SciDataContainer

Release 1.0.0

Reinhard Caspary, Sven Kleinert

May 31, 2023

CONTENTS

This documentation describes a lean container file format for the storage of scientific data in a way compliant to the FAIR principles of modern research data management. The standardized data container provides maximum flexibility and minimal restrictions. It is operating system independent and may be stored as local file as well as uploaded to a data storage server.

Data containers may be built and accessed using standard operating system tools. However, specialized tools make the workflow much more convenient. A Python library is already available, others will follow. Furthermore, a native GUI application for Microsoft Windows is on the way. All source code is available on GitHub.

This is a project of the cluster of Excellence PhoenixD funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2122, Project ID 390833453).

The data storage server is currently only available for PhoenixD members. However, we intend to make the whole project including the server publicly available. If you are from outside PhoenixD and wish to get early access, you are welcome. Please contact us.

CHAPTER

ONE

DATA CONTAINER CONCEPT

The basic concept of the data container is that it keeps the raw dataset, parameter data and meta data together. Parameter data is every data which scientists traditionally record in lab books like a description of the test setup, measurement settings, simulation parameters or evaluation parameters. The intention behind the container concept is to make datasets self-contained.

Each data container is identified by a UUID. The **Container** file is a ZIP package file. The data in the container is stored in **Items** (files in ZIP package), which are organized in **Parts** (folders in ZIP package). The standard file extension of a container file is `.zdc`.

There are no restrictions regarding data formats inside the container, but items should be stored in the JSON format, whenever possible. This makes the data readable for humans as well as machines. Furthermore, it allows to inspect, use and even create data container files with the tools provided by the operating system without any special software. We call the keys of JSON objects data **Attributes**.

Only the two items `content.json` and `meta.json` are required and must be located in the root part of the container. The optional root item `license.txt` may be used to store the text of the license for the dataset.

The data payload of a container consisting of the dataset and the parameter data should be stored in certain parts of the container. Although there are no restrictions in using parts, you should restrict yourself to a set of *suggested parts*.

1.1 Container Parameters

The parameters describing the container itself are stored in the required root item `content.json`, which contains a single JSON object. The following set of attributes is currently defined for this item:

- `uuid`: required UUID
- `replaces`: optional UUID of the predecessor of this dataset
- **`containerType`: required container type object**
  - `name`: required container name (camel case format)
  - `id`: optional identifier for standardized containers
  - `version`: required standard version, if `id` is given
- `created`: required creation timestamp (see *format*)
- `storageTime`: required timestamp of storage or freeze (see *format*)
- `static`: required boolean flag (see *container variants*)
- `complete`: required boolean flag (see *container variants*)
- `hash`: optional hex digest of SHA256 hash, required for *static containers*

- **`usedSoftware`: optional list of software objects**
  - `name`: required software name
  - `version`: required software version
  - `id`: optional software identifier (e.g. UUID or URL)
  - `idType`: required type of identifier, if `id` is given
- `modelVersion`: required data model version

1.2 Dataset Description

The meta data describing the data payload of the container is stored in the required root item `meta.json`, which contains a single JSON object. The following set of attributes is currently defined for this item:

- `author`: required name of the author
- `email`: required e-mail address of the author
- `organization`: optional affiliation of the author
- `comment`: optional comments on the dataset
- `title`: required title of the dataset
- `keywords`: optional list of keywords
- `description`: optional abstract for the dataset
- `timestamp`: optional creation timestamp of the dataset (see *format*)
- `doi`: optional digital object identifier of the dataset
- `license`: optional data license name (e.g. "MIT" or "CC-BY")

1.3 Timestamp Format

An ISO 8601 compatible string in a certain format is expected as value of timestamp attributes in `content.json` and `meta.json`. The required format contains the UTC date and time and the local timezone. For example:

```
"2023-02-17T15:23:57+0100"
```

1.4 Suggested Parts

Standardization simplifies data exchange as well as reuse of data. Therefore, it is suggested to store the data payload of a container in the following part structure:

- `/info`: informative parameters
- `/sim`: raw simulation results
- `/meas`: raw measurement results
- `/data`: parameters and data required to achieve results in `/sim` or `/meas`
- `/eval`: evaluation results derived from `/sim` and/or `/meas`

- `/log`: log files or other unstructured data

1.5 Container Variants

Our data model currently supports three variants of data containers, based on certain use cases. The distinction is mainly relevant for data storage and therefore of particular interest when you upload the container to a storage server. The respective variant is selected using the boolean attributes `static` and `complete` of the item `content.json`:

`static`	`complete`	Container variant
true	true	static container
true	false	(not allowed)
false	true	normal completed container
false	false	incomplete container

The **normal container** is generated and completed in a single step. This matches the typical workflow of generating data and saving all of it in one shot. However, if the data acquisition runs over a very long time like days or weeks, you may want to store also **incomplete containers**. In that case you can mark the container as containing incomplete data and update it as needed with increasing attribute `storageTime`. Each server upload will replace the previous container. With your final upload you mark the container as being complete.

Static containers are intended to carry static parameters in contrast to measurement or simulation data. An example would be a detailed description of a measurement setup, which is used for many measurements. Instead of including the large setup data with each individual measurement dataset, the whole setup may be stored as a single static dataset and referenced by its UUID as measurement parameter in subsequent containers. Static containers must contain a hash string. The data storage server refuses the upload of multiple containers with same `containerType` and `hash`.

CHAPTER

TWO

CONFIGURATION

2.1 Container File Extension

On Microsoft Windows you may inspect ZDC files with a double-click in the Windows Explorer. This requires that you register the extension `.zdc` in the same way as `.zip`. Run the following on the command prompt to achieve this behaviour:

```
reg copy HKCR\.zip HKCR\.zdc /s /f
```

2.2 Configuration File

Using a SciDataContainer software library or the GUI application makes the usage of data containers much more convenient. They can initialize and manage many container attributes automatically. The libraries and the GUI are also able to take some user specific attributes and parameters either from environment variables or a config file.

Name and location of the configuration file is `%USERPROFILE%\scidata.cfg` on Microsoft Windows and `~/.scidata` on other operating systems. The file is expected to be a text file. Leading and trailing white space is ignored, as well as lines starting with `#`. Parameters are taken from lines in the form `<key>=<value>`. White space before and after the equal sign is ignored. The keywords are case-insensitive.

The following parameters are supported:

Environment variable	Configuration key	Content
`DC_AUTHOR`	`author`	author of the dataset
`DC_EMAIL`	`email`	e-mail address of the author
`DC_SERVER`	`server`	name or address of the data storage server
`DC_KEY`	`key`	key for the storage server API

A value in the configuration file supersedes the content of the respective environment variable.

2.3 Example Configuration File

```
author = Jane Doe
email = jane.doe@example.com
server = data.example.com
key = 487cadbdcca5302b5d24f94609dbadda4f5b034d2f863ec22f9caa739b12690b
```

CHAPTER

THREE

PYTHON LIBRARY

This is the Python 3 implementation of `SciDataContainer`. The implementation is operating system independent. In order to simplify the generation of meta data, the `Container` class will try to insert default values for the author name and e-mail address from the configuration file.

The easiest way to install the latest version of the scidatacontainer package is using PIP:

```
pip install scidatacontainer
```

3.1 Basic Usage

3.1.1 Container Objects

As a simple application example, we generate and store a list of random integer numbers. Parameters are quantity and range of the numbers. At first, we import the Python package `random` module and the class `Container` from the package `scidatacontainer`:

```
>>> import random
>>> from scidatacontainer import Container
```

Then we generate a parameter dictionary and the actual test data:

```
>>> p = {"quantity": 8, "minValue": 1, "maxValue": 6}
>>> data = [random.randint(p["minValue"], p["maxValue"]) for i in range(p["quantity"])]
>>> data
[2, 5, 1, 3, 1, 4, 4, 4]
```

If a default author name and e-mail address was made available as explained in the *Configuration* section, there are just two additional attributes, which you must provide. One is the type of the container and the other a title of the dataset. Together with the raw data and the dictionary of parameters, we can now build the dictionary of container items:

```
>>> items = {
...          "content.json": {
...                            "containerType": {"name": "myRandInt"},
...                           },
...          "meta.json": {
...                         "title": "My first set of random numbers",
...                        },
...          "sim/dice.json": data,
...          "data/parameter.json": p,
...         }
```

Now we are ready to build the container, store it in a local file and get a short description of its content:

```
>>> dc = Container(items=items)
>>> dc.write("random.zdc")
>>> print(dc)
Complete Container
        type:        myRandInt
        uuid:        306e2c2d-a9f6-4306-8851-1ee0fceeb852
        created:     2023-02-28T10:03:44+0100
        storageTime: 2023-02-28T10:03:44+0100
        author:      Reinhard Caspary
```

Feel free to check the content of the file `random.zdc` now by opening it on the operating system level. Be reminded that the Windows Explorer requires the file extension `.zdc` to be registered first as in the *Configuration* section. Recovering the dataset from the local file as a new container object works straight forward:

```
>>> dc = Container(file="random.zdc")
>>> dc["sim/dice.json"]
[2, 5, 1, 3, 1, 4, 4, 4]
```

3.1.2 Server Storage

Container files can be stored on and retrieved from a specific data storage server. If the server name and an API key was made available as explained in the *Configuration* section, upload and download of a container is as simple as:

```
>>> dc.upload()
>>> dc = Container(uuid="306e2c2d-a9f6-4306-8851-1ee0fceeb852")
```

The server makes sure that UUIDs are unique. Once uploaded, a container can never be modified on a server. The only exemption are incomplete containers.

In the rare case that a certain container needs to be replaced, the attribute `replaces` may be used in `content.json`. Once uploaded, the server will always deliver the new container, even if the container with the old UUID is requested. Only the owner of a container is allowed to replace it.

3.1.3 Timestamps

You may use the function `timestamp()` to generate a timestamp in the format required by the `Container` class:

```
>>> from scidatacontainer import timestamp
>>> timestamp()
2023-03-24T21:50:34+0100
```

3.1.4 Incomplete Containers

As already mentioned, incomplete containers are a container variant which is intended for long running measurements or simulations. As long as the attribute `complete` in `content.json` has the value `False`, a container may be uploaded repeatedly, each time replacing the container with the same UUID on the server:

```
>>> items["content.json"]["complete"] = False
>>> dc = Container(items=items)
>>> dc.upload()
>>> dc["content.json"]["uuid"]
'306e2c2d-a9f6-4306-8851-1ee0fceeb852'
```

The server will only accept containers with increasing modification timestamps. Since the resolution of the internal timestamps is a second, you must wait at least one second before the next upload:

```
>>> dc = Container(uuid="306e2c2d-a9f6-4306-8851-1ee0fceeb852")
>>> dc["meas/newdata.json"] = newdata
>>> dc.upload()
```

For the final upload, the container must be marked as being complete. This makes this container immutable:

```
>>> dc = Container(uuid="306e2c2d-a9f6-4306-8851-1ee0fceeb852")
>>> dc["meas/finaldata.json"] = finaldata
>>> dc["content.json"]["complete"] = True
>>> dc.upload()
```

3.1.5 Static Containers

A static container is generated by calling the method `freeze()` of the container object. It is intended for static parameters in contrast to measurement or simulation data:

```
>>> dc = Container(items=items)
>>> dc.freeze()
>>> print(dc)
Static Container
        type:        myRandInt
        uuid:        2a7eb1c5-5fe8-4c92-be1d-2f1207b0d855
        hash:        bafc6813d92bd23b06b63eed035ba9b33415acc770c9128f47775ab2d55cc152
        created:     2023-03-01T21:01:20+0100
        storageTime: 2023-03-01T21:01:20+0100
        author:      Reinhard Caspary
```

Freezing a container will set the attribute `static` in `content.json` to `True`, which makes this container immutable and it calculates an SHA256 hash of the container content. When you try to upload a static container and there is another static container with the same attributes `containerType.name` and `hash`, the content of the current container object is silently replaced by the original one from the server.

3.2 Advanced Usage

3.2.1 Convenience Methods

The `Container` class provides a couple of convenience methods, which make it behave very similar to a dictionary:

```
>>> dc = Container(items=items)
>>> dc["content.json"]["uuid"]
'306e2c2d-a9f6-4306-8851-1ee0fceeb852'
>>> dc["log/console.txt"] = "Hello World!"
>>> "log/console.txt" in dc
True
>>> del dc["log/console.txt"]
>>> "log/console.txt" in dc
False
```

The method `keys()` returns a list of all full item names including the respective parts, `values()` a list of all item objects, and `items()` a list of all (name, item) tuples as you would expect from a dictionary object.

You may use the method `hash()` to calculate an SHA256 hash of the container content. The hex digest of this value is stored in the attribute `hash` of the item `container.json`.

Container objects generated from an items dictionary using the parameter `items=...` are mutable, which means that you can add, modify and delete items. As soon as you call one of the methods `write()`, `upload()`, `freeze()`, or `hash()`, the container becomes immutable. Containers loaded from a local file or a server are immutable as well.

An immutable container will throw an exception if you try to modify its content. However, this feature is not bulletproof. The `Container` class is not aware of any internal modifications of item objects.

You can convert an immutable container into a mutable one by calling its method `release()`. This generates a new UUID and resets the attributes `replaces`, `created`, `storageTime`, `hash` and `modelVersion`.

3.2.2 Server Storage

It is most convenient to store the server name and the API key in the configuration file. However, both values can also be specified as method parameters:

```
>>> dc.upload(server="...", key="...")
>>> dc = Container(uuid="306e2c2d-a9f6-4306-8851-1ee0fceeb852", server="...", key="...")
```

3.2.3 File Formats

The `Container` class can handle virtually any file format. However, in order to store and read a certain file format, it needs to know how to convert the respective Python object into a bytes stream and vice versa. File formats are identified by their file extension. The following file extensions are currently supported by the package `scidatacontainer` out of the box:

Extension	File format	Python object	Required packages
.json	JSON file (UTF-8 encoding)	dictionary or others	
.txt	Text file (UTF-8 encoding)	string	
.log	Text file (UTF-8 encoding)	string	
.pgm	Text file (UTF-8 encoding)	string	
.png	PNG image file	NumPy array	cv2, numpy
.npy	NumPy array	NumPy array	numpy
.bin	Raw binary data file	bytes	

Native support for image and NumPy objects is only available when your Python environment contains the packages cv2 and/or numpy. The container class tries to guess the format of items with unknown extension. However, it is more reliable to use the function `register()` to add alternative file extensions to already known file formats. The following commands will register the extension `.py` as a text file:

```
>>> from scidatacontainer import register
>>> register("py", "txt")
```

If you want to register another Python object, you need to provide a conversion class which can convert this object to and from a bytes string. This class should be inherited from the class `FileBase`. The storage of NumPy arrays for example may be realized by the following code:

```python
import io
import numpy as np
from scidatacontainer import FileBase, register

class NpyFile(FileBase):

        allow_pickle = False

        def encode(self):
                with io.BytesIO() as fp:
                        np.save(fp, self.data, allow_pickle=self.allow_pickle)
                        fp.seek(0)
                        data = fp.read()
                return data

        def decode(self, data):
                with io.BytesIO() as fp:
                        fp.write(data)
                        fp.seek(0)
                        self.data = np.load(fp, allow_pickle=self.allow_pickle)

register("npy", NpyFile, np.ndarray)
```

The third argument of the function `register()` sets this conversion class as default for NumPy array objects overriding any previous default class. This argument is optional.

Hash values are usually derived from the bytes string of an encoded object. If you require a different behaviour, you may also override the method `hash()` of the class `FileBase`.

3.3 SciDataContainer API

3.3.1 Container classes

`class scidatacontainer.`**`Container`**`(`*items: Optional[dict] = None*, *file: Optional[str] = None*, *uuid: Optional[str] = None*, *server: Optional[str] = None*, *key: Optional[str] = None*, *compression: int = 8*, *compresslevel: int = -1*`)`

Bases: `AbstractContainer`

Scientific data container.

`decode``(`*data: bytes*, *validate: bool = True*, *strict: bool = True*`)`

Take ZIP package as binary string. Read items from the package and store them in this object.

Parameters

- **`data`** – Bytestring containing the ZIP DataContainer.
- **`validate`** – If true, validate the content.
- **`strict`** – If true, validate the hash, too.

`encode``()`

Encode container as ZIP package. Return package as binary string.

`freeze``()`

Calculate the hash value of this container and make it static. The container cannot be modified any more when this method was called once.

`hash``()`

Calculate and save the hash value of this container.

`items``()`

Return this container as a dictionary of item objects (key, value) tuples.

`keys``()` → List[str]

Return a sorted list of the full paths of all items.

Returns

List of paths of Container items.

Return type

List[str]

`release``()`

Make this container mutable. If it was immutable, this method will create a new UUID and initialize the attributes replaces, createdstorageTime and modelVersion in the item “content.json”. It will also delete an existing hash and make it a new container.

`upload``(`*data: Optional[bytes] = None*, *server: Optional[str] = None*, *key: Optional[str] = None*`)`

Create a ZIP archive of the DataContainer and upload it to a server.

If data is passed to the function, data will be written to the file. Otherwise the byte representation of the class instance will be written to the file, which is what you typically want.

Parameters

- `data` – If given, data to write to the file.
- `server` – URL of the server.
- `key` – API Key from the server to identify yourself.

`validate_content()`

Make sure that the item "content.json" exists and contains all required attributes.

`validate_meta()`

Make sure that the item "meta.json" exists and contains all required attributes.

`values()` → List

Return a list of all item objects.

Returns
List of item objects of the Container.

Return type
List

`write`(*fn: str*, *data: Optional[bytes] = None*)

Write the container to a ZIP package file.

If data is passed to the function, data will be written to the file. Otherwise the byte representation of the class instance will be written to the file, which is what you typically want.

Parameters

- `fn` – Filename of export file.
- `data` – If given, data to write to the file.

`class scidatacontainer.AbstractContainer`(*items: Optional[dict] = None*, *file: Optional[str] = None*, *uuid: Optional[str] = None*, *server: Optional[str] = None*, *key: Optional[str] = None*, *compression: int = 8*, *compresslevel: int = -1*)

Bases: `ABC`

Scientific data container with minimal file support.

The following file types are supported:

- .json <-> dict
- .txt <-> str
- .bin <-> bytes

`decode`(*data: bytes*, *validate: bool = True*, *strict: bool = True*)

Take ZIP package as binary string. Read items from the package and store them in this object.

Parameters

- `data` – Bytestring containing the ZIP DataContainer.
- `validate` – If true, validate the content.
- `strict` – If true, validate the hash, too.

`encode()`

Encode container as ZIP package. Return package as binary string.

`freeze()`

Calculate the hash value of this container and make it static. The container cannot be modified any more when this method was called once.

`hash()`

Calculate and save the hash value of this container.

`items()`

Return this container as a dictionary of item objects (key, value) tuples.

`keys()` → List[str]

Return a sorted list of the full paths of all items.

Returns
List of paths of Container items.

Return type
List[str]

`release()`

Make this container mutable. If it was immutable, this method will create a new UUID and initialize the attributes replaces, createdstorageTime and modelVersion in the item "content.json". It will also delete an existing hash and make it a new container.

`upload`(*data: Optional[bytes] = None*, *server: Optional[str] = None*, *key: Optional[str] = None*)

Create a ZIP archive of the DataContainer and upload it to a server.

If data is passed to the function, data will be written to the file. Otherwise the byte representation of the class instance will be written to the file, which is what you typically want.

Parameters

- `data` – If given, data to write to the file.
- `server` – URL of the server.
- `key` – API Key from the server to identify yourself.

`validate_content()`

Make sure that the item "content.json" exists and contains all required attributes.

`validate_meta()`

Make sure that the item "meta.json" exists and contains all required attributes.

`values()` → List

Return a list of all item objects.

Returns
List of item objects of the Container.

Return type
List

`write`(*fn: str*, *data: Optional[bytes] = None*)

Write the container to a ZIP package file.

If data is passed to the function, data will be written to the file. Otherwise the byte representation of the class instance will be written to the file, which is what you typically want.

Parameters

- `fn` – Filename of export file.

- **data** – If given, data to write to the file.

3.3.2 File type support

scidatacontainer.**register**(*suffix: str*, *fclass: Type[*AbstractFile*]*, *pclass: Optional[Type[object]] = None*)

Register a suffix to a conversion class.

If the parameter class is a string, it is interpreted as known suffix and the conversion class of this suffix is registered also for the new one.

Parameters

- **suffix** – file suffix to identify this file type.
- **fclass** – Conversion class derived from AbstractFile.
- **pclass** – Python class that represents this object type.

Built-in conversion classes

class scidatacontainer.filebase.**AbstractFile**(*data*)

Base class for converting datatypes to their file representation.

abstract decode(*data: bytes*)

Decode the Container content from bytes. This is an abstract method and it neets to be overwritten by inheriting class.

abstract encode() → bytes

Encode the Container content to bytes. This is an abstract method and it needs to be overwritten by inheriting class.

Returns
Byte string representation of the object.

Return type
bytes

hash() → str

Return hex digest of SHA256 hash.

Returns
Hex digest of this object as string.

Return type
str

class scidatacontainer.filebase.**BinaryFile**(*data*)

Bases: *AbstractFile*

Data conversion class for a binary file.

decode(*data: bytes*)

Store bytes in this class.

encode() → bytes

Return byte string stored in this class.

Returns
Byte string representation of the object.

Return type
bytes

hash() → str

Return hex digest of SHA256 hash.

Returns
Hex digest of this object as string.

Return type
str

`class scidatacontainer.filebase.`**`TextFile`**(*data*)

Bases: *AbstractFile*

Data conversion class for a text file.

`charset = 'utf8'`

Character encoding used for translation from text to bytes.

Type
charset (str)

decode(*data: bytes*)

Decode text from given bytes string.

encode() → bytes

Encode text to bytes string.

Returns
Byte string representation of the object.

Return type
bytes

hash() → str

Return hex digest of SHA256 hash.

Returns
Hex digest of this object as string.

Return type
str

`class scidatacontainer.filebase.`**`JsonFile`**(*data*)

Bases: *AbstractFile*

Data conversion class for a JSON file represented as Python dictionary.

`charset = 'utf8'`

Character encoding used for translation from text to bytes.

Type
charset (str)

decode(*data: bytes*)

Decode dictionary from given bytes string.

encode() → bytes

Convert dictionary to pretty string representation with indentation and return it as bytes string.

Returns
Byte string representation of the object.

Return type
bytes

hash() → str

Return hex digest of the SHA256 hash calculated from the sorted compact representation. This should result in the same hash for semantically equal data dictionaries.

Returns
Hex digest of this object as string.

Return type
str

indent = 4

Indentation of exported JSON files.

Type
indent (int)

sortit(*data: Union[dict, list, tuple]*) → str

Return compact string representation with keys of all sub-dictionaries sorted.

Parameters
data – Dictionary, list or tuple to convert to string"

Returns
String representation of *data*

Return type
str

3.3.3 Convenience functions

scidatacontainer.**timestamp**() → str

Return the current ISO 8601 compatible timestamp as string.

Returns
timestamp as string

Return type
str

scidatacontainer.config.**load_config**(*config_path: Optional[str] = None*) → dict

Get config data from environment variables and config file.

This functions prefers values in the scidata config file and potentially overwrites values that are present as environmental variables.

Usually, users doen't need to call this function. However, it can be used for debugging purposes if the configuration parameters are not as expected.

Parameters
str – Path of the config file. If this is None, the default file will be used. This filename is only required for testing.

Returns
A dictionary containing information strings with keys "author", "email", "server", "key".

Return type
dict

CHAPTER

FOUR

DATA STORAGE SERVER

The data storage server provides a browser interface as well as a REST API. A user account is required to access the server and get an API key for the REST API via the browser interface.

4.1 Container Upload

Method
POST

URL
http://<server>/api/datasets/

Content
Container files

Header
Authorization: Token <key>

Response:

HTTP return code	Description	Returned content
`201 Created`	Successful container upload	
`400 Bad Request`	Existing static dataset with same `hash` and `containerType`	JSON object
`400 Bad Request`	Malformed or invalid container	
`403 Forbidden`	Unauthorized access	
`409 Conflict`	Existing completed dataset with same UUID	
`415 Unsupported`	Invalid container format	
`500 Server Error`	Internal server error	

4.2 Container Download

Method
GET

URL
http://<server>/api/datasets/<uuid>/download/

Header
Authorization: Token <key>

Response:

HTTP return code	Description	Returned content
`200 OK`	Success	Data container
`204 No Content`	Dataset deleted	
`301 Moved Permanently`	Dataset replaced	Last replacement of container
`403 Forbidden`	Unauthorized access	
`404 Not Found`	No dataset available	
`500 Server Error`	Internal server error	

INDEX