

User Tools: Basic Access API Design and Implementation

Min-Daou Gu

mg346@cs.cornell.edu

Department of Computer Science, Cornell University

December 16, 2005

Abstract

In this project, I designed and implemented two APIs, one is implemented as a dynamic linking library named BALOGIC and the other is an ATL Server web service named BAWS.

BALOGIC provides the basic functions to retrieve the crawls' information and page contents. BAWS functions as a wrapper for the underlying BALOGIC. It provides efficient, cross-platform, and language-independent web services.

Efficiency and robustness are the crucial concern when designing the API. In this report, design alternatives I faced and decided are discussed.

Introduction

The web lab project[1] stores enormous numbers of web pages from Internet Archive [2]. These pages are valuable for sociologists, computer scientists, and analysts to utilize and study.

To handle such huge amounts of data, it needs extremely delicate and well-designed mechanisms to transfer, load, pre-process, post-process, store, index, query, and display. In this phase of the project, we are focusing on the design of user tools. We seek best ways for the user tools to access the backend database servers. Meanwhile, we must assure that the user tools have the following features:

1. Efficiency
The user tool is open to the public. It must endure the high data flow and the computational load-bearing.
2. Easy to use
The users who use the user tools may vary from engineer to researcher. The tools must be easy to learn and to use.
3. Platform-crossing
Users who use the tools can not be limited by the OS platforms.

4. Programming Language Independency
The API should be a generic interface which can be incorporated with different programming languages.

5. Extensible
Adding new functions to the user tools should not be difficult. The flexibility and extensibility of the user tools must be kept.

This report introduces two APIs which are the user tools in the Web Lab project. These tools lets users retrieve basic information of the crawls and pages in the database.

One of the APIs is called BALOGIC which stands for the Basic Access LOGIC. It is a C++ native Microsoft dynamic linking library providing full functions and the best efficiency to access the backend server.

The other one is called BAWS standing for the Basic Access Web Service. It is an ATL Server web service based on COM interface.

Both API design are focusing on the five features listed above. BALOGIC assures the efficiency and BAWS is capable of crossing platform, implementing generically, and extending easily. Users may choose which to use according to their needs.

Besides, a generic logging mechanism is essential to the API design. This report proposes an multi-function logging module which can be easily merged into existing systems.

Related work

The Google Web API[3] which let software developers can query billions of web pages directly from their own computer programs. It uses the SOAP and WSDL standards so a developer can program in his or her favorite environment - such as Java, Perl, or Visual Studio .NET. The Google Web APIs service is a beta web program that enables developers to easily find and manipulate information on the

web.

Google API can be a classic model for our user tool design. It is easy to use and its web service interface makes it possible to collaborate with diverse implementation.

The Design and Alternatives

The framework of the Basic Access APIs shows in Figure 1. It is composed by different components. The kernel of the framework is the BALOGIC API dynamic linking library which accesses the MSSQL database *scidata1* via ADO (ActiveX Data Objects [4][5][6]). Users who pursues highly execution efficiency may use the BALOGIC directly. Otherwise, users may use the BAWS which is a wrapper of the BALOGIC. It exports a simplified and generic interface of the BALOGIC. The respective design issues and alternatives of each components in the Figure 1 are discussed as follows:

BALOGIC API

By introducing a base programming language API library, users can use BALOGIC API instead of communicating directly with BA web service API. BALOGIC API supports the full functionality to process users' requests. It converts users' requests to the corresponding SQL commands to retrieve and trim data from the database. BALOGIC API works in conjunction with Microsoft ADO to provide data access ability.

The BALOGIC API is implemented in native C++ code other than managed code because the speed is the most concern. Managed code is not adopted because it works slow comparing to native code. However, managed code is a fairly good choice for rapidly developing desktop applications.

Besides, static library is not introduced because if the library updates, it needs to recompile the whole project to build the object. It lacks the flexibility that DLL has. Using native code C++ and DLL would be a good balance between the efficiency and the flexibility.

Microsoft ADO

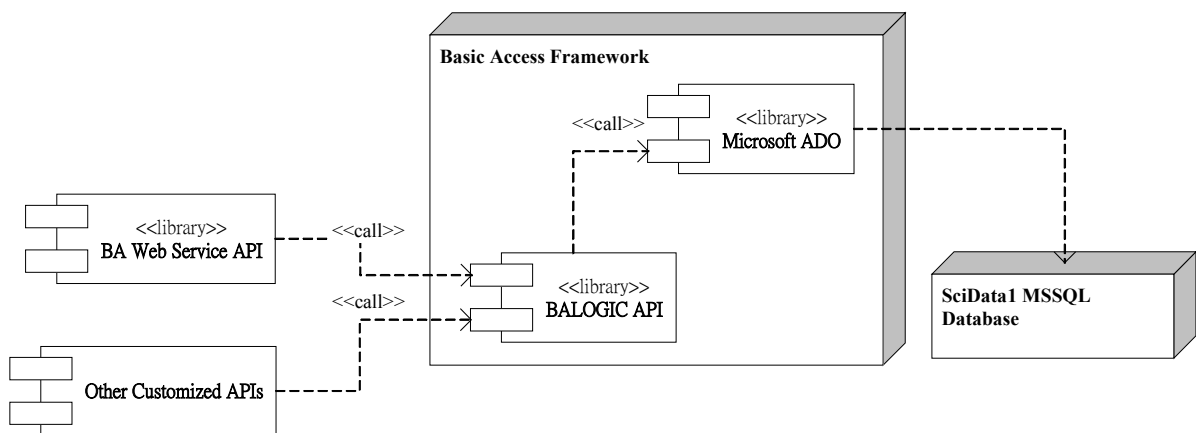
Microsoft ActiveX Data Objects (ADO) [4] has a set of COM classes with the capability to support all kinds of database operations. It enables our program to access and manipulate data from the data source through an OLE DB provider. We utilize ADO to access the MSSQL database.

There are alternatives to communicate with a database system. ADO, ODBC, and OLE DB are the common solution. ADO is adopted because it has so many advantages such as high speed, low memory overhead and the most important one, ease of use.

BA Web Service API

The most significant advantage of web service is that it allows interoperability between

Figure 1, Basic Access Framework



applications built on existing Internet standards such as XML and HTTP. As a result, any system capable of parsing text and communicating via a standard Internet transport protocol can communicate with a Web service, regardless of the platform, operating system, etc.

Users can use the functionality of a Web Service from a various number of applications, and they don't have to re-write the same code in every application that uses those functions.

Users invoke functions in our web services, send their query requests, and retrieve results in their native programming language. Our web service is implemented by using ATL Server and published on an IIS server.

Using ATL server web service ensures the advantages of C++ native code, that is, the speed. In the future, BAWS will be extended to take the responsibilities to handle the heavy web service requests and responses. Therefore, the efficiency is crucial to the utilization afterward.

The Implementation

Basic Access LOGIC

BALOGIC is a base API module implemented as a C++ dynamic linking library (DLL). The main purpose of introducing BALOGIC is to encapsulate database communication, SQL commands, and abstract data manipulations. This library supports basic functions to higher level API such as web service or users' customized API, which in turn builds more specific functions to be used by clients.

In general, BALOGIC wraps the ADO and exports functions which access the *scidata1* database.

In the current phase, BALOGIC provides functions like: retrieving all crawls information, retrieving crawls by date limitation, and retrieving crawls by URL. The most important function is to retrieve a specified web page content, however, it can not be done because the design of page content storage in the database has not been decided.

Basic Access Web Service

BAWS stands for Basic Access Web Service. To cross platform and programming language, we wrap Basic Access Logic DLL API and export its functions by BAWS to give users more alternatives and flexibilities to utilize our service.

BAWS is a set of API functions implemented by ATL Server Web Service [7][8][9] in Microsoft .Net Framework. Users can invoke BAWS API functions in their native programming languages to do basic manipulations through our web service. Please refer to the BAWS API specification section for all the details and usages.

ActiveX Template Library (ATL) Server is one of the newest technologies within Visual C++ .Net. It is an unmanaged native C++ program library for use when building web applications, XML web services, and other server applications. It is known that using managed code to build a web service is so simple in .Net framework. Why are we using ATL Server?

First of all, no other language within .NET Framework can beat the performance of C++ and this is a critical concern in the project. Second, ATL Server provides support for debugging and diagnostics. Lastly, ATL Server supports various other technologies, such as cookies, form processing, thread management, cryptography, and so on. Briefly, ATL Server brings BAWS the power and speed of C++.

Logging Module

The logging module is an static library implemented in native C++ code. The interface of the module is a variable parameters function which works like `sprintf()` in C. Please refer to the appendix C for more details.

The logging mechanism provides means for logging/debugging in situations of error in figure 2. The mechanism should help users or the developers of the API to track usage and locate errors more easily.

The logging module has the following requirement:

1. Dynamically switched on and off:

 Toggling logging mechanism will result

immediate effect after each API call. Developers do not need to recompile their programs to update the logging module.

2. Logging Level:

Not all messages need to be logged.

Crucial errors and minor errors have different priorities to log into files.

3. Log Rotation:

If the program continuously logs into a file without any control. That log file will expand unlimitedly. In case of that, the exceeding of disk quota may cause serious security problem. Hence, our logging module rotates the logging files and discards the old one automatically.

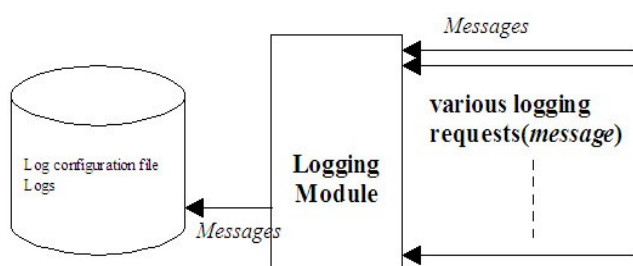


Figure 2, Logging Module Workflow

Results

The project's implementation environment is set up on a personal laptop with the IDE Microsoft Visual Studio .NET 2003 Professional edition installed. The OS on that machine is Microsoft Windows 2003 Enterprise Version. Both BALOGIC and BAWS described above have been implemented, deployed, and tested in this environment.

Both APIs work well by testing through a command-line executable test program. This program sent a request to the BAWS to retrieve all crawls. BAWS then invoked BALOGIC to handle the request. BALOGIC translated the request into SQL command, make use of ADO to retrieve the crawls information, and returned it back to the BAWS. Finally, BAWS sent back all crawls to the client side of the web service.

Conclusion

The achievement of this project is to build up the basic access tools for further advanced using. Having the advantages of the efficiency, flexibility, simplicity, and application

interoperability, BALOGIC and BAWS are capable of retrieving data from the database with perfect ease. I believe that these two APIs can be a strong stepping-stone for the further integration and extension.

Future work

The program and web service API of data retrieving are now prepared for utilizing.

The first application of BAWS will be the integration with *Retro Browser*[10] which provides the capability of viewing web pages in the past. It will function based on BAWS API. As soon as the database design of page contents storing completes, BAWS can serve Retro Browser the required page contents.

Other further works include that once the full-text index search engine is complete, this basic access API model can be extended to various useful user tool applications. Developers then focus on applying and expanding the API to a Google web API-like service providing advanced query methods for multi-disciplinary researchers.

Moreover, the authentication of the API and the web service must be enforced. It can be designed as an extension of the present service or an independent service collaborating with other services.

Acknowledgements

This work is part of the Web Laboratory, which is a joint project of Cornell University and the Internet Archive. This work is funded in part by National Science Foundation grants 0403340, 0127308, and 0537606.

I appreciate all the guidance and advice from professor William Arms and project leader Blazej Kot at Cornell University. Without them, I would never fulfill my project to this extent.

References

- [1] Web Lab Project
<https://gforge.cis.cornell.edu/projects/wri/>
- [2] Internet Archive
<http://www.archive.org/>
- [3] Google Web APIs
<http://www.google.com/apis/>
- [4] MSDN ActiveX Data Objects 2.8 Start Page

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/dasdkadooverview.asp>

[5] Using ADO with Visual C++

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/mdhowhowvcusersshouldreadadodocumentation.asp>

[6] ADO API Reference

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/mdmscadoapireference.asp>

[7] ATL Server Tutorial

<http://msdn2.microsoft.com/en-us/library/60kkc2ah.aspx>

[8] soap_method

<http://msdn2.microsoft.com/en-us/library/ayexh789.aspx>

[9] Supported Types in XML Web Services

Created with ATL Server

<http://msdn2.microsoft.com/en-us/library/hk5cz05e.aspx>

[10] Retro Browser

https://gforge.cis.cornell.edu/frs/shownotes.php?release_id=156

Appendix

- A. The API specification of Basic Access Logic Library
- B. The API specification of Basic Access Web Service
- C. The design specification of Logging Module

A. BALOGIC Class and Function Specification

Namespace: BasicAccessLogic

BA_DBConnector <class>

Introduction:

1. This class encapsulates the establishment of ADO connection. Every connection object represents single connection to a database, and is responsible of sending SQL commands to the database and retrieving returned data.
2. Each connection must be established using *establish()* function before sending requests. Invoke *close()* function after use and this will remove all the login information from a connection object and reset the state to un-established.
3. The class private member *_ConnectionPtr connector* which is a class object of the *ADO* does the primary database connection jobs. It is hidden to API user, but it is public to our *BA_BasicFunc* class for manipulation.
4. Upon destruction of the object *close()* function will be called first.

Class Private Members:

_ConnectionPtr connector

_bstr_t connection_string

_bstr_t uid

_bstr_t password

bool established

Class Public Members:

establish (const _bstr_t & connection_string, const _bstr_t & uid, const _bstr_t & password): int

Parameters:

connection_string: Indicates the information used to establish a connection to a data source.

Ref: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/ado270/htm/mdproconnectionstring.asp>

uid: User ID which is used to connect the data source.

password: Password of the user ID.

Description:

Establish a sustained ADO connection. Upon successful establishment, the *BA_DBConnector* object records connection details for subsequent use and sets *established* to *true*. If a connection has been established, you can not reestablish it without invoke *close()* first.

Return codes:

BA_SUCCESS

BA_GENERAL_ERROR

execute (const _bstr_t & sql_cmd): _RecordsetPtr

Parameters:

sql_cmd: A SQL command used to generate the returned record set pointer.

Description:

BA_DBConnection class provides an easier way to do SQL query. Users can give the SQL command string directly to this function and get the record set pointer as the return value. Users have to free the *_RecordsetPtr* pointer by themselves after use by using *_RecordsetPtr::Close()* function.

Return codes:

_RecordsetPtr

NULL

close(void): int

Description:

Disconnect the current ADO connection, reset connection details (*established*, *connection_string*, *uid*, and *password*) to the initial condition. It fails to close a connection if it has been already closed.

Return codes:

BA_SUCCESS

BA_GENERAL_ERROR

isEstablished(void): bool

Description:

Return *true* if an ADO connection has already been established and ready to use. Otherwise return *false*.

getConnectionString(void): const _bstr_t&

Description:

Return the *connection_string* of the current connection.

getUID(void): const _bstr_t&

Description:

Return the user ID of the current connection.

getPWD(void): const _bstr_t&

Description:

Return the password of the user ID of the current connection.

BA_Crawl <struct>

Introduction:

1. The structure *BA_Crawl* stores ID and Date information of a specified crawl.

Public Struct members:

_bstr_t crawlID

_bstr_t crawl_date FORMAT: **YYYY/MM/DD**

BA_BasicFunc <class>

Introduction:

1. BA_BasicFunc class supports all functionalities of our BALOGIC API. Users invoke functions in this class to retrieve structural data from the database.
2. This class includes all the functions of various manipulations. All methods are *static* in this class thus can be invoked directly without object construction.
3. *An BA_DBConnector* object must be constructed and established first for the functions in *BA_BasicFunc* class

Class Public Members:

*static int getAllCrawls(const BA_DBConnector & dbc,
vector<BA_Crawl> & crawls)*

Parameters:

dbc: An already established *BA_DBConnector* object.

crawls: An empty vector of structure *BA_Crawl* which is used to be filled up by this function.

Description:

The function fetches the crawlID and the date of all crawls in the database and then stores the <crawlID, crawl_date> pairs in the vector *crawls* which is composed of structure *BA_Crawl*.

Return codes:

BA_SUCCESS

BA_GENERAL_ERROR

BA_CONNECTION_UNESTABLISHED

```
static int getPageID(const BA_DBConnector& dbc,
                    vector<_bstr_t> & pageID,
                    const _bstr_t& URL,
                    const _bstr_t& date_begin,
                    const _bstr_t& date_end,
                    const _bstr_t& crawlID)
```

Parameters:

dbc: An already established *BA_DBConnector* object.

pageID: An empty vector of string which is used to be filled up by page ID.

URL: URL of the pages to be fetched. URL must not be blank

date_begin: The start of the date range. **DATE FORMAT: YYYYMMDD**

date_end: The end of the date range. *date_end* must be greater than *date_start*.

DATE FORMAT: YYYYMMDD

start: the start offset of fetched pages. *start* > 0.

number: the amount of pages to be fetched. 0 < The value of *number* <= 50.

crawlID: ID of a specified crawl. This parameter can be *omitted or empty*. In that case, all crawls will be searched for the request.

Description:

The function fetches pages according to the given crawlID, URL, and the date range in the database.

If the length of *date_begin* = 0, all crawls completed before *date_end* will be fetched.

If the length of *date_end* = 0, all crawls completed after *date_begin* will be fetched.

If both length of *date_begin* and *date_end* are zero, all crawls will be fetched.

The returned pages in the vector is sorted by their archive time from old to the latest.

Return codes:

BA_SUCCESS

BA_PAGE_NOT_FOUND

BA_GENERAL_ERROR

BA_CONNECTION_UNESTABLISHED

BA_INVALID_PARAMETER

```
static int getFileLocation(const BA_DBConnector& dbc,  
                           vector<_bstr_t> & fileLocation  
                           const _bstr_t& pageID)
```

Parameters:

dbc: An already established *BA_DBConnector* object.

fileLocation: An empty vector of string which is used to be filled up by file stored path.

pageID: Fetch file locations for this page ID.

Description:

The function fetches file locations for the given pageID.

Return codes:

BA_SUCCESS

BA_LOCATION_NOT_FOUND

BA_GENERAL_ERROR

BA_CONNECTION_UNESTABLISHED

BA_INVALID_PARAMETER

```
static int getCrawlsByURL(const BA_DBCConnector& dbc,  
                          const _bstr_t& URL,  
                          vector<BA_Crawl> & crawls)
```

Parameters:

dbc: An already established *BA_DBCConnector* object.

URL: URL string. URL must not be empty.

crawls: An empty vector of structure *BA_Crawl* which is used to be filled up by this function.

Description:

The function retrieves all *BA_Crawl* crawls in which the given URL string appears.

Return codes:

BA_SUCCESS

BA_GENERAL_ERROR

BA_CONNECTION_UNESTABLISHED

BA_INVALID_PARAMETER

```
static int getCrawlsByDate(      const BA_DBConnector& dbc,
                                const _bstr_t& date_begin,
                                const _bstr_t& int date_end
                                vector<BA_Crawl> & crawls)
```

Parameters:

dbc: An already established *BA_DBConnector* object.

date_begin: The start of the date range. **Date format: YYYYMMDD**

date_end: The end of the date range. **Date format: YYYYMMDD**

crawls: An empty vector of structure *BA_Crawl* which is used to be filled up by this function.

Description:

The function retrieves all *BA_Crawl* crawls which are completed within the given date range. If the length of *date_begin* = 0, all crawls completed before *date_end* will be fetched. If the length of *date_end* = 0, all crawls completed after *date_begin* will be fetched. If both length of *date_begin* and *date_end* are zero, all crawls will be fetched.

Return codes:

BA_SUCCESS

BA_GENERAL_ERROR

BA_CONNECTION_UNESTABLISHED

BA_INVALID_PARAMETER

Return code reference

These are the return codes defined and use within BALOGIC API.

Code	Define	Comment
0	BA_SUCCESS	Operation succeeded.
9001	BA_GENERAL_ERROR	General failure.
9005	BA_CONNECTION_UNESTABLISHED	Connection must be established first.
9010	BA_PAGE_NOT_FOUND	Can not find pages in database.
9015	BA_LOCATION_NOT_FOUND	Can not find file location in database.
9020	BA_INVALID_PARAMETER	Parameters invalid.

B. Basic Access Web Service API Specification:

Namespace:

BAWSService

Structure:

```
struct stCrawl
{
    BSTR    crawl_ID;
    BSTR    crawl_Date;
};
```

COM Interface:

```
interface IBAWSService
{
    [id(1)]    HRESULT getPage(    [in] BSTR URL,
                                [in] BSTR date_begin, [in] BSTR date_end,
                                [in] BSTR crawl_ID,
                                [out, retval] BSTR* page,
                                [out] BSTR* err_msg);

    [id(2)]    HRESULT getCrawls( [out] int* sizeOut,
                                [out, retval,size_is(*sizeOut)] stCrawl** pArOut,
                                [out] BSTR* err_msg);

    [id(3)]    HRESULT getCrawlsByURL([in]BSTR URL,
                                [out]int* sizeOut,
                                [out, retval,size_is(*sizeOut)] stCrawl** pArOut,
                                [out] BSTR* err_msg);

    [id(4)]    HRESULT getCrawlsByDate([in]BSTR date_begin,
                                [in]BSTR date_end,
                                [out]int* sizeOut,
                                [out, retval,size_is(*sizeOut)]stCrawl** pArOut,
                                [out] BSTR* err_msg);
};
```

class CBAWSService : public IBAWSService

Class Public Members:

[soap_method]

```
HRESULT getPage(          BSTR          URL,
                        BSTR          date_begin,
                        BSTR          date_end,
                        BSTR          crawl_ID,
                        BSTR*         page,
                        BSTR*         err_msg)
```

Parameters:

URL: URL of the page to be fetched. URL must **NOT** be empty.

date_begin: The start of the date range.

Date format:

YYYYMMDD

date_end: The end of the date range.

Date format:

YYYYMMDD

****date_end must be grater than date_start.***

crawl_ID: ID of a specified crawl. crawl_ID must **NOT** be empty.

page: The retrieved page content.

err_msg: If return is not S_OK, you can acquire error messages from this.

Description:

The function fetches the page according to the given crawl_ID, URL, and the date range in the database. If the length of *date_begin* = 0, the page archived before *date_end* will be fetched. If the length of *date_end* = 0, the page archived after *date_begin* will be fetched. If both length of *date_begin* and *date_end* are zero, no date range.

Return codes:

S_OK

E_FAIL

E_INVALIDARG

[soap_method]

```
HRESULT getCrawls(          int*          sizeOut,  
                        stCrawl**      pArOut,  
                        BSTR*          err_msg)
```

Parameters:

sizeOut: The number of the elements in the structure array.

pArOut: A pointer points to a structure array. This array is used to be filled with all crawls.

err_msg: If return is not S_OK, you can acquire error messages from this.

Description:

This function fetches the crawl's ID and the crawl's date of all crawls in the database and then stores the <crawl_ID, crawl_Date> pairs in the *structure array pArOur* which is composed of structure *stCrawl*. The size of the array changes dynamically, so this function also stores the size of the array into the arguments *sizeOut*.

Return codes:

S_OK
E_FAIL

[soap_method]

HRESULT getCrawlsByUrl(BSTR URL,
 int* sizeOut,
 stCrawl** pArOut,
 BSTR* err_msg)

Parameters:

URL: URL string. URL must **NOT** be empty.

sizeOut: The number of the elements in the structure array.

pArOut: A pointer points to a structure array. This array is used to be filled with all crawls.

err_msg: If return is not **S_OK**, you can acquire error messages from this.

Description:

The function retrieves all crawls in which the given URL string appears. This function fetches the crawl's ID and the crawl's date of those crawls in the database and then stores the <crawl_ID, crawl_Date> pairs in the *structure array pArOut* which is composed of structure *stCrawl*. The size of the array changes dynamically, so this function also stores the size of the array into the arguments *sizeOut*.

Return codes:

S_OK
E_FAIL
E_INVALIDARG

[soap_method]

```
HRESULT getCrawlsByDate(          BSTR          date_begin,  
                                BSTR          date_end,  
                                int*         sizeOut,  
                                stCrawl**    pArOut,  
                                BSTR*        err_msg)
```

Parameters:

date_begin: The start of the date range. **Date format:** **YYYYMMDD**

date_end: The end of the date range. **Date format:** **YYYYMMDD**

***date_end must be greater than date_start.**

sizeOut: The number of the elements in the structure array.

pArOut: A pointer points to a structure array. This array is used to be filled with all crawls.

err_msg: If return is not S_OK, you can acquire error messages from this.

Description:

The function retrieves all crawls which are completed within the given date range.

If the length of *date_begin* = 0, all crawls completed before *date_end* will be fetched.

If the length of *date_end* = 0, all crawls completed after *date_begin* will be fetched.

If both length of *date_begin* and *date_end* are zero, all crawls will be fetched.

Return codes:

S_OK

E_FAIL

E_INVALIDARG

C. Logging mechanism design

Procedure:

1. API user manually creates a text configuration file named “BALOGIC_log.cfg”. Please refer to the configuration file section for detailed format.
2. When BALOGIC function calls the logging module, logging configurations in the configuration file are checked. The logging mechanism is turned on only if the setting file is present and in its content the name “log_mode” is set to “Y”. To turn logging off, simply remove the file or set “log_mode” to “N”. Since the configuration file is checked at every log function, this ensures immediate effect after each function call.

Log configuration file spec:

Name: BALOGIC_log.cfg

Location

(win32): windows install path i.e. C:\windows\

Format:

log_mode=Y N	(Whether to turn logging on or off.)
log_level=1~3	(Level of log, <i>highest=3, default=1</i>)
log_path=C:\API_LOG\ log_size=500000	(Path name of the log, <i>default=current working dir.</i>) (Max size of the log file, <i>default size=500KB</i>)

Log file format:

BALOGIC.log

[TIME] [module] _FILE_, _LINE_, function arguments, return

Log file rotation:

Unlimited log is abandoned. There will be two files rotated for this log file:

1. BALOGIC.log
2. BALOGIC.bak

When exceeding the log file size limitation (Default= 500KB, this could be set in the log configuration file), the BALOGIC.log is renamed to “BALOGIC.bak” and replace the existing old backup log file - “BALOGIC.bak”. Therefore, a maximum disk space of 1MB is required for this rotation routine.

Logging level definition:

We choose to implement logging level for its efficiency in use. Logging Level would be level 1 ~ 3, 3 being the highest.

Level Definition:

- Level 1: Log only if error happens (default)
- Level 2: Log arguments and returns of functions.
- Level 3: Log database connection input output

Function design:

```
void apiLog(int level, format, .....)
```

Parameter:

level: logging level range=1~3.
format: variable parameter list.

Description:

Use this function to log messages. The *level* indicates the seriousness of the log. *Format* is like `sprintf` or `fprintf` in C.

Return:

void