DE LA RECHERCHE À L’INDUSTRIE

High Level Interfaces & their Application Programming Interface

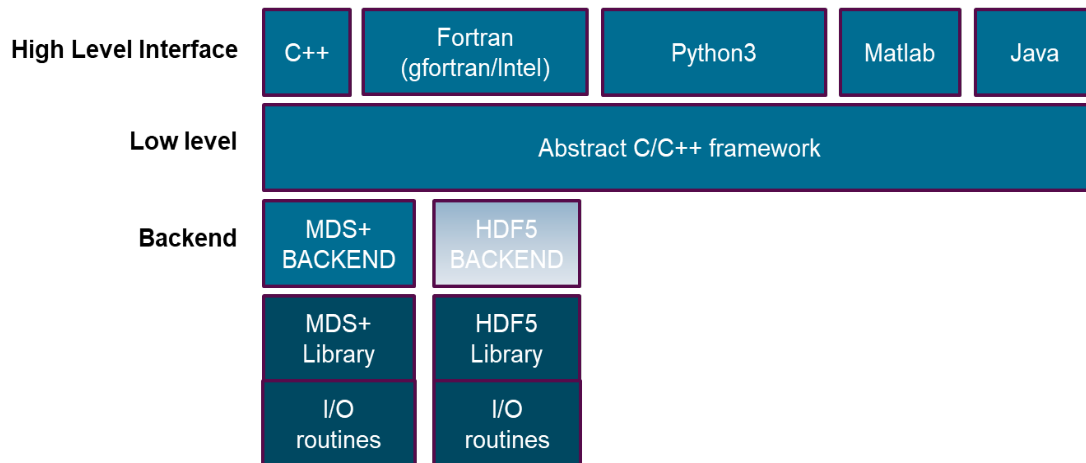
L. Fleury

20/09/2021

High Level Interfaces & API

- The IMAS Data Access Layer exposes a couple of operations (so-called **API**) for writing/reading IDSs data

- These data access operations are available from users code in the currently supported programming languages, the so-called **High Level Interfaces** (HLI):
  - Fortran
  - C++
  - Java
  - Python3
  - Matlab



API content & organization

- The **HLI API** can be divided into 2 sets of operations:
  - operations which apply on a **Data Entry**
  - operations which apply on an **IDS**
- A **Data Entry** is an IMAS concept for designating a collection of IDSs present in a local (pulse file) or a remote data source. A Data Entry is associated to a shot and a run number.
- The **HLI API** covers all available Access Layer features with the following exposed methods:

Operations on a Data Entry:

CREATE	(creates a new Data Entry)
OPEN	(opens an existing Data Entry)
CLOSE	(closes a Data Entry)

Operations on an IDS:

PUT	(writes data from an IDS to a Data Entry)
GET	(reads data of an IDS from an existing Data Entry)
PUT_SLICE	(writes a IDS time slice to a Data Entry)
GET_SLICE	(reads a time slice of an IDS from an existing Data Entry)
DELETE	(deletes an IDS from an existing Data Entry)

About the online tutorial

- In the online tutorial, each HLI API method is described using short code examples
- Code examples are using the Python **HLI**
- Provided Python3 programs can be executed interactively in a Python3 session or executed in a .py file
- API functions name and their signature may differ from one HLI to another
  - Documentation of all others HLIs is available in the User guide available from this page: https://confluence.iter.org/display/IMP/Integrated+Modelling+Home+Page

Creating a new Data Entry: create/close operations (1/2)

- The creation of a new Data Entry using the MDS+ or the HDF5 backend:
  - Consists in creating a new (MDS+/HDF5) pulse file on the disk
  - Requires to have at least an existing 'database' for hosting pulse file(s)

1. Let's first **create a new database** named 'data_access_tutorial' which will belong to the current user.
   From a new shell, execute the following commands:

```
>module load IMAS
>imasdb data_access_tutorial
```

The database is associated with a directory, let's **check** the latter has been successfully created by 'imasdb':

```
<g2lfleur@s52 ~>ls -alh ~/public/imasdb/data_access_tutorial
total 6.0K
drwxr-xr-x  3 g2lfleur g2itmdev 2.0K Sep 16 13:29 .
drwxr-xr-x  5 g2lfleur g2itmdev 2.0K Sep 16 13:29 ..
drwxr-xr-x 12 g2lfleur g2itmdev 2.0K Sep 16 13:29 3
```

Creating a new Data Entry: create/close operations (2/2)

2. Calling create() on an new Data Entry:

In this example, the MDS+ pulse file is created then closed. However no data have been yet saved to the pulse file.

```
1. import imas
   from imas import imasdef

2. #creating the Data Entry object 'data_entry' which handles the pulse file with
   shot=15000, run=1, belonging to database 'data_access_tutorial' of the current user,
   using the MDS+ backend
   data_entry = imas.DBEntry(imasdef.MDSPLUS_BACKEND, 'data_access_tutorial', 15000, 1)

3. #creating the pulse file handled by the Data Entry previously created
   data_entry.create()

   #now, we could perform some write operations using the put() operation
   #... This will be dealt with later

4. #closing the Data Entry
   data_entry.close()
```

```
<g2lfleur@s52 ~>ls -alh ~/public/imasdb/data_access_tutorial/3/0/
total 78M
drwxr-xr-x 2   g2lfleur g2itmdev 2.0K Sep 16 15:28 .
drwxr-xr-x 12 g2lfleur g2itmdev 2.0K Sep 16 13:29 ..
-rw-r--r-- 1     g2lfleur g2itmdev 42M Sep 16 15:28 ids_150000001.characteristics
-rw-r--r-- 1     g2lfleur g2itmdev 0       Sep 16 15:28 ids_150000001.datafile
-rw-r--r-- 1     g2lfleur g2itmdev 36M Sep 16 15:28 ids_150000001.tree
```

Opening an existing Data Entry: open/close operations

The following code opens the existing MDS+ pulse file created previously for shot=15000, run=1, from the 'data_access_tutorial' database of the current user:

```
1. import imas
   from imas import imasdef

2. #creating the Data Entry object 'data_entry' which handles the pulse file with shot=15000,
   run=1, belonging to database ‘data_access_tutorial' of the current user, using the MDS+ backend
   data_entry = imas.DBEntry(imasdef.MDSPLUS_BACKEND, 'data_access_tutorial’, 15000, 1)

3. #opening the pulse file handled by the Data Entry previously created
   data_entry.open()

   #now, we could perform some read/write operations using the get/put() operations
   #... This will be dealt with later

4. #closing the Data Entry
   data_entry.close()
```

Reading a magnetics IDS from WEST using get() (1/2)

Each IDS exposes the get() operation which reads **all** IDS data from an opened Data Entry.
When calling the get() operation on a IDS, **all** scalars and data arrays contained in the IDS are read.
All these data are put in memory.

ITER Physics Data Model Documentation for magnetics

Full path name	Description	Data Type	Coordinates
▸ ids_properties	Interface Data Structure properties. This element identifies the node above as an IDS	structure	
▾ flux_loop(i1)	Flux loops; partial flux loops can be described	struct_array [max_size=200 (limited in MDS+ backend only)]	1- 1...N
name	Name of the flux loop {static}	STR_0D	
identifier	ID of the flux loop {static}	STR_0D	
▸ type	Flux loop type. Available options (refer to the children of this identifier structure) : Name / Index / Description toroidal / 1 / Toroidal flux loop saddle / 2 / Saddle loop diamagnetic_internal / 3 / Diamagnetic internal loop diamagnetic_external / 4 / Diamagnetic external loop diamagnetic_compensation / 5 / Diamagnetic compensation loop diamagnetic_differential / 6 / Diamagnetic differential loop	structure	
▸ position(i2)	List of (R,Z,phi) points defining the position of the loop (see data structure documentation FLUXLOOPposition.pdf) {static}	struct_array [max_size=10 (limited in MDS+ backend only)]	1- 1...N
indices_differential(:)	Indices (from the flux_loop array of structure) of the two flux loops used to build the flux difference flux(second index) - flux(first index). Use only if ../type/index = 6, leave empty otherwise {static}	INT_1D	1- 1...2
area	Effective area (ratio between flux and average magnetic field over the loop) {static} [m^2]	FLT_0D	
gm9	Integral of 1/R over the loop area (ratio between flux and magnetic rigidity R0.B0). Use only if ../type/index = 3 to 6, leave empty otherwise. {static} [m]	FLT_0D	
▾ flux	Measured magnetic flux over loop in which Z component of normal to loop is directed downwards (negative grad Z direction) [Wb]. This quantity is COCOS-dependent, with the following transformation : Label / Expression psi_like / .fact_psi	structure	
data(:)	Data {dynamic} [as_parent]	FLT_1D	1- flux_loop(i1)/flux/time
validity_timed(:)	Indicator of the validity of the data for each time slice. 0: valid from automated processing, 1: valid and certified by the diagnostic RO; - 1 means problem identified in the data processing (request verification by the diagnostic RO), -2: invalid data, should not be used (values lower than -2 have a code-specific meaning detailing the origin of their invalidity) {dynamic}	INT_1D	1- flux_loop(i1)/flux/time
validity	Indicator of the validity of the data for the whole acquisition period. 0: valid from automated processing, 1: valid and certified by the diagnostic RO; - 1 means problem identified in the data processing (request verification by the diagnostic RO), -2: invalid data, should not be used (values lower than -2 have a code-specific meaning detailing the origin of their invalidity) {constant}	INT_0D	
time(:)	Time {dynamic} [s]	FLT_1D	1- 1...N

Reading a magnetics IDS from WEST using get() (2/2)

The code below reads an existing 'magnetics' IDS from a WEST pulse file:

```
1. #opening the existing Data Entry from WEST
data_entry = imas.DBEntry(imasdef.MDSPLUS_BACKEND, ‘west’, 54178, 0, ‘g2lfleur’)
(see code in a previous slide for completing this step)

2. #reading the 'magnetics' IDS from the Data Entry previously opened
magnetics_ids = data_entry.get('magnetics', 0) #argument 0 is the so-called IDS occurrence

3. #closing the Data Entry
data_entry.close()

#printing some IDS attributes
print('Number of flux loops = ', len(magnetics_ids.flux_loop))
print(‘First flux loop = ', magnetics_ids.flux_loop[0].flux.data)
print(‘Time basis = ', magnetics_ids.time)
```

Output:

```
Number of flux loops =  17
First flux loop =  [ 0.00065229  0.00163073  0.00489218... -0.01761185 -0.01663342  -0.01500269]
Time basis =  [ 1.83570397  1.86847198  1.90123999 ... 90.13289642 90.16566467  90.19843292]
```

Populating and writing a new magnetics IDS using put()

In order to write **all** data (scalars and data arrays) contained in an IDS to the pulse file created previously, we will call the put() operation which writes all static (non time dependent) AND dynamic data present in the IDS.

```
1. #opening the existing Data Entry (shot=15000, run=1)
   (see code in a previous slide)

2. #creating a new 'magnetics' IDS
   magnetics_ids = imas.magnetics()

3. #populating the ‘magnetics’ IDS
   magnetics_ids.ids_properties.homogeneous_time=1 #setting the homogeneous time (mandatory)
   magnetics_ids.ids_properties.comment='IDS created for testing the IMAS Data Access layer'
   magnetics_ids.flux_loop[0].flux.data = numpy.array([10,12,13.5,15]) #using some fancy values…
   magnetics_ids.time=numpy.array([0,1,2,3]) #the time(vector) must be not empty if
   homogeneous_time==1 otherwise an error will occur at runtime

4. #writing the magnetics IDS to the Data Entry
   data_entry.put(magnetics_ids, 0) #argument 0 is the so-called IDS occurrence

5. #closing the Data Entry
   data_entry.close()
```

An example of mixing get()/put() using 2 different Data Entries is given in the online tutorial

Steps for building an IDS with time slices using put_slice()

An IDS containing dynamic data structures can be built progressively using time slices.

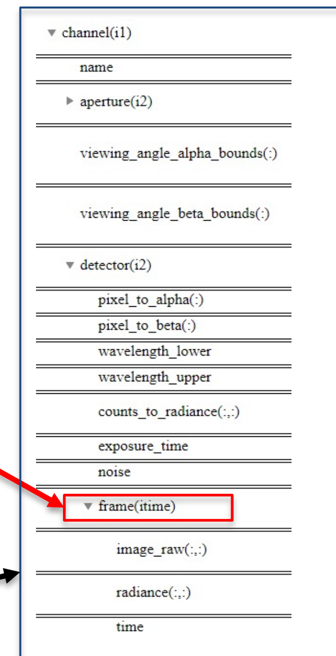
A dynamic data structure is either:

- an array with type INT_nD or FLT_nD where n=1 to 6, which holds a time coordinate
- or a so-called dynamic array of structures AOS[itime] where itime runs along a time index

In the online tutorial example, we illustrate the use of put_slice() on a 'camera_visible' IDS which contains the dynamic array of structures 'frame' (`camera_visible_ids.channel[0].detector[0].frame`).

Steps for building an IDS using time slices consist in:

1. opening an existing Data Entry (or creating a new one)
2. creating and initializing an IDS with static data
3. **setting the time slices container(s)**
4. **populating & writing time slices**
5. closing the Data Entry



Extract of the ITER Physics Data Model Documentation for 'camera_visible' IDS

Building a 'camera_visible' IDS using put_slice()

```
3. #setting time slices container(s) (camera_visible_ids.channel[0].detector[0].frame[0])
camera_visible_ids.channel.resize(1) #using only 1 channel (channel 0) for this example
camera_visible_ids.channel[0].detector.resize(1) #using only 1 detector for channel 0
camera_visible_ids.channel[0].detector[0].frame.resize(1) #it is the slice to be appended to the IDS
X = 3 #number of horizontal pixels of 2D 'image_raw' field
Y = 5 #number of vertical pixels of 2D 'image_raw' field
camera_visible_ids.channel[0].detector[0].frame[0].image_raw.resize(X,Y) #setting the size of the image
camera_visible_ids.time.resize(1) #the time vector contains only 1 element, it's the time of the slice
```

```
4. #populating & writing time slices
nb_slices=3 #number of time slices to be added
```

Only time slices of the 2D image_raw field of the first detector of the first channel are populated in this example:
camera_visible_ids.channel[0].detector[0].frame[0].image_raw

```
for i in range(nb_slices):
  camera_visible_ids.time[0] = float(i) #setting the time of the slice
  for j in range(X):
    for k in range(Y):
       #image_raw is a 2D array containing an image of X*Y pixels
       camera_visible_ids.channel[0].detector[0].frame[0].image_raw[j,k] = float(j + k +  i)
  if i==0:
    data_entry.put(camera_visible_ids) #the first slice i=0 has to be added using put()
  else:
    data_entry.put_slice(camera_visible_ids)  #appending current slice to the IDS using put_slice()
```

Getting a slice from previous ‘camera_visible’ IDS using get_slice()

Content of ‘camera_visible’ IDS:

(running previous code example using put_slice)

```
Slice time :0.0
Image raw:
[[0 1 2 3 4]
 [1 2 3 4 5]
 [2 3 4 5 6]]
-----
```

```
Slice time :1.0
Image raw:
[[1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]]
-----
```

```
Slice time :2.0
Image raw:
[[2 3 4 5 6]
 [3 4 5 6 7]
 [4 5 6 7 8]]
-----
```

Getting a time slice of ‘camera_visible’ IDS

```
1. #opening the existing Data Entry
(see code in a previous slide)

2. #getting a slice of ‘camera_visible’ IDS at the specified time using the closest interpolation
time_requested=1.
slice = data_entry.get_slice('camera_visible', time_requested, imasdef.CLOSEST_INTERP, 0)

print("Slice time: ", time_requested)
print("Image raw:")
print(slice.channel[0].detector[0].frame[0].image_raw)
print("-----")

3. #closing the Data Entry
data_entry.close()
```

Result:

(running this code)

```
Slice time : 1.0
Image raw:
[[1 2 3 4 5]
 [2 3 4 5 6]
 [3 4 5 6 7]]
-----
```