

# The ActivContentGateway API



Document Version 1.8.11

ACTIV Financial Systems, Inc.  
125 South Wacker Drive  
Suite 2325  
Chicago, IL 60606

## Contents

1.0 Introduction.....	4
2.0 Overview of the ActivContentGateway API .....	5
2.1 Server Connection.....	5
2.2 Content Gateway Discovery .....	5
2.3 Logon and Field Level Permissioning .....	5
2.4 Feed Level Subscription .....	5
2.5 Interactive Requests .....	6
2.6 Updates .....	6
2.7 Interactive Request Modes.....	6
2.8 Bandwidth Efficiency and Network Latency.....	7
3.0 The Content Platform Data Model.....	8
3.1 Record Data .....	8
3.2 Time Series Data.....	8
3.3 News Stories .....	8
4.0 The Navigation Model .....	9
4.1 Background to Navigation Data.....	9
4.2 Using the Navigation Model in Requests .....	10
5.0 The ActivContentGateway API in detail .....	14
5.1 Use of the C++ Standard Template Library.....	14
5.2 Use of ActivMiddleware Functions .....	14
5.3 Namespaces.....	14
5.4 The ContentGatewayClient Class .....	15
5.5 Connecting to a Content Gateway .....	16
5.6 Feed Level Subscription .....	18
5.7 Interactive Record Requests .....	19
5.8 Processing Record Updates.....	33
5.9 Parsing Field Data.....	34
5.10 Time Series Requests.....	35
5.11 Symbol Directory Requests .....	44
5.12 News Server Requests.....	46
5.13 Metadata Requests .....	49
5.14 Important Auxiliary Classes and Types in the ActivContentGateway API.....	50
6.0 References .....	52
Appendix 1 - Example RequestBlock and ResponseBlock usage .....	53
Appendix 2 - Symbology Reference .....	56
A2.1 Equities.....	56
A2.2 Equity Options .....	56
A2.3 Futures.....	57
A2.4 Future Options.....	57
A2.5 Future Spreads.....	57
A2.6 Market Makers .....	57
A2.7 Order Book.....	57
A2.8 Exchange Traded Funds.....	57

A2.9 Forex .....	58
A2.10 Index.....	58
A2.11 Mutual Fund.....	59
A2.12 Money Market.....	59
A2.13 Rankings.....	59
A2.14 Exchange Statistics .....	59
A2.15 News Stories .....	60
Appendix 3 - Exchange List .....	61
Appendix 4 - Event Types .....	64
A4.1 EVENT_TYPE_NONE .....	64
A4.2 EVENT_TYPE_TRADE .....	64
A4.3 EVENT_TYPE_TICK .....	64
A4.4 EVENT_TYPE_TRADE_CORRECTION.....	65
A4.5 EVENT_TYPE_TRADE_CANCEL .....	65
A4.6 EVENT_TYPE_TRADE_NON_REGULAR.....	65
A4.7 EVENT_TYPE_BBO_QUOTE.....	66
A4.8 EVENT_TYPE_QUOTE .....	66
A4.9 EVENT_TYPE_COMPOSITE_BBO_QUOTE .....	67
A4.10 EVENT_TYPE_CLOSING_QUOTE.....	67
A4.11 EVENT_TYPE_CLOSING_BBO_QUOTE .....	67
A4.12 EVENT_TYPE_OPEN .....	68
A4.13 EVENT_TYPE_CLOSE.....	68
A4.14 EVENT_TYPE_RESET .....	69
A4.15 EVENT_TYPE_NEWS .....	70
A4.16 EVENT_TYPE_NEWS_DELETE .....	70
A4.17 EVENT_TYPE_PURGE .....	70
A4.18 EVENT_TYPE_ALERT.....	70
A4.19 EVENT_TYPE_BBO_DEPTH .....	70
A4.20 EVENT_TYPE_ORDER.....	70
A4.21 EVENT_TYPE_HALT_RESUME.....	70
A4.22 EVENT_TYPE_IMBALANCE_VOLUME.....	71
A4.23 EVENT_TYPE_PRICE_INDICATION.....	71
A4.24 EVENT_TYPE_REFRESH .....	71
A4.25 EVENT_TYPE_REFRESH_CYCLE.....	71
A4.26 EVENT_TYPE_OPTION_REFRESH .....	71
A4.27 Other Event Types .....	72
Appendix 5 - Magazine list.....	73
Appendix 6 - Non common stock issues.....	76
Appendix 7 – Future Aliases.....	78

## ***1.0 Introduction***

This document describes the C++ ActivContentGateway API, a component of the ACTIV Content Platform.

A full introduction to the ACTIV Content Platform and the role of the ActivContentGateway API can be found in [\[1\]](#). We highly recommend reading this before continuing.

## **2.0 Overview of the *ActivContentGateway API***

The *ActivContentGateway API* is a set of C++ classes and types that together form the client interface to the Content Gateway.

The *ContentGatewayClient* class handles connection to the Content Gateway and provides overrideable virtual methods for receiving asynchronous responses and updates.

The *RealtimeRequestHelper*, *TimeSeriesRequestHelper*, *SymbolDirectoryRequestHelper* and *NewsRequestHelper* classes provide functionality for launching data requests and decoding response messages.

Various other support classes provide additional data types and helper functionality.

### **2.1 Server Connection**

The *ActivContentGateway API* is a traditional connection-based client. Communications with a Content Gateway begin when the client initiates a TCP/IP connection and end when the client disconnects. Delivery and correct ordering of data during the lifetime of a connection are guaranteed. Connection state changes are communicated reliably to both client and server.

### **2.2 Content Gateway Discovery**

The *ActivMiddleware Directory Service* supports various dynamic and static service location strategies. The *ActivContentGateway API* uses the Directory Service to locate and connect to a Content Gateway server, and to reconnect in the case of a service failure. The default connection and reconnection policies can be overridden by the application programmer if desired.

### **2.3 Logon and Field Level Permissioning**

The *ActivContentGateway API* must log-on to the Content Gateway before data can be accessed. The log-on credentials are used to establish a set of user permissions, which are enforced by the Gateway when any request for data is made.

Permission checks are performed at field level, not record level. Thus records containing data from mixed sources are fully supported by the permissioning system.

### **2.4 Feed Level Subscription**

Feed-level subscription is designed to support bulk delivery of ACTIV's feed. The feed-level user does not need to know which items to request – a feed-level application simply 'opens the pipe' and listens to data as it arrives. The data takes the form of refresh messages and updates. This approach is suitable for users wishing to write handlers to take ACTIV's feed data *en-bloc* into new or existing in-house systems.

The term *image* is used to refer to a complete copy of a record including all fields. Feed-level subscribers receive record images as part of a *refresh cycle*. Whenever the ticker plant has spare bandwidth on its output communications medium, it uses this to push out

record images. This allows downstream components to build up a complete copy of static or rarely-changing data in the ticker plant database. A complete refresh cycle typically lasts a few hours. Once one cycle completes a new one is started in an endlessly repeating process.

In addition to refresh messages, the feed-level subscriber receives real time updates. These are sent immediately on any change to a record and include **only** the fields that have changed (plus some additional information such as update Id and event type). Updates have higher priority than refreshes and are always output by the ticker plant immediately.

Downstream feed-level applications must handle both refresh messages and updates. There is no guarantee that a feed-level user will receive a refresh before receiving updates on a particular item; in fact this will rarely happen. Typical feed-level applications must 'listen' to a whole refresh cycle before their internal state is fully up-to-date.

It is worth noting that feed-level subscribers can also make use of interactive requests should they so choose – this may make sense under some conditions, e.g. to get access to the time series data stored by the Content Server, if the feed-level application is not databasing this itself.

## 2.5 Interactive Requests

The ActivContentGateway API offers a rich set of interactive requests for both record-based and time series data. Snapshot only and real time watches (with or without initial full data image) are supported. Requests for single and multiple record retrieval, partial matches and list requests are supported, as are flexible time series requests.

Interactive users do not need to handle refresh messages. The Content Server handles refreshes on their behalf to maintain an up-to-date full image of each record.

## 2.6 Updates

In addition to new values of changed fields, updates contain:

- an Update Id, which is a sequential per-record counter that increments on each update.
- an Event Type, which is an enumeration value indicating what caused the update (e.g. a new quote, a trade, a trade correction).

## 2.7 Interactive Request Modes

The ActivContentGateway API supports three request modes for interactive requests, these are:

- Asynchronous
- Synchronous
- Non-Blocking Synchronous

Asynchronous requests accept a RequestId which is used to identify the response when it is received via a callback. Asynchronous requests are preferred in most real-time applications since they offer the best possible system throughput.

Synchronous requests block until the full result set has been retrieved, and provide this data as output parameters to the request call. This is convenient for some kinds of application, e.g. where the result of a request is needed before processing can continue.

Non-Blocking Synchronous requests complete immediately, returning a SyncRequestId object. The user is free to go on and do additional processing, and in particular can launch additional overlapping requests with further SyncRequestId objects. Once the caller is ready, a call (with configurable timeout) is made using the SyncRequestId to pick up the results.

The Non-Blocking Synchronous request mode offers the key advantage of synchronicity (the results are returned in the callers own context) but is almost always a more performant choice than a fully synchronous request. It lends itself to tasks such as building a complex web page, where single-context execution is required, but overlapping of requests is desirable to reduce page build time.

Non-blocking synchronous requests are not currently implemented for Time Series requests.

Note that the Content Gateway receives all requests asynchronously, regardless of the request mode used by the ActivContentGateway API – synchronous request behavior is implemented entirely on the client side. The Gateway will generally overlap response processing to ensure fair treatment of all connected clients.

## **2.8 Bandwidth Efficiency and Network Latency**

ActivMiddleware is highly optimized for performance and minimal. To further reduce on-wire costs, the ActivContentGateway API fully supports features such as field-level requests, including, critically, field level permissioning of requests.

As will be seen in later sections, the ActivContentGateway API supports a very flexible request syntax allowing complex requests to be made in a single step. As an example, a single record request can be used to fetch listing level data for MSFT.Q whilst at the same time determining all indices of which MSFT.Q is a member, retrieving all option root symbols for MSFT.Q, and returning data fields on all of these items.

Complex requests are convenient for the application programmer, but crucially also offer bandwidth savings and significantly reduce the number of network round-trips required to complete complex data retrieval tasks. This leads to very low response latency. Low latency is always good for application performance, and is particularly significant when dealing with HTTP tunnels or other kinds of remote network link.

## **3.0 The Content Platform Data Model**

### **3.1 Record Data**

The ActivContentGateway API accesses record based data stored within the Content Server by making requests via the Content Gateway. The data is stored in ActivDatabase (a generic highly optimized real-time database management system) tables optimized for random access.

Each table has a primary key which is typically the *symbol* of a financial instrument, e.g. MSFT.Q, and a set of fields each identified by a *field Id*. Tables containing record data are generally referred to as *market data tables*. Every table is uniquely identified by an integer *table number*.

The set of fields (plus metadata such as the field type and maximum field length) for a given table is termed a *table template*. Differing kinds of data need different fields and hence live in different tables. Each table has its own table template, and the terms table number and table template are often used synonymously. A typical global market data system may have many hundreds of market data tables.

A given symbol can appear in only one market data table (but will also be used to identify the time series for this item in the time series database). The term *key* is used within this document to mean the combination of symbol and the table number of the table where that symbol resides.

### **3.2 Time Series Data**

Time Series data is held in data tables optimized for the serial nature of the data (adds are usually at the end of the series, lookups are usually sequential). The tables are keyed by time as well as symbol, and are referred to as *time series tables*. Time series tables have table numbers and table templates just as market data tables do.

The time series database can accumulate tick-by-tick data, end-of-day data and intraday bars of various bar periods. These series are stored in separate tables.

### **3.3 News Stories**

The ActivContentGateway API accesses news stories stored in the News Server by making requests via the Content Gateway. The News Server supports a rich query syntax that is described in [News Server Requests](#) below.



## 4.0 The Navigation Model

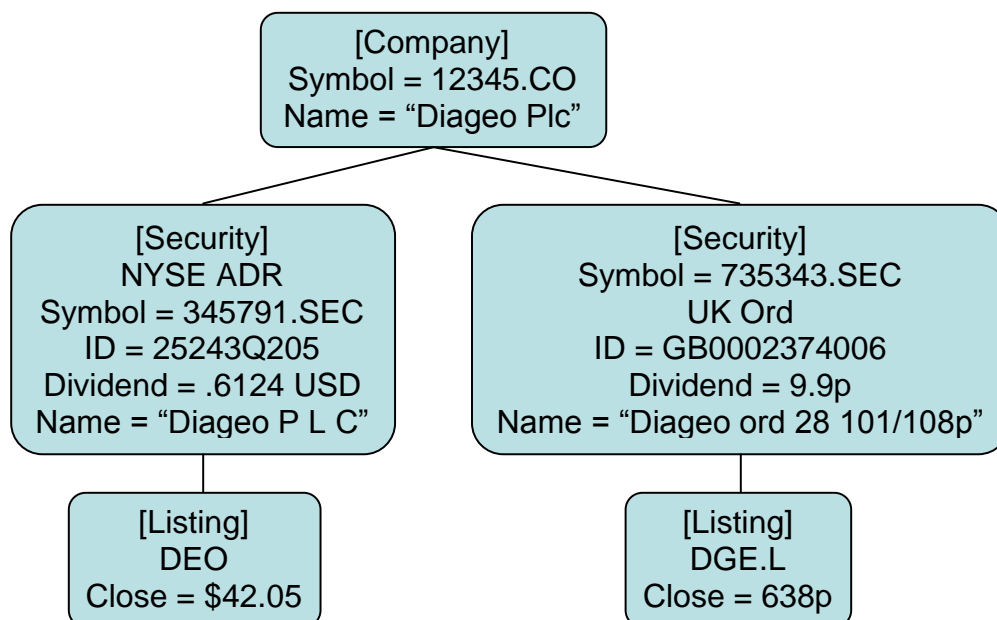
It is important to note that it is not necessary to use the navigation data model to make data requests in the ActivContentGateway API. However, the model allows far greater flexibility and expressiveness in each request and allows complex results to be retrieved in a single network round trip.

### 4.1 Background to Navigation Data

Navigation data is meta-data which expresses links between related items. This concept is well illustrated by considering the relationship between a company, the securities it has issued, and the prices at which those securities are traded on various regional exchanges - this is often referred to as the 'Company-Security-Listing' model. (Note that this model is a simplification of real-world corporate equity structure – an interesting discussion of this topic can be found in [\[2\]](#)).

'Company' refers to aspects of the company itself such as its name, business segments and total market capitalization. Although traditional market data systems often take an ad-hoc approach as to where to store this data, these items can best be represented in their own database record, a 'company level record'. 'Security' refers to the properties of each security – for example ISIN code, par value and dividend details. This data can best be held as 'security level records', one per security. 'Listing' refers to the details of trading a particular security on a particular exchange, for example last trade price, current quote and cumulative volume, and these also deserve their own database record. This is the 'listing level record'.

This model is illustrated in the following diagram:



The symbols employed for the company and securities are arbitrary values given for the purposes of example only. The database representation of the above (ignoring currency issues) is:

Companies table:

Symbol	Name	<Other fields>	...	...
12345.CO	Diageo Plc	...	...	...

Securities table:

Symbol	ID	Dividend	Name	...
345791.SEC	25243Q205	0.6124	Diageo P L C	...
735343.SEC	GB0002374006	9.9	Diageo ord 28 101/108p	

Listings table:

Symbol	Close	...	...	...
DEO	42.05	...	...	...
DGE.L	638	...	...	...

The above data model is clean and easily maintained, and avoids the problems of data duplication and inconsistencies typically present in market data systems which try to ‘cram’ the company and security level data into listing level records. However, a mechanism is needed to relate the symbols to each other. For example, it is clear from the diagram that DGE.L, 735343.SEC and 12345.CO are related entities, but we need a way to represent this programmatically. The ACTIV Content Platform does this through *relationships*, which will be discussed further in the next section.

Company-security-listing is one useful example of navigation but there are many others. Other commonly encountered relationships include option to option root to underlying, performance chains, index and sector index chains, and currency forward points chains, to name but a few. The ACTIV Content Platform supplies many navigation relationships as standard and allows administrators to define additional ones as necessary.

## 4.2 Using the Navigation Model in Requests

A major feature of the ACTIV Content Platform is its extremely powerful and extensible framework for building navigation models, and the ability to use the navigation model seamlessly when making requests for data.

The navigation model uses relationships to link entities such as Companies, Securities, Listings, Option Roots, Indexes and many others. Each relationship is identified by an integer *Relationship Id*.

To illustrate the idea of navigation-based requests, let’s look at some examples. A pseudo request syntax is used for clarity – the actual parameters to Feed API methods are C++ structures and are slightly more complex than those shown here.

### ***Canonical Requests***

The following illustrates how to request the close price for DGE.L (the UK listing of Diageo) along with the related dividend, security name, company name and operating profit.

1. GetEqual("12345.CO", {FIELD\_ID\_NAME, FIELD\_ID\_OP\_PROFIT});
2. GetEqual("56789.SEC", { FIELD\_ID\_NAME, FIELD\_ID\_DIV});
3. GetEqual("DGE.L", { FIELD\_ID\_CLOSE});

The GetEqual request takes a single symbol and a list of fields and returns the appropriate data from the database. (Note - these three separate requests could have been represented instead by a single GetMultipleEqual request which would have executed in a single network round trip.)

### ***Navigation-Based (Aliased) Requests***

This requests above are termed *canonical* because the real key is being specified with each. Here is an equivalent request made using the navigation model:

1. GetEqual("DGE.L",  
    {[RELID\_NONE] FIELD\_ID\_CLOSE},  
    {[RELID\_SECURITY] FIELD\_ID\_NAME, FIELD\_ID\_DIV},  
    {[RELID\_COMPANY] FIELD\_ID\_NAME, FIELD\_ID\_OP\_PROFIT});

This request is processed by the Content Gateway in three steps:

1. the record for DGE.L is found and the CLOSE field added to the results.
2. the SECURITY relationship for DGE.L is 'followed' to reach the relevant security symbol. The security record is located and the NAME and DIV fields are added to the results.
3. the COMPANY relationship for DGE.L is followed to reach the relevant company symbol. The associated record is located and the NAME and OP\_PROFIT fields are added to the results.

This request returns exactly the same results as the previous canonical request. Navigation requests of this kind can be arbitrarily complex, and usually execute in a single network round trip (very large results sets may be returned by the Content Gateway in multiple parts, but this is still preferable to multiple round trips).

There are two big advantages of the navigation-based request:

Firstly, the company and security symbols did not need to be known by the requesting application – only the listing level symbol was required. This will often be known or can be looked up. The only additional prior knowledge was the relationship names, which have fairly obvious meaning. The end result is that the user can access more of the available data with less prior knowledge about its symbology and location.

Secondly, many kinds of relationship resolve differently over time. For example, consider a request for the trade price of every Dow Jones Industrial Index constituent:

1. GetEqual(“=DJINDJ”, {[RELID\_LISTING] FIELD\_ID\_TRADE})

This request will always resolve to the current list of Dow Jones constituents, even though that changes over time.

### ***One-To-Many Relationships***

The above example makes it clear that some navigation relationships are one-to-many - this is accommodated by the ActivContentGateway API. A request for a single symbol may result in a table of results.

### ***Aliasing***

Here are two navigation-based requests that return identical results:

1. GetEqual(“DGE.L”, {[RELID\_COMPANY] FIELD\_ID\_OP\_PROFIT})
2. GetEqual(“56789.SEC”, {[RELID\_COMPANY] FIELD\_ID\_OP\_PROFIT})

In general there may be any number of ways in which a particular result may be requested using navigation-based requests – it depends on the relationships set up on your system. For this reason, navigation-based requests are also referred to as *aliased requests*. Aliasing has important implications for data caching, since the returned symbols don’t in general match the requested symbol. Responses from the ActivContentGateway API always include enough information for the user to be able to maintain this mapping – see ResponseParameters which discusses the response format in detail.

### ***Updating Navigation Links***

We noted above that relationships, such as that between the Dow Jones Industrial Average and its constituents, are liable to change over time.

When a relationship Id is used in a requests, it is resolved based on the current state of the navigation database, and record images are returned to the user followed by real time updates as they occur. Whenever a request is made, the most up-to-date navigation data is used to resolve it, so new requests always receive the correct data.

Subscriptions that were already open at the time of the navigation are not currently affected, although future Content Gateway versions will detect these and update which symbols are subscribed to in real time.

### ***Field Scope***

Another way to look at relationship Ids is to regard them as a field scope. Market data systems typically have fields which may have broad applicability – for example ‘Name’. FIELD\_ID\_NAME is present in many table templates, so the intended meaning is not clear until a symbol is specified. The meaning of the field is relative to the supplied symbol, and the field is said to be *implicitly scoped*.

On the other hand {[RELID\_SECURITY]FIELD\_ID\_NAME} has an explicit meaning, regardless of whether the symbol provided in the request refers to a listing, security or any other entity. The Content Gateway will always look for a navigation path to a security, and if one exists will follow it and return the security name. This field is said to be *explicitly scoped*. We can write fully scoped fields as <scope>.<field id> for brevity, e.g. SECURITY.NAME.

The idea of explicitly scoped fields is potentially interesting – here is an example GUI grid display which can accept any kind of entity and uses navigation to try to show meaningful data:

Symbol	SECURITY.NAME	COMPANY.NAME	LISTING.CLOSE
735343.SEC	Diageo ord 28 101/108p	Diageo Plc	638
DGE.L	Diageo ord 28 101/108p	Diageo Plc	638

Here is how the grid looks without explicitly scoped fields:

Symbol	NAME	NAME	CLOSE
735343.SEC	Diageo ord 28 101/108p	Diageo ord 28 101/108p	n/a
DGE.L	n/a	n/a	638

Of course, such a grid may have to handle one-to-many display issues in practice. For example, LISTING.CLOSE for a security symbol will typically result in multiple targets (one for each listing). GUIs can handle this using a drill-down mechanism – e.g. double click the cell to see the complete list of values.

## **5.0 The ActivContentGateway API in detail**

### **5.1 Use of the C++ Standard Template Library**

Many of the types used in the ActivContentGateway API are typedefs of STL types, particularly `std::vector`. `std::string` is also used extensively. Familiarity with basic STL operations (e.g. iteration) is assumed. For further information on the STL, please consult an appropriate text book, such as [\[3\]](#).

### **5.2 Use of ActivMiddleware Functions**

The ActivContentGateway API relies on ActivMiddleware to provide services such as network communications and service location.

`ContentGatewayClient` is the main class in the ActivContentGateway API. It is derived from `Activ::Component`, making all middleware base `Component` functionality (such as messaging, alarm callbacks, remote console user-interface support and access to directory services) available. This additional functionality is optional – the API described in this document is all you need to write fully functional Feed API applications. Some of the enhanced middleware functionality is licensed separately from the ActivContentGateway API.

The `ContentGatewayClient` class constructor requires an `Application` object reference (which can be `Activ::AgentApplication`, or `Activ::ThinApplication`) and a `ManagedEndPoint` object reference.

`AgentApplication` provides all necessary middleware core services and allows you to write stand-alone applications. `ThinApplication` expects to connect via the ACTIV Shared Memory transport to an `AgentApplication` running in a different process on the same machine, and should be used where multiple clients wish to share middleware services.

### **5.3 Namespaces**

The ActivContentGateway API makes use of the following namespaces:

#### *Activ*

This is the top level namespace. All ACTIV headers reside in this namespace, or a child-namespace.

#### *Activ::ContentPlatform*

All headers in the `include/ActivContentPlatform` directory reside in this namespace, or a child-namespace.

#### *Activ::ContentPlatform::Feed*

Headers which relate to `ActivFeed`, but not specifically to APIs which can access the feed, reside in this namespace. For example, `include/ActivContentPlatform/ActivFeedCommon`.

*Activ::ContentPlatform::FeedApi*

Headers which are common to all ActivFeed APIs (currently the ActivContentGateway API and the ActivContentServer API) reside in this namespace.

*Activ::ContentPlatform::ContentGatewayApi*

Headers which relate to the ActivContentGateway API itself reside in this namespace. ACTIV makes use of the ‘using’ directive in the sample applications provided with the SDK for code clarity and brevity:

```
using namespace Activ;  
using namespace Activ::ContentPlatform;
```

We will assume the above in the code example that follow.

## 5.4 The ContentGatewayClient Class

ContentGatewayClient provides methods for:

- connecting to Content Gateways and connection management;
- starting and stopping feed-level subscription;
- making interactive requests (via helper classes);
- receiving asynchronous data and notifications;
- validating and decoding asynchronous responses;
- handling multi-part responses.

The ContentGatewayClient uses helper classes to implement interactive request functionality for both record, time series and news requests. Interactive requests are described in detail below.

ContentGatewayClient is seldom used directly. Instead application programmers usually derive their own class from ContentGatewayClient so that they can override virtual methods to receive asynchronous notifications. Nonetheless we continue to refer to ‘ContentGatewayClient’ in this document for simplicity.

## Creating a ContentGatewayClient

The following code fragment shows how to initialize a ContentGatewayClient. In this example, the code constitutes the ‘main’ function of a console application.

```
int main(int argc, char *argv[])  
{  
    AgentApplication::Settings settings;  
    AgentApplication application(settings);  
    ContentGatewayApi::ContentGatewayClient client(application,  
        application.GetManagedEndPoint());  
  
    application.Run();  
    return EXIT_SUCCESS;  
}
```

## Threading

Application gives you control over the threading model for your program. In the above example, the main thread is being used to drive the Application object, and all callbacks will be driven from this thread. Run() does not return until the Application has been shut down (typically via its user interface).

If you wish to start a separate thread to drive callbacks, call

```
application.StartThread();
```

This call returns immediately, allowing you to proceed with other processing on your thread. If you do this, you must take care of thread synchronization issues between your thread and the callback thread. You should also stop the callback thread once you are finished using:

```
application.PostDiesToThreads();  
application.WaitForThreadsToExit();
```

When WaitForThreadsToExit() returns thread cleanup is complete and you can exit your application.

StartThread() can be called once only – this is enforced within the API. It is not possible to guarantee correct ordering of data if more than one thread is servicing callbacks.

If you intend to access the ContentGatewayClient from a single thread (i.e. receive updates and make all API method calls from the same thread), you could instantiate a client using the ContentGatewayClient::Settings object as follows:

```
ContentGatewayApi::ContentGatewayClient::Settings settings;  
settings.m_createMutex = false;  
ContentGatewayApi::ContentGatewayClient client(application,  
    application.GetManagedEndPoint(), settings);
```

This saves a small amount of thread synchronization overhead.

## 5.5 Connecting to a Content Gateway

The ContentGatewayClient class offers the following functionality to support connection and management of connection state:

- The Connect() and Disconnect() methods.
- A ConnectParameters member class.
- Overrideable OnConnect(), OnDisconnect() and OnBreak() methods with default handlers.
- A current connection state via the GetState() method.
- The GetServiceLocation() method which returns the url of the Content Gateway that the ContentGatewayClient is currently connected to.
- The GetContentGatewayInfo() method which returns detailed information about the Content Gateway the ContentGatewayClient is currently connected to, including hostname, OS it is running on, uptime, etc.



Connecting to a Content Gateway is a two stage process. The first step is to discover the url of a Content Gateway which is accomplished using the FindServices() method provided by the ServiceApi class of ActivMiddleware.

The second stage is to call the Connect() method, passing a filled in a ConnectParameters object and a timeout value. Please see any of the provided samples for detailed examples of using the FindServices() and Connect() methods.

A timeout value of zero indicates that an Asynchronous connect will be performed. In this case, you can override OnConnect() to be notified when connection completes – you should call the inherited version of OnConnect() from your override as this updates internal Feed API state; this function returns STATUS\_CODE\_SUCCESS if connection was successful.

A non-zero timeout value leads to a synchronous connect. On return from Connect(), provided the return value is good, connection and authentication has succeeded. The timeout is in milliseconds.

At any time the current state of the connection can be determined by calling GetState().

The ConnectParameters class contains the following members:

m_serviceId	Service Id of content gateway.
m_url	Location of a content gateway as returned from the FindServices() method.
m_userName	User name.
m_password	Password.
m_policy	Action to take on a connection break. The default is to continually retry connecting to the same gateway.
m_failoverSeconds	How long to attempt reconnection to the current gateway before failing over to another.
m_attributeQuery, m_agentScope	If the API is set to automatically failover (via m_policy), use these parameters to discover a suitable gateway to failover to. Generally you'll leave them set to the default values.
m_userType	The gateway can restrict logins for the provided user name to certain types of application such as ActivWorkstation, third-party ISV applications, or general API users. Unless your user name is permissioned this way, the default USER_TYPE_API will probably suffice.
m_infoString	An arbitrary string than you can set.

Basic authentication is piggy-backed onto the connection protocol to avoid an additional round trip - set `m_userName` and `m_password` appropriately. In future, additional authentication schemes may be added.

`OnBreak()` is called when a connection to a gateway is lost unexpectedly. By default, the client will attempt to reconnect to the gateway if the connection is broken for any reason. If `m_policy` is set to `POLICY_FIND_NEXT_ALTERNATIVE`, after `m_failoverSeconds` the client re-resolves the service Id with the directory service and tries to connect to the next available server. If no connection is possible, this process is repeated – all available servers are tried in a round-robin fashion until a connection succeeds. `POLICY_FIND_RANDOM_ALTERNATIVE` is similar, except instead of attempting connections in round-robin, a random gateway will be chosen from those available.

If overriding `OnBreak()`, call the inherited method if standard reconnection policies are being used. If the standard policies are not appropriate, your `OnBreak` handler can use the appropriate Middleware services to implement your own custom policy – this is beyond the scope of this document.

Disconnection from the Gateway is instigated by calling `Disconnect()` and is always performed asynchronously. To receive notification of a completed disconnection, override `OnDisconnect()` and call the inherited `OnDisconnect()` handler which returns `STATUS_CODE_SUCCESS` if disconnection was successful.

## 5.6 Feed Level Subscription

Feed level subscription provides ‘raw’ access to ACTIV’s complete feed, after filtering the content at field level according to user permissions.

The `ContentGatewayClient` supports feed level subscription via the `SubscribeToFeed()` and `UnsubscribeFromFeed()` methods.

Various modes of feed level subscription are available via the `SubscribeFeedParameters` object passed to the `SubscribeToFeed()` method. It is possible to receive all record update messages, receive only updates from a subset of event types or just to receive new record and record delete messages. Record updates are delivered via the `OnRecordUpdate()` callback.

The message stream is closed by calling `UnsubscribeFromFeed()`.

The `SubscribeFeedParameters` class contains the following members:

<code>m_type</code>	<code>TYPE_FULL</code> receive all updates  <code>TYPE_ADD_OR_DELETE</code> receive only record add or delete messages  <code>TYPE_EVENT_TYPE_INCLUDE_LIST</code>
---------------------	---

	receive only updates with event types specified in m_eventTypeList  TYPE_EVENT_TYPE_EXCLUDE_LIST receive only updates with event types not specified in m_eventTypeList.
m_eventTypeList	List of event types

For information on processing the received update and refresh data, see 5.8 Processing Record Updates.

## 5.7 Interactive Record Requests

### Request Types

The following interactive requests return record data:

- GetEqual, GetMultipleEqual
- GetMultiplePatternMatch, GetMultiplePatternMatchList
- GetMatch, GetMultipleMatch
- GetFirst, GetLast
- GetNext, GetPrevious

**GetEqual** and **GetMultipleEqual** accept a key or list of keys respectively and return the matching record(s) from the Content Server database. If a table number is not provided, the gateway will search all appropriate Content Server tables.

**GetMultiplePatternMatch** accepts a symbol pattern and returns all matching records. The pattern syntax is described here:

<http://support.activfinancial.com/modules/smartfaq/faq.php?faqid=57>

**GetMultiplePatternMatchList** is similar to **GetMultiplePatternMatch** but with support for a list of symbol patterns instead of just one. The ordering of returned records will be all symbols that match the first symbol pattern, followed by all symbols matching the second symbol pattern, and so on.

A **GetMatch** request will use a heuristic to search for a matching symbol. Two are currently provided:

**MATCH\_TYPE\_COMPOSITE** – attempts to find a composite record that matches the provided symbol or, if no composite exists, the first regional record for that symbol.

**MATCH\_TYPE\_PRIMARY** – attempts to find the primary exchange record for the provided symbol, or, if not found, any record that matches.

**GetMatch** is provided primarily for applications where the user does not wish to know the details of the ACTIV symbology with respect to exchange codes; instead of being

required to enter a fully qualified symbol such as 'IBM.N', they can enter 'IBM' and get either the primary or composite IBM record.

Some examples may clarify this:

<i>Match type</i>	<i>Input symbol</i>	<i>Returned symbol</i>	<i>Reason</i>
Composite	MSFT	MSFT.	A composite for MSFT was found
Composite	MSFT.	MSFT.	MSFT. already the composite record
Composite	MSFT.Q	MSFT.Q	MSFT.Q was an exact match *
Composite	MSFT.PA	MSFT.PA	MSFT.PA was an exact match *
Composite	ADGO	ADGO.QB	No composite, so first regional returned
Composite	ADGO.	ADGO.QB	No composite, so first regional returned
Composite	ADGO.QB	ADGO.QB	No composite, so first regional returned
Primary	MSFT	MSFT.Q	Primary exchange for MSFT found
Primary	MSFT.	MSFT.Q	Primary exchange for MSFT. found
Primary	MSFT.Q	MSFT.Q	MSFT.Q already the primary record
Primary	MSFT.PA	MSFT.Q	Primary exchange for MSFT found

\* A future ContentGateway release will change the functionality of the composite GetMatch in this case to return the more consistent MSFT.

Note the SymbolId provided to GetMatch can leave the table number undefined in a similar vein to GetEqual.

**GetMultipleMatch** is similar to GetMatch but accepts a list of symbols to search for.

**GetFirst**, **GetLast**, **GetNext** and **GetPrevious** together support forward and reverse table walks. GetFirst and GetLast return, respectively, the N records at the start or end of a specified table (where N is specified by the caller). GetNext and GetPrevious accept a table number and symbol and return the N records following or preceding the specified symbol (note that the provided symbol does not actually have to exist).

### ***Common Characteristics***

The interactive record requests have many characteristics in common:

- All use very similar RequestParameters objects – there are slight differences depending whether the request needs a key, a wildcard symbol pattern, or a list of keys, etc.
- All have identical ResponseParameters.
- All allow both canonical and navigation-based requests to be made (and the two can be mixed together).
- All support all three request modes, namely Asynchronous, Synchronous and Non-Blocking Synchronous.
- All give the option of *not* receiving any, or certain, or all fields from the record (subject to permissioning restrictions).

- All give the option of subscribing to the results of the request. See Subscription and Updates for further information.

To make the interface to the various request types as consistent as possible, helper classes are used.

## **Record Request Helper Classes**

Each request type has an associated helper class which works in conjunction with ContentGatewayClient to allow requests to be made simply and efficiently. They are named GetFirst, GetLast, GetNext, etc. to match the name of the request.

The helper class:

- makes the correct request parameters available to you under the standard name 'RequestParameters'.
- Makes the correct response parameters available to you under the standard name 'ResponseParameters'.
- Has methods which let you submit your request as Synchronous, Asynchronous or Non-Blocking Synchronous.
- Has methods which decode asynchronous responses for you.

The helper classes are nested classes of the ContentGatewayClient, so are accessible as (for example) ContentGatewayClient::GetFirst. If you are within a class derived from ContentGatewayClient, you can dispense with the scope operator and just write GetFirst.

## **Example Record Requests**

To demonstrate the interaction of ContentGatewayClient and the Record Request helpers, this section presents example record requests for all three request modes. The code fragments introduce types and concepts which have not yet been discussed – details of these will be presented in later sections.

### ***Synchronous Request Mode***

See DoSynchronousGetFirstRequest() in DocumentationSample.

### ***Asynchronous Request Mode***

Here is an example of how to make an asynchronous GetEqual request for Microsoft Corporation's NASDAQ quote, MSFT.Q. We ask for only the ASK and BID prices. The request is canonical, i.e. does not make use of navigation.

See DoAsynchronousGetEqualRequest() in DocumentationSample.

The above launches the request and returns immediately. To receive the data we need to override a callback.

See OnGetEqualResponse() in DocumentationSample.

Each record request type has a corresponding overrideable response handler in ContentGatewayClient.

Asynchronous requests offer the highest possible performance and throughput for your application.

### ***Non-Blocking Synchronous Request Mode***

Here is an example non-blocking synchronous version of the above request.

See DoNonBlockingSynchronousGetEqualRequest() in DocumentationSample.

Non-Blocking Synchronous requests are almost always a higher performance choice than pure Synchronous requests, because they allow you to overlap requests or do other work while the request is executing. They still offer you the convenience of picking up the results within your own thread context and flow-of-control, rather than via some separate callback function (as in the asynchronous case). They are not as performant as fully asynchronous requests.

## **RequestParameters**

Having looked at the basic form of all three record request modes, we need to look in more detail at the parameters being used.

### ***Common Parameters***

RequestParameters for all record requests include the following members:

m_flags	See the values below.
m_permissionLevel	See the values below.
m_maxResponseBlocks	Maximum response list size.
m_requestBlockList	List of Request Blocks.
m_subscribeParameters	Options for subscribing to the results of the request. See below.

### **Flags**

Possible values for m\_flags are:

FLAG_NONE	No special action.
FLAG_IGNORE_ALIAS	The Content Server database can contain 'alias' records, which allow mapping of a symbol to a record with a different symbol. For example, an alias might exist that maps from DJI.CB to =DJI.CB (the Dow Industrial Average). Normally, if a GetEqual is made for DJI.CB, the record for =DJI.CB is actually returned (the alias lookup happens implicitly). By setting FLAG_IGNORE_ALIAS, this will not be the case and instead you'll receive

	the DJI.CB record. This record just contains a field containing the symbol of the target of the alias - in this case =DJI.CB. It is highly unlikely client applications will require FLAG_IGNORE_ALIAS.
--	---

### Permission Level

Possible values for m\_permissionLevel are:

PERMISSION_LEVEL_DEFAULT	Return the requested symbols at the default permission level for this user. If the user is permissioned for both realtime and delayed data for a given symbol, the default will usually be for realtime.
PERMISSION_LEVEL_REALTIME	Return the requested symbols from realtime tables, if the user is permissioned accordingly.
PERMISSION_LEVEL_DELAYED	Return the requested symbols from delayed tables, if the user is permissioned accordingly.

### MaxResponseBlocks

The m\_maxResponseBlocks member indicates the maximum number of response blocks that should be returned in a single response. If the total response is too large to send in one go, it will be sent as a multi-part response with a maximum of m\_maxResponseBlocks response blocks per partial response.

### RequestBlockList

The m\_requestBlockList member specifies the fields being requested. This is a list of RequestBlock objects, each of which has the following members:

m_flags	See the values below.
m_relationshipId	The navigation relationship scoping each field in the list.
m_fieldIdList	The list of fields.

### Flags

Possible values for m\_flags are:

FLAG_NONE	No special action.
FLAG_ALL_FIELDS	Return all available fields for this relationship Id (m_fieldIdList is ignored).

Refer to 4.2 Using the Navigation Model in Requests for a discussion of canonical and navigational requests and field scope. Canonical requests are made by specifying RELATIONSHIP\_ID\_NONE. Navigation requests are made by specifying a valid relationship Id value for m\_relationshipId, this is then the scope of each field in the list.

Canonical and navigational requests can be mixed together in a request, just use RELATIONSHIP\_ID\_NONE in one RequestBlock and valid relationship Ids in others.

The Content Gateway will fail requests if the same relationship Id appears in more than one request block. It is never necessary to do this – just specify the additional fields you require in the first block.

A record request takes a list of request blocks, and this list applies to all keys specified in the request. For example, if you do a GetMultipleEqual on MSFT.Q and DELL.Q, the same request blocks apply to both. If you need different request blocks for each key, you must launch two separate requests.

The combination of a requested symbol and a request block is called a *partial request*. The number of partial requests in a complete request is therefore equal to the product of the number of requested symbols and the number of request blocks. The response will have at least this number of response blocks (more if a one-to-many navigation link was used).

### ***Request-Specific Parameters***

In addition to the common parameters, record requests need the following additional parameters depending on request type.

GetFirst::RequestParameters

GetLast::RequestParameters

m_tableNumber	The table number whose first or last records should be returned.
m_numberOfRecords	How many records to return.

GetNext::RequestParameters

GetPrevious::RequestParameters

m_symbolId	GetNext and GetPrevious return the previous or next record to the one specified by this key.
m_numberOfRecords	How many records to return.

GetEqual::RequestParameters

m_symbolId	The key of the record that should be returned.
------------	--

GetMultipleEqual::RequestParameters

m_symbolIdList	A list of keys for the records that should be returned.
----------------	---



#### GetMultiplePatternMatch::RequestParameters

m_tableNumber	The table number to search.
m_symbolPattern	The string pattern to search for.
m_requestFlags	REQUEST_FLAG_NONE  REQUEST_FLAG_WATCH as well as returning the current records that match the pattern, additionally watch for new records that match the pattern. Setting this option requires subscribing to the results of the request also.  REQUEST_FLAG_REVERSE_ORDER return the results of the request in reverse order.

#### GetMultiplePatternMatchList::RequestParameters

m_tableNumber	The table number to search.
m_symbolPatternList	A list of string patterns to search for.
m_requestFlags	REQUEST_FLAG_NONE  REQUEST_FLAG_WATCH as well as returning the current records that match the patterns, additionally watch for new records that match any of the patterns. Setting this option requires subscribing to the results of the request also.  REQUEST_FLAG_REVERSE_ORDER return the results of the request in reverse order.

#### GetMatch::RequestParameters

m_symbolId	The (partial) key of the record that should be searched for.
m_matchType	As described in section 5.7: MATCH_TYPE_COMPOSITE or MATCH_TYPE_PRIMARY

#### GetMultipleMatch::RequestParameters

m_symbolIdList	The (partial) keys of the records that should be searched for.
m_matchType	As described in section 5.7: MATCH_TYPE_COMPOSITE or

## ResponseParameters

All record requests share a common response data format. There are no bespoke parameters for any record request type. A ResponseParameters object contains the following fields:

m_responseBlockList	List of response blocks, described below.
m_subscriptionCookie	Subscription cookie, described below.

The response format is somewhat complex, so the discussion is split into sub-sections below.

### *Cookie*

If subscription was used in the request, this cookie must be used when unsubscribing. Otherwise it is set to SUBSCRIPTION\_COOKIE\_UNDEFINED.

For further discussion of subscription cookies, please see Subscription and Updates.

### *ResponseBlockList*

The m\_responseBlockList member contains the bulk of the response data as a list of ResponseBlocks.

The Response Block is the most significant object in record data responses. It defines a set of field data values for a particular partial request.

Typical responses include at least one response block for each RequestBlock in the request. If, however, a one-to-many navigation link was used in a RequestBlock it will generate multiple ResponseBlocks, one for each target of the navigation link. Furthermore, GetMultipleEqual, GetMultipleMatch, GetMultiplePatternMatch and GetMultiplePatternMatchList operations will return additional request blocks for each requested symbol. It is easy to see that a ‘multiple’ request which also uses one-to-many navigation links can potentially lead to a large response.

Response blocks have the following format:

m_status	The status of this block, described below.
m_flags	Described below.
m_resolvedKey	The requested key, with valid table number. Note the symbol in this key may not match that provided to the original request; various transformations may take place on the provided key to resolve it to the actual key of a record in the

	database.
m_responseKey	The returned key. For canonical requests (RELATIONSHIP_ID_NONE) this will equal m_resolvedKey. Otherwise, it is the target of a navigation lookup on m_resolvedKey.
m_relationshipId	The relationship Id of this part of the response.
m_permissionId	The permission group m_responseKey is associated with.
m_permissionLevel	The permission level of the returned record.
m_permissionLevelData	Data specific to different values of m_permissionLevel. Currently if m_permissionLevel is PERMISSION_LEVEL_DELAYED, the m_delayPeriod member of m_permissionLevelData will contain the amount of time (in minutes) the returned data is delayed by.
m_fieldData	The field data scoped by this relationship Id.

### Relating Response Data to your Request

A typical request may include many request blocks, using various different relationshipIds and fields. Requests like GetMultipleEqual also provide a list of keys in one step. The result can be a long list of response blocks.

Here are example request and response blocks for a GetMultipleEqual request for two stocks. We ask for the last close price and a security level item, the current dividend yield.

Requested key 1: MSFT.Q

Requested key 2: MMM.N

#### RequestBlock1:

m_flags	FLAG_NONE
m_relationshipId	RELATIONSHIP_ID_NONE
m_fieldIdList	FID_CLOSE

#### RequestBlock2:

m_flags	FLAG_NONE
m_relationshipId	RELATIONSHIP_ID_SECURITY
m_fieldIdList	FID_CURRENT_YIELD

As long as the request succeeds, the response blocks will look like this (ignoring table numbers, which will be present in both of the key fields):

#### ResponseBlock1:

m_status	STATUS_SUCCESS
m_flags	FLAG_NONE

m_resolvedKey	MSFT.Q
m_responseKey	MSFT.Q
m_relationshipId	RELATIONSHIP_ID_NONE
m_permissionId	PERMISSION_ID_NASDAQ_LEVEL1
m_permissionLevel	PERMISSION_LEVEL_REALTIME
m_fieldData	26.00

#### ResponseBlock2:

m_status	STATUS_SUCCESS
m_flags	FLAG_NONE
m_resolvedKey	MMM.N
m_responseKey	MMM.N
m_relationshipId	RELATIONSHIP_ID_NONE
m_permissionId	PERMISSION_ID_NYSE
m_permissionLevel	PERMISSION_LEVEL_REALTIME
m_fieldData	139.35

#### ResponseBlock3:

m_status	STATUS_SUCCESS
m_flags	FLAG_NONE
m_resolvedKey	MSFT.Q
m_responseKey	MSFT.SEC
m_relationshipId	RELATIONSHIP_ID_SECURITY
m_permissionId	PERMISSION_ID_SECURITY
m_permissionLevel	PERMISSION_LEVEL_REALTIME
m_fieldData	0.31

#### ResponseBlock4:

m_status	STATUS_SUCCESS
m_flags	FLAG_NONE
m_resolvedKey	MMM.N
m_responseKey	MMM.SEC
m_relationshipId	RELATIONSHIP_ID_SECURITY
m_permissionId	PERMISSION_ID_SECURITY
m_permissionLevel	PERMISSION_LEVEL_REALTIME
m_fieldData	1.89

Each response block holds enough information to let you match it back to the relevant part of the request. The combination of m\_resolvedKey and m\_relationshipId gives you the relevant request block and requested symbol.

Significantly, the response block gives you the mapping of alias request key to canonical response key. From the above, we now know that our (aliased) request for MSFT.Q, RELATIONSHIP\_ID\_SECURITY led us to the canonical key MSFT.SEC. Similarly we know that MMM.N, RELATIONSHIP\_ID\_SECURITY led us to MMM.SEC. This may be important to us when we come to receive updates on MSFT.SEC and MMM.SEC, since the updates are marked only with the canonical keys.

There are some further example request and response blocks in Appendix 1 - Example RequestBlock and ResponseBlock usage.

## Status

The status field applies only to this ResponseBlock. Remember that there is also a status associated with the response as a whole. This is determined by calling ContentGatewayClient::IsValidResponse().

m\_status field can take any of the following values:

STATUS_SUCCESS	This part of the request was successful.
STATUS_SOURCE_NOT_FOUND	The requested key could not be found as a source for this relationship Id.
STATUS_SOURCE_NOT_PERMISSIONED	This user is not permissioned for this source.
STATUS_RELATIONSHIP_NOT_FOUND	The requested relationship could not be found.
STATUS_RELATIONSHIP_NOT_PERMISSIONED	This user is not permissioned to use the requested relationship.
STATUS_NAVIGATION_NOT_FOUND	No instances of the requested relationship Id were found.
STATUS_NAVIGATION_NOT_PERMISSIONED	The user is not permissioned to use this instance of this relationship.
STATUS_TARGET_NOT_FOUND	The requested relationship was found, an instance was found for the requested key, but the target of the instance does not exist.
STATUS_TARGET_NOT_PERMISSIONED	The requested relationship was found, an instance was found for the requested key, but the user is not permissioned to see the target data.
STATUS_INVALID_SOURCE_TABLE	An invalid table number was used in a request.
STATUS_PERMISSION_LEVEL_NOT_AVAILABLE	User has requested data from a permission level that is not configured in the Content Gateway. Most likely requesting delayed data when the Content Gateway only provides realtime data.
STATUS_QUOTA_LIMIT_NOT_SUBSCRIBED	A user might be permissioned with a limited number of allowed subscriptions. This status would be received if the user requested a record with subscription but was already subscribed to their quota of symbols. The data in the ResponseBlock is still good, but no updates will be received.

STATUS\_SUCCESS or STATUS\_QUOTA\_LIMIT\_NOT\_SUBSCRIBED indicates that the data in this ResponseBlock is good.

To understand the relationship Id status values, bear in mind that a relationship Id defines a mapping between a source symbol and a target symbol or symbols. The source symbol is the one you specified in the request, the target is the one that the navigation walk leads you to.

This mapping is held in a database table in the Content Server. When navigation resolution takes place, either the source or the target of the mapping may be missing from the table (STATUS\_SOURCE\_NOT\_FOUND, STATUS\_TARGET\_NOT\_FOUND), there may be no entries at all for this relationship Id (STATUS\_NAVIGATION\_NOT\_FOUND), or your user Id may not be permissioned to use the relationships (STATUS\_NAVIGATION\_NOT\_PERMISSIONED). Finally, the ContentGateway may not recognize the relationship Id that you have specified (STATUS\_RELATIONSHIP\_NOT\_FOUND) or your user Id may not be permissioned to use it (STATUS\_RELATIONSHIP\_NOT\_PERMISSIONED).

### Flags

The `m_flags` member can have the following values:

FLAG_NONE	No flags set.
FLAG_STALE	Record image contains fields which are potentially stale. Stale data occurs when the Content Server detects it has missed an update from the incoming broadcast feed (each record has an Update ID which allows this to be detected). This is unusual, but can occur occasionally particularly with satellite delivery during adverse weather or solar conditions.

### Field Data

`m_fieldData` is a self-describing but compressed block of field values and field-level status. The same format is used for record updates. Please see 5.9 Parsing Field Data for a full description of how to decode this data.

### Relationship Id

The `m_relationshipId` value matches the relationship Id that you specified in the request block.

### Request and Response Keys

The `m_resolvedKey` field indicates to which requested symbol a particular response block relates. You may have specified the requested symbol(s) directly (GetEqual or GetMultipleEqual) or it may be implicit (GetFirst, GetNext, GetLast, GetPrevious, GetMultiplePatternMatch, GetMultiplePatternMatchList). Note that `m_resolvedKey` always includes the correct table number even if you did not specify it in the request.

The value of `m_responseKey` is the canonical key of the target data, regardless of whether you made a canonical or aliased request. The fields in the response block always refer to the `m_responseKey`. Again, `m_responseKey` always contains the correct table number.

## Aliasing, Responses and Update Handling

Earlier examples have shown that the response key in the response from an aliased request may not match the key asked for in the request. For this reason, responses are always marked with both the (aliased) request key (`m_resolvedKey` plus `m_relationshipId`) and the (canonical) response key (`m_responseKey`).

Updates are marked with canonical keys only (it does not make sense to try to mark updates with every possible alias which could refer to them). A client making a subscription request generally needs to remember the mapping of alias to canonical key as it cannot match updates to the related request without this knowledge. In the above example, `MSFT.Q` was the requested symbol, yet updates will be received for `MMM.SEC` as well as `MSFT.Q`.

## Subscription and Updates

Receiving updates to records returned by an interactive request is controlled by the `m_subscribeParameters` member of the `RequestParameters` classes. It is similar to the subscription options available when using the `SubscribeToFeed()` method. Here are the members of the `SubscribeParameters` class:

<code>m_type</code>	<code>TYPE_NONE</code> do not subscribe (the default)  <code>TYPE_FULL</code> receive all updates  <code>TYPE_ADD_OR_DELETE</code> receive only record add or delete messages  <code>TYPE_EVENT_TYPE_FILTER_INCLUDE_LIST</code> receive only updates with event types specified in <code>m_eventTypeList</code>  <code>TYPE_EVENT_TYPE_FILTER_EXCLUDE_LIST</code> receive only updates with event types <i>not</i> specified in <code>m_eventTypeList</code> .
<code>m_eventTypeList</code>	List of event types for <code>TYPE_EVENT_TYPE_FILTER * LIST</code>

If the value of `m_subscribeParameters.m_type` is `TYPE_NONE`, users will receive a 'snapshot' of the data they requested, with no subsequent updates.

If two requests are made for the same symbol, one specifying a subscription type of `TYPE_EVENT_TYPE_FILTER_INCLUDE_LIST` and one `TYPE_EVENT_TYPE_`

FILTER\_EXCLUDE\_LIST for the same event type, inclusion is given preference – i.e. the client will receive any updates for that symbol.

The Gateway always guarantees coherency between the full record image returned in the initial response and the subsequent data updates, i.e. there are no gaps in the data stream (missed updates) and no duplicates (updates sent to you that had in fact already been applied to the record at the time the full image was returned). This is extremely important for downstream applications, especially if they perform cumulative calculations or trigger actions based on received values.

If subscription was requested, a valid m\_subscriptionCookie is returned in ResponseParameters, provided the request succeeded. When the updates are no longer desired, the subscription should be canceled by passing this cookie to one of the unsubscribe methods in ContentGatewayClient.

This unsubscribe mechanism is designed to work correctly with both canonical and navigation-based requests, even if the resolution of the navigation-based request has changed since it was made (see Updating Navigation Links).

### ***Partial Unsubscribe***

Where a multiple symbol request has been made, Unsubscribe(cookie, sourceSymbolIdList) may be used to unsubscribe from part of the result set.

Note - the granularity of partial unsubscribe is limited to those symbols specified in the **request**; not those received in the response as part of a navigation walk. For example, if I make a GetEqual request for =DJI.DJ but use navigation to retrieve all Dow Jones Industrial constituents, I can unsubscribe from =DJI.DJ but not from individual constituents. If on the other hand I make a GetMultipleEqual request for MSFT.Q and IBM.N I can unsubscribe separately from either symbol.

### ***Subscription information***

It is possible to query the content gateway to retrieve the current active subscriptions. This is accomplished using the ContentGatewayClient::GetSubscriptionInfo() method. It takes two parameters. The first parameter is the permission level for which you wish to retrieve the subscription information (currently Feed::PERMISSION\_LEVEL\_REALTIME and Feed::PERMISSION\_LEVEL\_DELAYED are supported). The second parameter is a reference to a SubscriptionInfoList object. If the call is successful, the SubscriptionInfoList object will be populated with all the symbols that are currently subscribed at the given permission level and the number of times each symbol has been subscribed.

### ***Deletes***

A delete is an update that has the FLAG\_DELETE bit set. This informs the user that any cached values for the deleted item should be invalidated, and that no further updates will be received on the item.



## Multi-Part Responses

Record requests based on `GetMultipleEqual`, `GetMultiplePatternMatch`, `GetMultiplePatternMatchList`, `GetMultipleMatch` or aliased requests using one-to-many navigation links can have large results sets. The ContentGateway may split these up and send them to the client in multiple parts – this is termed a *multi-part response*. The Content Gateway does this to protect other clients from being ‘locked out’ by large requests. The Gateway tries to give each connected client a fair share of its resources.

The effect of a multi-part response from the `ActivContentGateway` API programmer’s point of view depends on the request mode used. For synchronous requests, there is no apparent difference as far as the programmer is concerned – the request call blocks until all parts have been received and a single `ResponseParameters` object is returned with the complete data set. (Remember that synchronicity is implemented purely on the client side – the Content Gateway receives all requests asynchronously).

For Asynchronous requests, the `Response` callback will be called multiple times with the same `RequestId`; that of the original request. The `ContentGatewayClient` provides the `IsCompleteResponse()` method to allow you to test for the last part of a multi-part response. Note that the decoder functions in the request helper classes do not clear the `ResponseParameters` passed to them – you can pass the same `ResponseParameters` object from each `Response` callback to build up the complete result set.

For the non-blocking synchronous case, the status code returned by the helper class `GetResponse()` method is `STATUS_CODE_PENDING` if there is an additional part to be retrieved. The user should continue to call `GetResponse()` until it returns `STATUS_CODE_SUCCESS` or an error code.

## 5.8 Processing Record Updates

Both feed-level and interactive users receive record updates by overriding the virtual `ContentGatewayClient` method `OnRecordUpdate`. Feed-level users also receive refreshes this way.

See `OnRecordUpdate()` in `DocumentationSample`.

### RecordUpdate

The `RecordUpdate` structure contains the following fields:

<code>m_flags</code>	<code>FLAG_NONE</code> - nominal update  <code>FLAG_DELETE</code> - record has been deleted  <code>FLAG_STALE</code> - some fields (other than those in this update) may be stale  <code>FLAG_REFRESH</code> - this update is a complete refresh image of the record containing all fields
----------------------	--

	FLAG_QUOTA_LIMIT_NOT_SUBSCRIBED - this is a new record matching a watch the user had open, but the user has not been subscribed due to hitting their subscription quota
m_symbolId	Canonical key of the update.
m_updateId	Update Id.
m_eventType	Event type indicating the cause of the update.
m_permissionId	The permission group m_symbolId is associated with.
m_permissionLevel	The permission level of the returned record.
m_permissionLevelData	Data specific to different values of m_permissionLevel. Currently if m_permissionLevel is PERMISSION_LEVEL_DELAYED, the m_delayPeriod member of m_permissionLevelData will contain the amount in minutes the returned data is delayed by.
m_fieldData	The updated fields.

The m\_symbolId is always the canonical key of the item being updated. Aliases are never sent in updates.

m\_updateId is a sequential counter which is incremented each time an update is sent.

m\_eventType is an enumeration which indicates the cause of the update. See the header file include/ActivContentPlatform/ActivFeedCommon/EventTypes.h for a current list.

Updates include only the fields that have changed, and this information is present in m\_fieldData, unless FLAG\_REFRESH is set in which case all fields are present.

The RecordUpdate class provides the self-explanatory helper methods IsNewRecord() and IsDelete().

The 'stale' concept is described further here:  
<http://support.activfinancial.com/modules/smartfaq/faq.php?faqid=50>

## 5.9 Parsing Field Data

Field data is sent as a self-describing but opaque data block. The format is the same regardless of whether the data was sourced from a record response, an update or (for feed-level users) a refresh message.

To decode the field data, use the FieldListValidator helper object. It accepts a reference to the m\_fieldData object of an update or response and provides the following functionality:

- Parses the data into a map of field Id to field data and status

- Provides random access to fields via GetField(FieldId)
- Provides forward const iterator support through the fields

The validator caches the field data it parses, which then has a lifetime independent of the `m_fieldData` used to initialize it. This means the validator can be retained as an efficient field data cache even after you return from the function or callback where you received the update message or response.

The access methods of the `FieldListValidator` return `Field` objects, which contain the following members:

<code>m_fieldId</code>	The Id of the field.
<code>m_fieldStatus</code>	The status of this field.
<code>m_rules</code>	Special rules which might apply to this field.
<code>m_pIFieldType</code>	An <code>IFieldType</code> pointer to the data.

Field status can have any of these values:

<code>FIELD_STATUS_DEFINED</code>	The field data is good.
<code>FIELD_STATUS_UNDEFINED</code>	The field has the explicit value 'undefined'.
<code>FIELD_STATUS_NOT_FOUND</code>	The field was not found.
<code>FIELD_STATUS_NOT_PERMISSIONED</code>	The user was not permissioned to access this field.

`IFieldType *` is a pointer to one of the basic field types supported in the ACTIV Content Platform (these include numeric fields, string fields, binary arrays, datetime etc). The field data classes are polymorphic, deriving from `IFieldType` which provides various access and arithmetic functions. The sample application `FieldTypesSample` describes the use of these classes in more detail.

See `DisplayFieldData()` in `DocumentationSample` for a simple example usage of `FieldListValidator`.

Rather than construct a `FieldListValidator` object each time an update or response is received, it is highly preferable to store a `FieldListValidator` (e.g. as a member of a class) and reuse it with the `Initialize()` method to save construction time and make use of the internal caching of `IFieldType` objects.

## 5.10 Time Series Requests

Time series data is collected by a Time Series Server (TSS); essentially it is a historical series of the changes taking place to the records in the market data tables. However, in addition to basic tick-by-tick and daily closing price series, the TSS maintains Intraday Bar series of various periods. Client access to the TSS is via the `ActivContentGateway` API.

The time series API offers separate functions for retrieving historical (i.e. closing price), intraday and tick-by-tick data. There are various options for filtering (e.g. filtering out irregular trades) and specifying the field list required in the response.

There is very flexible support for specifying time periods. Absolute time, trading or calendar day counts, data point counts and data point addresses can be used in any combination. This greatly simplifies making time series data requests for technical analysis purposes.

Relationship Id navigation is not used in the time series request model. The returned data always relates to the canonical key specified in the request.

There is no equivalent of feed-level access for the time series database – all requests are interactive. Feed-level API users who need access to time series data can either

- use the interactive requests provided by the ActivContentGateway API, or
- maintain their own time series database from the record refresh and update messages that they receive.

Note that ActivFeed carries record-based Corporate Actions information, including that required to maintain a historical database (for example, details of share splits, dividend payments and mergers).

## **Field Sets**

The results of a time series request have different characteristics depending on whether historical bars, intraday bars or tick-by-tick data is requested and are therefore represented by separate objects with appropriate access methods. Currently Tick, HistoryBar and IntradayBar objects are defined. These all derive from a common base, FieldSet, which provides limited common functionality such as conversion of the data to string form for debugging purposes.

## **Request Types**

The following interactive requests return time series data:

- GetHistory
- GetIntraday
- GetTicks

These return closing price history series, intraday bar series of various periods, and tick-by-tick data series respectively. The TSS can maintain a configurable amount of tick-by-tick and intraday bar history (subject to available storage space). 10 years or more of closing price history are available, depending on which stock is requested.

## ***Common Characteristics***

- Time series requests all take the same RequestParameters object. Some values (e.g. field filters) are applicable to only some request types, as described in the following sections. The Content Gateway fails requests that have invalid parameter combinations.

- All three request types support both synchronous and asynchronous request modes.
- All three request types support a GetFirst / GetNext style request mode. This differs somewhat from the record request non-blocking synchronous implementation, and is discussed further below.

## Helper Classes

As is the case for record requests, there is a helper class for each time series request type, and these are implemented as nested classes of ContentGatewayClient. The helpers make correctly typed RequestParameters and ResponseParameters objects available and allow requests to be initiated using any of the available request modes. They also provide decoders for asynchronous responses.

## Overrides

The ContentGatewayClient class provides the following virtual methods which may be overridden to receive asynchronous time series responses:

```
virtual void OnGetFirstHistoryResponse(HeapMessage &response);
virtual void OnGetNextHistoryResponse(HeapMessage &response);
virtual void OnGetFirstIntradayResponse(HeapMessage &response);
virtual void OnGetNextIntradayResponse(HeapMessage &response);
virtual void OnGetFirstTicksResponse(HeapMessage &response);
virtual void OnGetNextTicksResponse(HeapMessage &response);
```

## Example Time Series Requests

Here are some examples of typical time series requests which demonstrate the available request types and the role of the helper classes.

### *Synchronous GetFirst / GetNext for Closing Price History*

See DoSynchronousGetHistoryRequest() in DocumentationSample.

### *Asynchronous GetFirst / GetNext for Tick Series*

See DoAsynchronousGetTickRequest(), OnGetFirstTicksResponse() and OnGetNextTicksResponse() in DocumentationSample.

### *Extracting Time Series Data*

Once you have a valid ResponseParameters object you can extract the time series points from the m\_fieldSetList member. This is an stl vector where each entry is a Tick, a HistoryBar or an IntradayBar depending on the request you made. It is normally preferable to use a separate handler for each of these types, since each provides functionality and accessors specific to the series being returned. For a simple example of handling a tick series response, see DisplayFieldSetList() in DocumentationSample.

For full details of the access methods of Tick, HistoryBar and IntradayBar, please see the relevant header files.

## Request Modes for Time Series Requests

The synchronous and asynchronous behavior for time series requests is the same as for record requests; the former blocks the caller's thread until the results are available, returning these as out parameters; the latter returns immediately and the results are delivered by an asynchronous callback at some later time.

There is no direct equivalent of non-blocking synchronous request mode for time series requests. Instead there is a request mode (referred to as GetFirst/GetNext) which allows the client to retrieve multi-part response messages whilst controlling the rate at which time series data is received. This cookie-based mechanism is explained in the following section.

### *TimeSeriesCookie*

A time series request may specify a very large data set. For example, a request for 5 days of tick data (quotes plus trades) for MSFT.Q may contain over a million data points.

There are two reasons why the initial response to a time series request may not contain the full data set:

1. The Gateway imposes a limit on the size of responses, to protect other Feed API clients against being 'locked out' while the Gateway services a very large request.
2. It is good practice for Feed API application programmers to specify a maximum response size when they make the request. The response will then contain at most this number of data points. This allows the programmer to control the amount of data received and avoid being 'swamped'.

In either case the return code of the initial time series request will be STATUS\_CODE\_PENDING. The user needs to be a way to request the remainder of the series, and this is supported by the use of TimeSeriesCookie.

All time series responses include a TimeSeriesCookie. This contains all of the state necessary for the Content Gateway to continue a fetch operation where it left off. The cookie allows the server side of time series requests to be implemented statelessly, which enhances scalability.

The time series request helper classes offer the methods SendGetNextRequest and PostGetNextRequest which allow the next part of the dataset to be retrieved. Call either of these quoting the cookie. They both return a further cookie and the next part of the response. They return STATUS\_CODE\_SUCCESS if the final part of the response has been retrieved, and STATUS\_CODE\_PENDING otherwise. The process should be repeated until STATUS\_CODE\_SUCCESS (or an error code) is returned.

Because the server side of the request is stateless, there is no obligation on the ActivContentGateway API programmer to complete the request in any particular time frame, or indeed at all. The only proviso is that the data must still be in the database by the time the cookie is used, otherwise the next part of the request will fail.

The synchronous method pair `SendGetFirstRequest` / `SendGetNextRequest` is similar in some respects to the non-blocking synchronous request mode used in the record request model.

There is no record request equivalent of the asynchronous `PostGetFirstRequest` / `PostGetNextRequest` pair. Similarly, there is no time series equivalent of the asynchronous multi-part responses that the Content Gateway supports for record requests – additional time series response parts are never sent by the Content Gateway unless the client explicitly requests them using a `SendGetNextRequest` or `PostGetNextRequest` call.

## RequestParameters

All time series requests share the same parameter format, which has the following structure:

<code>m_symbolId</code>	The requested symbol.
<code>m_flags</code>	Request flags.
<code>m_maxFieldSets</code>	Max number of returned data points.
<code>m_seriesType</code>	The series type, e.g. tick or intraday.
<code>m_recordFilterType</code>	The record filter.
<code>m_fieldFilterType</code>	The field filter.
<code>m_periodList</code> <code>m_delayPeriod</code> <code>m_specifiedInterval</code>	The range of data points to return. Used internally; no need to set. For <code>INTRADAY_SERIES_TYPE_SPECIFIED_MINUTE</code> , the bar period to use. Valid values are 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60

It is not necessary to specify a table number when setting `m_symbolId` – use `Feed::TABLE_NO_UNDEFINED`.

Flags can be either `FLAG_NONE` or `FLAG_REVERSE`. The former causes results to be returned in decreasing date order (i.e. most recent data point first), while the latter causes the order to be reversed (oldest data point first).

A maximum of `m_maxFieldSets` data points will be returned from your request. If the number of data points in the entire requested series exceeds this value, you can request the remainder using `GetNext` with the cookie.

Series type can take the following values:

<code>TICK_SERIES_TYPE</code>	Tick-by-tick data.
<code>INTRADAY_SERIES_TYPE_1_MINUTE</code>	1 Minute intraday bars.
<code>INTRADAY_SERIES_TYPE_5_MINUTE</code>	5 Minute intraday bars.
<code>INTRADAY_SERIES_TYPE_SPECIFIED_MINUTE</code>	Variable period intraday bars.
<code>HISTORY_SERIES_TYPE_DAILY</code>	Daily bars
<code>HISTORY_SERIES_TYPE_WEEKLY</code>	Weekly bars
<code>HISTORY_SERIES_TYPE_MONTHLY</code>	Monthly bars

Only history series types will be accepted for `GetHistory` requests, and only intraday series types will be accepted by `GetIntraday` requests. `GetTick` requests currently have only one possible series type.

RecordFilterType currently has values defined only for tick and intraday series:

TICK_RECORD_FILTER_TYPE_ALL	Return all ticks.
TICK_RECORD_FILTER_TYPE_ALL_TRADES	Return all trades but no quotes.
TICK_RECORD_FILTER_TYPE_REGULAR_TRADES	Return only 'normal' trades.
TICK_RECORD_FILTER_TYPE_ALL_QUOTES	Return all quotes.
TICK_RECORD_FILTER_TYPE_ALL_BIDS	Return all bid quotes.
TICK_RECORD_FILTER_TYPE_ALL_ASKS	Return all ask quotes.
INTRADAY_RECORD_FILTER_TYPE_REGULAR_TRADES	Return only 'normal' trades.
INTRADAY_RECORD_FILTER_TYPE_ALL	Return all trades.
INTRADAY_RECORD_FILTER_TYPE_ALL_PLUS_VWAP	Return all trades + VWAP.

The tick database records both quotes (separate asks and bids) and trades.

TICK\_RECORD\_FILTER\_TYPE\_REGULAR\_TRADES excludes trade types such as late trades, but the exact definition of what is included is exchange-specific.

FieldFilterType can take the following values:

INTRADAY_FIELD_FILTER_TYPE_FULL_BAR	Full intraday bar.
INTRADAY_FIELD_FILTER_TYPE_MINI_BAR	Reduced intraday bar.
HISTORY_FIELD_FILTER_TYPE_FULL_BAR	Full history bar.
HISTORY_FIELD_FILTER_TYPE_MINI_BAR	Reduced history bar.

There is no field filter for tick series. Mini bars include Date/Time, Open, High, Low, Close and Volume values but exclude 'value added' values such as VWAP-related fields.

### ***Specifying Periods for Time Series***

The range of data points in a time series request can be specified very flexibly using the m\_periodList member of the request parameters object.

m\_periodList is a typedef for an stl vector, and can accept any number of Period objects.

Each period object has a PeriodType taken from the following selection:

PERIOD_TYPE_NOW	Current time.	A
PERIOD_TYPE_INFINITY	Plus or minus infinity.	A
PERIOD_TYPE_LOCAL_DATE_TIME	Local date time.	A
PERIOD_TYPE.UTC_DATE_TIME	UTC date time.	A
PERIOD_TYPE_LATEST_TRADING_DATE	The most recent trading date.	A
PERIOD_TYPE_TRADING_DAY_COUNT	A count of trading days.	R
PERIOD_TYPE_DATA_POINT_COUNT	A count of data points.	R
PERIOD_TYPE_DATA_POINT_ADDRESS	The address of a specific data point.	A

Periods are designated as absolute or relative, depending on whether their value denotes a fixed point in the database. The third column of the above matrix denotes absolute types by 'A' and relatives by 'R'.

The rules for building a valid m\_periodList are as follows:



1. Any number of Periods may be added to the list.
2. The first Period in the list defines the oldest data point in the request range.
3. The last Period in the list defines the youngest data point in the request range.
4. At least one absolute period must be specified.
5. No more than two absolute periods may be specified.
6. A relative period may not appear between two absolute periods.
7. There may be any number of relative periods before an absolute.
8. There may be any number of relative periods after an absolute.
9. If a sequence of more than one relative is adjacent to an absolute, the relatives are applied in order of proximity to the absolute and the effect is cumulative.

The Content Gateway will validate the contents of the list and fail requests with invalid combinations.

The idea is that the list is interpreted in chronological order, from the first entry to the last. Absolute periods can be mapped onto fixed points in the time series database and allow the meaning of relative periods to be understood as offsets. The offsets are before the absolute period (going back in time) if the relative value appears in front of the absolute in the list, and vice versa.

Here are some examples of the uses of the period list.

#### Two DateTimes

PeriodList entries (in order, top is first in list):

PeriodType	Value
PERIOD TYPE LOCAL DATE TIME	20-JUL-1999
PERIOD TYPE LOCAL DATE TIME	20-JUL-2000

The time series will begin on 20 July 1999 (the least recent point) and run until 20 July 2000 (the most recent point). The Content Gateway will reject time series requests if the specified Periods in the list are not in chronological order (oldest time is first in list).

#### Use now to get a series ending at the current time

PeriodList:

PeriodType	Value
PERIOD TYPE LOCAL DATE TIME	1-JUN-2003 12:45
PERIOD TYPE NOW	n/a

The time series will begin at 1-JUN-2003 12:45 and run until the latest point available in the database at the time the request is processed.

#### Use a count to get additional data points

PeriodList:

PeriodType	Value
PERIOD TYPE DATA POINT COUNT	50
PERIOD TYPE LOCAL DATE TIME	1-JUN-2003 12:45
PERIOD TYPE NOW	n/a

The time series will run from 50 points before the point at 1<sup>st</sup> June 2003 12:45pm until the latest available point. This kind of request can be useful when performing technical analysis on charts – for example, when calculating a 50 point moving average. To avoid gaps in the display, 50 additional points are needed at the beginning of the main data range. It is generally not possible to guess what DateTime to use to get these additional points.

This case illustrates the mixture of relative and absolute periods in the list. Because it is followed by 1-JUN-2003 12:45, the data point count is interpreted as an offset going back in time from that date time.

Relatives may not appear between two absolutes. There is no sense in the following:

PeriodType	Value
PERIOD_TYPE_LOCAL_DATE_TIME	1-MAR-2003 12:45
PERIOD_TYPE_DATA_POINT_COUNT	50
PERIOD_TYPE_LOCAL_DATE_TIME	1-JUN-2003 12:45

There may or may not be as many as 50 points between these two date times in the database – the data point count is irrelevant either way, since the data range is unambiguously established by the two absolute Periods.

#### Use minus infinity to retrieve all data

PeriodList:

PeriodType	Value
PERIOD_TYPE_INFINITY	n/a
PERIOD_TYPE_NOW	n/a

The time series will contain the entire series held in the database.

#### Use plus infinity to request coherent real-time updates

PeriodList:

PeriodType	Value
PERIOD_TYPE_TRADING_DAY_COUNT	1
PERIOD_TYPE_INFINITY	n/a

The time series will contain all points from one trading day ago until now. In addition, a subscription will be made for record updates on the specified symbol, and the Content Gateway guarantees that the updates are coherent with the end of the returned time series. Please note: the update feature is not implemented yet. To get real-time updates, please subscribe to the real-time record as discussed in Section 5.7 of this document.

#### Use a specific data point address

PeriodList:

PeriodType	Value
PERIOD_TYPE_DATA_POINT_ADDRESS	<address returned by TimeSeriesCookie>
PERIOD_TYPE_DATA_POINT_COUNT	100

The data point address of the last point in a series is available from the TimeSeriesCookie object. The above request is a convenient way to request an additional 100 points of data starting precisely where a previous request left off.

The best way to become familiar with the capabilities of the period list is to make some trial requests using the ActivContentGatewayApiSample program.

## ResponseParameters

The time series response parameters class has a single member m\_fieldSetList, which is an stl vector of HistoryBar, IntradayBar or Tick objects depending on the request made.

These objects derive from a common base class called FieldSet. FieldSet supports the basic access functions GetField, SetField and GetType which allow the derived types to be treated in a polymorphic way. Normally, however, the accessor methods of the derived type should be used for efficiency and flexibility.

The following section gives a brief overview of the data available in each object.

### *HistoryBar*

The HistoryBar class is used to store daily, weekly, and monthly data points. Fields supported include date, open, high, low, close, total volume and VWAP fields (total value, total price, and tick count).

All historical data available through the Content Gateway has split factors applied. Future versions of the Content Platform and Feed API may represent aggregate dividend and split adjustment fields in the bar, instead of adjusting the closing price.

Depending on request parameters, some fields may not be populated – specifically, the VWAP related fields are not transmitted if the HISTORY\_FIELD\_FILTER\_TYPE\_MINI\_BAR filter is specified. Unused fields are not transmitted over the network.

Type	Field	Comment
Date	m_date	Date of bar.
Rational	m_open	Open.
Rational	m_high	High.
Rational	m_low	Low.
Rational	m_close	Close.
Rational	m_totalPrice	Sum of prices in bar.
Rational	m_totalValue	Sum of (price * size).
Rational	m_totalVolume	Total volume.
uint32_t	m_tickCount	Number of ticks in bar.

### *IntradayBar*

Pre-built Intraday Bars are constructed by the Content Server from tick data for various time intervals.

Type	Field	Comment
------	-------	---------

DateTime	m_dateTime	Date and time of bar.
Rational	m_open	Open.
Rational	m_high	High.
Rational	m_low	Low.
Rational	m_close	Close.
Rational	m_totalPrice	Sum of prices in bar.
Rational	m_totalValue	Sum of (price * size).
Rational	m_totalVolume	Total volume.
uint32_t	m_tickCount	Number of ticks in this bar.
byte_t	m_flag	Trade type code.

## ***Tick***

The Tick class represent a single trade or quote made on an instrument.

Type	Field	Comment
byte_t[4]	m_conditions	Trade or quote conditions.
DateTime	m_dateTime	Date and time of tick.
Rational	m_price	Price.
Rational	m_size	Size.
std::string	m_exchange	Exchange code.
TickType	m_tickType	Tick Type, see below.
Feed::UpdateId	m_updateId	Update Id, see below.

Valid tick types are:

TICK_TYPE_NONE	Hidden tick.
TICK_TYPE_TRADE	Trade.
TICK_TYPE_CORRECTED_TRADE	Trade correction.
TICK_TYPE_DELETED_TRADE	Trade delete.
TICK_TYPE_FILTERED_TRADE	Trade price is marked suspect.
TICK_TYPE_FORM_T_TRADE	Form T trade.
TICK_TYPE_BID	Bid.
TICK_TYPE_ASK	Ask.

## **5.11 Symbol Directory Requests**

The Symbol Directory is a sub-process of the Content Gateway; it keeps track of all instruments and certain fields which can be searched upon. Currently these fields are FID\_NAME (311) and FID\_LOCAL\_CODE (301). More fields will be added in due time.

The symbol directory API offers functions for searching instrument names and local codes to get matching symbols. There is also an option to filter the matching symbols on entity types.

## **Request Types**

The following interactive requests return symbol directory data:

- GetSymbols

The GetSymbols is available in synchronous, asynchronous and non-blocking synchronous modes.

## Helper Classes

As is the case for record requests, there is a helper class for the symbol directory request type, and this is implemented as a nested class of ContentGatewayClient. The helpers make correctly typed RequestParameters and ResponseParameters objects available and allow requests to be initiated using any of the available request modes. They also provide decoders for asynchronous responses.

## Overrides

The ContentGatewayClient class provides the following virtual method which may be overridden to receive asynchronous symbol directory responses:

```
virtual void OnGetSymbolsResponse(HeapMessage &response);
```

## Example Symbol Directory Requests

Here are some examples of typical symbol directory requests which demonstrate the available request types and the role of the helper classes.

### *Synchronous Request Mode*

See DoSynchronousGetSymbolsRequest() in DocumentationSample.

### *Asynchronous Request Mode*

See DoAsynchronousGetSymbolsRequest() and OnGetSymbolsResponse() in DocumentationSample.

### *Non-Blocking Synchronous Request Mode*

See DoNonBlockingSynchronousGetSymbolsRequest() in DocumentationSample.

## RequestParameters

The symbol directory request parameters format has the following structure:

m_fieldId	The field id to search on
m_searchString	The string to search with
m_filterType	Type of request
m_matchType	Whether to search for partial matches or exact matches
m_entityTypeList	List of entity types if entity type filter request

Currently the only fields available to search on are FID\_NAME (311) and FID\_LOCAL\_CODE (301). Other fields to search on will be added, such as FID\_ISIN (293).

When searching for a name, the symbol directory tries to match the string against all words in the name, and a trailing wildcard character ("\*") is not required. When searching for a local code, the symbol directory tries an exact match on the local code.

`m_filterType` can be `FILTER_TYPE_FULL`, `FILTER_TYPE_ENTITY_TYPE_INCLUDE_LIST` or `FILTER_TYPE_ENTITY_TYPE_EXCLUDE_LIST`.

`FILTER_TYPE_FULL` will perform no filtering on the results and is the default, `FILTER_TYPE_ENTITY_TYPE_INCLUDE_LIST` will only include entity types in the list and `FILTER_TYPE_ENTITY_TYPE_EXCLUDE_LIST` will exclude entity types in the list from the response.

`m_matchType` can be `MATCH_TYPE_ANY` to perform partial matches on the provided string, or `MATCH_TYPE_EXACT` for an exact match. The default is `MATCH_TYPE_ANY`.

## ResponseParameters

The symbol directory response parameters format has the following structure:

<code>m_fieldId</code>	The field id to search on
<code>m_symbolResponseList</code>	List of <code>SymbolResponse</code> objects

The `SymbolResponse` object represents one instrument that matched the request and has the following structure:

<code>m_symbolId</code>	The matching symbol
<code>m_flags</code>	Response flags
<code>m_matchData</code>	The entire field which had a successful match
<code>m_entityType</code>	The entity type of the instrument

The symbol can be used in subsequent calls to the Content Gateway, such as `GetEqual`.

Flags can be `FLAG_NONE` or `FLAG_SAME_AS_LAST`. They are used internally and can be safely ignored.

The match data will be the entire field, specified by the field id in the response parameters (i.e. Name or local code currently), which matched the request.

## 5.12 News Server Requests

The News Server stores all news stories for up to 12 months allowing the user to perform searching using various parameters. Access to the news server is via the Content Gateway, using the `ActivContentGateway` API.

The news server API offers functions for searching for news stories via word matching with logical operators, symbol matching and date, magazine, category and supplier filtering. You can also specify which fields of the news story you would like returned, the number of news stories returned, and which symbol to start the searching from (for a follow up, get next type request). You can also subscribe to a news query so subsequent news stories which match your query will be received via the `OnNewsUpdate()` callback.

## Request Types

The following interactive requests return news server data:

- GetNewsStories

The GetNewsStories request is available in synchronous, asynchronous and non-blocking synchronous modes.

## Helper Classes

As is the case for record requests, there is a helper class for the news server request type, and this is implemented as a nested class of ContentGatewayClient. The helper makes correctly typed RequestParameters and ResponseParameters objects available and allows requests to be initiated using any of the available request modes. It also provides a decoder for asynchronous responses.

## Overrides

The ContentGatewayClient class provides the following virtual methods which may be overridden to receive asynchronous news server responses or updates for a subscription request:

```
virtual void OnGetNewsStoriesResponse(HeapMessage &response);
virtual void OnNewsUpdate(HeapMessage &response);
```

## Example News Server Requests

Here are some examples of typical news server requests which demonstrate the available request types and the role of the helper classes.

### *Synchronous Request Mode*

See DoSynchronousNewsServerRequest() in DocumentationSample.

### *Asynchronous Request Mode*

See DoAsynchronousNewsServerRequest() and OnGetNewsStoriesResponse() in DocumentationSample.

### *Non-Blocking Synchronous Request Mode*

See DoNonBlockingSynchronousNewsServerRequest() in DocumentationSample. This example also shows how to subscribe to new stories matching the query; see OnNewsUpdate() for an example of processing these updates.

## RequestParameters

The news server request parameters format has the following structure:

m_query	The query
m_startDate	The start date of the search
m_endDate	The end date of the search
m_startSymbol	The symbol to start the search from
m_numberOfRecords	The number of records to return
m_maxRecordsPerResponse	Maximum number of records in each response
m_fieldIdList	The field id list to return
m_flags	FLAG_NONE - Nominal request

	FLAG_SUBSCRIBE - Subscribe to this query
m_permissionLevel	Requested permission level

m\_query is the query to specify what stories you wish to be returned. It can contain tags and operators for those tags. The operators are 'AND OR NOT ()', e.g.

'body=(microsoft OR apple) AND NOT ibm'. Note that the NOT operators can only be ANDed. To retrieve every story, use \* (an asterisk). Tags are specified as: tag=value. Valid tags are:

body	Query a news story body.
cat	Query the category code.
head	Query the headline.
magazine	The magazine to search on.
newssymbol	The actual symbol of a story.
supplier	The supplier of news stories to search on.
symbol	A stock symbol referenced in a story.

Here are some example queries:

- cat=industrial AND symbol=IBM.
- magazine=(DJDN or BIZ) AND symbol=MSFT. AND body=report
- body=(apple AND intel)
- head=yahoo
- newssymbol=241b3044%BIZ
- supplier=comtex

m\_startSymbol is used to specify where to start the search from and should be populated with the last symbol received from a previous request (see below). All other fields in the request should be the same as the initial request, this is to approximate a GetNext type request.

The news server response parameters format has the following structure:

m_subscriptionCookie	Subscription cookie.
m_storyList	List of returned stories.

m\_subscriptionCookie will be populated if the request had the subscription flag set; it can be used to unsubscribe from the query using the Unsubscribe() method. It is also used in news updates to identify which query the update matched (see below).

Each entry in m\_storyList represents one news story that matched the query and has the following structure:

m_symbol	The news story symbol
m_permissionIdList	List of permissions for this story
m_fieldData	The field data of the story



`m_symbol` can be used in subsequent calls to retrieve more fields, or used in `m_startSymbol`.

The news update format has the following structure:

<code>m_flags</code>	Update flags
<code>m_newsStory</code>	The news story
<code>m_updateId</code>	Update id
<code>m_subscriptionCookieList</code>	List of subscription cookies this news story matches

`m_newsStory` has the same structure as each entry in `m_storyList` above.  
`m_subscriptionCookieList` will contain a subscription cookie for each request you made that this news story matches. This can be used to work out which news stories match which query, by matching against the subscription cookies returned in the response (see above).

## News Story Specific Fields

- `FID_LANGUAGE` - the language of the news story e.g. 'en', 'fr', 'en-us'.
- `FID_CHARACTER_SET` - the character set of the news story that should be used to display the story e.g. 'ISO-8859-1'.
- `FID_PREVIOUS_NEWS_SYMBOL`, `FID_NEXT_NEWS_SYMBOL` - for those sources that allow news stories to be linked together these two fields refer to the previous and next symbol. This is usually to provide further updates to a breaking story or to shorten larger stories. (Presently only supported by Dow Jones.)
- `FID_NEWS_STATE` - contains state information about the news story (see `Enumerations.h`). The `NEWS_STATE_HOT` indicator is used to identify an important story, like a news flash. The `NEWS_STATE_TEMPORARY` indicator is for stories that should not be archived. The `NEWS_STATE_DISABLE_NEWS_ALERT` indicator is for stories that should not create a news alert, this is for less important stories.
- `FID_EXPIRATION_DATE` - is the date the story should be removed, and only applies to temporary news stories.

## 5.13 Metadata Requests

Using the `ActivContentGateway` API, you can access metadata about the feed itself. These metadata methods are available in the `MetaData` nested class of `ContentGatewayClient`.

Currently supported are the following:

- Getting a list of the permission ids your user id can access, and the fields you have access to within each permission id: `GetPermissionInfo()`
- A list of all tables that data can be requested from: `GetTableInfoList()`
- A detailed specification of a particular table containing a list of the fields in that table, along with the type of each field (unsigned int, rational, etc.): `GetTableSpecification()`

- A list of all navigational relationships that are available, or those that are available for a particular source context: `GetRelationshipInfoList()`
- A list of all the field ids currently available on the feed, along with a description of the field, its type and the C++ name for the field: `GetUniversalFieldHelperMap()`
- Information about a particular field id: `GetUniversalFieldHelper()`

As usual, the `ActivContentGatewayApiSample` demonstrates each of these methods. Additionally, the `MetaDataSample` can use the metadata methods to generate up-to-date copies of the `FieldIds.h`, `RelationshipIds.h` and `TableNumbers.h` header files. This may be useful when new content is added to the feed and you do not wish to hardcode any new table numbers or field ids in your code.

Note that only synchronous mode is supported for metadata requests so as such they should not be called regularly or in time critical code. The intention is they would be used on startup, or maybe once after a successful login.

An exception to this is the field metadata methods, which cache the field information locally in your `ContentGatewayClient` once retrieved from the `ContentGateway`. In this case, it is safe to use the field metadata APIs as a replacement for the `FieldIds.h` interface (which is static and can only return useful information about fields known at the time the `ActivContentGateway` API was compiled).

## 5.14 Important Auxiliary Classes and Types in the `ActivContentGateway` API

Here is a short description of some other important auxiliary classes and types in the `ActivContentGateway` API.

### ***Feed::TableNo***

```
typedef uint16_t TableNo;
```

Identifies a market data table or time series data table within the content platform.

### ***Feed::FieldId***

```
typedef uint16_t FieldId;
```

`FieldId` identifies a field in an `Activ Content Server` data table.

### ***Feed::RelationshipId***

```
typedef uint16_t RelationshipId;
```

Identifies a navigation relationship.

### ***Feed::PermissionId***

```
typedef uint16_t PermissionId;
```

Identifies the permission group a record is associated with. For example, `PERMISSION_ID_NASDAQ_LEVEL1`.

### ***Feed::Context***

```
typedef uint16_t SymbolContext;
```

Context Identifiers are used by the Content Gateway to assist with navigation lookups, and may be regarded as opaque by the application programmer. Each symbol in the data universe belongs to exactly one context, which is represented by a context Id.

***ContentGatewayApi::SymbolId***

SymbolId represents a *fully qualified key*. 'Fully qualified' means that the correct table number is present.

Table number is the market data table where the specified symbol resides. Responses always contain a valid value for table number, but it is not mandatory to specify one when you make a request.

## **6.0 References**

[1] `ActivContentPlatformOverview.pdf`. Find this in the `docs/ActivContentPlatform` directory of your SDK.

[2] Discussion Paper - In Search of Unique Instrument Identifier  
Szeto, Iman et al.  
Reference Data User Group (RDUG)  
&  
Reference Data Coalition (REDAC).  
June 2003

[3] The C++ Standard Library. A Tutorial and Reference.  
Josuttis, Nicolai M.  
Addison-Wesley 1999.  
ISBN 0-201-37926-0

## Appendix 1 - Example RequestBlock and ResponseBlock usage

Here are some further examples which illustrate the use of request and response block. Note not all members of the RequestBlock and ResponseBlock classes are shown; only those of interest.

1. A simple GetEqual request for “MSFT.Q” for fields ASK and BID.

Requested Key : MSFT.Q

RequestBlock:

m_flags	FLAG_NONE
m_relationshipId	RELATIONSHIP_ID_NONE
m_fieldIdList	FID_ASK, FID_BID

ResponseBlock:

m_status	STATUS_SUCCESS
m_resolvedKey	MSFT.Q
m_responseKey	MSFT.Q
m_relationshipId	RELATIONSHIP_ID_NONE
m_fieldData	26.89, 26.91

This is a canonical request, so the resolved keys and response keys are equal and the relationship used is RELATIONSHIP\_ID\_NONE. There is only one response block, matching the single canonical request block.

2. An aliased (navigation-based) GetEqual request for the close price of each Dow Jones Industrial Average constituent:

Requested Key: =DJI.DJ

RequestBlock:

m_flags	FLAG_NONE
m_relationshipId	RELATIONSHIP_ID_LISTING
m_fieldIdList	FID_CLOSE

ResponseBlock1:

m_status	STATUS_SUCCESS
m_resolvedKey	=DJI.DJ
m_responseKey	MMM.N
m_relationshipId	RELATIONSHIP_ID_LISTING
m_fieldData	141.4

ResponseBlock2:

m_status	STATUS_SUCCESS
m_resolvedKey	=DJI.DJ
m_responseKey	AA.N
m_relationshipId	RELATIONSHIP_ID_LISTING
m_fieldData	26.71

**ResponseBlock3:**

m_status	STATUS SUCCESS
m_resolvedKey	=DJI.DJ
m_responseKey	MO.N
m_relationshipId	RELATIONSHIP_ID LISTING
m_fieldData	40.99

... and so on for the remaining 27 constituents.

This was an aliased request, so the request and response keys do not match.

3. A canonical GetMultiplePatternMatch request for latest trade prices for anything matching “MSFT”.

Requested Pattern: MSFT\*

**RequestBlock:**

m_flags	FLAG NONE
m_relationshipId	RELATIONSHIP_ID NONE
m_fieldIdList	FID TRADE

**ResponseBlock1:**

m_status	STATUS SUCCESS
m_resolvedKey	MSFT.
m_responseKey	MSFT.
m_relationshipId	RELATIONSHIP_ID NONE
m_fieldData	26.89

**ResponseBlock2:**

m_status	STATUS SUCCESS
m_resolvedKey	MSFT.A
m_responseKey	MSFT.A
m_relationshipId	RELATIONSHIP_ID NONE
m_fieldData	26.87

**ResponseBlock3:**

m_status	STATUS SUCCESS
m_resolvedKey	MSFT.C
m_responseKey	MSFT.C
m_relationshipId	RELATIONSHIP_ID NONE
m_fieldData	26.89

... and so on for all further symbols that begin with “MSFT”.

This was a canonical request, but the request specified (indirectly) multiple symbols, so there are multiple response blocks. The response keys together indicate the list of matches made by the Content Gateway for the “MSFT\*” pattern.

4. An aliased GetMultipleEqual request for the current yield of the primary Microsoft Corp. and 3M Corp. securities traded on NASDAQ and NYSE respectively.

Requested Key 1: MSFT.Q

Requested Key 2: MMM.N

RequestBlock:

m_flags	FLAG_NONE
m_relationshipId	RELATIONSHIP_ID_SECURITY
m_fieldIdList	FID_CURRENT_YIELD

ResponseBlock1:

m_status	STATUS_SUCCESS
m_resolvedKey	MSFT.Q
m_responseKey	MSFT.SEC
m_relationshipId	RELATIONSHIP_ID_SECURITY
m_fieldData	0.31

ResponseBlock2:

m_status	STATUS_SUCCESS
m_resolvedKey	MMM.N
m_responseKey	MMM.SEC
m_relationshipId	RELATIONSHIP_ID_SECURITY
m_fieldData	1.89

(The security keys in the results above are fictitious values used only for the purposes of example.)

There are two response blocks. This is because there were two symbols in the request but only one request block, and the navigation link used was one-to-one.

The `m_resolvedKey` value allows us to relate the response blocks back to the requested symbols.

An excellent way to become familiar with the response format is to try out some sample requests using the `ActivContentGatewayApiSample` program, which ships as part of the `ActivFeed API SDK`.

## **Appendix 2 - Symbolology Reference**

Exchange provided ticker symbols are always part of the symbol and are typically used without alteration. Generally, a LocalCode field will also be provided in the database record that only contains the exchange provided ticker symbol. Most symbols contain the exchange delimiter ‘.’ (dot) followed by the exchange code (see section A at the end of this document). The exchange portion of the symbol need not be at the end of the symbol (e.g. Market Market Quotes).

### **A2.1 Equities**

<Issue Symbol>.<Exchange Code>

Examples: IBM.N, MSFT.Q

Composite U.S. equity symbols have no exchange code, but end in a period:

Examples: IBM.

### **A2.2 Equity Options**

#### **Option Roots**

<Root Symbol>.<Exchange Code>

Examples: MSQ.OCC

#### **Option Deliverables**

<Root Symbol>#<Index Number>.<Exchange Code>

The index number is present only to ensure uniqueness of the symbol in cases where multiple deliverable records are associated with an option root. Exchange code is the same as that of the associated option root symbol.

Examples: MSQ#1.OCC

#### **Option Contracts**

<Root Symbol>/<Month Code>/<Strike Price Code>.<Exchange Code>

Examples: IBM/A/A.AO, MSQ/A/A.XO



### **A2.3 Futures**

<Root Symbol>/<Year><Month Code>.<Exchange Code>

Examples: C/08H.CB, LC/08Z.CM

### **A2.4 Future Options**

<Root Symbol>/<Year><Month Code>/<Strike Price ><Put/Call>.<Exchange Code>

Examples: C/8G/24000C.CB, LH/8G/42000P.CM

### **A2.5 Future Spreads**

<Root Symbol>/<Year1><Month Code1>—<Year2><Month Code2>.<Exchange Code>

Examples: C/08H-08K.CB, LH/08J-08K.CM

### **A2.6 Market Makers**

<Underlying Symbol>:<Market Maker Id>

Examples: MSFT.Q:LEHM, INTC.Q:ISLD

### **A2.7 Order Book**

<Underlying Symbol>;<Order Id>

### **A2.8 Exchange Traded Funds**

< Underlying Symbol >.<Exchange Code>

Examples: NIR.A, ONEQI.Q

## **A2.9 Forex**

### **Spots**

<Base Currency><Value Currency>.<Exchange Code>

Examples: USDGBP.TF, AUDEUR.TF

### **Spot quotes**

<Base Currency><Value Currency>.<Exchange Code>:<Market Maker Id>

Examples: USDGBP.TF:BARC, CADDKK.TF:DRES

### **Forwards**

<Base Currency><Value Currency>/<Forward Code>.<Exchange Code>

where Forward Codes is one of:

ON - Overnight

SN - Spot Next

TN - Tomorrow Next

<Number><Duration> where <Duration> can be one of:

W - Weeks

M - Months

Y - Years

Examples: USDGBP/1W.TF, CHFEUR/2Y.TF

### **Forward quotes**

<Base Currency><Value Currency>/<Forward Code>.<Exchange Code>:<Market Maker Id>

Examples: NZDHKD/TN.TF:UBSW

## **A2.10 Index**

=< Issue Symbol>.<Exchange Code>

Examples: =COMP.Q, =XAX.A

## **A2.11 Mutual Fund**

<Fund Id>.<Exchange Code>

Examples: BMCGX.Q, JAMFX.Q

## **A2.12 Money Market**

<Fund Id>.<Exchange Code>

Examples: AALXX.Q, ITTXX.Q

## **A2.13 Rankings**

CUMVAL/ACT+<Exchange Code>

Contains the top 20 most active equities based on cumulative value.

CUMVOL/ACT+<Exchange Code>

Contains the top 20 most active equities based on cumulative volume.

PERCENT/ADV+<Exchange Code>

Contains the top 20 advancing equities based on percent change.

PERCENT/DEC+<Exchange Code>

Contains the top 20 declining equities based on percent change.

Example: CUMVAL/ACT+Q, CUMVOL/ACT+Q, PERCENT/ADV+Q,  
PERCENT/DEC+Q

## **A2.14 Exchange Statistics**

ISSUES/DAY+<Exchange Code>

Contains the number of equities that are up/down/unchanged on the day.

ISSUES/TICK+<Exchange Code>

Contains the number of equities that are up/down on the last tick.

CUMVAL/DAY+<Exchange Code>

Contains the cumulative value of equities that are up/down/unchanged on the day.

CUMVOL/DAY+<Exchange Code>

Contains the cumulative volume of equities that are up/down/unchanged on the day.

Examples: ISSUES/DAY+Q, ISSUES/TICK+Q, CUMVAL/DAY+Q,  
CUMVOL/DAY+Q

TICK+<Exchange Code>

Number of advancing equities - Number of declining equities.

TRIN+<Exchange Code>

(Number of advancing equities / Number of declining equities) / (Advancing cumulative  
volume / Declining cumulative volume).

Examples: TICK+Q, TRIN+Q

## **A2.15 News Stories**

< News ID > /HL% <Magazine>

Example: 090b5625/HL%BIZ

### **A2.15 News Story**

< News ID > % <Magazine>

Example: 090b5625%BIZ

### ***Appendix 3 - Exchange List***

<b>Exchange Name</b>	<b>Exchange Code</b>
AMEX	A
AMEX Options	AO
BSE	B
BOX	BO
Bovespa	BS
NSX	C
CBOT	CB
CBOT Globex	CBE
Instinet	CBX
CBOE Futures Exchange	CF
CME	CM
CME Globex	CMG
CNQ Exchange	CQ
COMEX	CX
COMEX Globex	CXA
DME	DME
NYSE Arca ECN	EA
BATS	EB
INET ECN	EI
Eurex	EX
FSE	FU
HKFE	HF
HKEX	HK
ICE	ICE
JASDAQ	JQ
KCBT	KC
KCBOT Globex	KCE
LSE	L
LIFFE	LI
LME	LM
Montreal Exchange	M

MGEX	MG
MGEX Globex	MGE
Chicago Stock Exchange	MW
NYSE	N
ICE	NB
ICE Electronic	NBE
NSE	NG
NYSE Best Quotes	NQ
NYMEX	NX
NYMEX Clearport	NXC
NYMEX Europe Electronic	NXE
NYMEX Globex	NXG
NYMEX Europe	NXL
US Options Composite	O
NYSE Open Book	OB
One Chicago	OC
OSE	OS
Euronext Paris	P
NYSE Arca	PA
NYSE Arca Options	PO
Pink Sheets	PS
Pure Trading	PT
NASDAQ	Q
NASDAQ - Bulletin Board	QB
NASDAQ - ADF	QD
NASDAQ OTC	QO
OTCQX	QX
SGX	SG
SSE	SP
TSE	T
TOCOM	TC
Tenfore	TF
TSX	TO

TSX Venture	TV
US Composite	US
CBSX	W
WCE	WC
PHLX	X
PBOT	XB
Eurex	XEUR
PHLX Options	XO
ISE	Y

This exchange list is subject to change, the latest exchange list can be extracted from TABLE\_NO\_EXCHANGE (37). An example of how to do this using one of the ContentGateway API samples is shown below:

```
SnapshotViewer_x86_win32_vc71_mss.exe -u username -p password -t 37 -f
456;311 -o exchanges.txt
```

## **Appendix 4 - Event Types**

This appendix describes each of the event types used on ACTIV Feed and the fields associated with each event. The definition of event type can be found in the header file `include/ActivContentPlatform/ActivFeedCommon/EventTypes.h` (included as part of the ActivFeed API SDK), together with details of all allowed values for this field. Every update message disseminated on ACTIV Feed includes an event type. Some of the events have a specific set of fields associated with them which will always be included in a message. However, messages with these event types may also contain other fields besides the event specific set. Additionally, DCSes can be configured such that *only* the changed fields are forwarded (i.e. there would be no guarantee for a given event type that field N is always present).

### **A4.1 EVENT\_TYPE\_NONE**

Used when there is no specific event associated with the message.

Fields always present:

None.

### **A4.2 EVENT\_TYPE\_TRADE**

Indicates that a trade has occurred.

Fields always present:

FID\_TRADE

FID\_TRADE\_CONDITION

FID\_TRADE\_DATE

FID\_TRADE\_EXCHANGE (only present for composite symbols)

FID\_TRADE\_SIZE

FID\_TRADE\_TIME

The following fields are also always present if they exist in the table that was updated by the message:

FID\_TRADE\_BUYER\_ID

FID\_TRADE\_SELLER\_ID

### **A4.3 EVENT\_TYPE\_TICK**

Tick events are used when the trade fields associated with an instrument are updated at fixed time intervals. A good example of an instrument type that would have tick events rather than trade events is index.

Fields always present:



FID\_TRADE  
FID\_TRADE\_CONDITION  
FID\_TRADE\_DATE  
FID\_TRADE\_SIZE  
FID\_TRADE\_TIME

#### **A4.4 EVENT\_TYPE\_TRADE\_CORRECTION**

Indicates that a previous trade was reported incorrectly and includes the corrected values for the fields.

Fields always present:

FID\_TRADE  
FID\_TRADE\_CONDITION  
FID\_TRADE\_DATE  
FID\_TRADE\_SIZE  
FID\_TRADE\_EXCHANGE (only present for composite symbols)  
FID\_TRADE\_TIME

#### **A4.5 EVENT\_TYPE\_TRADE\_CANCEL**

Indicates that a previously reported trade has been cancelled.

Fields always present:

FID\_TRADE  
FID\_TRADE\_CONDITION  
FID\_TRADE\_DATE  
FID\_TRADE\_EXCHANGE (only present for composite symbols)  
FID\_TRADE\_SIZE  
FID\_TRADE\_TIME

#### **A4.6 EVENT\_TYPE\_TRADE\_NON\_REGULAR**

Indicates that the information contained in the message is for a non-regular trade. An example of a non-regular trade would be a late trade. This event type was formerly called the now deprecated EVENT\_TYPE\_TRADE\_LATE.

Fields always present:

FID\_TRADE  
FID\_TRADE\_CONDITION  
FID\_TRADE\_DATE  
FID\_TRADE\_EXCHANGE (only present for composite symbols)

FID\_TRADE\_SIZE  
FID\_TRADE\_TIME

In the case of late reported trades, the update message will include rules information for each of these fields that indicates that the value received in the message should not overwrite any cached (i.e. latest) value.

#### **A4.7 EVENT\_TYPE\_BBO\_QUOTE**

Indicates that the message contains best bid/offer information.

Fields always present:

If any of the bid fields have changed as a result of an update then all of the following fields will be present in the resulting message:

FID\_BID  
FID\_BID\_CONDITION  
FID\_BID\_EXCHANGE (only present for composite symbols)  
FID\_BID\_SIZE  
FID\_BID\_TIME

If any of the ask fields have changed as a result of an update then all of the following fields will be present in the resulting message:

FID\_ASK  
FID\_ASK\_CONDITION  
FID\_ASK\_EXCHANGE (only present for composite symbols)  
FID\_ASK\_SIZE  
FID\_ASK\_TIME

If either a bid or an ask field has changed then the following fields will also be included in the message:

FID\_QUOTE\_DATE

#### **A4.8 EVENT\_TYPE\_QUOTE**

Indicates that the message contains the latest bid and ask information for a particular market maker.

Fields always present:

Exactly the same as detailed in EVENT\_TYPE\_BBO\_QUOTE.

#### **A4.9 EVENT\_TYPE\_COMPOSITE\_BBO\_QUOTE**

Used internally in the system between the Upstream and Downstream content servers. This event type is forwarded by the Downstream Content Server as an EVENT\_TYPE\_BBO\_QUOTE message and is processed in exactly the same way.

Fields always present:

Exactly the same as detailed in EVENT\_TYPE\_BBO\_QUOTE.

#### **A4.10 EVENT\_TYPE\_CLOSING\_QUOTE**

Used internally in the system between the Upstream and Downstream Content Servers to indicate that a market maker has stopped quoting for the day. It also includes the last values for the quote fields. This event type is forwarded by the Downstream Content Server as an EVENT\_TYPE\_QUOTE message.

Fields always present:

Exactly the same as detailed in EVENT\_TYPE\_BBO\_QUOTE.

On receipt of this event type, the Downstream Content Server will indicate that the quote has closed by setting the STATE\_BIT\_CLOSED (see Enumerations.h) in the FID\_STATE field (field id 361). It will also set the following fields:

FID\_CLOSING\_BID  
FID\_CLOSING\_ASK  
FID\_CLOSING\_QUOTE\_DATE.

If this results in a change to the value of one of these fields, it will also be included in the update message.

#### **A4.11 EVENT\_TYPE\_CLOSING\_BBO\_QUOTE**

Used internally in the system between the Upstream and Downstream Content Servers to indicate that an instrument has stopped quoting for the day. It also includes the last values for the quote fields. This event type is forwarded by the Downstream Content Server as an EVENT\_TYPE\_BBO\_QUOTE message.

Fields always present:

Exactly the same as detailed in EVENT\_TYPE\_BBO\_QUOTE.

On receipt of this event type, the Downstream Content Server will set the following fields:

FID\_CLOSING\_BID

FID\_CLOSING\_BID\_EXCHANGE (only present for composite symbols)  
FID\_CLOSING\_ASK  
FID\_CLOSING\_ASK\_EXCHANGE (only present for composite symbols)  
FID\_CLOSING\_QUOTE\_DATE.

If this results in a change to the value of one of these fields, it will also be included in the update message.

#### **A4.12 EVENT\_TYPE\_OPEN**

Indicates that the instrument has opened.

Fields always present:

None.

On receipt of this event type, the Downstream Content Server will clear the Closed bit in the FID\_STATE field to indicate that the instrument is now open.

See Enumerations.h for definitions of the State field.

#### **A4.13 EVENT\_TYPE\_CLOSE**

Indicates that the instrument has closed.

Fields always present:

None.

On receipt of this event type, the Downstream Content Server will set the following fields:

FID\_CLOSE  
FID\_CLOSE\_DATE  
FID\_CLOSE\_EXCHANGE (only present for composite symbols)  
FID\_CLOSE\_STATUS

The following fields will also be set but are only applicable for certain instrument types:

FID\_CLOSE\_CONDITION  
FID\_CLOSE\_CUMULATIVE\_VALUE  
FID\_CLOSE\_CUMULATIVE\_VALUE\_STATUS  
FID\_CLOSE\_CUMULATIVE\_VOLUME  
FID\_CLOSE\_CUMULATIVE\_VOLUME\_DATE  
FID\_CLOSE\_CUMULATIVE\_VOLUME\_STATUS

If this results in a change to the value of one of these fields, it will also be included in the update message.

The Downstream Content Server will also set the Closed bit in the FID\_STATE field to indicate that the instrument is now closed.

See Enumerations.h for details on how to interpret the Status fields and for definitions of the State field.

#### **A4.14 EVENT\_TYPE\_RESET**

Used to initiate the archiving and clear down of certain fields in readiness for the next trading day.

Fields always present:

FID\_RESET\_DATE

On receipt of this event type, the Downstream Content Server will set the following fields:

FID\_PREVIOUS\_ASK  
FID\_PREVIOUS\_BID  
FID\_PREVIOUS\_QUOTE\_DATE  
FID\_PREVIOUS\_CLOSE  
FID\_PREVIOUS\_CLOSE\_DATE  
FID\_PREVIOUS\_CUMULATIVE\_PRICE  
FID\_PREVIOUS\_CUMULATIVE\_VALUE  
FID\_PREVIOUS\_CUMULATIVE\_VOLUME  
FID\_PREVIOUS\_CUMULATIVE\_VOLUME\_DATE  
FID\_PREVIOUS\_NET\_CHANGE  
FID\_PREVIOUS\_OPEN  
FID\_PREVIOUS\_PERCENT\_CHANGE  
FID\_PREVIOUS\_TRADE\_HIGH  
FID\_PREVIOUS\_TRADE\_LOW  
FID\_PREVIOUS\_TRADING\_DATE  
FID\_PREVIOUS\_OPEN\_INTEREST  
FID\_PREVIOUS\_OPEN\_INTEREST\_DATE

Note that some of these fields are only applicable for certain instrument types.

The Downstream Content Server will then clear down fields so they are ready to receive values on the next trading day. The fields that get cleared vary according to instrument type but will typically include all bid, ask, trade and volume fields.

#### **A4.15 EVENT\_TYPE\_NEWS**

Indicates that the message is a news story.

#### **A4.16 EVENT\_TYPE\_NEWS\_DELETE**

Indicates that a news story should be deleted.

#### **A4.17 EVENT\_TYPE\_PURGE**

Indicates that a record is going to be purged from the Downstream Content Server database. It does not mean that the record has to be deleted on any client systems: It is simply used as a mechanism to clear out old records from tables where only a limited history is kept, e.g. a corporate actions table.

#### **A4.18 EVENT\_TYPE\_ALERT**

Indicates that some important property of a record has changed, generally relating to the trading of the record. For example, messages with this event type are generated when an instrument enters a restricted trading state.

#### **A4.19 EVENT\_TYPE\_BBO\_DEPTH**

Indicates that any of up to 10 levels of Bid or Ask depth fields have changed. For example FID\_BID2, FID\_BID2\_COUNT, FID\_BID2\_SIZE, FID\_BID2\_TIME.

Fields always present:

None.

#### **A4.20 EVENT\_TYPE\_ORDER**

Indicates that the message contains order details.

Fields always present:

None.

#### **A4.21 EVENT\_TYPE\_HALT\_RESUME**

Indicates that the instrument has either halted trading or has resumed being traded.

Fields always present:

FID\_STATE

The FID\_STATE field indicates whether the instrument has halted or resumed. See Enumerations.h for details on how to interpret the State field.

#### **A4.22 EVENT\_TYPE\_IMBALANCE\_VOLUME**

Indicates that the instrument has an imbalance of order volume.

Fields always present:

FID\_IMBALANCE\_BUY\_VOLUME  
FID\_IMBALANCE\_SELL\_VOLUME  
FID\_IMBALANCE\_VOLUME\_TIME

#### **A4.23 EVENT\_TYPE\_PRICE\_INDICATION**

Indicates that price indications are available for the instrument.

Fields always present:

FID\_HIGH\_INDICATION\_PRICE  
FID\_LOW\_INDICATION\_PRICE  
FID\_INDICATION\_PRICE\_TIME

#### **A4.24 EVENT\_TYPE\_REFRESH**

Used internally on the upstream content servers to request a refresh of a record to be sent to all downstream content servers.

Fields always present:

None.

#### **A4.25 EVENT\_TYPE\_REFRESH\_CYCLE**

Used internally between the upstream and downstream content servers to indicate that a complete refresh of the upstream database has been sent.

Fields always present:

None.

#### **A4.26 EVENT\_TYPE\_OPTION\_REFRESH**

Used internally by the options feed handler translator devices to get a complete refresh of an options record to the content servers.

Fields always present:

None.

#### **A4.27 Other Event Types**

The following event types are used for corporate actions and are used by the Downstream Content Server to identify when adjustments need to be made to its database:

**EVENT\_TYPE\_IPO**  
**EVENT\_TYPE\_IPO\_CANCEL**  
**EVENT\_TYPE\_DELIST**  
**EVENT\_TYPE\_DELIST\_CANCEL**  
**EVENT\_TYPE\_DIVIDEND**  
**EVENT\_TYPE\_DIVIDEND\_CANCEL**  
**EVENT\_TYPE\_DIVIDEND\_CORRECTION**  
**EVENT\_TYPE\_SPLIT**  
**EVENT\_TYPE\_SPLIT\_CANCEL**  
**EVENT\_TYPE\_SPLIT\_CORRECTION**  
**EVENT\_TYPE\_SYMBOL\_CHANGE**  
**EVENT\_TYPE\_SYMBOL\_CHANGE\_CANCEL**  
**EVENT\_TYPE\_OPTION\_SYMBOL\_CHANGE**  
**EVENT\_TYPE\_OPTION\_SPLIT**  
**EVENT\_TYPE\_PRICE\_ADJUSTMENT**  
**EVENT\_TYPE\_PRICE\_ADJUSTMENT\_CANCEL**

The following event types are used internally by the system:

**EVENT\_TYPE\_TIME\_SERIES\_CORRECTION**  
**EVENT\_TYPE\_TIME\_SERIES\_DELETE**

The following event types are currently unused:

**EVENT\_TYPE\_FORCE\_PROCESS\_REFRESH**  
**EVENT\_TYPE\_CACHE\_FLUSH**



## **Appendix 5 - Magazine list**

<b>Source Name</b>	<b>Magazine</b>	<b>Supplied by</b>
ABIX Australasian Business Intelligence	ABX	Comtex
AFX News Asia	AFA	Comtex
AFX News Europe	AFE	Comtex
AFX News UK	AFK	Comtex
Asia in Focus	AIF	Comtex
Al-Bawaba	ALB	Comtex
AM Best	AMB	Comtex
A&G Information	ANG	Comtex
All Africa.com	ANS	Comtex
Associated Press Online	APO	Comtex
Asia Pulse	APU	Comtex
Alestron, Inc. (formerly AsiaPort now Sinocast)	ASI	Comtex
Associated Press WorldStream	AWS	Comtex
Business News Americas	BAM	Comtex
BBC Monitoring	BBC	Comtex
Business Wire	BIZ	Comtex
FIND -- FedBizOps	CBD	Comtex
CCN Matthews	CCN	Comtex
China IT Watch	CIT	Comtex
Canada NewsWire	CNW	Comtex
Collegiate Presswire	COL	Comtex
Canadian Press (AP)	CPW	Comtex
Christian Science Monitor	CSM	Comtex
DailyFX	DFX	Comtex
DMEurope.com	DME	Comtex
Datamonitor	DTM	Comtex
Edgar Online -- Glimpse Feed	EDG	Comtex
Edgar Online 8-K Glimpse	EDK	Comtex
EFE News	EFE	Comtex
PRNewswire Europe	EUR	Comtex
FedNet Government News	FDN	Comtex
FIND, Inc. -- Fed Register	FFR	Comtex
FIND, Inc. -- Government Info	FGI	Comtex
OsterDowJones	FWN	Comtex
OsterDowJones Select	FWS	Comtex
HUGIN, AS	HUG	Comtex

Street Insider	IDR	Comtex
Inman News Service	INM	Comtex
MarketWire (formerly Internet Wire)	INW	Comtex
IPO Monitor	IPM	Comtex
Inter Press Services	IPS	Comtex
Investrend Financial Wire	ITR	Comtex
JapanCorp.net	JCN	Comtex
JAGNotes -- Equity Analysis	JEA	Comtex
JAGNotes -- Unrestricted	JGN	Comtex
Knobias.com	KAS	Comtex
Knight Ridder News -- unrestricted	KNO	Comtex
Knight Ridder News -- restricted	KNS	Comtex
Knight-Ridder/Tribune Business News -- unrestricted	KRO	Comtex
Knight-Ridder/Tribune Business News -- restricted	KRT	Comtex
Kyodo News	KYO	Comtex
Internet Securities Inc.	LTN	Comtex
Midnight Trader	MID	Comtex
M2 - Airline Industry Information	MTA	Comtex
M2 - 10Meters.com	MTM	Comtex
M2 - Communications (multiple)	MTO	Comtex
M2 - DVD News	MTV	Comtex
Newsbytes News Network	NBY	Comtex
New World Publishing	NWP	Comtex
Access Intelligence, LLC Lite	PHL	Comtex
Access Intelligence, LLC Full	PHP	Comtex
PrimeZone Media Network	PMZ	Comtex
CompanyNewsGroup	PRL	Comtex
PRNewswire	PRN	Comtex
InterPress Service -- QNA/WAM	QNA	Comtex
Resource News International	RNI	Comtex
RosBusinessConsulting	ROS	Comtex
RWE Australian Business	RWE	Comtex
SinoCast China Financial Watch	SFW	Comtex
SinoCast China Transportation Watch	TRW	Comtex
Trade Signals	TSG	Comtex
University Wire	UNW	Comtex
United Press International	UPI	Comtex
US Newswire	USN	Comtex
The Deal -- CW Select	VCB	Comtex

Vickers Stock Research -- Insider Trading US	VKR	Comtex
FIND, Inc. - Washington Day Book	WDB	Comtex
Xinhua -- Economic News	XEC	Comtex
Xinhua Financial News	XFN	Comtex
Xinhua News Agency	XIN	Comtex
Dow Jones News Service	DJDN	Dow Jones

## **Appendix 6 - Non common stock issues**

There are several different notations from different exchanges for non common stock issues. Activ provides a uniform symbology which will be available as alias symbols in addition to the already existing exchange notations. This uniform symbology is described below:

Each ticker code can be optionally followed by any of the following three combinations:

<ticker>,<series or class>.<exchange>  
<ticker>,<suffix>.<exchange>  
<ticker>,<suffix>,<series or class>.<exchange>

Where <suffix> is a two character string representing an issue type other than common stock and <series or class> is a single character representing the class.

Note that the <suffix> and the <series or class> are preceded by a comma character.

The following suffixes are currently used:

1B	First Convertible Bond
1P	First Preferred, Same Company
2B	Second Convertible Bond
2P	Second Preferred, Same Company
3B	Third Convertible Bond
3P	Third Preferred, Same Company
4P	Fourth Preferred, Same Company
AD	American Depository Receipts
BD	Bonds
BR	Bankruptcy
CB	Convertible Bond
CC	Convertible Called
CL	Called
CV	Convertible
DB	Debentures
DQ	Delinquent In Filings
EC	Emerging company
FO	Foreign
IQ	Issuer Qualifications
IR	Installment Receipts
LV	Limited Voting Shares
MF	Mutual Funds
MP	Miscellaneous Situations (Such As Certificates Of Preferred When Issued)
MS	Miscellaneous
MV	Multiple Voting Shares

NO	Notes
NS	Notes
NT	Notes
NV	Non-Voting Shares
NW	New
PC	Preferred Called
PD	Preferred When Distributed
PI	Preferred When Issued
PP	Partial Paid
PR	Preferred Issue
PV	Preferred Convertible
RT	Rights
RV	Restricted Voting Shares
RW	Rights When Issued
SB	Shares Of Beneficial Interest
SV	Subordinate Voting Shares
TE	Test symbol
UN	Units
US	Trading In Us Dollars
VO	Voting
WA	Warrants Same Company
WD	When Distributed
WH	Warrants When Issued
WI	When Issued
WS	Warrants
WW	With Warrants Or With Rights

## **Appendix 7 – Future Aliases**

ACTIV provide a shorthand alias for futures by stripping out the expiration date separator (“/”) and the immediate zero (“0”) after that if it exists. For example:

```
A/07J.OC -> A7J.OC
A1/07J.OC -> A17J.OC
AL/10F.CX -> AL10F.CX
```

ACTIV also manually maintains a “rolling contract” alias determined by our Data Quality team based on open interest and cumulative volume. For example:

```
A/*.OC -> A/07J.OC
```

This mapping will be reviewed periodically and updated to reflect the current most active contract.

For completeness, ACTIV also provide a shorthand for the above alias by removing the expiration date separator.

```
A*.OC -> A/*.OC -> A/07J.OC
```

This can be seen in the GetMatch example below using the ActivContentGatewayApiSample:

```
# 15
Enter table number <65535>:
Enter symbol: A*
<C>omposite, <P>rimary search type <C>:
Enter relationship id <0>:
Enter field id:
Return all permissioned fields (y/n) <n>:
Enter relationship id:
Subscribe to <N>othing, <A>dds or deletes, <I>nclude Event types,
<E>xclude Event Types, <F>ull <N>:
Ignore alias (y/n) <n>:
Max response blocks in one response <64>:
Enter permission level <255>:
Send request (y/n) <y>:
Returned success using request id 13.
# 15:55:27 [ApiSample:3] MyContentGatewayClient::OnGetMatchResponse():
Request id [0d 00 00 00] - status success
15:55:27 [ApiSample:3] Subscription cookie ..... 4294967295
15:55:27 [ApiSample:3] **** Response block 1/1 ****
15:55:27 [ApiSample:3] Resolved key ..... A*.OC [600]
15:55:27 [ApiSample:3] Relationship id 0 ..... success
15:55:27 [ApiSample:3] Flags ..... none
15:55:27 [ApiSample:3] Response key ..... A/07J.OC
[16]
15:55:27 [ApiSample:3] Permission id ..... 44
15:55:27 [ApiSample:3] Permission level ..... 0
15:55:27 [ApiSample:3] No field data.
15:55:27 [ApiSample:3] MyContentGatewayClient::OnGetMatchResponse():
Request id [0d 00 00 00] now complete
```