp = zarr.group(store)
>>> sub_grp = root_grp.create_group('foo')
>>> a = sub_grp.create_dataset('bar', shape=(20, 20), chunks=(10, 10))
>>> a[:] = 42
>>> a.attrs['comment'] = 'answer to life, the universe and everything'
>>> store.close()
```

What has been stored:

```
>>> import zipfile
>>> zf = zipfile.ZipFile('data/group.zip', mode='r')
>>> for name in sorted(zf.namelist()):
...     print(name)
.zgroup
foo/.zgroup
foo/bar/.zarray
foo/bar/.zattrs
foo/bar/0.0
foo/bar/0.1
foo/bar/1.0
foo/bar/1.1
```

## Changes

### Version 2 clarifications

The following changes have been made to the version 2 specification since it was initially published to clarify ambiguities and add some missing information.

- The specification now describes how bytes fill values should be encoded and decoded for arrays with a fixed-length byte string data type ([#165](#), [#176](#)).
- The specification now clarifies that units must be specified for `datetime64` and `timedelta64` data types ([#85](#), [#215](#)).
- The specification now clarifies that the `'zattrs'` key does not have to be present for either arrays or groups, and if absent then custom attributes should be treated as empty.

- The specification now describes how structured datatypes with subarray shapes and/or with nested structured data types are encoded in array metadata ([#111](#), [#296](#)).

## Changes from version 1 to version 2

The following changes were made between version 1 and version 2 of this specification:

- Added support for storing multiple arrays in the same store and organising arrays into hierarchies using groups.
- Array metadata is now stored under the “.zarray” key instead of the “meta” key.
- Custom attributes are now stored under the “.zattrs” key instead of the “attrs” key.
- Added support for filters.
- Changed encoding of “fill\_value” field within array metadata.
- Changed encoding of compressor information within array metadata to be consistent with representation of filter information.