openPMD-api Documentation

Release 0.2.0-alpha

The openPMD Community

Jun 12, 2018

INSTALLATION

openPMD C++ & Python API

This library provides an abstract API for openPMD file handling. It provides both support for writing & reading into various formats and works both serial and parallel (MPI). Implemented backends include HDF5 and ADIOS.

Note: Are you looking for our latest Doxygen docs for the API?

See http://www.openpmd.org/openPMD-api

Attention: This library is just getting started. Please stay tuned for updates and contact us on GitHub if you want to try it.

openPMD-api is a library using semantic versioning, starting with version 1.0.0.

The supported version of the openPMD standard are reflected as follows: `standardMAJOR.apiMAJOR.apiMINOR`.

openPMD-api version	supported openPMD standard versions
`0.1.0-0.2.0` (alpha)	`1.0.0-1.1.0`
`1.0.0+`	`1.0.1-1.1.0` (not released yet)
`2.0.0+`	`2.0.0+` (not released yet)

CHAPTER 1

Installation

1.1 Installation

Choose *one* of the install methods below to get started:

1.1.1 Using the Spack Package

A package for openPMD-api is available on the Spack package manager.

```
spack install openpmd-api  # optional: +python
spack load --dependencies openpmd-api
```

1.1.2 Using the conda Package

A package for serial openPMD-api is available on the Conda package manager.

```
conda install -c conda-forge openpmd-api
```

1.1.3 From Source with CMake

You can also install `openPMD-api` from source with CMake. This requires that you have all *dependencies* installed on your system. The developer section on *build options* provides further details on variants of the build.

On Linux platforms:

```
git clone https://github.com/openPMD/openPMD-api.git

mkdir -p openPMD-api-build
cd openPMD-api-build

# optional for some tests
.travis/download_samples.sh

# for own install prefix append:
```

(continues on next page)

(continued from previous page)

```
#   -DCMAKE_INSTALL_PREFIX=$HOME/somepath
# for options append:
#   -DopenPMD_USE_...=...
cmake ../openPMD-api

make -j

# optional
make test

# sudo is only required for system paths
sudo make install
```

On Windows platforms, replace the last steps with:

```
cmake -G "NMake Makefiles" ../openPMD-api

nmake
nmake install
```

1.2 Changelog

1.2.1 0.2.0-alpha

Date: 2018-06-11

Initial Numpy Bindings

Adds first bindings for record component reading and writing. Fixes some minor CMake issues.

Changes to "0.1.1-alpha"

Features

- Python: first NumPy bindings for record component chunk store/load #219
- CMake: add new `BUILD_EXAMPLES` option #238
- CMake: build directories controllable #241

Bug Fixes

- forgot to bump `version.hpp`/`__version__` in last release
- CMake: Overwritable Install Paths #237

1.2.2 0.1.1-alpha

Date: 2018-06-07

ADIOS1 Build Fixes & Less Flushes

We fixed build issues with the ADIOS1 backend. The number of performed flushes in backends was generally minimized.

Changes to "0.1.0-alpha"

Bug Fixes

- SerialIOTest: `loadChunk` template missing for ADIOS1 #227
- prepare running serial applications linked against parallel ADIOS1 library #228

Other

- minimize number of flushes in backend #212

1.2.3 0.1.0-alpha

Date: 2018-06-06

This is the first developer release of openPMD-api.

Both HDF5 and ADIOS1 are implemented as backends with serial and parallel I/O support. The C++11 API is considered alpha state with few changes expected to come. We also ship an unstable preview of the Python3 API.

CHAPTER 2

Usage

2.1 First Steps

For brevity, all following examples assume the following includes/imports:

2.1.1 C++11

```
#include <openPMD/openPMD.hpp>

using namespace openPMD;
```

2.1.2 Python

```
import openPMD
```

2.2 Serial API

...

2.2.1 Reading

C++

```
#include <openPMD/openPMD.hpp>

#include <iostream>
#include <memory>


using std::cout;
```

(continues on next page)

(continued from previous page)

```
using namespace openPMD;

int main()
{
    Series series = Series(
        "../samples/git-sample/data%T.h5",
        AccessType::READ_ONLY
    );
    cout << "Read a Series with openPMD standard version "
         << series.openPMD() << '\n';

    cout << "The Series contains " << series.iterations.size() << " iterations:";
    for( auto const& i : series.iterations )
        cout << "\n\t" << i.first;
    cout << '\n';

    Iteration i = series.iterations[100];
    cout << "Iteration 100 contains " << i.meshes.size() << " meshes:";
    for( auto const& m : i.meshes )
        cout << "\n\t" << m.first;
    cout << '\n';
    cout << "Iteration 100 contains " << i.particles.size() << " particle species:
↪";
    for( auto const& ps : i.particles )
        cout << "\n\t" << ps.first;
    cout << '\n';

    MeshRecordComponent E_x = i.meshes["E"]["x"];
    Extent extent = E_x.getExtent();
    cout << "Field E/x has shape (";
    for( auto const& dim : extent )
        cout << dim << ',';
    cout << ") and has datatype " << E_x.getDatatype() << '\n';

    Offset chunk_offset = {1, 1, 1};
    Extent chunk_extent = {2, 2, 1};
    auto chunk_data = E_x.loadChunk<double>(chunk_offset, chunk_extent);
    cout << "Queued the loading of a single chunk from disk, "
            "ready to execute\n";
    series.flush();
    cout << "Chunk has been read from disk\n"
         << "Read chunk contains:\n";
    for( size_t row = 0; row < chunk_extent[0]; ++row )
    {
        for( size_t col = 0; col < chunk_extent[1]; ++col )
            cout << "\t"
                 << '(' << row + chunk_offset[0] << '|' << col + chunk_offset[1] <
↪< '|' << 1 << ")\t"
                 << chunk_data.get()[row*chunk_extent[1]+col];
        cout << '\n';
    }

    return 0;
}
```

An extended example can be found in `examples/6_dump_filebased_series.cpp`.

Python

```
import openPMD


if __name__ == "__main__":
    series = openPMD.Series("../samples/git-sample/data%T.h5",
                            openPMD.Access_Type.read_only)
    print("Read a Series with openPMD standard version %s" %
          series.openPMD)

    print("The Series contains {0} iterations:".format(len(series.iterations)))
    for i in series.iterations:
        print("\t {0}".format(i))
    print("")

    i = series.iterations[100]
    print("Iteration 100 contains {0} meshes:".format(len(i.meshes)))
    for m in i.meshes:
        print("\t {0}".format(m))
    print("")
    print("Iteration 100 contains {0} particle species:".format(
        len(i.particles)))
    for ps in i.particles:
        print("\t {0}".format(ps))
    print("")

    E_x = i.meshes["E"]["x"]
    shape = E_x.shape

    print("Field E.x has shape {0} and datatype {1}".format(
          shape, E_x.dtype))

    offset = openPMD.Extent([1, 1, 1])
    extent = openPMD.Extent([2, 2, 1])
    # TODO buffer protocol / numpy bindings
    # chunk_data = E_x[1:3, 1:3, 1:2]
    chunk_data = E_x.load_chunk(offset, extent)
    # print("Queued the loading of a single chunk from disk, "
    #       "ready to execute")
    series.flush()
    print("Chunk has been read from disk\n"
          "Read chunk contains:")
    print(chunk_data)
    # for row in range(2):
    #     for col in range(2):
    #         print("\t({0}|{1}|{2})\t{3}".format(
    #             row + 1, col + 1, 1, chunk_data[row*chunk_extent[1]+col])
    #         )
    #     print("")
```

2.2.2 Writing

C++

```
#include <openPMD/openPMD.hpp>

#include <iostream>
#include <memory>
#include <numeric>
```

(continues on next page)

(continued from previous page)

```
using std::cout;
using namespace openPMD;

int main(int argc, char *argv[])
{
    // user input: size of matrix to write, default 3x3
    size_t size = (argc == 2 ? atoi(argv[1]) : 3);

    // matrix dataset to write with values 0...size*size-1
    std::vector<double> global_data(size*size);
    std::iota(global_data.begin(), global_data.end(), 0.);

    cout << "Set up a 2D square array (" << size << 'x' << size
         << ") that will be written\n";

    // open file for writing
    Series series = Series(
        "../samples/3_write_serial.h5",
        AccessType::CREATE
    );
    cout << "Created an empty " << series.iterationEncoding() << " Series\n";

    MeshRecordComponent E =
      series
          .iterations[1]
          .meshes["E"][MeshRecordComponent::SCALAR];
    cout << "Created a scalar mesh Record with all required openPMD attributes\n";

    Datatype datatype = determineDatatype(shareRaw(global_data));
    Extent extent = {size, size};
    Dataset dataset = Dataset(datatype, extent);
    cout << "Created a Dataset of size " << dataset.extent[0] << 'x' << dataset.
↪extent[1]
         << " and Datatype " << dataset.dtype << '\n';

    E.resetDataset(dataset);
    cout << "Set the dataset properties for the scalar field E in iteration 1\n";

    series.flush();
    cout << "File structure and required attributes have been written\n";

    Offset offset = {0, 0};
    E.storeChunk(offset, extent, shareRaw(global_data));
    cout << "Stored the whole Dataset contents as a single chunk, "
            "ready to write content\n";

    series.flush();
    cout << "Dataset content has been fully written\n";

    return 0;
}
```

An extended example can be found in `examples/7_extended_write_serial.cpp`.

Python

```
import openPMD
import numpy as np
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    # user input: size of matrix to write, default 3x3
    size = 3

    # matrix dataset to write with values 0...size*size-1
    global_data = np.arange(size*size, dtype=np.double)

    print("Set up a 2D square array ({0}x{1}) that will be written".format(
        size, size))

    # open file for writing
    series = openPMD.Series(
        "../samples/3_write_serial_py.h5",
        openPMD.Access_Type.create
    )

    print("Created an empty {0} Series".format(series.iteration_encoding))

    print(len(series.iterations))
    E = series.iterations[1].meshes["E"][openPMD.Mesh_Record_Component.SCALAR]

    datatype = openPMD.Datatype.DOUBLE
    # datatype = openPMD.determineDatatype(global_data)
    extent = openPMD.Extent([size, size])
    dataset = openPMD.Dataset(datatype, extent)

    print("Created a Dataset of size {0}x{1} and Datatype {2}".format(
        dataset.extent[0], dataset.extent[1], dataset.dtype))

    E.reset_dataset(dataset)
    print("Set the dataset properties for the scalar field E in iteration 1")

    # writing fails on already open file error
    series.flush()
    print("File structure has been written")

    # offset = openPMD.Offset([0, 0])
    offset = openPMD.Extent([0, 0])
    # TODO implement slicing protocol
    # E[offset[0]:extent[0], offset[1]:extent[1]] = global_data
    E.store_chunk(offset, extent, global_data)
    print("Stored the whole Dataset contents as a single chunk, " +
          "ready to write content")

    series.flush()
    print("Dataset content has been fully written")
```

2.3 Parallel API

The following examples show parallel reading and writing of domain-decomposed data with MPI.

2.3.1 Reading

```
#include <openPMD/openPMD.hpp>
```

(continues on next page)

(continued from previous page)

```
#include <mpi.h>

#include <iostream>
#include <memory>


using std::cout;
using namespace openPMD;

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int mpi_size;
    int mpi_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);


    /* note: this scope is intentional to destruct the openPMD::Series object
     *       prior to MPI_Finalize();
     */
    {
        Series series = Series(
            "../samples/git-sample/data%T.h5",
            AccessType::READ_ONLY,
            MPI_COMM_WORLD
        );
        if( 0 == mpi_rank )
            cout << "Read a series in parallel with " << mpi_size << " MPI ranks\n
↪";

        MeshRecordComponent E_x = series.iterations[100].meshes["E"]["x"];

        Offset chunk_offset = {
            static_cast< long unsigned int >(mpi_rank) + 1,
            1,
            1
        };
        Extent chunk_extent = {2, 2, 1};

        auto chunk_data = E_x.loadChunk<double>(chunk_offset, chunk_extent);

        if( 0 == mpi_rank )
            cout << "Queued the loading of a single chunk per MPI rank from disk, "
                    "ready to execute\n";
        series.flush();

        if( 0 == mpi_rank )
            cout << "Chunks have been read from disk\n";

        for( int i = 0; i < mpi_size; ++i )
        {
            if( i == mpi_rank )
            {
                cout << "Rank " << mpi_rank << " - Read chunk contains:\n";
                for( size_t row = 0; row < chunk_extent[0]; ++row )
                {
                    for( size_t col = 0; col < chunk_extent[1]; ++col )
                        cout << "\t"
```

(continues on next page)

(continued from previous page)

```
                                << '(' << row + chunk_offset[0] << '|' << col + chunk_
↪offset[1] << '|' << 1 << ")\t"
                                << chunk_data.get()[row*chunk_extent[1]+col];
                        cout << std::endl;
                    }
                }

                // this barrier is not necessary but structures the example output
                MPI_Barrier(MPI_COMM_WORLD);
            }
        }

        // openPMD::Series MUST be destructed at this point
        MPI_Finalize();

        return 0;
}
```

2.3.2 Writing

```
#include <openPMD/openPMD.hpp>

#include <mpi.h>

#include <iostream>
#include <memory>


using std::cout;
using namespace openPMD;

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);

    int mpi_size;
    int mpi_rank;

    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

    /* note: this scope is intentional to destruct the openPMD::Series object
     *       prior to MPI_Finalize();
     */
    {
        // allocate a data set to write
        std::shared_ptr< double > global_data(new double[mpi_size], [](double *p)
↪{ delete[] p; });
        for( int i = 0; i < mpi_size; ++i )
            global_data.get()[i] = i;
        if( 0 == mpi_rank )
            cout << "Set up a 1D array with one element per MPI rank (" << mpi_size
                 << ") that will be written to disk\n";

        std::shared_ptr< double > local_data{new double};
        *local_data = global_data.get()[mpi_rank];
        if( 0 == mpi_rank )
            cout << "Set up a 1D array with one element, representing the view of␣
↪the MPI rank\n";
```

(continues on next page)

(continued from previous page)

```
        // open file for writing
        Series series = Series(
            "../samples/5_parallel_write.h5",
            AccessType::CREATE,
            MPI_COMM_WORLD
        );
        if( 0 == mpi_rank )
          cout << "Created an empty series in parallel with "
               << mpi_size << " MPI ranks\n";

        MeshRecordComponent id =
            series
                .iterations[1]
                .meshes["id"][MeshRecordComponent::SCALAR];

        Datatype datatype = determineDatatype(local_data);
        Extent dataset_extent = {static_cast< long unsigned int >(mpi_size)};
        Dataset dataset = Dataset(datatype, dataset_extent);

        if( 0 == mpi_rank )
            cout << "Created a Dataset of size " << dataset.extent[0]
                 << " and Datatype " << dataset.dtype << '\n';

        id.resetDataset(dataset);
        if( 0 == mpi_rank )
            cout << "Set the global on-disk Dataset properties for the scalar␣
↪field id in iteration 1\n";

        series.flush();
        if( 0 == mpi_rank )
            cout << "File structure has been written to disk\n";

        Offset chunk_offset = {static_cast< long unsigned int >(mpi_rank)};
        Extent chunk_extent = {1};
        id.storeChunk(chunk_offset, chunk_extent, local_data);
        if( 0 == mpi_rank )
            cout << "Stored a single chunk per MPI rank containing its␣
↪contribution, "
                     "ready to write content to disk\n";

        series.flush();
        if( 0 == mpi_rank )
            cout << "Dataset content has been fully written to disk\n";
    }

    // openPMD::Series MUST be destructed at this point
    MPI_Finalize();

    return 0;
}
```

CHAPTER 3

Development

3.1 Contribution Guide

3.1.1 GitHub

The best starting point is the GitHub issue tracker.

For existing tasks, the labels good first issue and help wanted are great for contributions. In case you want to start working on one of those, just *comment* in it first so no work is duplicated.

New contributions in form of pull requests always need to go in the `dev` (development) branch. The `master` branch contains the last stable release and receives updates only when a new version is drafted.

Maintainers organize prioritites and progress in the projects tab.

3.1.2 Style Guide

For coding style, please try to follow the guides in ComputationalRadiationPhysics/contributing for new code.

3.2 Repository Structure

3.2.1 Branches

- `master`: the latest stable release, always tagged with a version
- `dev`: the development branch where all features start from and are merged to
- `release-X.Y.Z`: release candiate for version `X.Y.Z` with an upcoming release, receives updates for bug fixes and documentation such as change logs but usually no new features

3.2.2 Directory Structure

- `include/`
  - C++ header files

  - set `-I` here
  - prefixed with project name
- `src/`
  - C++ source files
- `lib/`
  - `python/`
    - modules, e.g. for RT interfaces, pre* & post-processing
    - set `PYTHONPATH` here
- `examples/`
  - read and write examples
- `samples/`
  - example files; need to be added manually with: `.travis/download_samples.sh`
- `test/`
  - unit tests which are run on `make test`
- `.travis/`
  - setup scripts for our continuous integration system
- `docs/`
  - documentation files
- `cmake/`
  - CMake scripts

3.3 How to Write a Backend

Adding support for additional types of file storage or data transportation is possible by creating a backend. Backend design has been kept independent of the openPMD-specific logic that maintains all constraints within a file. This should allow easy introduction of new file formats with only little knowledge about the rest of the system.

3.3.1 File Formats

To get started, you should create a new file format in `include/openPMD/IO/Format.hpp` representing the new backend. Note that this enumeration value will never be seen by users of openPMD-api, but should be kept short and concise to improve readability.

```
enum class Format
{
     JSON
};
```

In order to use the file format through the API, you need to provide unique and characteristic filename extensions that are associated with it. This happens in `src/Series.cpp`:

```
Format
determineFormat(std::string const& filename)
{
    if( auxiliary::ends_with(filename, ".json") )
        return Format::JSON;
}
```

```
std::string
cleanFilename(std::string const& filename, Format f)
{
    switch( f )
    {
        case Format::JSON:
            return auxiliary::replace_last(filename, ".json", "");
    }
}
```

```
std::function< bool(std::string const&) >
matcher(std::string const& name, Format f)
{
    switch( f )
    {
        case Format::JSON:
        {
            std::regex pattern(auxiliary::replace_last(name + ".json$", "%T",
↪"[[:digit:]]+"));
            return [pattern](std::string const& filename) -> bool { return␣
↪std::regex_search(filename, pattern); };
        }
    }
}
```

Unless your file format imposes additional restrictions to the openPMD constraints, this is all you have to do in the frontend section of the API.

3.3.2 IO Handler

Now that the user can specify that the new backend is to be used, a concrete mechanism for handling IO interactions is required. We call this an `IOHandler`. It is not concerned with any logic or constraints enforced by openPMD, but merely offers a small set of elementary IO operations.

On the very basic level, you will need to derive a class from `AbstractIOHandler`:

```
/* file: include/openPMD/IO/JSON/JSONIOHandler.hpp */
#include "openPMD/IO/AbstractIOHandler.hpp"

namespace openPMD
{
class JSONIOHandler : public AbstractIOHandler
{
public:
    JSONIOHandler(std::string const& path, AccessType);
    virtual ~JSONIOHandler();

    std::future< void > flush() override;
}
} // openPMD
```

```
/* file: src/IO/JSON/JSONIOHandler.cpp */
#include "openPMD/IO/JSON/JSONIOHandler.hpp"

namespace openPMD
{
JSONIOHandler::JSONIOHandler(std::string const& path, AccessType at)
        : AbstractIOHandler(path, at)
{ }
```

(continues on next page)

(continued from previous page)

```
JSONIOHandler::~JSONIOHandler()
{ }

std::future< void >
JSONIOHandler::flush()
{ return std::future< void >(); }
} // openPMD
```

Familiarizing your backend with the rest of the API happens in just one place in `src/IO/AbstractIOHandler.cpp`:

```
#if openPMD_HAVE_MPI
std::shared_ptr< AbstractIOHandler >
AbstractIOHandler::createIOHandler(std::string const& path,
                                   AccessType at,
                                   Format f,
                                   MPI_Comm comm)
{
    switch( f )
    {
        case Format::JSON:
            std::cerr << "No MPI-aware JSON backend available. "
                         "Falling back to the serial backend! "
                         "Possible failure and degraded performance!" << std::endl;
            return std::make_shared< JSONIOHandler >(path, at);
    }
}
#endif

std::shared_ptr< AbstractIOHandler >
AbstractIOHandler::createIOHandler(std::string const& path,
                                   AccessType at,
                                   Format f)
{
    switch( f )
    {
        case Format::JSON:
            return std::make_shared< JSONIOHandler >(path, at);
    }
}
```

In this state, the backend will do no IO operations and just act as a dummy that ignores all queries.

3.3.3 IO Task Queue

Operations between the logical representation in this API and physical storage are funneled through a queue `m_work` that is contained in the newly created IOHandler. Contained in this queue are `IOTask` s that have to be processed in FIFO order (unless you can prove sequential execution guarantees for out-of-order execution) when `AbstractIOHandler::flush()` is called. A **recommended** skeleton is provided in `AbstractIOHandlerImpl`. Note that emptying the queue this way is not required and might not fit your IO scheme.

Using the provided skeleton involves

- deriving an IOHandlerImpl for your IOHandler and
- delegating all flush calls to the IOHandlerImpl:

```
/* file: include/openPMD/IO/JSON/JSONIOHandlerImpl.hpp */
#include "openPMD/IO/AbstractIOHandlerImpl.hpp"
```

(continues on next page)

(continued from previous page)

```
namespace openPMD
{
class JSONIOHandlerImpl : public AbstractIOHandlerImpl
{
public:
    JSONIOHandlerImpl(AbstractIOHandler*);
    virtual ~JSONIOHandlerImpl();

    virtual void createFile(Writable*, Parameter< Operation::CREATE_FILE > const&)␣
↪override;
    virtual void createPath(Writable*, Parameter< Operation::CREATE_PATH > const&)␣
↪override;
    virtual void createDataset(Writable*, Parameter< Operation::CREATE_DATASET >␣
↪const&) override;
    virtual void extendDataset(Writable*, Parameter< Operation::EXTEND_DATASET >␣
↪const&) override;
    virtual void openFile(Writable*, Parameter< Operation::OPEN_FILE > const&)␣
↪override;
    virtual void openPath(Writable*, Parameter< Operation::OPEN_PATH > const&)␣
↪override;
    virtual void openDataset(Writable*, Parameter< Operation::OPEN_DATASET > &)␣
↪override;
    virtual void deleteFile(Writable*, Parameter< Operation::DELETE_FILE > const&)␣
↪override;
    virtual void deletePath(Writable*, Parameter< Operation::DELETE_PATH > const&)␣
↪override;
    virtual void deleteDataset(Writable*, Parameter< Operation::DELETE_DATASET >␣
↪const&) override;
    virtual void deleteAttribute(Writable*, Parameter< Operation::DELETE_ATT >␣
↪const&) override;
    virtual void writeDataset(Writable*, Parameter< Operation::WRITE_DATASET >␣
↪const&) override;
    virtual void writeAttribute(Writable*, Parameter< Operation::WRITE_ATT > const&
↪) override;
    virtual void readDataset(Writable*, Parameter< Operation::READ_DATASET > &)␣
↪override;
    virtual void readAttribute(Writable*, Parameter< Operation::READ_ATT > &)␣
↪override;
    virtual void listPaths(Writable*, Parameter< Operation::LIST_PATHS > &)␣
↪override;
    virtual void listDatasets(Writable*, Parameter< Operation::LIST_DATASETS > &)␣
↪override;
    virtual void listAttributes(Writable*, Parameter< Operation::LIST_ATTS > &)␣
↪override;
}
} // openPMD
```

```
/* file: include/openPMD/IO/JSON/JSONIOHandler.hpp */
#include "openPMD/IO/AbstractIOHandler.hpp"
#include "openPMD/IO/JSON/JSONIOHandlerImpl.hpp"

namespace openPMD
{
class JSONIOHandler : public AbstractIOHandler
{
public:
    /* ... */
private:
    JSONIOHandlerImpl m_impl;
}
```

(continues on next page)

(continued from previous page)

```
} // openPMD
```

```
/* file: src/IO/JSON/JSONIOHandler.cpp */
#include "openPMD/IO/JSON/JSONIOHandler.hpp"

namespace openPMD
{
/*...*/
std::future< void >
JSONIOHandler::flush()
{
    return m_impl->flush();
}
} // openPMD
```

Each IOTask contains a pointer to a `Writable` that corresponds to one object in the openPMD hierarchy. This object may be a group or a dataset. When processing certain types of IOTasks in the queue, you will have to assign unique FilePositions to these objects to identify the logical object in your physical storage. For this, you need to derive a concrete FilePosition for your backend from `AbstractFilePosition`. There is no requirement on how to identify your objects, but ids from your IO library and positional strings are good candidates.

```
/* file: include/openPMD/IO/JSON/JSONFilePosition.hpp */
#include "openPMD/IO/AbstractFilePosition.hpp"

namespace openPMD
{
struct JSONFilePosition : public AbstractFilePosition
{
    JSONFilePosition(uint64_t id)
        : id{id}
    { }

    uint64_t id;
};
} // openPMD
```

From this point, all that is left to do is implement the elementary IO operations provided in the IOHandlerImpl. The `Parameter` structs contain both input parameters (from storage to API) and output parameters (from API to storage). The easy way to distinguish between the two parameter sets is their C++ type: Input parameters are `std::shared_ptr` s that allow you to pass the requested data to their destination. Output parameters are all objects that are *not* `std::shared_ptr` s. The contract of each function call is outlined in `include/openPMD/IO/AbstractIOHandlerImpl`.

```
/* file: src/IO/JSON/JSONIOHandlerImpl.cpp */
#include "openPMD/IO/JSONIOHandlerImpl.hpp"

namespace openPMD
{
void
JSONIOHandlerImpl::createFile(Writable* writable,
                              Parameter< Operation::CREATE_FILE > const&␣
↪parameters)
{
    if( !writable->written )
    {
        path dir(m_handler->directory);
        if( !exists(dir) )
            create_directories(dir);

        std::string name = m_handler->directory + parameters.name;
```

(continues on next page)

(continued from previous page)

```
        if( !auxiliary::ends_with(name, ".json") )
            name += ".json";

        uint64_t id = /*...*/
        ASSERT(id >= 0, "Internal error: Failed to create JSON file");

        writable->written = true;
        writable->abstractFilePosition = std::make_shared< JSONFilePosition >(id);
    }
}
/*...*/
} // openPMD
```

Note that you might have to keep track of open file handles if they have to be closed explicitly during destruction of the IOHandlerImpl (prominent in C-style frameworks).

3.4 Build Dependencies

Section author: Axel Huebl

`openPMD-api` depends on a series of third-party projects. These are currently:

3.4.1 Required

- CMake 3.10.0+
- C++11 capable compiler, e.g. g++ 4.9+, clang 3.9+, VS 2015+

3.4.2 Shipped internally

The following libraries are shipped internally for convenience:

- MPark.Variant 1.3.0+
- Catch2 2.2.1+

3.4.3 Optional: I/O backends

- HDF5 1.8.13+
- ADIOS1 1.13.1+
- ADIOS2 2.1+ (*not yet implemented*)

while those can be build either with or without:

- MPI 2.3+, e.g. OpenMPI or MPICH2

3.4.4 Optional: language bindings

- Python:
  - Python 3.X+
  - pybind11 2.2.1+
- Python (*not yet implemented*):
  - mpi4py?

- numpy-dev?
- xtensor-python 0.17.0+?

3.5 Build Options

Section author: Axel Huebl

3.5.1 Variants

The following options can be added to the `cmake` call to control features. CMake controls options with prefixed `-D`, e.g. `-DopenPMD_USE_MPI=OFF`:

CMake Option	Values	Description
`openPMD_USE_MPI`	**AUTO**/ON/OFF	Enable MPI support
`openPMD_USE_HDF5`	**AUTO**/ON/OFF	Enable support for HDF5
`openPMD_USE_ADIOS1`	**AUTO**/ON/OFF	Enable support for ADIOS1
`openPMD_USE_ADIOS2`	AUTO/ON/**OFF**	Enable support for ADIOS2 [1]
`openPMD_USE_PYTHON`	**AUTO**/ON/OFF	Enable Python bindings
`openPMD_USE_INVASIVE_TESTS`	**AUTO**/ON/OFF	Enable unit tests that modify source code [2]
`PYTHON_EXECUTABLE`	(first found)	Path to Python executable

[1] *not yet implemented*

[2] e.g. C++ keywords, currently disabled only for MSVC

3.5.2 Shared or Static

By default, we will build as a static library and install also its headers. You can only build a static (`libopenPMD.a` or `openPMD.lib`) or a shared library (`libopenPMD.so` or `openPMD.dll`) at a time.

The following options can be tried to switch between static and shared builds and control if dependencies are linked dynamically or statically:

CMake Option	Values	Description
`BUILD_SHARED_LIBS`	ON/**OFF**	Build the C++ API as shared library
`HDF5_USE_STATIC_LIBRARIES`	ON/**OFF**	Require static HDF5 library
`ADIOS_USE_STATIC_LIBS`	ON/**OFF**	Require static ADIOS1 library

Note that python modules (`openPMD.cpython.[...].so` or `openPMD.pyd`) are always dynamic libraries. Therefore, if you want to build the python module and prefer static dependencies, make sure to provide all of dependencies build with position independent code (`-fPIC`). The same requirement is true if you want to build a *shared* C++ API library with *static* dependencies.

3.5.3 Debug

By default, the `Release` version is built. In order to build with debug symbols, pass `-DCMAKE_BUILD_TYPE=Debug` to your `cmake` command.

3.5.4 Shipped Dependencies

Additionally, the following libraries are shipped internally for convenience. These might get installed in your *CMAKE_INSTALL_PREFIX* if the option is `ON`.

The following options allow to switch to external installs of dependencies:

CMake Option	Values	Installs	Library	Version
`openPMD_USE_INTERNAL_VARIANT`	**ON**/OFF	Yes	MPark.Variant	1.3.0+
`openPMD_USE_INTERNAL_CATCH`	**ON**/OFF	No	Catch2	2.2.1+

3.5.5 Tests

By default, tests and examples are built. In order to skip building those, pass `-DBUILD_TESTING=OFF` or `-DBUILD_EXAMPLES=OFF` to your `cmake` command.

3.6 Sphinx

Section author: Axel Huebl

In the following section we explain how to contribute to this documentation.

If you are reading the HTML version on http://openPMD-api.readthedocs.io and want to improve or correct existing pages, check the "Edit on GitHub" link on the right upper corner of each document.

Alternatively, go to *docs/source* in our source code and follow the directory structure of reStructuredText (`.rst`) files there. For intrusive changes, like structural changes to chapters, please open an issue to discuss them beforehand.

3.6.1 Build Locally

This document is build based on free open-source software, namely Sphinx, Doxygen (C++ APIs as XML) and Breathe (to include doxygen XML in Sphinx). A web-version is hosted on ReadTheDocs.

The following requirements need to be installed (once) to build our documentation successfully:

```
cd docs/

# doxygen is not shipped via pip, install it externally,
# from the homepage, your package manager, conda, etc.
# example:
sudo apt-get install doxygen

# python tools & style theme
pip install -r requirements.txt # --user
```

With all documentation-related software successfully installed, just run the following commands to build your docs locally. Please check your documentation build is successful and renders as you expected before opening a pull request!

```
# skip this if you are still in docs/
cd docs/

# parse the C++ API documentation,
#   enjoy the doxygen warnings!
doxygen
# render the `.rst` files and replace their macros within
#   enjoy the breathe errors on things it does not understand from doxygen :)
```

(continues on next page)

(continued from previous page)

```
make html

# open it, e.g. with firefox :)
firefox build/html/index.html

# now again for the pdf :)
make latexpdf

# open it, e.g. with okular
build/latex/openPMD-api.pdf
```

3.6.2 Useful Links

- A primer on writing restFUL files for sphinx
- Why You Shouldn't Use "Markdown" for Documentation
- Markdown Limitations in Sphinx

3.7 Doxygen

Section author: Axel Huebl

An online version of our Doxygen build can be found at

http://www.openPMD.org/openPMD-api/

We regularly update it via

```
git checkout gh-pages

# optional argument: branch or tag name
./update.sh

git commit -a
git push
```

This section explains what is done when this script is run to build it manually.

3.7.1 Requirements

First, install Doxygen and its dependencies for graph generation.

```
# install requirements (Debian/Ubuntu)
sudo apt-get install doxygen graphviz

# enable HTML output in our Doxyfile
sed -i 's/GENERATE_HTML.*=.*NO/GENERATE_HTML     = YES/' docs/Doxyfile
```

3.7.2 Build

Now run the following commands to build the Doxygen HTML documentation locally.

```
cd docs/

# build the doxygen HTML documentation
doxygen

# open the generated HTML pages, e.g. with firefox
firefox html/index.html
```