POWERFACTORY

PowerFactory 2020

Application Programming Interface (API)

PF2020

POWER SYSTEM SOLUTIONS

MADE IN GERMANY

Publisher:
DIgSILENT GmbH
Heinrich-Hertz-Straße 9
72810 Gomaringen / Germany
Tel.: +49 (0) 7072-9168-0
Fax: +49 (0) 7072-9168-88
info@digsilent.de

Please visit our homepage at:
`https://www.digsilent.de`

March 3, 2020
Revision 1

Contents

1 Introduction

The DIgSILENT ***PowerFactory*** Application Programming interface (API) offers third party applications the possibility to embed ***PowerFactory*** functionality into their own program. It offers direct access to the ***PowerFactory*** data model and gives access to the varied calculations and its results.

The API is designed as an automation interface; hence it requires detailed knowledge of the ***PowerFactory*** data model and how to achieve certain tasks manually, including knowledge about the participating objects and commands. It does not provide a pure calculation engine which can be fed with an abstract calculation topology.

Technically, the interface is realized in C++ and provided as a DLL that can dynamically be linked to any external application. The design idea was to keep the interface as small as possible while providing access to almost all ***PowerFactory*** data and function.

This document presents the structure of the API and how to include it in third party applications.

Experience with the internal scripting language DPL is helpful but not required.

Source Code snippets presented below are intended for demonstration only and are incomplete in such as sense that they cannot be compiled independently. For a working example on how to use the API please refer to the Visual Studio example project also contained in the package.

2 Overview

The ***PowerFactory*** API is a logical layer on top of the ***PowerFactory*** application that encapsulates the internal data structures and makes them available to external applications. Its purpose is to give a consistent interface being close to the ***PowerFactory*** data model. The API takes care about internal memory management and data persistency. It does not allow any external applications to access ***PowerFactory*** directly, all interaction from a 3rd party application are relayed via the API.

2.1 API Versioning Concept

Beginning with ***PowerFactory*** 16, the API has been redesigned to guarantee binary compatibility with future ***PowerFactory*** versions. This helps 3rd party application developers as they do not need to rebuild their application just to be able to make use of a newer ***PowerFactory*** version.

A consequence of this is the introduction of namespaces and folders to separate ***PowerFactory*** API versions from each other. Additionally all plain C functions defined in Api.hpp are suffixed by the version they belong too.

We advise to use typedefs for our API classes in 3rd party code to make the transition to a new API version as easy as possible, e.g.:

```
typedef api::v2::DataObject DataObject;
typedef api::v2::Application Application;
typedef api::v2::Api Api;
typedef api::v2::ExitError ExitError;
```

2.1.1 Migrating from API versions before 16.0

The API was split into multiple headers. Before recompiling any third party applications usage of the new headers must be ensured.

Due to the introduction of a namespace to separate different API versions the above mentioned typedefs should be introduced in third party applications. All compiler errors should be resolved to use these typedefs.

Another change is the introduction of a DataObjectPtr return type for all methods of the Value type which returned a DataObject in the old version. The Value type is version independent, therefore it does not need to know anything about the specific API version it is currently used with. All DataObjectPtr return values can be casted to the api version DataObject which is currently used in the application.

2.2 Version Overview

The ***PowerFactory*** API is currently available in following versions

Version	*Namespace*	*Availability*
1	api::v1	since ***PowerFactory*** 2016
2	api::v2	since ***PowerFactory*** 2018

These versions differ in the following ways

- **Version 1 (api::v1)**

  Initial version of the API.

- **Version 2 (api::v2)**

  Extension of Version 1. The following new functions for accessing content of OutputWindow have been added:

  - const Value* api::v2::OutputWindow::GetContent()
  - const Value* api::v2::OutputWindow::GetContent(MessageType filter)
  - void api::v2::OutputWindow::Save(const char* filePath)

  All functions from v1 are also available in v2.

  Introduced the exception ExitError to signal ***PowerFactory*** start-up errors and some run-time crashes.

Please note, all ***PowerFactory*** API versions are completely independent and cannot be mixed. Each 3rd party application has to decide for one of them. For new applications it is always recommended to use the latest interface version, e.g. api::v2.

Only the Value class is shared and identical for all versions.

2.3 Interface Data Model

The API consists of 5 different classes

Api	The entry point class
Application	Exposes the single running instance of ***PowerFactory*** .
OutputWindow	Allows to use the ***PowerFactory*** output window to display warnings, errors, etc.
DataObject	Encapsulates a ***PowerFactory*** object, e.g. an ElmTerm, ComImport, etc. and acts as a Proxy.
ExitError	Exception thrown when ***PowerFactory*** can not be started or on some crashes during run-time (new in api::v2).
Value	Encapsulation of data values acting as input or output for the API functions. A Value is a kind of variant used to offer a consistent interface while respecting different memory managements on the ***PowerFactory*** and external application side. The data stored in a Value object can be of different type (i.e. string, double, vector, DataObject, etc.).

2.4 Related Files

The interface consists of the following C++ include files

Api.hpp	Contains the definition of the Api class
Application.hpp	Contains the definition of the Application class
OutputWindow.hpp	Contains the definition of the OutputWindow class
DataObject.hpp	Contains the definition of the DataObject class
ExitError.hpp	Contains the definition of the ExitError class (new in api::v2)
ApiValue.hpp	Contains the definition of the Value class

and one static library which needs to be linked to the 3rd party application

digapivalue.lib	Contains the implementation of the Value class

The API itself is implemented in digapi.dll. This library needs to be run time dynamically linked to your application. (See 2.5).

2.5 Compiler Settings

To avoid errors at runtime, the compiler settings for the third party application using the API should be set as following:

- Character Set

  No multi-byte / Unicode character set

- Calling Convention

  cdecl

2.6 Loading digapi.dll

Run time dynamic linking is achieved by calling the Windows API method LoadLibraryEx. A detailed documentation can be found in the MSDN. Below is a snippet showing the preferred way of doing it.

```
void LoadPowerFactoryApi()
{
  HINSTANCE  dllHandle = LoadLibraryEx(
                           TEXT("E:\\\pf\\digapi.dll"),
                           NULL,
                           LOAD_WITH_ALTERED_SEARCH_PATH);
  if (!dllHandle) {
    throw std::exception("digapi.dll␣could␣not␣be␣loaded");
  }
  // use the handle
}
```

Using LoadLibraryEx with a full path to the digapi.dll has the benefit that the 3rd party application does not need to reside in the ***PowerFactory*** installation directory.

The modifier LOAD˙WITH˙ALTERED˙SEARCH˙PATH is required because ***PowerFactory*** will in turn load additional libraries that would not be found if this modifier is omitted.

2.7 Extracting Methods from digapi.dll

The ***PowerFactory*** API provides a handful of C methods defined in Api.hpp which serve as entry point (CreateXXX() and DestroyXXX()). They are located at the bottom of the file, wrapped in an extern "C" section.

To access such a method from a dynamically loaded DLL the Windows API provides the method GetProcAddress which is documented in the MSDN. The basic idea is to retrieve a function pointer from the DLL and cast it to the desired function signature. From there on it can be called like any other function. The snippet below shows the important steps, comprehensive error handling has been omitted.

Example for api::v2

```
extern "C" {
typedef Api* (__cdecl *CREATEAPI)(const char* username,
                                  const char* password,
                                  const char* commandLineArguments);
}

Api* CreateApiInstance(const char* withUsername,
                       const char* withPassword,
                       const char* withCommandLineArguments,
                       HINSTANCE fromDll)
{
  Api* instance(NULL);

  CREATEAPI createApi = (CREATEAPI)GetProcAddress(fromDll,
                                                  "CreateApiInstanceV2");
  try {
    createApi(withUsername,
              withPassword,
              withCommandLineArguments);
  }
  catch(const api::v2::ExitError& ex) {
    std::cout << "API␣instance␣creation␣failed␣with␣error␣code␣"
              << ex.GetCode() << std::endl;
    throw;
  }
  return instance;
}
```

2.8 Memory Management

PowerFactory has its own memory management. Therefore it must be guaranteed memory allocated from ***PowerFactory*** will only be released by ***PowerFactory*** and memory allocated by the 3rd party application will only be released by the 3rd party application.

2.8.1 Objects of Type Value

This leads to the requirement of having a transfer object for complex types between these two worlds, the Value object. The Value object is a variant type and provides the necessary methods to retrieve the stored content. A Value object returned by the API must not be deleted. Ownership remains on the API side.

Likewise will the API never delete a Value object passed to it. It is therefore safe to create Value objects on the stack and pass them to the API even though the API requires a pointer to the object.

To clean up Value objects returned from the API the method ReleaseValue is provided on an instance of the API object. Calling this method tells the API you do not need this object any more and it can be destroyed. Accessing the Value object afterwards leads to undefined behaviour.

As this Value is identical for all interface versions, it's located in the root namespace ”api” (api::Value).

2.8.2 Objects of Type DataObject

A DataObject instance encapsulates a ***PowerFactory*** object and provides access to it. It is a proxy object and also bound to the rules mentioned above. A DataObject returned by the API must not be deleted directly. The API object provides the method ReleaseObject to tell the API you do not need the proxy any more and it can be deleted. This will only release the proxy object, the ***PowerFactory*** object itself remains.

DataObjects cannot be instantiated by 3rd party code directly.

DataObjects behave a bit differently than Value objects. By default, each request for a DataObject proxy for a specific ***PowerFactory*** object will return the very same proxy object until ReleaseObject is called for that proxy. DataObject instances are recycled as long as the same ***PowerFactory*** object is requested. This is done for performance reasons and for convenience regarding object identification by pointer comparison. However, it requires a careful thinking about the responsibility for a DataObject proxy returned by the API.

Users feeling uncomfortable with this behaviour can use the method SetObjectReusingEnabled to disable the object re-using. But this can have a negative impact on the API performance and has to be evaluated individually.

Calling any method on a DataObject which was previously released leads to undefined behaviour.

3 Working with an API Instance

The entry point to ***PowerFactory*** is an instance of the API. It needs to be created via the C method CreateApiInstanceVX and should be destroyed by DestroyApiInstanceVX.

3.1 Create an Instance of the API

Use the function CreateApiInstanceVX to create an instance of the API; only one instance can be created at the time. Trying to create multiple instances will lead to undefined behaviour. When creating an API instance fails e.g. an error occured while trying to access licence, then CreateApiInstanceV1 returns NULL and CreateApiInstanceV2 throws an ExitError exception. For an example see 2.7.

```
v1::Api* CreateApiInstanceV1(const char* username,
                             const char* password,
                             const char* commandLineArguments);

v2::Api* CreateApiInstanceV2(const char* username,
                             const char* password,
                             const char* commandLineArguments);
```

Parameters:

username	Name of the user to log in to ***PowerFactory*** . Can be NULL to enforce the default behaviour as if ***PowerFactory*** was started via shortcut.
password	The password for the user which should be logged in. Can be NULL to omit the password.
commandLine	Additional command line options. These need to be specified in the same way as if ***PowerFactory*** was started via a command shell. Can be NULL to omit the command line arguments.

3.2 Delete an Instance of the API

Using DestroyApiInstanceVX, the corresponding instance of the API will be deleted and the memory occupied by the created objects within this instance of the API will be freed.

If CreateApiInstanceVX has started ***PowerFactory*** , then calling DestroyApiInstanceV1 additionally terminates the whole proccess and DestroyApiInstanceV2 only terminates ***PowerFactory*** (but ***PowerFactory*** can not be re-started in the same process).

DestroyApiInstanceVX parameters:

instance	Pointer to a ***PowerFactory*** API instance which should be destroyed. The destroy function is version specific and can only release instances of the corresponding version.

3.3 Handle run-time exit errors

Consider the situation that the instance of the API was created with CreateApiInstanceV2 and that ***PowerFactory*** crashes e.g. during some calculation. In the most cases an ExitError exception is throw shortly before ***PowerFactory*** terminates. So for a 3rd party application with a global exception handler it is possible to do some clean-up in this situation.

4 Working with *PowerFactory* objects

PowerFactory presents data and functionality encapsulated in objects. The class of an object can be seen in ***PowerFactory*** as a suffix to an objects name in the dialogue title while editing an object. Below is an example for a load flow command, whose class is ComLdf.

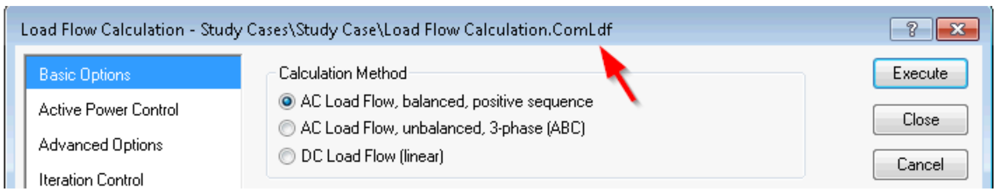
The class defines the attributes and operations an object provides.

Objects are organised in a tree structure, much like the folders of the Windows file system. Navigating in the tree is done via GetChildren and GetParent of an object. These methods are accessible on the DataObject proxy. Certain folders can be used as entry point for the navigation, e.g. the current user or the active project. The Application class of the API provides various methods to access specific folders.

4.1 Accessing Object Attributes

To access any attribute of an object, its type must be known. Therefore, the function GetAttributeType returns the type of the corresponding attribute. The attribute types are defined in DataObject.hpp

```
enum AttributeType {
  TYPE_INVALID      = -1,

  TYPE_INTEGER      = 0,
  TYPE_INTEGER_VEC  = 1,

  TYPE_DOUBLE       = 2,
  TYPE_DOUBLE_VEC   = 3,
  TYPE_DOUBLE_MAT   = 4,

  TYPE_OBJECT       = 5,
  TYPE_OBJECT_VEC   = 6,

  TYPE_STRING       = 7,
  TYPE_STRING_VEC   = 8,

  TYPE_INTEGER64    = 9,
  TYPE_INTEGER64_VEC = 10,
};
```


Load Flow Calculation - Study Cases\Study Case\Load Flow Calculation.ComLdf

Calculation Method
[✓ AC Load Flow, balanced, positive sequence]
AC Load Flow, unbalanced, 3-phase (ABC)
DC Load Flow (linear)

Figure 4.1: Dialogue of load flow command (ComLdf)

Once the type has been identified, methods like GetAttributeInt, GetAttributeDouble, GetAttributeObject, GetAttributeString return respectively the content of the attribute as an int, double, DataObject or Value.

Container types can be accessed by using GetAttributeContainer or the appropriate overload of the methods above which also accept indices. The size of a container can be determined by calling GetAttributeSize.

A few DataObject methods will either return a Value object or require one as input. The Value object encapsulates non-primitive types like strings or containers and provides methods to access the encapsulated values.

4.2 Modifying Objects and Attributes

Objects and their attributes can be modified using one of the type bound methods.

For matrices and vectors, overloaded methods of the above ones allowing targeting a specific element (row and column indices) are available. The method ResizeAttribute resizes a vector or matrix attribute.

Blocks of attributes can be accessed or modified at once using DefineTransferAttributes, GetAttributes and SetAttributes.

4.3 Creating and Deleting Objects

As ***PowerFactory*** stores all objects in a tree hierarchy objects need to be created with a parent so they can be sorted into the right place in the tree. The DataObject provides the method CreateObject which needs to be given a class name for the correct ***PowerFactory*** object to be created.

The created object can be used immediately.

To delete an object the method DeleteObject must be called on itself.

After calling DeleteObject the DataObject pointer may no longer be used and should be released See 2.8.2.

5 Executing Commands

The Application and DataObject classes contain an Execute method to execute generically commands.

The syntax of the method is

```
Value* Execute(const char* command, const Value* inArgs, int* error /*=NULL*/);
```

where

command	is the name of the function to be executed
inArgs	is(are) the required argument(s), depending on the executed function
error	is the returned error code (0 on success)

The available commands are listed in the Python Function Reference.

Commands with more then one argument can be executed with 'inArgs' of type vector.

Example:

```
Application* app(apiInstance->GetApplication());

// getting all versions of the active project
const Value* activeProject(app->Execute("GetActiveProject", NULL, error));
const Value* versions(app->Execute("GetVersions", NULL, error));

// collecting all lines
const Value* lines(app->Execute("GetCalcRelevantObjects", &Value("*.ElmLne"), error));

// collecting lines which are not out of service
Value arguments(Value::VECTOR);
arguments.VecInsertString("*.ElmLne");
arguments.VecInsertInteger(0);
const Value* linesInService(app->Execute("GetCalcRelevantObjects", &arguments, error));

// releasing received values
apiInstance->ReleaseValue(activeProject)
apiInstance->ReleaseValue(versions)
apiInstance->ReleaseValue(lines)
apiInstance->ReleaseValue(linesInService)
```

6 Class Reference

6.1 api::v1::Api

6.1.1 GetVersion

```
Value* GetVersion()
```

Returns the version of the current API instance

Arguments:

none

Return value:

A pointer of type Value pointing to a string with the version of the current API.

Example:

The following example displays in the command window the version of the api used to create the running instance.

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
std::cout << std::endl
          << apiInstance->GetVersion()->GetString()
          << std::endl;
```

6.1.2 ReleaseObject

```
int ReleaseObject(const DataObject* object)
```

Releases a DataObject instance. All DataObject pointers returned by an api must be released using this function. Calling delete from an external DLL is not possible as the API instance has its own memory management. Each object must only be released once.

Arguments:

const DataObject*	the pointer to the object to be released

Return value:

0 on success, non-zero on error

Example:

```
DataObject* user(apiInstance->GetApplication()->GetCurrentUser());
apiInstance->ReleaseObject(user);
```

6.1.3 ReleaseValue

```
int ReleaseValue(const Value* object)
```

Releases an API value instance. All API value pointers returned by the api must be released using this function. Calling delete from an external DLL is not possible as the API instance has its own memory management.

Arguments:

const Value*	pointer to the object to be released

Return value:

0 on success, non-zero on error

Example:

```
DataObject* user(apiInstance->GetApplication()->GetCurrentUser());
Value* name = user->GetName();
apiInstance->ReleaseValue(name);
apiInstance->ReleaseObject(user);
```

6.1.4 GetApplication

```
Application* GetApplication()
```

The function returns an instance of Application. There exists only one Application object per Api instance. This object must not be deleted.

Arguments:

None

Return value:

Pointer to the instance of an Application object, never NULL

Example:

The following example displays in the command window the version of the running instance of ***PowerFactory*** .

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
std::cout << "PowerFactory␣version:"
          << apiInstance->GetApplication()->GetVersion()->GetString()
          << std::endl;
```

6.1.5 IsDebug

```
bool IsDebug()
```

Arguments:

None

Return value:

True if ***PowerFactory*** is in debug mode; false otherwise.

Example:

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
if(apiInstance->IsDebug())
{
  std::cout << "PowerFactory␣running␣in␣debug␣mode" << std::endl;
}
```

6.2 api::v1::Application

6.2.1 GetVersion

```
const Value* GetVersion()
```

The function returns the version of the currently running ***PowerFactory*** , e.g. ”14.0.505”

Arguments:

None

Return value:

Pointer to a Value object holding version information of ***PowerFactory*** application; returned string is never null and must be released when no longer in use.

Example:

The following example displays in the command window the version of the running instance of ***PowerFactory*** .

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
std::cout << "PowerFactory␣version:"
          << apiInstance->GetApplication()->GetVersion()->GetString()
          << std::endl;
```

6.2.2 GetTempDirectory

```
const Value* GetTempDirectory()
```

Returns the path to the temporary directory of the running instance of ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value pointing to a string with the temporary directory of ***PowerFactory*** .

Example:

```
std::cout << apiInstance->GetApplication()->GetTempDirectory()->GetString()
          << std::endl;
```

6.2.3 GetWorkingDirectory

```
const Value* GetWorkingDirectory()
```

Returns the path to the working directory of the running instance of ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value holding a string with the working directory of ***PowerFactory*** .

Example:

```
Application* application(apiInstance->GetApplication());
std::cout << application->GetWorkingDirectory()->GetString()
          << std::endl;
```

6.2.4 GetInstallationDirectory

```
const Value* GetInstallationDirectory()
```

Returns the path to the installation directory of the running instance of ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value holding a string with the installation directory of ***PowerFactory*** .

6.2.5 GetLanguageCode

```
const Value* GetLanguageCode()
```

Returns the language code of the currently used display language in ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value holding a string with the language code.

6.2.6 GetOutputWindow

```
OutputWindow* GetOutputWindow()
```

This function returns an instance of the running ***PowerFactory*** output window. Each api instance has only one output window instance.

Arguments:

None

Return value:

Returns a pointer to an instance of OutputWindow, never NULL

6.2.7 GetCurrentUser

```
DataObject* GetCurrentUser()
```

This function returns the current user.

Arguments:

None

Return value:

Returns a pointer to the currently logged in user object.

Example:

The following example displays the name of the current user in the command window.

```
int error = 0;
DataObject* user = apiInstance->GetApplication()->GetCurrentUser();
std::cout << *(user->GetAttributeString("loc_name",error)) << std::endl;
```

6.2.8 GetActiveProject

```
DataObject* GetActiveProject()
```

Returns a pointer to the currently active ***PowerFactory*** project, NULL if there is no active project.

Arguments:

None

Return value:

Pointer to the currently active ***PowerFactory*** project

6.2.9 GetActiveStudyCase

```
DataObject* GetActiveStudyCase()
```

Returns a pointer to the currently active study case, NULL if there is no active case.

Arguments:

None

Return value:

Pointer of type Value to the currently active study case

6.2.10 GetCalcRelevantObjects

```
const Value* GetCalcRelevantObjects()
```

This function returns all objects that are currently relevant for calculation: lines, nodes, switches, transformers, etc. and their types.

Arguments:

None

Return value:

Returns a pointer of type Value to a vector of objects relevant for calculation, never NULL. The container must be released if no longer in use.

6.2.11 GetClassId

```
int GetClassId(const char* className)
```

This function returns the class identifier integer of the considered class className. Each class name corresponds to one unique index. The mapping of class name might be different according to the build version of ***PowerFactory*** , but it is guaranteed that it will not change while an Api instance exists. This indices should not be stored statically but rather be generated dynamically in the code using the GetClassId method.

Arguments:

const char* className	the name of the considered class

Return value:

Returns an integer representing the index (¿0) of the considered class; 0 if className is not a valid class name.

Example:

```
int filterID = apiInstance->GetApplication()->GetClassId("intPrj");
std::cout << "Project␣ID␣(intPrj):" << filterID << std::endl;
```

6.2.12 GetClassDescription

```
const Value* GetClassDescription(const char* className)
```

Returns the description of a ***PowerFactory*** class.

Arguments:

const char* className	name of the considered ***PowerFactory*** class

Return value:

Returns the description text, never NULL; but the string is empty for invalid class names

Example:

The following example displays the description text of the class intPrj

```
Application* application(apiInstance->GetApplication());
std::cout << "intPrj␣description␣text:"
          << application->GetClassDescription("intPrj")->GetString()
          << std::endl;
```

6.2.13 AttributeMode

```
enum AttributeMode {
  MODE_DISPLAYED    = 0,
  MODE_INTERNAL     = 1
};
```

Enumerated type for accessing object attributes in ***PowerFactory*** .

MODE_DISPLAYED	as displayed in PF (unit conversion applied)
MODE_INTERNAL	as internally stored

6.2.14 SetAttributeMode

```
void SetAttributeMode(AttributeMode mode)
```

Changes the way how attribute values are accessed

Arguments:

AttributeMode mode	the way the attribute values should be accessed

Return value:

void

6.2.15 GetAttributeMode

```
AttributeMode GetAttributeMode()
```

Returns the mode how object attributes are accessed.

Arguments:

None

Return value:

Current mode as AttributeMode

6.2.16 SetWriteCacheEnabled

```
void SetWriteCacheEnabled(bool enabled)
```

This function intends to optimize performances. In order to modify objects in ***PowerFactory*** , those must be set in a special edit mode before any value can be changed. Switching back and forth between edit mode and stored mode is time consuming; enabling the write cache flag will set objects in edit mode and they will not be switched back until WriteChangeToDb is called.

Default value: disabled.

Arguments:

bool enabled	true = enabled; false = disabled

Return value:

void

Example:

```
apiInstance->GetApplication()->SetWriteCacheEnabled(true);
```

6.2.17 IsWriteCacheEnabled

```
bool IsWriteCacheEnabled()
```

Returns whether or not the cache method for optimizing performances is enabled.

Arguments:

None

Return value:

boolean: whether or not the cache method for optimizing performances is enabled

Example:

```
bool cacheEnabled;
cacheEnabled = apiInstance->GetApplication()->IsWriteCacheEnabled();
```

6.2.18 SetObjectReusingEnabled

```
void SetObjectReusingEnabled(bool enabled)
```

This option allows to change the behavior of creation vs. re-using api::DataObject instances for identical ***PowerFactory*** objects. When enabled access to the same ***PowerFactory*** object will result in usage of same api::DataObject instance until the instance has explicitly been released via ReleaseObject() call.

This affects the behavior of all api functions returning api::DataObject pointers.

Default value: enabled.

Arguments:

bool enabled	true = enabled; false = disabled

Return value:

void

6.2.19 IsObjectReusingEnabled

```
bool IsObjectReusingEnabled()
```

Returns whether or not object reusing is currently enabled.

Arguments:

None

Return value:

boolean: whether or not object reusing is enabled

6.2.20 GetAttributeType

```
DataObject::AttributeType GetAttributeType(const char* classname,
                                           const char* attribute)
```

This function returns the actual type of an object attribute.

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which type must be returned

Return value:

Returns the type of the attribute or TYPE_INVALID on error (no attribute of that name exists)

TYPE_INVALID	error
TYPE_INTEGER	int
TYPE_INTEGER_VEC	vector of type int
TYPE_DOUBLE	double
TYPE_DOUBLE_VEC	vector of double
TYPE_DOUBLE_MAT	2D matrix of double
TYPE_OBJECT	DataObject
TYPE_OBJECT_VEC	vector of DataObject
TYPE_STRING	string literal
TYPE_STRING_VEC	vector of string literals
TYPE_INTEGER64	__int64
TYPE_INTEGER64_VEC	vector of __int64

6.2.21 GetAttributeDescription

```
const Value* GetAttributeDescription(const char* classname,
                                     const char* attribute)
```

Returns the description of an attribute, NULL if the attribute does not exist

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which description must be returned

Return value:

Pointer of type Value to the current attribute description (=string)

6.2.22 GetAttributeUnit

```
const Value* GetAttributeUnit(const char* classname, const char* attribute)
```

Returns the unit of an attribute, e.g. km, MW...; NULL if the given attribute name does not exist; the string is empty for attributes without units

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which description must be returned

Return value:

Pointer of type Value to the units of the considered attribute (=string)

6.2.23 GetAttributeSize

```
void GetAttributeSize(const char* name,
                      const char* attribute,
                      int& countRows,
                      int& countCols)
```

This function returns the size of object attribute if this attribute is a vector or a matrix.

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which description must be returned
int& countRows	the returned number of rows
int& countCols	the returned number of columns

Return value:

void

6.2.24 IsAttributeReadOnly

```
bool IsAttributeReadOnly(const char* classname, const char* attribute) const
```

Checks whether the attribute with given name can be written via the api. Typical read-only attributes are calculation results.

Please note, write access can still fail if the object that should be modified is part of an inactive project or access rights are missing.

Arguments:

const char* classname	class name for which the attribute is considered
const char* attribute	attribute that should be checked

Return value:

boolean: whether or not object reusing is enabled

6.2.25 DefineTransferAttributes

```
virtual void DefineTransferAttributes(const char* classname, const char* attributes, int* erro
```

Allows to specify for a given classname a comma separated list of attributes which should be written in one go. The values can then be passed using DataObject::SetAttributes.

Arguments:

const char* classname	class name for which the attribute is considered
const char* attributes	attribute that should be checked

Return value:

void

6.2.26 Execute

```
Value* Execute(const char* command, const Value* inArgs, int* error)
```

This function executes a command on the instance of the application.

The available commands are listed in the Python Function Reference. Commands with more then one argument can be executed with 'inArgs' of type vector.

Arguments:

const char* command	the command that should be executed
const Value* inArgs	input arguments for the command to be executed
int* error	returned error code (0 on success)

Return value:

The function returns a pointer of type Value to the result of the command if applicable.

Example:

```
Application* app(apiInstance->GetApplication());

// getting the active project
const Value* activeProject(app->Execute("GetActiveProject", NULL, error));

// collecting all lines
const Value* lines(app->Execute("GetCalcRelevantObjects", &Value("*.ElmLne"), error));

// collecting lines which are not out of service
Value arguments(Value::VECTOR);
arguments.VecInsertString("*.ElmLne");
arguments.VecInsertInteger(0);
const Value* linesInService(app->Execute("GetCalcRelevantObjects", &arguments, error));

// releasing received values
apiInstance->ReleaseValue(activeProject)
apiInstance->ReleaseValue(lines)
apiInstance->ReleaseValue(linesInService)
```

6.2.27 WriteChangesToDb

```
void WriteChangesToDb()
```

This function combined with SetWriteCacheEnabled is meant to optimize performances. If the write cache flag is enabled all objects remain in edit mode until WriteChangesToDb is called and all the modifications made to the objects are saved into the database.

Arguments:

None

Return value:

void

6.3 api::v1::OutputWindow

6.3.1 MessageType

```
enum MessageType
{
  M_PLAIN = 0,
  M_ERROR = 1,
  M_WARN  = 2,
  M_INFO  = 4
};
```

Enumerated type which defines the message type.

M_PLAIN	message not prepended by any text
M_ERROR	message prepended by error prefix, will also popup as error dialog
M_WARN	message prepended by warning prefix
M_INFO	message prepended by info prefix

6.3.2 Print

```
void Print(MessageType type, const char* msg)
```

This function prints a message to the PowerFactory output window.

Arguments:

MessageType type	type of message
const char* msg	the actual message to be displayed

Return value:

void

Example:

The following example displays a message in the PowerFactory output window.

```
OutputWindow* outWindow = apiInstance->GetApplication()->GetOutputWindow();
outWindow->Print(OutputWindow::M_INFO,"Running␣PowerFactory␣from␣the␣API");
```

6.3.3 Clear

```
void Clear()
```

Empties the content of the output window.

Arguments:

None

Return value:

void

Example:

The following example displays a message in the PowerFactory output window and clears it.

```
OutputWindow* outWindow = apiInstance->GetApplication()->GetOutputWindow();
outWindow->Print(OutputWindow::M_INFO,"Running␣PowerFactory␣from␣the␣API");
outWindow->Clear();
```

6.4 api::v1::DataObject

6.4.1 AttributeType

```
enum AttributeType {
  TYPE_INVALID = -1,
  TYPE_INTEGER = 0,
  TYPE_INTEGER_VEC = 1,
  TYPE_DOUBLE = 2,
  TYPE_DOUBLE_VEC = 3,
  TYPE_DOUBLE_MAT = 4,
  TYPE_OBJECT = 5,
  TYPE_OBJECT_VEC = 6,
  TYPE_STRING = 7,
  TYPE_STRING_VEC = 8,
  TYPE_INTEGER64 = 9,
  TYPE_INTEGER64_VEC = 10,
};
```

Enumerated type for defining the type of object attributes

TYPE˙INVALID	error or not existing attribute
TYPE˙INTEGER	integer
TYPE˙INTEGER˙VEC	vector of integers
TYPE˙DOUBLE	double
TYPE˙DOUBLE˙VEC	vector of double
TYPE˙DOUBLE˙MAT	matrix of doubles
TYPE˙OBJECT	DataObject
TYPE˙OBJECT˙VEC	vector of DataObject
TYPE˙STRING	string literal
TYPE˙STRING˙VEC	vector of string literals
TYPE˙INTEGER64	˙˙int64
TYPE˙INTEGER64˙VEC	vector of ˙˙int64

6.4.2 GetClassName

```
const Value* GetClassName()
```

Returns the class name of the considered DataObject (ex: ElmTerm, etc.)

Arguments:

None

Return value:

Pointer of type Value to the class name of the object, never NULL.

6.4.3 GetClassId

```
int GetClassId()
```

Returns the id of the class of the current object.

Arguments:

None

Return value:

Integer representing the index number of the class of the considered object

6.4.4 GetName

```
const Value* GetName()
```

Returns the name of the object, attribute loc_name, in PowerFactory.

Arguments:

None

Return value:

Pointer of type Value with the name of the object (string); never NULL

6.4.5 GetFullName

```
const Value* GetFullName(DataObject* relParent)
```

Returns the name, including the path, of the current object relative to a parent object

Arguments:

DataObject* relParent	starting point of the path

Return value:

Pointer of type value with the full path to the object (=string); never NULL

6.4.6 GetChildren

```
const Value* GetChildren(bool recursive)
```

Returns a collection of children objects for the current object. If the recursive flag is set to false, only the direct children of the object are returned else the function explores the full tree starting at the considered object.

Arguments:

bool recursive	false: only direct children are returned; true: the complete descendant tree is returned.

Return value:

The returned value is pointer of type Value pointing to a vector of DataObject; it is never NULL.

6.4.7 GetParent

```
DataObject* GetParent()
```

Returns the parent object of the current object.

Arguments:

None

Return value:

The parent object of the current object; NULL if the object has no parent.

6.4.8 IsNetworkDataFolder

```
bool IsNetworkDataFolder()
```

Checks whether the object is a network data folder (IntBmu, IntZone, etc.)

Arguments:

None

Return value:

Returns true if the object is a network data folder, false otherwise.

6.4.9 IsHidden

```
bool IsHidden()
```

Checks whether the object is active or not (depending on currently activated variation stage)

Arguments:

None

Return value:

Returns true if the object is inactive (stored in a in-active variation stage or deleted in the current stage)

6.4.10 IsDeleted

```
bool IsDeleted()
```

Checks whether the object is deleted (stored in the recycle bin).

Arguments:

None

Return value:

Returns true if the object is in the recycled bin; false otherwise.

6.4.11 IsAttributeReadOnly

```
bool IsAttributeReadOnly(const char* attribute) const
```

Checks whether the attribute with given name can be written via the api. Typical read-only attributes are calculation results.

Please note, write access can still fail if the object that should be modified is part of an inactive project or access rights are missing.

Arguments:

const char* attribute	attribute that should be checked

Return value:

boolean: whether or not object reusing is enabled

6.4.12 GetAttributeType

```
DataObject::AttributeType GetAttributeType(const char* attribute)
```

Returns the type of an attribute;

Arguments:

const char* attribute	the considered attribute for which the type must be returned

Return value:

Returns a value of type AttributeType with the type of the considered attribute.

6.4.13 GetAttributeDescription

```
const Value* GetAttributeDescription(const char* attribute)
```

Returns the description of an attribute of the current object; null if the required attribute does not exist.

Arguments:

const char* attribute	the considered attribute

Return value:

A pointer of type Value to the description of the attribute (string).

Example:

This example returns the description of the Project settings (pPrjSettings) attribute of the active project

```
const Value* activProj = apiInstance->GetApplication()->GetActiveProject();
const DataObject* prj = activProj->GetDataObject();
const Value* descr = prj->GetAttributeDescription("pPrjSettings");
printf("Attribute␣description␣(pPrjSettings):␣%s\n", descr->GetString());
```

6.4.14 GetAttributeUnit

```
const Value* GetAttributeUnit(const char* attribute)
```

Returns the unit of an attribute of the considered object (km, MW, etc.); NULL if the attribute does not exist; empty string for attributes without units.

Arguments:

const char* attribute	the attribute name whose unit is requested

Return value:

Pointer of type Value to the units of the given attribute of the considered object (=string)

Example:

The following example displays the units of the attribute Retention period

```
const Value* activProj = apiInstance->GetApplication()->GetActiveProject();

if(activProj)
{
  DataObject* projectProxy = activProj->GetDataObject();
  const Value* units = projectProxy->GetAttributeUnit("st_retention");
  printf("retention␣period␣units:␣%s\n",units->GetString());
}
```

6.4.15 GetAttributeSize

```
void GetAttributeSize(const char* attribute, int& countRows, int& countCols)
```

Returns the number of rows and columns for attributes of type matrix and vector.

Arguments:

const char* attribute	the considered attribute
int& countRows	number of rows (return value)
int& countCols	number of columns (return value)

Return value:

void

Example:

```
DataObject* myMatrix;
int* row;
int* col;
myMatrix->GetAttributeSize(  m a t r i x  , row, col);
```

6.4.16 GetAttributeInt

```
int GetAttributeInt(const char* attribute, int* error)
int GetAttributeInt(const char* attribute, int row, int col, int* error)
```

Returns the value of an attribute as an integer.

Arguments:

const char* attribute	the attribute name
int row	the row index of the element to be extracted if attribute is a matrix or a vector
int col	the row index of the element to be extracted if attribute is a matrix
int* error	returned error code

Return value:

Value of the considered attribute as an integer

Example: The following example displays the value and the units of the attribute Retention period

```
const Value* activProj = apiInstance->GetApplication()->GetActiveProject();

if(activProj)
{
  DataObject* projectProxy = activProj->GetDataObject();
  const Value* units = projectProxy->GetAttributeUnit("st_retention");
  int nbDays = activProj->GetDataObject()->GetAttributeInt("st_retention");
  printf( retention period: %d %s  , nbDays, units->GetString());
}
```

6.4.17 GetAttributeInt64

```
__int64 GetAttributeInt64(const char* attribute, int* error)
__int64 GetAttributeInt64(const char* attribute, int row, int col, int* error)
```

Returns the value of an attribute as an 64 Bit integer.

Arguments:

const char* attribute	the attribute name
int row	the row index of the element to be extracted if attribute is a matrix or a vector
int col	the row index of the element to be extracted if attribute is a matrix
int* error	returned error code

Return value:

Value of the considered attribute as an 64 Bit integer

6.4.18 GetAttributeDouble

```
double GetAttributeDouble(const char* attribute, int* error)
double GetAttributeDouble(const char* attribute,
                          int         row,
                          int         col,
                          int*        error)
```

Returns the value of an attribute as a double.

Arguments:

const char* attribute	the considered attribute
int row	the row index of the element to be extracted if attribute is a matrix or a vector
int col	the row index of the element to be extracted if attribute is a matrix
int* error	returned error code

Return value:

Value of the considered attribute as a double

6.4.19 GetAttributeObject

```
DataObject* GetAttributeObject(const char* attribute, int* error)
DataObject* GetAttributeObject(const char* attribute, int row, int* error)
```

Returns the value of an attribute as a pointer to a DataObject.

Arguments:

const char* attribute	the considered attribute
int row (optional)	the row index of the element to be extracted if attribute is a matrix or a vector
int* error (optional)	the returned error code

Return value:

Pointer to DataObject containing the corresponding attribute

6.4.20 GetAttributeString

```
const Value* GetAttributeString(const char* attribute, int* error)
const Value* GetAttributeString(const char* attribute, int row, int* error)
```

Returns the value of an string attribute.

Arguments:

const char* attribute	the considered attribute
int row (optional)	the row index of the element to be extracted if attribute is a matrix or a vector
int* error (optional)	returned error code

Return value:

Pointer of type Value to the attribute.

6.4.21 GetAttributeContainer

```
const Value* GetAttributeContainer(const char* attribute, int* error=0) const
```

Returns the value of a container attribute, i.e. a double vector or a collection of objects as a Value object holding a vector of values.

Arguments:

const char* attribute	the considered attribute
int* error (optional)	returned error code

Return value:

Pointer of type Value holding a vector of values.

6.4.22 SetAttributeObject

```
void SetAttributeObject(const char* attribute, DataObject* obj, int* error)
void SetAttributeObject(const char* attribute,
                        DataObject* obj,
                        int         row,
                        int*        error)
```

Sets the corresponding object attribute.

Arguments:

const char* attribute	the considered attribute
DataObject* obj	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector
int* error	returned error code

Return value:

void

6.4.23 SetAttributeString

```
void SetAttributeString(const char* attribute, const char* value, int* error)
void SetAttributeString(const char* attribute,
                        const char* value,
                        int         row,
                        int         col,
                        int*        error)
```

Sets the corresponding string attribute.

Arguments:

const char* attribute	the considered attribute
const char* value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.4.24 SetAttributeContainer

```
void SetAttributeContainer(const char* attribute, const Value* value, int* error=0)
```

Sets the value of a container attribute, i.e. a vector of doubles or a collection of objects. The Value object passed must be a VECTOR type.

Arguments:

const char* attribute	the considered attribute
const Value* value	the new value of the attribute
int* error	returned error code

Return value:

void

6.4.25 SetAttributeDouble

```
void SetAttributeDouble(const char* attribute, double value, int* error)
void SetAttributeDouble(const char* attribute,
                        double      value,
                        int         row,
                        int         col,
                        int*        error)
```

Sets the corresponding double attribute.

Arguments:

const char* attribute	the considered attribute
double value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.4.26 SetAttributeInt

```
void SetAttributeInt(const char* attribute, int value, int* error)
void SetAttributeInt(const char* attribute,
                     int         value,
                     int         row,
                     int         col,
                     int*        error)
```

Sets the corresponding double attribute.

Arguments:

const char* attribute	the considered attribute
int value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.4.27 SetAttributeInt64

```
void SetAttributeInt64(const char* attribute, __int64 value, int* error)
void SetAttributeInt64(const char* attribute,
__int64     value,
int          row,
int          col,
int*         error)
```

Sets the corresponding integer attribute.

Arguments:

const char* attribute	the considered attribute
¨int64 value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.4.28 GetAttributeNames

```
const Value* GetAttributeNames() const
```

Returns a Value object of type VECTOR holding the names of all attributes for this object.

Return value:

Pointer of type Value holding a vector of string values.

6.4.29 ResizeAttribute

```
void ResizeAttribute(const char* attribute,
                     int         rowSize,
                     int         colSize,
                     int*        error)
```

Resize an attribute of the current object if this attribute is a matrix or a vector

Arguments:

const char* attribute	the considered attribute
int rowSize	the new number of rows for the element
int colSize	the new number of columns for the element
int* error	returned error code

Return value:

void

6.4.30 CreateObject

```
DataObject* CreateObject(const char* className, const char* locname)
```

Create a new object of a given class inside the considered object (if this one can hold the new object) Returns a DataObject pointer to the newly created object if successful; NULL otherwise.

Arguments:

const char* className	the class name of the object to be created
const char* locname	the name of the new object

Return value:

Pointer to the newly created DataObject

6.4.31 DeleteObject

```
void DeleteObject(int* error=0)
```

Deletes an object from ***PowerFactory*** , e.g. a network element or even a project.

Arguments:

int* error	returned error code (0 on success)

Return value:

void

6.4.32 GetAttributes

```
const Value* GetAttributes(int* error=0) const
```

Returns the values of all attributes specified via Application::DefineTransferAttributes as Value object holding a VECTOR.

Arguments:

int* error	returned error code (0 on success)

Return value:

Pointer of type Value holding a vector of values.

6.4.33 SetAttributes

```
void SetAttributes(const Value* values, int* error=0)
```

Sets all attributes specified via Application::DefineTransferAttributes to the values passed as Value object holding a VECTOR.

Arguments:

const Value* values	the new values for the attributes
int* error	returned error code (0 on success)

Return value:

void

6.4.34 IsSame

```
bool IsSame(const DataObject* other) const
```

When Application::IsObjectReusingEnabled is false, the identity of two DataObject pointers can not be compared using the simple equality operator as two DataObjects might point to the same ***PowerFactory*** object. In these cases IsSame must be used for comparison.

Arguments:

const DataObject* other	other object for comparison

Return value:

boolean: whether or not the objects represent the same ***PowerFactory*** object

6.4.35 Execute

```
Value* Execute(const char* command, const Value* inArgs, int* error)
```

This function executes a command on the instance of the object.

The available commands are listed in the Python Function Reference. Commands with more then one argument can be executed with 'inArgs' of type vector.

Arguments:

const char* command	the command that should be executed
const Value* inArgs	input arguments for the command to be executed
int* error	returned error code (0 on success)

Return value:

The function returns a pointer of type Value to the result of the command if applicable.

Example:

```
Application* app(apiInstance->GetApplication());

// getting all versions of the active project
const Value* activeProject(app->Execute("GetActiveProject", NULL, error));
const Value* versions(app->Execute("GetVersions", NULL, error));

// releasing received values
apiInstance->ReleaseValue(activeProject)
apiInstance->ReleaseValue(versions)
```

6.4.36 WriteChangesToDb

```
void WriteChangesToDb()
```

This function combined with SetWriteCacheEnabled is meant to optimize performances. If the write cache flag is enabled all objects remain in edit mode until WriteChangesToDb is called and all the modifications made to the objects are saved into the database.

Arguments:

None

Return value:

void

6.5 api::v2::Api

6.5.1 GetVersion

```
Value* GetVersion()
```

Returns the version of the current API instance

Arguments:

none

Return value:

A pointer of type Value pointing to a string with the version of the current API.

Example:

The following example displays in the command window the version of the api used to create the running instance.

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
std::cout << std::endl
          << apiInstance->GetVersion()->GetString()
          << std::endl;
```

6.5.2 ReleaseObject

```
int ReleaseObject(const DataObject* object)
```

Releases a DataObject instance. All DataObject pointers returned by an api must be released using this function. Calling delete from an external DLL is not possible as the API instance has its own memory management. Each object must only be released once.

Arguments:

const DataObject*	the pointer to the object to be released

Return value:

0 on success, non-zero on error

Example:

```
DataObject* user(apiInstance->GetApplication()->GetCurrentUser());
apiInstance->ReleaseObject(user);
```

6.5.3 ReleaseValue

```
int ReleaseValue(const Value* object)
```

Releases an API value instance. All API value pointers returned by the api must be released using this function. Calling delete from an external DLL is not possible as the API instance has its own memory management.

Arguments:

const Value*	pointer to the object to be released

Return value:

0 on success, non-zero on error

Example:

```
DataObject* user(apiInstance->GetApplication()->GetCurrentUser());
Value* name = user->GetName();
apiInstance->ReleaseValue(name);
apiInstance->ReleaseObject(user);
```

6.5.4 GetApplication

```
Application* GetApplication()
```

The function returns an instance of Application. There exists only one Application object per Api instance. This object must not be deleted.

Arguments:

None

Return value:

Pointer to the instance of an Application object, never NULL

Example:

The following example displays in the command window the version of the running instance of ***PowerFactory*** .

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
std::cout << "PowerFactory␣version:"
          << apiInstance->GetApplication()->GetVersion()->GetString()
          << std::endl;
```

6.5.5 IsDebug

```
bool IsDebug()
```

Arguments:

None

Return value:

True if ***PowerFactory*** is in debug mode; false otherwise.

Example:

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
if(apiInstance->IsDebug())
{
  std::cout << "PowerFactory␣running␣in␣debug␣mode" << std::endl;
}
```

6.6 api::v2::Application

6.6.1 GetVersion

```
const Value* GetVersion()
```

The function returns the version of the currently running ***PowerFactory*** , e.g. ”14.0.505”

Arguments:

None

Return value:

Pointer to a Value object holding version information of ***PowerFactory*** application; returned string is never null and must be released when no longer in use.

Example:

The following example displays in the command window the version of the running instance of ***PowerFactory*** .

```
int errorCode = 0;
apiInstance = CreateApiInstanceEx(NULL, NULL, NULL, errorCode);
std::cout << "PowerFactory␣version:"
          << apiInstance->GetApplication()->GetVersion()->GetString()
          << std::endl;
```

6.6.2 GetTempDirectory

```
const Value* GetTempDirectory()
```

Returns the path to the temporary directory of the running instance of ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value pointing to a string with the temporary directory of ***PowerFactory*** .

Example:

```
std::cout << apiInstance->GetApplication()->GetTempDirectory()->GetString()
          << std::endl;
```

6.6.3 GetWorkingDirectory

```
const Value* GetWorkingDirectory()
```

Returns the path to the working directory of the running instance of ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value holding a string string with the working directory of ***PowerFactory*** .

Example:

```
Application* application(apiInstance->GetApplication());
std::cout << application->GetWorkingDirectory()->GetString()
          << std::endl;
```

6.6.4 GetInstallationDirectory

```
const Value* GetInstallationDirectory()
```

Returns the path to the installation directory of the running instance of ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value holding a string with the installation directory of ***PowerFactory*** .

6.6.5 GetLanguageCode

```
const Value* GetLanguageCode()
```

Returns the language code of the currently used display language in ***PowerFactory*** .

Arguments:

None

Return value:

A pointer of type Value holding a string with the language code.

6.6.6 GetOutputWindow

```
OutputWindow* GetOutputWindow()
```

This function returns an instance of the running ***PowerFactory*** output window. Each api instance has only one output window instance.

Arguments:

None

Return value:

Returns a pointer to an instance of OutputWindow, never NULL

6.6.7 GetCurrentUser

```
DataObject* GetCurrentUser()
```

This function returns the current user.

Arguments:

None

Return value:

Returns a pointer to the currently logged in user object.

Example:

The following example displays the name of the current user in the command window.

```
int error = 0;
DataObject* user = apiInstance->GetApplication()->GetCurrentUser();
std::cout << *(user->GetAttributeString("loc_name",error)) << std::endl;
```

6.6.8 GetActiveProject

```
DataObject* GetActiveProject()
```

Returns a pointer to the currently active ***PowerFactory*** project, NULL if there is no active project.

Arguments:

None

Return value:

Pointer to the currently active ***PowerFactory*** project

6.6.9 GetActiveStudyCase

```
DataObject* GetActiveStudyCase()
```

Returns a pointer to the currently active study case, NULL if there is no active case.

Arguments:

None

Return value:

Pointer of type Value to the currently active study case

6.6.10 GetCalcRelevantObjects

```
const Value* GetCalcRelevantObjects()
```

This function returns all objects that are currently relevant for calculation: lines, nodes, switches, transformers, etc. and their types.

Arguments:

None

Return value:

Returns a pointer of type Value to a vector of objects relevant for calculation, never NULL. The container must be released if no longer in use.

6.6.11 GetClassId

```
int GetClassId(const char* className)
```

This function returns the class identifier integer of the considered class className. Each class name corresponds to one unique index. The mapping of class name might be different according to the build version of ***PowerFactory*** , but it is guaranteed that it will not change while an Api instance exists. This indices should not be stored statically but rather be generated dynamically in the code using the GetClassId method.

Arguments:

const char* className	the name of the considered class

Return value:

Returns an integer representing the index (¿0) of the considered class; 0 if className is not a valid class name.

Example:

```
int filterID = apiInstance->GetApplication()->GetClassId("intPrj");
std::cout << "Project␣ID␣(intPrj):" << filterID << std::endl;
```

6.6.12 GetClassDescription

```
const Value* GetClassDescription(const char* className)
```

Returns the description of a ***PowerFactory*** class.

Arguments:

const char* className	name of the considered ***PowerFactory*** class

Return value:

Returns the description text, never NULL; but the string is empty for invalid class names

Example:

The following example displays the description text of the class intPrj

```
Application* application(apiInstance->GetApplication());
std::cout << "intPrj␣description␣text:"
          << application->GetClassDescription("intPrj")->GetString()
          << std::endl;
```

6.6.13 AttributeMode

```
enum AttributeMode {
  MODE_DISPLAYED    = 0,
  MODE_INTERNAL     = 1
};
```

Enumerated type for accessing object attributes in ***PowerFactory*** .

MODE_DISPLAYED	as displayed in PF (unit conversion applied)
MODE_INTERNAL	as internally stored

6.6.14 SetAttributeMode

```
void SetAttributeMode(AttributeMode mode)
```

Changes the way how attribute values are accessed

Arguments:

AttributeMode mode	the way the attribute values should be accessed

Return value:

void

6.6.15 GetAttributeMode

```
AttributeMode GetAttributeMode()
```

Returns the mode how object attributes are accessed.

Arguments:

None

Return value:

Current mode as AttributeMode

6.6.16 SetWriteCacheEnabled

```
void SetWriteCacheEnabled(bool enabled)
```

This function intends to optimize performances. In order to modify objects in ***PowerFactory*** , those must be set in a special edit mode before any value can be changed. Switching back and forth between edit mode and stored mode is time consuming; enabling the write cache flag will set objects in edit mode and they will not be switched back until WriteChangeToDb is called.

Default value: disabled.

Arguments:

bool enabled	true = enabled; false = disabled

Return value:

void

Example:

```
apiInstance->GetApplication()->SetWriteCacheEnabled(true);
```

6.6.17 IsWriteCacheEnabled

```
bool IsWriteCacheEnabled()
```

Returns whether or not the cache method for optimizing performances is enabled.

Arguments:

None

Return value:

boolean: whether or not the cache method for optimizing performances is enabled

Example:

```
bool cacheEnabled;
cacheEnabled = apiInstance->GetApplication()->IsWriteCacheEnabled();
```

6.6.18 SetObjectReusingEnabled

```
void SetObjectReusingEnabled(bool enabled)
```

This option allows to change the behavior of creation vs. re-using api::DataObject instances for identical ***PowerFactory*** objects. When enabled access to the same ***PowerFactory*** object will result in usage of same api::DataObject instance until the instance has explicitly been released via ReleaseObject() call.

This affects the behavior of all api functions returning api::DataObject pointers.

Default value: enabled.

Arguments:

bool enabled	true = enabled; false = disabled

Return value:

void

6.6.19 IsObjectReusingEnabled

```
bool IsObjectReusingEnabled()
```

Returns whether or not object reusing is currently enabled.

Arguments:

None

Return value:

boolean: whether or not object reusing is enabled

6.6.20 GetAttributeType

```
DataObject::AttributeType GetAttributeType(const char* classname,
                                           const char* attribute)
```

This function returns the actual type of an object attribute.

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which type must be returned

Return value:

Returns the type of the attribute or TYPE_INVALID on error (no attribute of that name exists)

TYPE_INVALID	error
TYPE_INTEGER	int
TYPE_INTEGER_VEC	vector of type int
TYPE_DOUBLE	double
TYPE_DOUBLE_VEC	vector of double
TYPE_DOUBLE_MAT	2D matrix of double
TYPE_OBJECT	DataObject
TYPE_OBJECT_VEC	vector of DataObject
TYPE_STRING	string literal
TYPE_STRING_VEC	vector of string literals
TYPE_INTEGER64	__int64
TYPE_INTEGER64_VEC	vector of __int64

6.6.21 GetAttributeDescription

```
const Value* GetAttributeDescription(const char* classname,
                                     const char* attribute)
```

Returns the description of an attribute, NULL if the attribute does not exist

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which description must be returned

Return value:

Pointer of type Value to the current attribute description (=string)

6.6.22 GetAttributeUnit

```
const Value* GetAttributeUnit(const char* classname, const char* attribute)
```

Returns the unit of an attribute, e.g. km, MW...; NULL if the given attribute name does not exist; the string is empty for attributes without units

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which description must be returned

Return value:

Pointer of type Value to the units of the considered attribute (=string)

6.6.23 GetAttributeSize

```
void GetAttributeSize(const char* name,
                      const char* attribute,
                      int& countRows,
                      int& countCols)
```

This function returns the size of object attribute if this attribute is a vector or a matrix.

Arguments:

const char* classname	class name for which the attribute type is considered
const char* attribute	attribute which description must be returned
int& countRows	the returned number of rows
int& countCols	the returned number of columns

Return value:

void

6.6.24 IsAttributeReadOnly

```
bool IsAttributeReadOnly(const char* classname, const char* attribute) const
```

Checks whether the attribute with given name can be written via the api. Typical read-only attributes are calculation results.

Please note, write access can still fail if the object that should be modified is part of an inactive project or access rights are missing.

Arguments:

const char* classname	class name for which the attribute is considered
const char* attribute	attribute that should be checked

Return value:

boolean: whether or not object reusing is enabled

6.6.25 DefineTransferAttributes

```
virtual void DefineTransferAttributes(const char* classname, const char* attributes, int* erro
```

Allows to specify for a given classname a comma separated list of attributes which should be written in one go. The values can then be passed using DataObject::SetAttributes.

Arguments:

const char* classname	class name for which the attribute is considered
const char* attributes	attribute that should be checked

Return value:

void

6.6.26 Execute

```
Value* Execute(const char* command, const Value* inArgs, int* error)
```

This function executes a command on the instance of the application.

The available commands are listed in the Python Function Reference. Commands with more then one argument can be executed with 'inArgs' of type vector.

Arguments:

const char* command	the command that should be executed
const Value* inArgs	input arguments for the command to be executed
int* error	returned error code (0 on success)

Return value:

The function returns a pointer of type Value to the result of the command if applicable.

Example:

```
Application* app(apiInstance->GetApplication());

// getting the active project
const Value* activeProject(app->Execute("GetActiveProject", NULL, error));

// collecting all lines
const Value* lines(app->Execute("GetCalcRelevantObjects", &Value("*.ElmLne"), error));

// collecting lines which are not out of service
Value arguments(Value::VECTOR);
arguments.VecInsertString("*.ElmLne");
arguments.VecInsertInteger(0);
const Value* linesInService(app->Execute("GetCalcRelevantObjects", &arguments, error));

// releasing received values
apiInstance->ReleaseValue(activeProject)
apiInstance->ReleaseValue(lines)
apiInstance->ReleaseValue(linesInService)
```

6.6.27 WriteChangesToDb

```
void WriteChangesToDb()
```

This function combined with SetWriteCacheEnabled is meant to optimize performances. If the write cache flag is enabled all objects remain in edit mode until WriteChangesToDb is called and all the modifications made to the objects are saved into the database.

Arguments:

None

Return value:

void

6.7 api::v2::OutputWindow

6.7.1 MessageType

```
enum MessageType
{
  M_PLAIN = 0,
  M_ERROR = 1,
  M_WARN  = 2,
  M_INFO  = 4
};
```

Enumerated type which defines the message type.

M_PLAIN	message not prepended by any text
M_ERROR	message prepended by error prefix, will also popup as error dialog
M_WARN	message prepended by warning prefix
M_INFO	message prepended by info prefix

6.7.2 Print

```
void Print(MessageType type, const char* msg)
```

This function prints a message to the PowerFactory output window.

Arguments:

MessageType type	type of message
const char* msg	the actual message to be displayed

Return value:

void

Example:

The following example displays a message in the PowerFactory output window.

```
OutputWindow* outWindow = apiInstance->GetApplication()->GetOutputWindow();
outWindow->Print(OutputWindow::M_INFO,"Running␣PowerFactory␣from␣the␣API");
```

6.7.3 Clear

```
void Clear()
```

Empties the content of the output window.

Arguments:

None

Return value:

void

Example:

The following example displays a message in the PowerFactory output window and clears it.

```
OutputWindow* outWindow = apiInstance->GetApplication()->GetOutputWindow();
outWindow->Print(OutputWindow::M_INFO,"Running␣PowerFactory␣from␣the␣API");
outWindow->Clear();
```

6.7.4 GetContent

```
Value* GetContent()
Value* GetContent(MessageType filter)
```

Returns the textual content of the ***PowerFactory*** output window. An optional MessageType filter allows to obtain messages of a specific type only.

Arguments:

MessageType filter	optional, type of message to obtain

Return value:

The function returns an Value pointer of type Value::VECTOR containing the text from the output window.

Example:

The following example demonstrates how text from output window can be obtained and printed to std::out line by line.

```
OutputWindow* outWindow = apiInstance->GetApplication()->GetOutputWindow();

const Value* val = apiInstance->GetApplication()->GetOutputWindow()->GetContent();
for(unsigned int i=0; i<val->VecGetSize(); i++) {
  const Value* line = val->VecGetValue(i);
  std::cout << line->GetString() << std::endl;
}

OutputWindow* outWindow = apiInstance->GetApplication()->GetOutputWindow();
outWindow->Print(OutputWindow::M_INFO,"Running␣PowerFactory␣from␣the␣API");
```

6.7.5 Save

```
void Save(const char* file)
```

Saves the content of the ***PowerFactory*** output window to a file.

Arguments:

const char* file	name and path of the output file. The format will be selected depending on the given file extension. .txt plain text file .htm html formatted file .csv comma-separated value file .out ***PowerFactory*** specific out format

Return value:

void

Example:

The following example saves the content of output window to a text file

```
OutputWindow* outWindow = apiInstance->GetApplication()->GetOutputWindow();
outWindow->Save("D:\\Output.txt");
```

6.8 api::v2::DataObject

6.8.1 AttributeType

```
enum AttributeType {
  TYPE_INVALID = -1,
  TYPE_INTEGER = 0,
  TYPE_INTEGER_VEC = 1,
  TYPE_DOUBLE = 2,
  TYPE_DOUBLE_VEC = 3,
  TYPE_DOUBLE_MAT = 4,
  TYPE_OBJECT = 5,
  TYPE_OBJECT_VEC = 6,
  TYPE_STRING = 7,
  TYPE_STRING_VEC = 8,
  TYPE_INTEGER64 = 9,
  TYPE_INTEGER64_VEC = 10,
};
```

Enumerated type for defining the type of object attributes

TYPE_INVALID	error or not existing attribute
TYPE_INTEGER	integer
TYPE_INTEGER_VEC	vector of integers
TYPE_DOUBLE	double
TYPE_DOUBLE_VEC	vector of double
TYPE_DOUBLE_MAT	matrix of doubles
TYPE_OBJECT	DataObject
TYPE_OBJECT_VEC	vector of DataObject
TYPE_STRING	string literal
TYPE_STRING_VEC	vector of string literals
TYPE_INTEGER64	__int64
TYPE_INTEGER64_VEC	vector of __int64

6.8.2 GetClassName

```
const Value* GetClassName()
```

Returns the class name of the considered DataObject (ex: ElmTerm, etc.)

Arguments:

None

Return value:

Pointer of type Value to the class name of the object, never NULL.

6.8.3 GetClassId

```
int GetClassId()
```

Returns the id of the class of the current object.

Arguments:

None

Return value:

Integer representing the index number of the class of the considered object

6.8.4 GetName

```
const Value* GetName()
```

Returns the name of the object, attribute loc_name, in PowerFactory.

Arguments:

None

Return value:

Pointer of type Value with the name of the object (string); never NULL

6.8.5 GetFullName

```
const Value* GetFullName(DataObject* relParent)
```

Returns the name, including the path, of the current object relative to a parent object

Arguments:

DataObject* relParent	starting point of the path

Return value:

Pointer of type value with the full path to the object (=string); never NULL

6.8.6 GetChildren

```
const Value* GetChildren(bool recursive)
```

Returns a collection of children objects for the current object. If the recursive flag is set to false, only the direct children of the object are returned else the function explores the full tree starting at the considered object.

Arguments:

bool recursive	false: only direct children are returned; true: the complete descendant tree is returned.

Return value:

The returned value is pointer of type Value pointing to a vector of DataObject; it is never NULL.

6.8.7 GetParent

```
DataObject* GetParent()
```

Returns the parent object of the current object.

Arguments:

None

Return value:

The parent object of the current object; NULL if the object has no parent.

6.8.8 IsNetworkDataFolder

```
bool IsNetworkDataFolder()
```

Checks whether the object is a network data folder (IntBmu, IntZone, etc.)

Arguments:

None

Return value:

Returns true if the object is a network data folder, false otherwise.

6.8.9 IsHidden

```
bool IsHidden()
```

Checks whether the object is active or not (depending on currently activated variation stage)

Arguments:

None

Return value:

Returns true if the object is inactive (stored in a in-active variation stage or deleted in the current stage)

6.8.10 IsDeleted

```
bool IsDeleted()
```

Checks whether the object is deleted (stored in the recycle bin).

Arguments:

None

Return value:

Returns true if the object is in the recycled bin; false otherwise.

6.8.11 GetAttributeType

```
DataObject::AttributeType GetAttributeType(const char* attribute)
```

Returns the type of an attribute;

Arguments:

const char* attribute	the considered attribute for which the type must be returned

Return value:

Returns a value of type AttributeType with the type of the considered attribute.

6.8.12 GetAttributeDescription

```
const Value* GetAttributeDescription(const char* attribute)
```

Returns the description of an attribute of the current object; null if the required attribute does not exist.

Arguments:

const char* attribute	the considered attribute

Return value:

A pointer of type Value to the description of the attribute (string).

Example:

This example returns the description of the Project settings (pPrjSettings) attribute of the active project

```
const Value* activProj = apiInstance->GetApplication()->GetActiveProject();
const DataObject* prj = activProj->GetDataObject();
const Value* descr = prj->GetAttributeDescription("pPrjSettings");
printf("Attribute␣description␣(pPrjSettings):␣%s\n", descr->GetString());
```

6.8.13 GetAttributeUnit

```
const Value* GetAttributeUnit(const char* attribute)
```

Returns the unit of an attribute of the considered object (km, MW, etc.); NULL if the attribute does not exist; empty string for attributes without units.

Arguments:

const char* attribute	the attribute name whose unit is requested

Return value:

Pointer of type Value to the units of the given attribute of the considered object (=string)

Example:

The following example displays the units of the attribute Retention period

```
const Value* activProj = apiInstance->GetApplication()->GetActiveProject();

if(activProj)
{
  DataObject* projectProxy = activProj->GetDataObject();
  const Value* units = projectProxy->GetAttributeUnit("st_retention");
  printf("retention␣period␣units:␣%s\n",units->GetString());
}
```

6.8.14 GetAttributeSize

```
void GetAttributeSize(const char* attribute, int& countRows, int& countCols)
```

Returns the number of rows and columns for attributes of type matrix and vector.

Arguments:

const char* attribute	the considered attribute
int& countRows	number of rows (return value)
int& countCols	number of columns (return value)

Return value:

void

Example:

```
DataObject* myMatrix;
int* row;
int* col;
myMatrix->GetAttributeSize( m a t r i x  , row, col);
```

6.8.15 GetAttributeInt

```
int GetAttributeInt(const char* attribute, int* error)
int GetAttributeInt(const char* attribute, int row, int col, int* error)
```

Returns the value of an attribute as an integer.

Arguments:

const char* attribute	the attribute name
int row	the row index of the element to be extracted if attribute is a matrix or a vector
int col	the row index of the element to be extracted if attribute is a matrix
int* error	returned error code

Return value:

Value of the considered attribute as an integer

Example: The following example displays the value and the units of the attribute Retention period

```
const Value* activProj = apiInstance->GetApplication()->GetActiveProject();

if(activProj)
{
  DataObject* projectProxy = activProj->GetDataObject();
  const Value* units = projectProxy->GetAttributeUnit("st_retention");
  int nbDays = activProj->GetDataObject()->GetAttributeInt("st_retention");
  printf( retention period: %d % s  , nbDays, units->GetString());
}
```

6.8.16 GetAttributeInt64

```
__int64 GetAttributeInt64(const char* attribute, int* error)
__int64 GetAttributeInt64(const char* attribute, int row, int col, int* error)
```

Returns the value of an attribute as an 64 Bit integer.

Arguments:

const char* attribute	the attribute name
int row	the row index of the element to be extracted if attribute is a matrix or a vector
int col	the row index of the element to be extracted if attribute is a matrix
int* error	returned error code

Return value:

Value of the considered attribute as an 64 Bit integer

6.8.17 GetAttributeDouble

```
double GetAttributeDouble(const char* attribute, int* error)
double GetAttributeDouble(const char* attribute,
                          int         row,
                          int         col,
                          int*        error)
```

Returns the value of an attribute as a double.

Arguments:

const char* attribute	the considered attribute
int row	the row index of the element to be extracted if attribute is a matrix or a vector
int col	the row index of the element to be extracted if attribute is a matrix
int* error	returned error code

Return value:

Value of the considered attribute as a double

6.8.18 GetAttributeObject

```
DataObject* GetAttributeObject(const char* attribute, int* error)
DataObject* GetAttributeObject(const char* attribute, int row, int* error)
```

Returns the value of an attribute as a pointer to a DataObject.

Arguments:

const char* attribute	the considered attribute
int row (optional)	the row index of the element to be extracted if attribute is a matrix or a vector
int* error (optional)	the returned error code

Return value:

Pointer to DataObject containing the corresponding attribute

6.8.19 GetAttributeString

```
const Value* GetAttributeString(const char* attribute, int* error)
const Value* GetAttributeString(const char* attribute, int row, int* error)
```

Returns the value of an string attribute.

Arguments:

const char* attribute	the considered attribute
int row (optional)	the row index of the element to be extracted if attribute is a matrix or a vector
int* error (optional)	returned error code

Return value:

Pointer of type Value to the attribute.

6.8.20 GetAttributeContainer

```
const Value* GetAttributeContainer(const char* attribute, int* error=0) const
```

Returns the value of a container attribute, i.e. a double vector or a collection of objects as a Value object holding a vector of values.

Arguments:

const char* attribute	the considered attribute
int* error (optional)	returned error code

Return value:

Pointer of type Value holding a vector of values.

6.8.21 SetAttributeObject

```
void SetAttributeObject(const char* attribute, DataObject* obj, int* error)
void SetAttributeObject(const char* attribute,
                        DataObject* obj,
                        int         row,
                        int*        error)
```

Sets the corresponding object attribute.

Arguments:

const char* attribute	the considered attribute
DataObject* obj	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector
int* error	returned error code

Return value:

void

6.8.22 SetAttributeString

```
void SetAttributeString(const char* attribute, const char* value, int* error)
void SetAttributeString(const char* attribute,
                        const char* value,
                        int         row,
                        int         col,
                        int*        error)
```

Sets the corresponding string attribute.

Arguments:

const char* attribute	the considered attribute
const char* value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.8.23 SetAttributeContainer

```
void SetAttributeContainer(const char* attribute, const Value* value, int* error=0)
```

Sets the value of a container attribute, i.e. a vector of doubles or a collection of objects. The Value object passed must be a VECTOR type.

Arguments:

const char* attribute	the considered attribute
const Value* value	the new value of the attribute
int* error	returned error code

Return value:

void

6.8.24 SetAttributeDouble

```
void SetAttributeDouble(const char* attribute, double value, int* error)
void SetAttributeDouble(const char* attribute,
                        double      value,
                        int         row,
                        int         col,
                        int*        error)
```

Sets the corresponding double attribute.

Arguments:

const char* attribute	the considered attribute
double value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.8.25 SetAttributeInt

```
void SetAttributeInt(const char* attribute, int value, int* error)
void SetAttributeInt(const char* attribute,
                         int          value,
                         int          row,
                         int          col,
                         int*         error)
```

Sets the corresponding double attribute.

Arguments:

const char* attribute	the considered attribute
int value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.8.26 SetAttributeInt64

```
void SetAttributeInt64(const char* attribute, __int64 value, int* error)
void SetAttributeInt64(const char* attribute,
__int64     value,
int          row,
int          col,
int*         error)
```

Sets the corresponding integer attribute.

Arguments:

const char* attribute	the considered attribute
¨int64 value	the new value of the attribute
int row	the row index of the element to be extracted if attribute is a vector or a matrix
int col	the column index of the element to be extracted if attribute is a vector or a matrix
int* error	returned error code

Return value:

void

6.8.27 GetAttributeNames

```
const Value* GetAttributeNames() const
```

Returns a Value object of type VECTOR holding the names of all attributes for this object.

Return value:

Pointer of type Value holding a vector of string values.

6.8.28 ResizeAttribute

```
void ResizeAttribute(const char* attribute,
                     int         rowSize,
                     int         colSize,
                     int*        error)
```

Resize an attribute of the current object if this attribute is a matrix or a vector

Arguments:

const char* attribute	the considered attribute
int rowSize	the new number of rows for the element
int colSize	the new number of columns for the element
int* error	returned error code

Return value:

void

6.8.29 CreateObject

```
DataObject* CreateObject(const char* className, const char* locname)
```

Create a new object of a given class inside the considered object (if this one can hold the new object) Returns a DataObject pointer to the newly created object if successful; NULL otherwise.

Arguments:

const char* className	the class name of the object to be created
const char* locname	the name of the new object

Return value:

Pointer to the newly created DataObject

6.8.30 DeleteObject

```
void DeleteObject(int* error=0)
```

Deletes an object from ***PowerFactory*** , e.g. a network element or even a project.

Arguments:

int* error	returned error code (0 on success)

Return value:

void

6.8.31 GetAttributes

```
const Value* GetAttributes(int* error=0) const
```

Returns the values of all attributes specified via Application::DefineTransferAttributes as Value object holding a VECTOR.

Arguments:

int* error	returned error code (0 on success)

Return value:

Pointer of type Value holding a vector of values.

6.8.32 SetAttributes

```
void SetAttributes(const Value* values, int* error=0)
```

Sets all attributes specified via Application::DefineTransferAttributes to the values passed as Value object holding a VECTOR.

Arguments:

const Value* values	the new values for the attributes
int* error	returned error code (0 on success)

Return value:

void

6.8.33 IsSame

```
bool IsSame(const DataObject* other) const
```

When Application::IsObjectReusingEnabled is false, the identity of two DataObject pointers can not be compared using the simple equality operator as two DataObjects might point to the same ***PowerFactory*** object. In these cases IsSame must be used for comparison.

Arguments:

const DataObject* other	other object for comparison

Return value:

boolean: whether or not the objects represent the same ***PowerFactory*** object

6.8.34 Execute

```
Value* Execute(const char* command, const Value* inArgs, int* error)
```

This function executes a command on the instance of the object.

The available commands are listed in the Python Function Reference. Commands with more then one argument can be executed with 'inArgs' of type vector.

Arguments:

const char* command	the command that should be executed
const Value* inArgs	input arguments for the command to be executed
int* error	returned error code (0 on success)

Return value:

The function returns a pointer of type Value to the result of the command if applicable.

Example:

```
Application* app(apiInstance->GetApplication());
```

```
// getting all versions of the active project
const Value* activeProject(app->Execute("GetActiveProject", NULL, error));
const Value* versions(app->Execute("GetVersions", NULL, error));

// releasing received values
apiInstance->ReleaseValue(activeProject)
apiInstance->ReleaseValue(versions)
```

6.8.35 WriteChangesToDb

```
void WriteChangesToDb()
```

This function combined with SetWriteCacheEnabled is meant to optimize performances. If the write cache flag is enabled all objects remain in edit mode until WriteChangesToDb is called and all the modifications made to the objects are saved into the database.

Arguments:

None

Return value:

void

6.9 api::v2::ExitError

6.9.1 GetCode

```
int GetCode()
```

Returns the code of the exit error.

Return value:

Integer representing the exit error. A detailed description of all exit errors can be found in the Error Code reference.

6.10 api::Value

The Value type is a variant type which can hold values of various types. It is used to transport data between the API and the 3rd party application.

6.10.1 Type

```
enum Type {
  UNKNOWN,
  INTEGER,
  DOUBLE,
  STRING,
  INTEGER64,
  DATAOBJECT,
  VECTOR,
  DOUBLEMATRIX,
  VALUE
};
```

Enumerated type to define the type of a Value object.

6.10.2 Constructor / Destructor

```
Value(const int val)
Value(const __int64 val)
Value(const double val)
Value(const char* val)
Value(DataObject* val)
Value(Type type)
~Value()
```

Constructs Value objects of passed type.

6.10.3 GetType

```
const Type GetType()
```

Returns the actual type of the Value object.

Arguments:

None

Return value Type of the value object encoded as Type.

6.10.4 GetInteger

```
int GetInteger(int* error)
```

Returns an integer if the Value object is of integer type, otherwise returns 0.

Arguments:

int* error	returned error code

Return value:

Returns an integer if the Value object is of integer type, otherwise returns 0.

6.10.5 GetInteger64

```
__int64 GetInteger64(int* error)
```

Returns a long integer if the Value object is of long integer type, otherwise returns 0.

Arguments:

int* error	returned error code

Return value:

Returns a long integer if the Value object is of long integer type, otherwise returns 0.

6.10.6 GetDouble

```
double GetDouble(int* error)
```

Returns a double if the Value object is of type double, otherwise returns 0.

Arguments:

int* error	returned error code

Return value:

Returns a double if the Value object is of type double, otherwise returns 0.

6.10.7 GetString

```
const char* GetString(int* error)
```

Returns a string if the Value object is of type string, otherwise returns NULL.

Arguments:

int* error	returned error code

Return value:

Returns a string if the Value object is of type string, otherwise returns NULL.

6.10.8 GetDataObject

```
DataObject* GetDataObject(int* error)
```

Returns a pointer to a DataObject if the Value object is of type DataObject, NULL otherwise. DataObject must be released when no longer in use.

Arguments:

```
int* error (optional): returned error code
```

Return value:

Returns a pointer to a DataObject if the Value object is of type DataObject, NULL otherwise.

6.10.9 MatGetRowCount

```
unsigned int MatGetRowCount(int* error=0)
```

Returns the number of rows/elements if the Value is matrix/vector

Arguments:

int* error	returned error code

Return value:

Number of rows as unsigned int

6.10.10 MatGetColCount

```
unsigned int MatGetColCount(int* error=0)
```

returns the number of columns if the Value is a matrix

Arguments:

int* error	returned error code

Return value:

Number of columns as unsigned integer.

6.10.11 MatSetDouble

```
void MatSetDouble(const unsigned int row,
                  const unsigned int col,
                  const double       val,
                  int*               error=0)
```

Set the value of a double in a matrix at position (row, col)

Arguments:

const unsigned int row	row index
const unsigned int col	column index
const double val	double value to be inserted in the matrix
int* error	returned error code

Return value:

void

6.10.12 MatGetDouble

```
double MatGetDouble(const unsigned int row,
                    const unsigned int col,
                    int*               error=0)
```

Returns the value of the double in the considered matrix at position (row, col)

Arguments:

const unsigned int row	row index of the considered matrix element
const unsigned int col	column index of the considered matrix element
int* error	returned error code

Return value:

The value of the double at position (row, col)

6.10.13 SetValue

```
void SetValue(const int val)
void SetValue(const __int64 val)
void SetValue(const double val)
void SetValue(const char* val)
void SetValue(DataObject* val)
```

Set the value of a Value type object; value can be of types int, ¨int64, double, char* and DataObject*.

Arguments:

According to the type of the value to be set:

const int val	if the value to be set is an integer
const __int64 val	if the value to be set is a long integer
const double val	if the value to be set is a double
const char* val	if the value to be set is a string
const DataObject* val	if the value to be set is a DataObject

Return value:

void

6.10.14 VecGetInteger

```
int VecGetInteger(const unsigned int index, int* error)
```

Returns the integer stored in a vector at position index

Arguments:

const unsigned int index	index of the element to be read
int* error	returned error code

Return value:

integer stored in a vector at position index

6.10.15 VecGetInteger64

```
__int64 VecGetInteger64(const unsigned int index, int* error)
```

Returns a long integer stored in a vector at position index.

Arguments:

const unsigned int index	index of the element to be read
int* error	returned error code

Return value:

Returns a __int64 stored in a vector at position index.

6.10.16 VecGetDouble

```
double VecGetDouble(const unsigned int index, int* error)
```

Returns a double stored in a vector a position index.

Arguments:

const unsigned int index	index of the element to be read
int* error	returned error code

Return value:

Returns a double stored in a vector a position index

6.10.17 VecGetString

```
const char* VecGetString(const unsigned int index, int* error)
```

Returns a string of characters stored in a vector at position index.

Arguments:

const unsigned int index	index of the element to be read
int* error	returned error code

Return value:

Returns a string of characters stored in a vector at position index.

6.10.18 VecGetDataObject

```
DataObject* VecGetDataObject(const unsigned int index, int* error)
```

Returns an object stored in a vector at position index.

Arguments:

const unsigned int index	index of the element to be read
int* error (optional)	returned error code

Return value:

Returns an object stored in a vector at position index, NULL if the position does not hold a DataObject.

6.10.19 VecGetValue

```
const Value* VecGetValue(const unsigned int idx, int* error=0)
```

Returns a pointer to a Value stored in a vector at position idx

Arguments:

const unsigned int index	index of the element to be read
int* error	returned error code

Return value:

Returns a pointer to a Value if the considered element is a Value, NULL otherwise.

6.10.20 VecClear

```
void VecClear(int* error=0)
```

Empties the considered vector

Arguments:

int* error	returned error code

Return value:

void

6.10.21 VecGetSize

```
unsigned int VecGetSize(int* error)
```

Returns the size of the considered vector.

Arguments:

int* error	returned error code

Return value:

the size of the considered vector

6.10.22 VecGetType

```
Type VecGetType(const unsigned int index, int* error)
```

Returns the type (integer, string, etc.) of an element stored in a vector at position index.

Arguments:

const unsigned int index	index of the element to be read
int* error (optional)	returned error code

Return value:

The stored type encoded as Type

6.10.23 VecInsertInteger

```
void VecInsertInteger(const int val, int* error=0)
```

Pushes the new value to the end of the vector.

Arguments:

const int val	integer value to be inserted in the vector
int* error (optional)	returned error code

Return value:

void

6.10.24 VecInsertInt64

```
void VecInsertInt64(const __int64 val, int* error=0)
```

Pushes the new value to the end of the vector.

Arguments:

const ¨int64 val	integer value to be inserted in the vector
int* error	returned error code

Return value:

void

6.10.25 VecInsertDouble

```
void VecInsertDouble(const double val, int* error=0)
```

Pushes the new value to the end of the vector.

Arguments:

const double val: double value to be inserted in the vector int* error (optional): returned error code

Return value:

void

6.10.26 VecInsertString

```
void VecInsertString(const char* val, int* error=0)
```

Pushes the new value to the end of the vector.

Arguments:

const char* val	string to be inserted in the vector
int* error (optional)	returned error code

Return value:

void

6.10.27 VecInsertDataObject

```
void VecInsertDataObject(DataObject* val, int* error=0)
```

Pushes the new value to the end of the vector.

Arguments:

DataObject* val	pointer to the DataObject to be inserted in the vector
int* error	returned error code

Return value:

void

6.10.28 VecInsertValue

```
void VecInsertValue(const Value* val, int* error=0)
```

Pushes the new value to the end of the vector.

Arguments:

const Value* val	pointer to the Value to be inserted in the vector
int* error	returned error code

Return value:

void