

TUTORIAL

Virtual Object Layer (VOL)

Connector Construction Basics

February 2022



Dana Robinson
The HDF Group

Overview

Purpose of This Tutorial



Be the "Hello, world!" of VOL connector implementation

Outline



Part I

- VOL Overview
- VOL Toolkit Repository
- Constructing a Connector
- Mapping

Part II

- Tutorial VOL Connector
- Boilerplate
- File Operations
- Group Operations
- Dataset Operations

Assumptions and Limitations

You should...

- Have basic HDF5 knowledge
- Understand what the virtual object layer is (there will be a quick review)
- Be able to read C code

Limitations

- The tutorial VOL connector is designed for POSIX systems (i.e., no Windows yet)
- Covers terminal VOL connectors only (i.e., no pass-through)
- This is an introductory tutorial and sticks to the basics

*** Important HDF5 Version Note ***

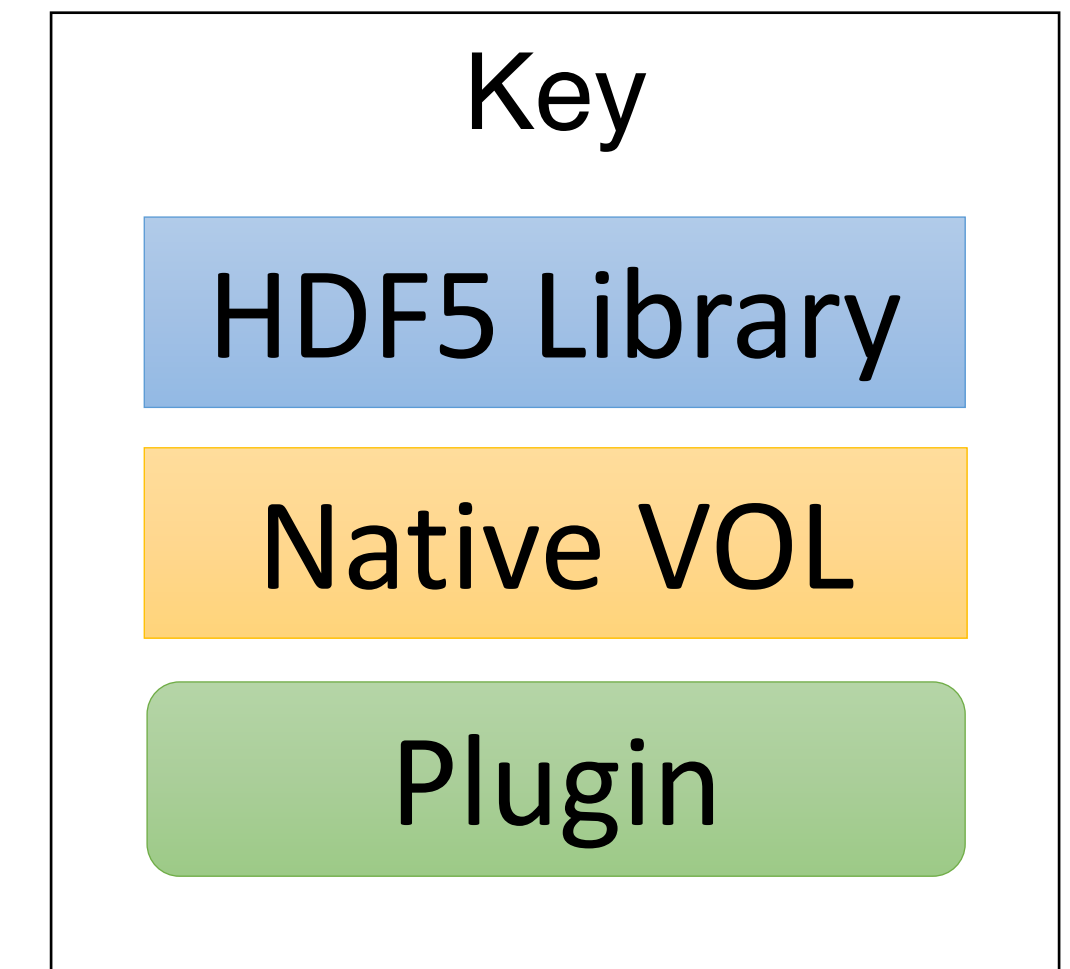
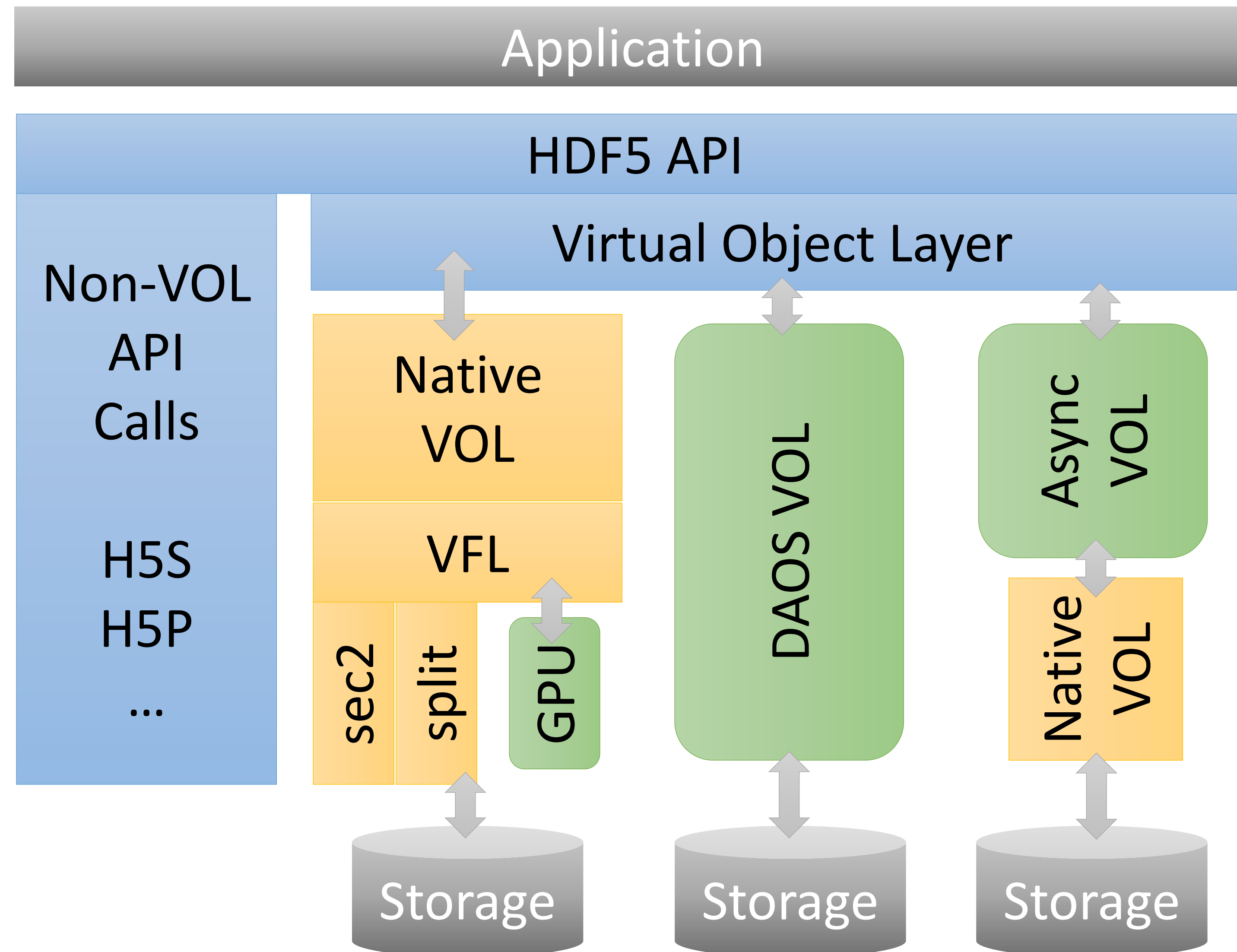
- **ALL VOL CONNECTOR DEVELOPMENT SHOULD TARGET HDF5 1.13.x**
- Do **NOT** use HDF5 1.12.x
- There were important changes to the VOL interface in HDF5 1.13.0 that could not be moved to 1.12.x without breaking binary compatibility
- Everything in the VOL toolkit repository and this tutorial targets HDF5 1.13.0
- Note that HDF5 1.13.0 is an experimental branch
 - It is possible that the VOL interface could be changed in the 1.13.x versions that will be released before HDF5 1.14.0
 - Should be fairly stable, though

Part I

Basics

VOL Overview

VOL Architecture



Virtual Object Layer (VOL)



- Sits between public HDF5 API calls and storage-oriented code
- Allows the creation of VOL connectors that perform arbitrary operations when storage-oriented calls (e.g., H5Dread) are called
- Passthrough connectors perform operations (logging, caching, mirroring, etc.) and then invoke another VOL connector layered underneath
- Terminal VOL connectors do not pass operations to other VOL connectors in a chain and are typically designed to map HDF5 file objects and metadata to storage
- VOL connectors can be written by users and loaded as plugins
- Non-storage HDF5 API calls do not go through the VOL (dataspace and property list calls, etc.)

The VOL Toolkit Repository

VOL Toolkit Repository



- Location: <https://github.com/HDFGroup/vol-toolkit>
- All your VOL construction needs in a single location
- Does not contain original content
- Designed to bring important content from other repositories together with consistent versioning
- Content is mainly included as git submodules, though the docs are currently copied in
- Tags will identify "HDF5 1.13.0", etc. versions of the toolkit
- Includes an appropriate version of HDF5

Documentation



Two documents are included in the toolkit

User's Guide

- Covers basic VOL operations like registration, handling plugin paths, etc.

Connector Author's Guide

- Helpful instructions for constructing VOL connectors
- RM for "connector author" calls that aren't covered in the main HDF5 API docs

Both were copied from the hdf5doc repository (<https://github.com/HDFGroup/hdf5doc/tree/master/RFCs/HDF5/VOL>)

Documentation - Public HDF5 Headers



There are several header files you may want to inspect while working on a VOL connector

H5VLpublic.h

- Public HDF5 VOL header
- Things needed by VOL users (registration API calls, etc.)

H5VLconnector.h

- Useful for constructing any VOL connector

H5VLconnector_passthru.h

- Useful for constructing pass-through VOL connectors
- Not useful for terminal connectors

These are all public headers, are distributed with the library, and included under `hdf5.h`

Documentation - Public HDF5 Headers (2)



H5Xdevelop.h - e.g., **H5ESdevelop.h**, **H5Idevelop.h**

- Headers that contain useful API calls for VOL, VFD, etc. authors
- Public, not really intended to be a part of the API an average user sees

These are all public headers, are distributed with the library, and included under `hdf5.h`

H5PLextern.h

- Plugin functionality
- NOT included in `hdf5.h`!

Templates



Two template repositories are linked in the toolkit

vol-template (<https://github.com/HDFGroup/vol-template>)

- Template for building terminal VOL connectors
- Build files + stubs.
- Developed and supported by THG
- Officially a "template repository" on github so you can clone + rename

vol-external-passthrough (<https://github.com/hpc-io/vol-external-passthrough>)

- Template for constructing pass-through connectors
- Has no-op, pass-through stubs for all callbacks
- Developed and supported by NERSC

Production Connectors (NOT in Toolkit)

When developing your own connector, it can be VERY helpful to see what others have done

Examples:

vol-daos (<https://github.com/HDFGroup/vol-daos>)

- Terminal VOL connector based on Intel's DAOS developed by THG
- Largely complete coverage of the HDF5 API
- Supports parallel HDF5 and async I/O

vol-async (<https://github.com/hpc-io/vol-async>)

vol-cache (<https://github.com/hpc-io/vol-cache>)

- Pass-through VOL connectors developed by NERSC
- Support parallel HDF5 (both) and async I/O (vol-async)

Find a full list here: <https://portal.hdfgroup.org/display/support/Registered+VOL+Connectors>

Testing Suite



A subset of the HDF5 library tests has been collected in a separate repository

vol-tests (<https://github.com/HDFGroup/vol-tests>)

- Requires CMake
- Supports parallel connectors and async
- No Windows support
- Tests a lot of the HDF5 API
- Tests the HDF5 command-line tools
- Expect a lot of failed tests until you have significant HDF5 API coverage in your connector
- Instructions for use located in the repository's README

Tutorial VOL Connector



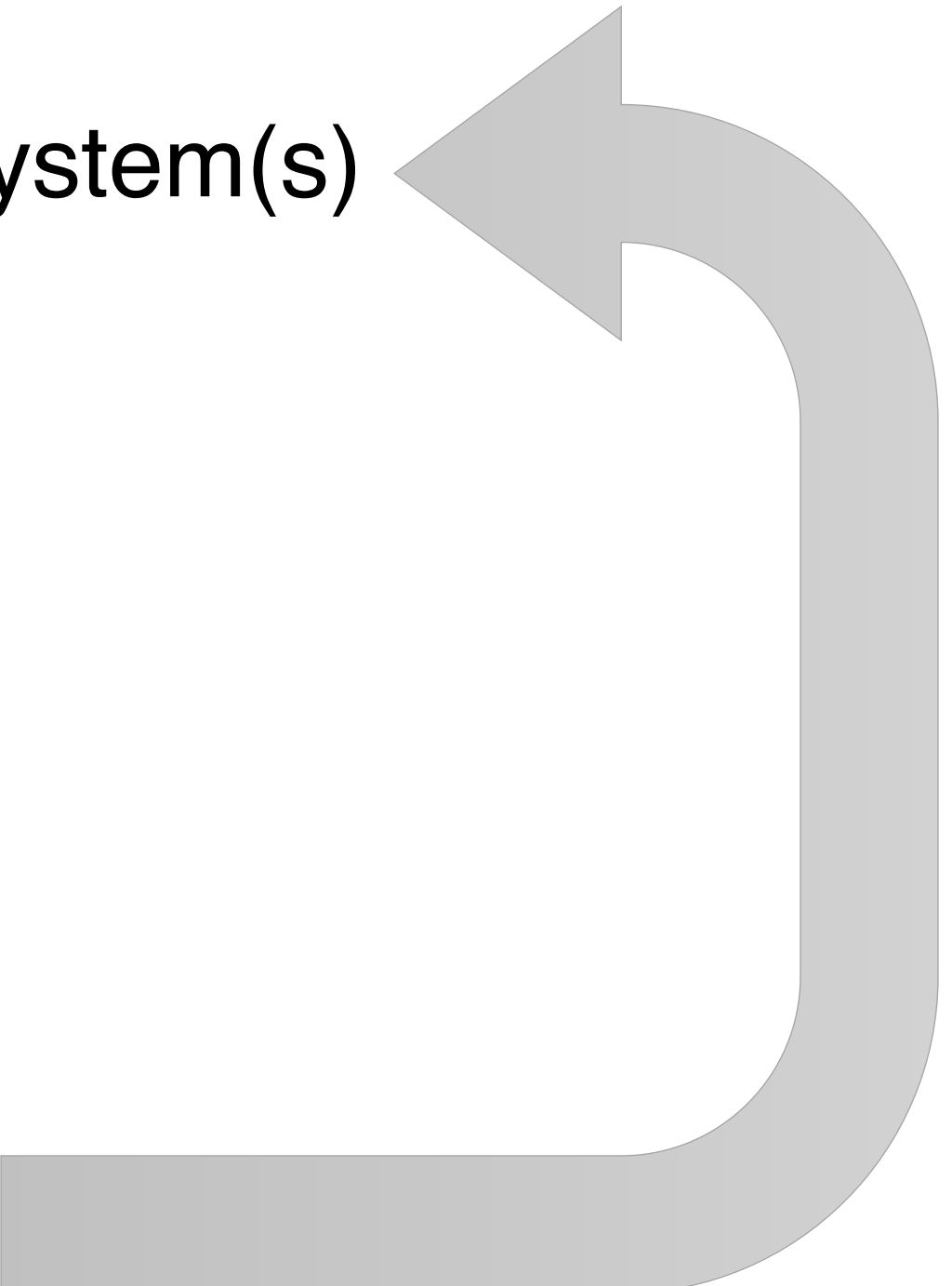
The connector covered in this tutorial

vol-tutorial (<https://github.com/HDFGroup/vol-tutorial.git>)

- Built using the template terminal VOL connector as a starting point
- Will include these slides

Constructing a Terminal Connector

Process

1. Select your data storage system(s)
 2. Clone the template repository (optional)
 3. Think about how to map HDF5 concepts to "things" in your data storage system(s)
 4. Start implementing functionality via callbacks in the `H5VL_class_t` struct
 - a. Boilerplate
 - b. File create/open/close/is accessible/delete
 - c. Group create/open/close
 - d. Dataset create/open/close/read/write
 - e. ...
 5. Repeat steps 3 & 4 as you work to make tests pass and discover issues.
- 

Mapping

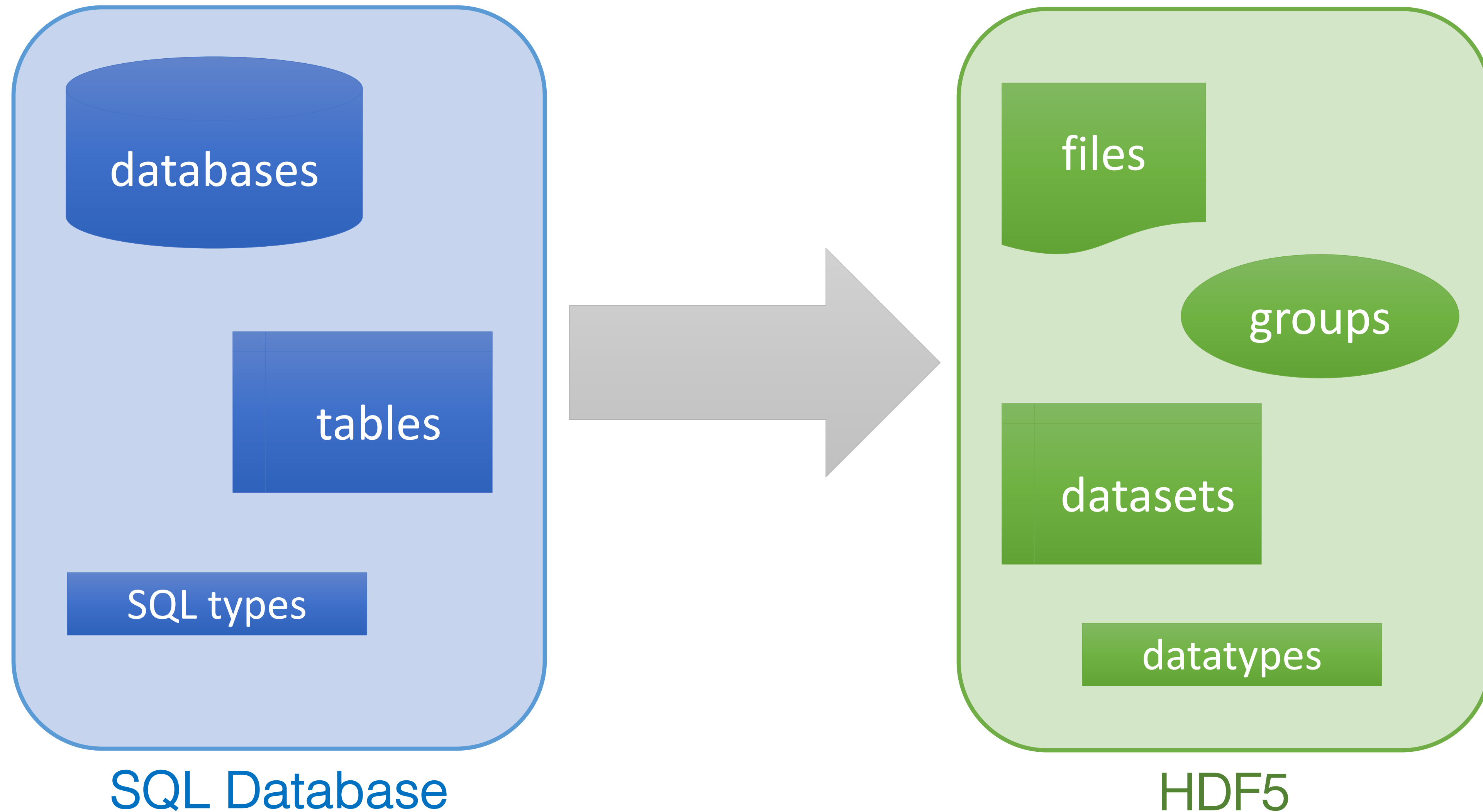


You will need to figure out how to implement several things using the primitives available in your data source

- Files
- File Objects
 - Groups
 - Datasets
 - Stored Datatypes
- Attributes
- Links
- References
- Tokens

You may not need all these things if you will be implementing a subset of the HDF5 API

Mapping (cont)



Mapping (cont)



Operations

You will also need to think about the operations that HDF5 permits (partial I/O, iterating, etc.) and how to map those to the operations provided by your storage system(s).

Type conversion may also be tricky. HDF5 has a rich type system and that may be difficult to map to your storage scheme(s).

Implementing Callback Functionality



HDF5 API calls invoke callbacks via the connector's `H5VL_class_t` struct

```
static const H5VL_class_t tutorial_vol_g = {
    H5VL_VERSION,          /* VOL class struct version */
    TUTORIAL_VOL_CONNECTOR_VALUE, /* value */
    TUTORIAL_VOL_CONNECTOR_NAME, /* name */
    0,                     /* connector version */
    0,                     /* capability flags */
    NULL,                  /* initialize */
    NULL,                  /* terminate */
    {
        /* file_cls */
        NULL, /* create */
        NULL, /* open */
        NULL, /* get */
        NULL, /* specific */
        NULL, /* optional */
        NULL, /* close */
    },
    ...
}
```

Implementing Callback Functionality (2)



The VOL Connector Authors Guide includes an appendix that maps callbacks to HDF5 API calls

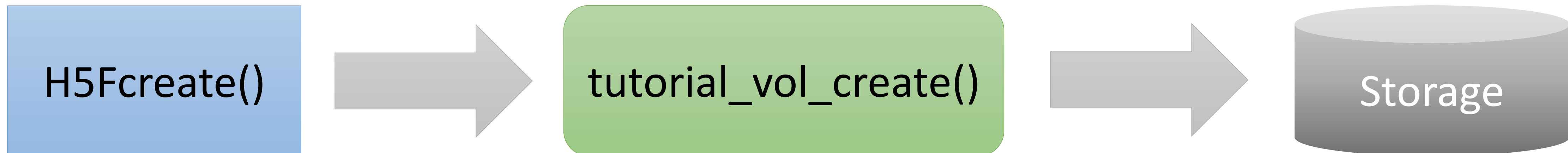
Appendix A Mapping of VOL Callbacks to HDF5 API Calls

VOL Callback	HDF5 API Call
FILE	
create	H5Fcreate
open	H5Fopen
get	H5Fget_access_plist H5Fget_create_plist H5Fget_fleno H5Fget_intent H5Fget_name H5Fget_obj_count H5Fget_obj_ids
specific	H5Fdelete H5Fflush H5Fis_accessible H5Fis_hdf5 (deprecated, hard-coded to use native connector) H5Freopen
close	H5Fclose

Implementing Callback Functionality (3)

For every HDF5 API call you want to support, you will have to implement the appropriate callback via your connector's class struct

The library will then invoke your callback via the VOL when working with files that were opened using your connector



State Management via hid_t IDs

Create/open callbacks typically return void pointers to whatever per-object/file/thing structure you create to manage your connector's state

Example: The create callback in the H5VL_file_class_t group

```
void *(*create)(const char *name, unsigned flags, hid_t fcpl_id,  
hid_t fapl_id, hid_t dxpl_id, void **req);
```

These pointers are then associated with the hid_t ID that is assigned to your newly created file/object/thing

```
hid_t fid = H5Fcreate(...); /* pointer to buffer stored in fid */
```

State Management via hid_t IDs (2)

The pointers managed by the IDs are then passed as parameters to callbacks that use the open object/file/thing.

Example: The get callback in the H5VL_file_class_t group

```
herr_t (*get)(void *obj, H5VL_datatype_get_args_t *args, hid_t  
dxpl_id, void **req);
```

And you can inspect and/or manipulate that state as you see fit

State Management via `hid_t` IDs (3)

HDF5 `hid_t` IDs are reference counted and when the reference count drops to zero, the tracked thing's close callback will be invoked

The state pointer will be passed in as a parameter

Example: The close callback in the `H5VL_file_class_t` group

```
herr_t (*close)(void *file, hid_t dxpl_id, void **req);
```

And you can then dispose of any resources you used

Mapping

Mapping HDF5 "Things" to Storage

This is probably the most difficult part of constructing a connector!

It is difficult to give specific advice as every situation will have different needs and constraints

Potential Problems

- The HDF5 data model may not map well to your storage scheme
- Common HDF5 data access patterns may not be performant for you
- You may need to do extra work outside of a normal HDF5 workflow

Don't feel you have to implement everything! (more on this later...)

Hybrid Systems



- Don't feel constrained to one data storage system
- Storing datasets, groups, attributes, internal metadata using different schemes is possible
 - You could store datasets/attributes in Redis and groups/links in Neo4j, for example
- Do what you need to do to make the system useful, robust, and performant
- You have a lot of freedom!

Files



An HDF5 file is a container for data and file metadata

- It does NOT have to be a "container" in your data storage scheme in the way that a single file or database contains data
- What is important is that it is the starting point for traversing the group/link graph
- Needs to store the root group
- Probably want some sort of marker to indicate "this is HDF5"

File Objects



Groups, datasets, stored datatypes are **HDF5 file objects**

They are manipulated via their own API calls (e.g., H5D for datasets) and as a class via the H5O API calls (H5Ovisit, etc.)

Groups

A collection of links

Datasets

n-dimensional data of a particular type

Stored Datatypes

A set of type information (class, size, byte order, signed/unsigned, etc.)

Note that HDF5 attributes are considered file metadata and *not* file objects

File Objects (cont)



Considerations

- They are targets of links, source might have been passed in as a file ID (implying root group)
- Need a way to handle them as a class in H5O calls
- Iteration calls like H5Ovisit can iterate by creation order or name
- API calls may locate objects by name or creation order index
- API calls exist that return the property lists and dataspace used to create the object. You'll need to cache or construct these
- Remember, some HDF5 API calls may not make sense for you. You may be able to no-op some of them (e.g., H5Dflush/refresh)

Attributes



Attributes are NOT file objects so they don't have to be treated like groups, datasets, etc.

- Need to be attachable to file objects
- Iteration calls like H5Aiterate can iterate by creation order or name
- API calls may locate attributes by name or creation order index
- API calls exist that return the property lists and dataspace used to create the attribute. You'll need to cache or construct these

Tokens (H50_token_t)

- VOL-agnostic replacement for HDF5 haddr_t addresses in the public API
 - 128 bits, interpreted by the VOL connector (defined in H5public.h)
- Exposed via H50get_info/iterate and used in H50open_by_token
- Also used in references
- Allows directly opening objects
- If you need more than 128 bits to uniquely identify an object, consider storing a key into some sort of dictionary where you could store more data

```
#define H50_MAX_TOKEN_SIZE (16)
```

```
typedef struct H50_token_t {  
    uint8_t __data[H50_MAX_TOKEN_SIZE];  
} H50_token_t;
```

Infrastructure



Some HDF5 functionality is VOL-independent

IDs (H5I API calls)

- A buffer you define will be associated with the `hid_t` ID at create/open time
- You close it in the close callback
- Library handles ID tracking and management otherwise

Property Lists (H5P API calls)

- Can inspect properties in lists passed to callbacks
- You can also add your own properties (via `H5Pset/get`)
- Some HDF5 API calls return property lists (e.g., `H5Fget_create_plist`) so you'll need to save or construct them if you want to support those calls
- Could use `H5Pencode/decode` or just handle the properties you care about individually

Infrastructure (2)

Dataspaces (H5S API calls)

- Inspect dataspaces to determine # dimensions, sizes, etc. when creating datasets, setting up I/O, etc.
- You will have to construct dataset dataspace to return to the user (e.g., H5Dget_space)

Non-committed Datatypes (many H5T API calls)

- HDF5 has a rich type system
- You will need to map this (or a subset) to your storage
- Many H5T API calls can help you create, inspect, and convert types

Native HDF5 API Calls



When creating a VOL connector, there are some things that are NOT considered to be a part of the abstract VOL model

- Chunking
- Compression
- Virtual File Drivers
- Caching (chunk, metadata)

Many API calls are considered to be a part of the native VOL connector

- Handled via the native VOL connector's optional callbacks
- A full list is included in the connector author's guide
- Example: `H5Fset_libver_bounds()`
- You can implement native calls in your own connector

How Much Do You Need To Implement?



- This depends on the needs of your users
- You do NOT need to implement the full HDF5 API, all callbacks, etc.
 - No VOL connector does this!
 - The native HDF5 connector doesn't even do this! (no H5M map calls, for example)
- Just implement what you need
 - Some API calls may not fit your storage system well
 - Some API calls may not be applicable to your use case(s)
- Consider the command-line tools
 - Being able to handle h5dump, h5ls, etc. is very useful
 - The tools use a lot of HDF5 API calls to do their work
 - We'll add a list of the calls used by the tools to the connector author's guide

Part II

Constructing a Simple Connector

Environment Setup

Building the Library



- HDF5 repository: (<https://github.com/HDFGroup/hdf5>)
- Must be HDF5 1.13.0 or later (develop branch okay)
 - HDF5 1.13.x versions **may change** the VOL interface
 - HDF5 1.14.x versions will have a **stable** VOL interface
- Don't build with the memory sanity checking feature!
 - This turns on our heap canary and tracking system
 - Can cause problems when reallocating buffers, etc.
 - `--enable-memory-alloc-sanity-check` (Autotools)
 - - **default ON in debug**, OFF in release
 - `HDF5_MEMORY_ALLOC_SANITY_CHECK` (CMake)
 - - default OFF (always)

The Tutorial VOL Connector

A Tutorial VOL Connector

- Terminal VOL connector
- Maps the HDF5 API to file system objects
- Constructed in stages
- Built as a plugin
- Both CMake and Autotools supported
- Designed for learning, not robustness!
 - Fragile
 - No real error checking
 - Don't code like this in real life

Sample tutorial "file" layout

```
$ tree dataset_ops.h5tut/  
dataset_ops.h5tut/  
├── dataset_group  
│   └── new_dset  
│       ├── new_dset.data  
│       ├── new_dset.dataspace  
│       ├── new_dset.datatype  
│       └── new_dset.fillval  
└── TUTORIAL_VOL_CONNECTOR_FILE
```

Repository Structure



- Location: <https://github.com/HDFGroup/vol-tutorial>
- Each section of this tutorial has an associated branch listed on the title card
- When following along, check out the branch and build the connector

Boilerplate

```
git checkout boilerplate
```

Boilerplate



- The class struct
- Versioning (VOL API and connectors)
- Connector names and values, registration

File Operations

```
git checkout file_operations
```

File Mapping



A tutorial VOL file is simply a directory that contains a special marker file

I'm using the extension .h5tut to indicate "file-ness"

Be careful with using .h5, .hdf5, etc.

The marker file has the name TUTORIAL_VOL_CONNECTOR_FILE and no content

Sample output

```
$ tree file_ops.h5tut/  
file_ops.h5tut/  
└─ TUTORIAL_VOL_CONNECTOR_FILE
```

Missing Callback Errors



- Sometimes the HDF5 library will invoke an unexpected callback
- The error stack will tell you what's missing (usually the last entry)

HDF5-DIAG: Error detected in HDF5 (1.13.1-1) thread 0:

```
#000: ../../hdf5/src/H5F.c line 661 in H5Fcreate(): unable to synchronously create file
major: File accessibility
minor: Unable to create file
```

```
#001: ../../hdf5/src/H5F.c line 615 in H5F__create_api_common(): unable to create file
major: File accessibility
minor: Unable to open file
```

```
#002: ../../hdf5/src/H5VLcallback.c line 3428 in H5VL_file_create(): file create failed
major: Virtual Object Layer
minor: Unable to create file
```

```
#003: ../../hdf5/src/H5VLcallback.c line 3390 in H5VL__file_create(): VOL connector has no 'file
create' method
major: Virtual Object Layer
minor: Feature is unsupported
```

File Operations



- Adding a callback
- Create
- Missing callback errors
- Open and close
- Delete, handling "specific" callbacks
- IsAccessible, "this is HDF5" markers

Group Operations

```
git checkout group_operations
```

Group Mapping

Groups are the next obvious thing to implement as every file contains a root group

Implemented as directories

Metadata, attributes, etc. could be stored in each group as needed

Sample output

```
$ tree group_ops.h5tut/  
group_ops.h5tut/  
├── group_1  
│   └── group_2  
└── TUTORIAL_VOL_CONNECTOR_FILE
```


Group Operations



- Create
- Open and close
- Handling the root group (has no ID, etc.)

Dataset Operations

```
git checkout dataset_operations
```

Dataset Mapping

Datasets are implemented as directories containing (text) data files

Storing all file objects in directories will more easily lead to equal treatment of file objects under the H5O API calls, etc.

Metadata like the dataspace, datatype, and fill value are stored in separate files

Sample output

```
$ tree dataset_ops.h5tut/  
dataset_ops.h5tut/  
├── dataset_group  
│   └── new_dset  
│       ├── new_dset.data  
│       ├── new_dset.dataspace  
│       ├── new_dset.datatype  
│       └── new_dset.fillval  
└── TUTORIAL_VOL_CONNECTOR_FILE
```

Dataset Operations



- Handling datatypes
- Handling dataspace
- Create
- Open and close
- Read and write

Recap

What Have We Covered?

- Reviewed the VOL architecture
- Looked at the "VOL toolkit" repository
- Showed how VOL connector functionality is implemented
- Discussed mapping storage/functionality to HDF5 API calls and VOL callbacks
- Implemented a simple VOL connector based on file system objects like directories and text files

Problems? Suggestions?



If you discover any bugs or have suggestions for anything in the VOL toolkit, please let us know!

- Post on the forum (<https://forum.hdfgroup.org>)
- Email the THG help desk (help@hdfgroup.org)
- Email me (derobins@hdfgroup.org)

THANK YOU!

Questions & Comments?