

BLOOMBERG L.P. OPEN API

PUBLISHING DEVELOPER GUIDE

Version: 1.2
Last Updated: 07/31/2015

Related Documents

Document Name
Core User Guide
Core Developer Guide
Enterprise User Guide
Enterprise Developer Guide
Publishing User Guide
Reference Guide - Bloomberg Services and Schemas
Reference Symbology Guide

All materials including all software, equipment and documentation made available by Bloomberg are for informational purposes only. Bloomberg and its affiliates make no guarantee as to the adequacy, correctness or completeness of, and do not make any representation or warranty (whether express or implied) or accept any liability with respect to, these materials. No right, title or interest is granted in or to these materials and you agree at all times to treat these materials in a confidential manner. All materials and services provided to you by Bloomberg are governed by the terms of any applicable Bloomberg Agreement(s).

Contents

1. About This Guide.....	4
2. Publishing Overview	4
2.1. Simple Broadcast	4
2.1.1 Creating a Session.....	5
2.1.2 Obtaining Authorization.....	5
2.1.3 Creating the Topic.....	7
2.1.4 Publishing Events.....	7
2.2. Interactive Publisher	8
2.2.1. Registration	9
2.2.2. Event Handling	9
2.2.3. Publication.....	11
3. Publishing Classes.....	12
3.1. ProviderSession	12
3.1.1. Session Options/Software Failover	13
3.2. Services	13
3.2.1. Registering a Service	13
3.3. Topics and TopicList.....	14
3.4. Events and EventFormatter.....	14
3.4.1. Example Event Structure	15
3.4.2. Event Formatter.....	15
4. Local Publishing	16
4.1. Publisher Groups.....	17
4.2. Launchpad Page Publishing.....	17
5. Platform Failover	19
6. Contribution.....	20
6.1. Contribution Example Code.....	20

1. About This Guide

The Publishing Developer's Guide is designed to help understand the concept of Publishing and how Bloomberg L.P. API allows customer applications to publish data as well as consume it. Customer data can be published for distribution within the customer's enterprise, contributed to the Bloomberg infrastructure, distributed to others, or used for warehousing.

2. Publishing Overview

The Bloomberg API allows customer applications to publish data as well as consume it. Customer data can be published for distribution within the customer's enterprise, contributed to the Bloomberg infrastructure, distributed to others, or used for warehousing.

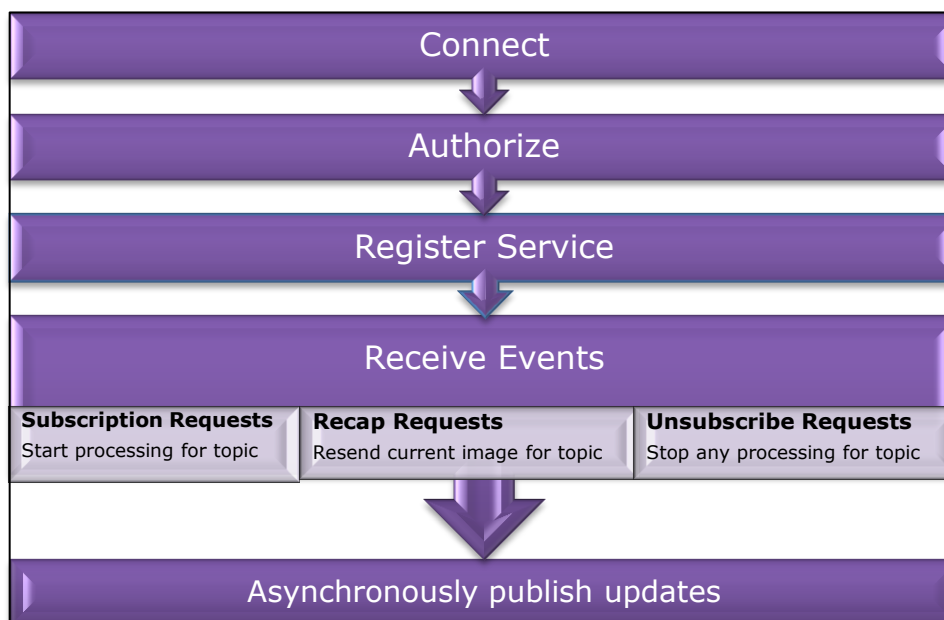


Figure 1: Publishing Flow

Publishing applications might simply “**broadcast**” data or they can be “**interactive**”, responding to feedback from the infrastructure about the currently active subscriptions from data consumers. This will illustrate both paradigms. There are two Programming Examples:

- `BroadcastOneTopic.cpp`
- `InteractivePublisher.cpp`.

2.1. Simple Broadcast

In a simple broadcast, the publishing application sends data but has no indication if anyone is consuming that data. In this simple example, data will be produced for a single topic. The major stages are:

- Creating a session
- Obtaining authorization
- Creating the topic

- Publishing events for the topic to the designated service

2.1.1 Creating a Session

Sessions for publication are created in the same manner as those for consuming data. The key difference is that they are managed by an instance of `ProviderSession` instead of `Session`.

The event handler plays no significant role in this example and will NOT be examined.

```
// BroadcastOneTopic.cpp
... ..

int main()
{
    SessionOptions sessionOptions;
    sessionOptions.setServerHost("platform");
    sessionOptions.setServerPort(8197);

    sessionOptions.setAuthenticationOptions("AuthenticationType=OS_LOGON")
    ; MyEventHandler myEventHandler;

    ProviderSession session(sessionOptions, &myEventHandler, 0);
    if (!session.start()) {
        std::cerr <<"Failed to start session." << std::endl; return 1;
    }... ..
}
```

2.1.2 Obtaining Authorization

The authorization stage, if successful, provides a valid `Identity` object which is required for later operations. Authorization is done by the `"//blp/apiauth"` service on receipt of an authorization request.

☞ For additional information on **"Authorization and Permissioning Systems"** refer to the **Enterprise User Guide**.

```
Name TOKEN("token");
Name TOKEN_SUCCESS("TokenGenerationSuccess"); Name
TOKEN_FAILURE("TokenGenerationFailure");
Name AUTHORIZATION_SUCCESS("AuthorizationSuccess");
EventQueue tokenEventQueue;
session.generateToken(CorrelationId(), &tokenEventQueue);
std::string token;
Event event = tokenEventQueue.nextEvent();
if (event.eventType() == Event::TOKEN_STATUS) {
    MessageIterator iter(event);
    while (iter.next()) {
        Message msg = iter.message();
        msg.print(std::cout);
        if (msg.messageType() == TOKEN_SUCCESS) {
            token = msg.getElementAsString(TOKEN);
        }
    }
}
```

```

        else if (msg.messageType() == TOKEN_FAILURE) {
            break;
        }
    }

    }
    if (token.length() == 0) {
        std::cout << "Failed to get token" << std::endl;
    }
    session.openService("//blp/apiauth");
    Service authService = session.getService("//blp/apiauth");
    Request authRequest =
    authService.createAuthorizationRequest();
    authRequest.set(TOKEN, token.c_str());

    EventQueue authQueue;
    Identity providerIdentity = session.createIdentity();
    session.sendAuthorizationRequest(
        authRequest, &providerIdentity, CorrelationId(), &authQueue);

else if (event.eventType() == EventType.RESPONSE
        || event.eventType() == EventType.PARTIAL_RESPONSE
        || event.eventType() == EventType.REQUEST_STATUS)
{ for (Message msg: event) {
    if
        (msg.correlationID().equals(d_authorizationResponseCorrelationId
        )) { Object authorizationResponseMonitor =
            msg.correlationID().obje
            ct(); synchronized (authorizationResponseMonitor) {
                if (msg.messageType() == AUTHORIZATION_SUCCESS) {
                    d_authorizationResponse = Boolean.TRUE;
                    authorizationResponseMonitor.notifyAll();
                }
                else if (msg.messageType() == AUTHORIZATION_FAILURE) {
                    d_authorizationResponse = Boolean.FALSE;
                    System.err.println("Not authorized: " +
                        msg.getElement("reason"));
                }
            }
            else {
                assert d_authorizationResponse == Boolean.TRUE;
                System.out.println("Permissions updated");
            }
        }
    }
}
}
}

```

2.1.3 Creating the Topic

Before publishing data, the application must create a Topic object on the appropriate service. This example uses synchronous method `createTopics()` of the `ProviderSession` to create a Topic on `//blp/test` service from a topic string `"testtopic"`.

```
... ..
const std::string myService = "//blp/test";
const std::string myTopic = "testtopic";
TopicList topicList;
topicList.add((myService + "/ticker/" + myTopic).c_str(),
    CorrelationId((long long)1));

session.createTopics(
    &topicList,
    ProviderSession::AUTO_REGISTER_SERVICES,
    providerIdentity);

Topic topic;
for (size_t i = 0; i < topicList.size(); ++i) {
    if (topicList.statusAt(i) == TopicList::CREATED) {
        topic = session.getTopic(topicList.messageAt(i));
    }
}
... ..
```

2.1.4 Publishing Events


In this example, data is published by sending events to the designated service, `//blp/test`. Event objects are obtained from the service and populated with the topic and the application specific data. In this simple example, each event contains a single data message; however, in general, each event can contain multiple messages. In this simple example, the data is just an integer value that is incremented and published every ten seconds.

```
... ..
Name messageType ("MyMessageType");
Name fieldType ("MyFieldType");

Service service = session.getService(myService.c_str());
for (int value = 1; true; ++value, sleep(10)) {
    Event event = service.createPublishEvent();
    EventFormatter eventFormatter(event);
    eventFormatter.appendMessage(messageType, topic);
    eventFormatter.setElement(fieldName, value);

    session.publish(event);
}

session.stop();
return 0;
}
```

 *Note: The standard C library 'sleep' function is used above. The argument specifies the number of seconds to sleep.*

2.2. Interactive Publisher

The Bloomberg infrastructure can send events to provider applications when data is needed for a given topic. These events allow the customer applications to **"interact"** with the Bloomberg infrastructure. Data for a topic need be published only when it is known to have subscribers.

In this simple example, data is published, only as needed, for a set of topics on a single service. The major steps involved are:

- Creating a session
- Obtaining authorization
- Registering for subscription start and stop messages
- Handling subscription start/stop events, which add and remove topics to the active publication set
- Creating a topic
- Publishing events for the active topics of the designated service

The details for creating a session, obtaining a provider identity, and authorization are the same as for 'Simple Broadcast'; they will not be detailed again.

This design requires the management of a collection of **"active"** topics for publication. That collection will be populated (and depopulated) by event handling threads and accessed for periodic publication by the main thread. A map will be used to store pairs of topic/CUSIP pairs (keyed on topic). The topics are provided in the start and stop messages, and CUSIPs are obtained by requesting resolution of the received topics.

The multiple threads of this application must not concurrently access the collection; standard template library (STL) containers are not thread-safe in that respect. Since there is only one **"reading"** thread in this application, a simple mutex suffices. A pthread mutex was chosen because it is familiar to many readers. The event handler is designed to hold pointers to the collection of active topics and to the mutex that manages access to that collection.

```
// InteractivePublisher.cpp
... ..

int main(int argc, char **argv)
{
    Publications activePublications; pthread_mutex_t
    activePublicationsMutex;
    pthread_mutex_init(&activePublicationsMutex, NULL); MyEventHandler
    myEventHandler(&activePublications,
                  &activePublicationsMutex);

    SessionOptions sessionOptions;
    sessionOptions.setServerHost("192.168.9.155");
    sessionOptions.setServerPort(8197);
    //sessionOptions.setAuthenticationOptions("AuthenticationType=OS_LOGON");
    sessionOptions.setAuthenticationOptions("AuthenticationMode=APPLICATION_ONLY;

    ApplicationAuthenticationType=APPNAME_AND_KEY;ApplicationName=blp:APP_BBOX");
```



```

ProviderSession session(sessionOptions, &myEventHandler, 0);
if (!session.start()) {
    std::cerr << "Failed to start session." << std::endl;
    return -1;
}

```

2.2.1. Registration

On completion of service registration, the application can expect subscription start and subscription stop messages in the context of subscription status events.

```

... .. create 'activePublication' collection, the managing mutex,
        and the event handler ... ..
... .. create 'session' and obtain 'Identity'... ..

const char *myService = "//blp/mktdata8";
if (!session.registerService(myService, providerIdentity)) {
    std::cerr << "Failed to register " << myService << std::endl;
    return -1;
}

... ..
}

```

2.2.2. Event Handling

The event handler in this example is detailed below. The relevant event type is `TOPIC_STATUS`. The `TOPIC_STATUS` event has three message types of interest: `TOPIC_CREATED`, `TOPIC_SUBSCRIBED`, and `TOPIC_UNSUBSCRIBED`.

On receipt of **"started"** type messages, the event handler adds the topic to a set of topics that require asynchronous topic creation. Once all of the messages in the event have been examined, that list (if non-empty) is sent for resolution. Use of the session's `createTopicsAsync` method means that the operation does not block. Rather, the result is returned in a separate event of type `TOPIC_CREATED`.

When messages indicating successful topic creation are received, the event handler extracts the topic and the corresponding string, creates an item, and adds that item to the collection of active publications. Since a topic may have received a **"stop"** message while it was being created, there is first a check to see if the topic is still in the "needed" set before it is added to the **"active"** collection.

On receipt of a **"stopped"** type, the event handler extracts the topic from the message and deletes the corresponding item in the collection of active publications or the collection of topics needing creation.

☞ *Note that all operations use the provided mutex to provide exclusive access for each other.*

```

bool MyEventHandler::processEvent(const Event& event, ProviderSession*
session)
{
    switch (event.eventType()) {
        case Event::TOPIC_STATUS: {
            TopicList topicList;

```

```

        MessageIterator iter(event);
        while (iter.next()) {
            Message msg = iter.message();
            std::cout << msg << std::endl;
            if (msg.messageType() == TOPIC_SUBSCRIBED)
            { Topic topic;
              try {
                  topic = session->getTopic(msg);
              }
              catch (blpapi::Exception &) {
              }
              if (!topic.isValid()) {
                  topicList.add(msg);
              }
              else if (d_actPub_p->find(topic) == d_actPub_p->end())
              { std::string topicStr =
msg.getElementAsString("topic");
                pthread_mutex_lock(d_actMutex_p);
                PublicationItem publicationItem(topic, topicStr);
                d_actPub_p->insert(publicationItem);
                pthread_mutex_unlock(d_actMutex_p);
            }
        }
        else if (msg.messageType() == TOPIC_UNSUBSCRIBED) {
            Topic topic;
            try {
                topic = session->getTopic(msg);

                pthread_mutex_lock(d_actMutex_p);
                Publications::iterator it = d_actPub_p->find(topic);
                if (it != d_actPub_p->end()) {
                    d_actPub_p->erase(it);
                }
                pthread_mutex_unlock(d_actMutex_p);
            }
            catch (blpapi::Exception &) {
            }
        }
        else if (msg.messageType() == TOPIC_CREATED) {
            try {
                Topic topic = session->getTopic(msg);
                std::string topicStr =
msg.getElementAsString("topic");
                pthread_mutex_lock(d_actMutex_p);
                PublicationItem publicationItem(topic, topicStr);
                d_actPub_p->insert(publicationItem);
                pthread_mutex_unlock(d_actMutex_p);
            } catch (blpapi::Exception &e) {
                std::cerr
                    << "Exception in Session::getTopic(): "
                    << e.description()
                    << std::endl;
                continue;
            }
        }
    }

```

```

        }
    }
    if (topicList.size()) {
        session->createTopicsAsync(topicList);
    }
}
break
;
defau
lt:
    printMessages(event);
}
return true;
}

```

2.2.3. Publication

The publication loop in this example is, in many ways, similar to that used in the first example. There is a value that is incremented every ten seconds and is used to create an event for publication.

```


Service service = session.getService(myService);
Name messageType("MyMessageType");
Name fieldName("MyFieldName");
for (int value = 1; true; ++ value, sleep(10)) {
    pthread_mutex_lock(&activePublicationsMutex);

    if (0 == activePublications.size()) {
        continue;
    }

    Event event = service.createPublishEvent(); EventFormatter
    eventFormatter(event);
    for (Publications::iterator iter = activePublications.begin();
        iter != activePublications.end();
        ++iter) {
        const std::string& cusip = iter->second;
        eventFormatter.appendMessage(messageType, iter->first);
        eventFormatter.setElement(fieldName, myValueFor(cusip,
value));
    }
    pthread_mutex_unlock(&activePublicationsMutex);
    session.publish(event);
}
session.stop();

return 0;
}

```

 **Note:** The standard C library '*sleep*' function is used above. The argument specifies the number of seconds to sleep.

However, there are some differences, rather than a single fixed topic, publication is made for all of the topics in the collection of active publications.

Note that the mutex is acquired before iterating over that collection.

There is at most one published event per cycle. Each event may have multiple messages, each with data for a specific topic. Although sending an empty event would not be harmful, if the collection of active publications is empty, no event is published for that cycle. The published data might vary by topic. Details of the myValueFor function are not important and, therefore, not shown.

3. Publishing Classes

This section gives an overview on the Publishing classes for the Platform. The figure below displays the Class Representations in detail, each one of these will be explained in detail.

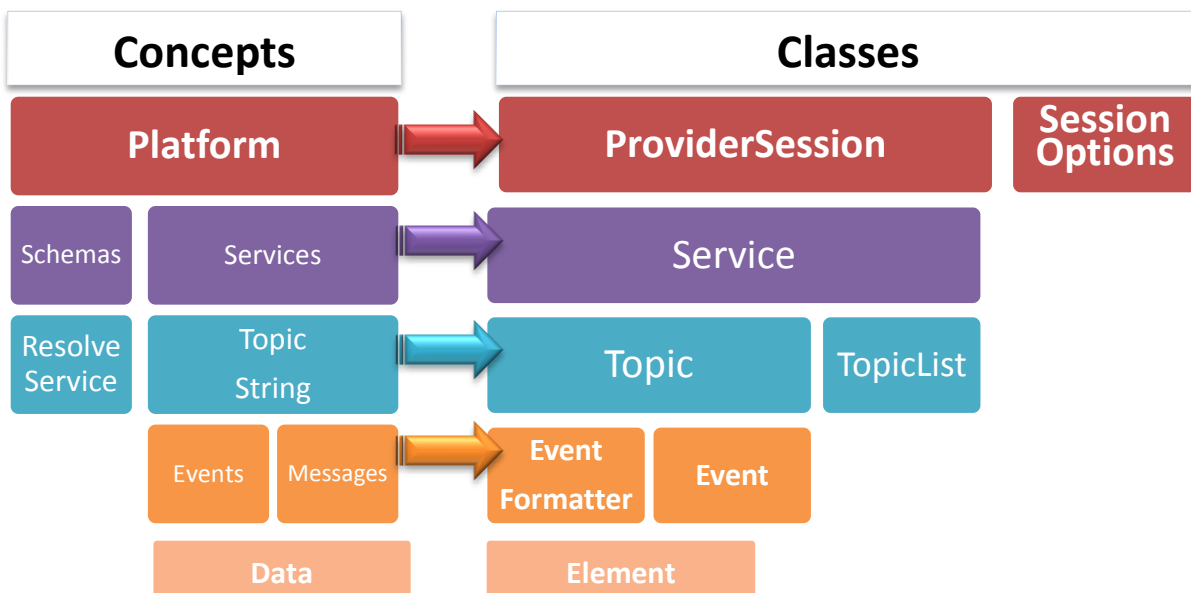


Figure 2: Platform Class Representation

3.1. ProviderSession

The ProviderSession object provides the context of a customer application's connection to the Bloomberg infrastructure via the Bloomberg API. Having a ProviderSession object, customer applications can use them to create Service objects for using specific Bloomberg services.

The SessionOptions class is used to initiate ProviderSession:

```

SessionOptions sessionOptions;
ProviderSession session(sessionOptions,
    &myEventHandler, 0);
if (!session.start())
{
    HandleFailure();
    return;
}
  
```

3.1.1. Session Options/Software Failover

The SessionOptions class provides all of the initialization information that the ProviderSession requires.

Many of its methods are used to implement a software failover mechanism. Given a list of possible hosts (d_hosts) and a port to connect on (d_port), the SessionOptions class can be populated with a number of potential hosts to connect to.

```
for (size_t i = 0; i < d_hosts.size(); ++i)
{
    sessionOptions.setServerAddress(d_hosts[i],
                                    d_port, i);
}
```

The options to enable the failover are as follows:

```
sessionOptions.setAutoRestartOnDisconnection(true);
sessionOptions.setNumStartAttempts(d_hosts.size());
```

Once the API receives a disconnection event, or tries to connect to a server that isn't active, it will advance to the next host on the list.

3.2. Services

All Bloomberg data provided by the Bloomberg API is accessed through a "**service**". The Schema defines the format of requests to the service and the events returned from that service. The customer application's interface to a Bloomberg service is a Service object. Multiple applications can publish to a single service with the same topics. Platform provides the ability to assign priorities:

- Priority
 - Integer Value
 - Higher the value the more preferred
- Group
 - Set of publishers for a service that support identical sets of topics
 - Possibly different priorities within group

3.2.1. Registering a Service

There are two ways to registering a service:

- a. **ServiceRegistrationOptions** is one where a **setGroupID** is to set the identifier for the group that the application belongs to and the other one is **setServicePriority** where the priority is set for the application for this service.
- b. **ProviderSession** is the second way where the **registerService (service_name, app_identity, service_options)** helps in registering the service with the ProviderSession and Platform and **getService(service_name)** is used to retrieve a Service object for use after it's been registered.

3.3. Topics and TopicList

The concept of a Topic is an “**instrument**” of data that is to be distributed and all the messages are created with the topic object. A **Topic** (Concept) is an instrument of data that is to be distributed. All messages are created using the topic object.

The **TopicList** is used to construct a list of topics to be created:

```
topicList.add("//viper/mktdata/ABCDE Equity",
CorrelationId(1));
```

It is possible to check status of a topic:

```
for (size_t i = 0; i < topicList.size(); ++i)
{
    int status = topicList.statusAt(i);
    if (status == TopicList::CREATED)
```

The **ProvideSession** help to create Topics(topic_list,); and also updates the status in the TopicList.

3.4. Events and EventFormatter

An event is encapsulates data to be sent to the platform that is created via a Service’s createPublishEvent.

This is populated via the EventFormatter object:

```
Event event = service.createPublishEvent();
```

An EventFormatter is constructed with the Event it updates:

```
EventFormatter = new EventFormatter(event);
```

The ProviderSession.publish(Event) sends the event to the platform.

3.4.1. Example Event Structure

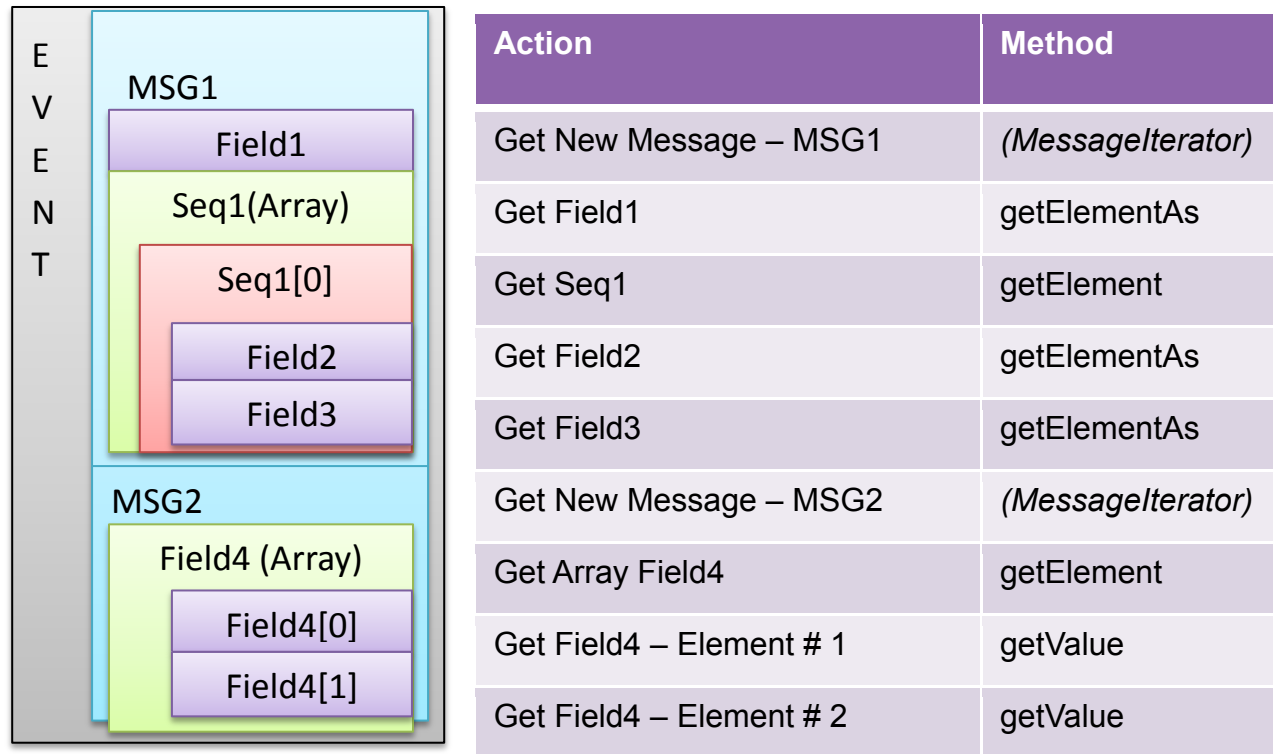


Figure 3: Event Structure - example

3.4.2. Event Formatter

Note: In both deployments, the end-user application and the customer's active Bloomberg Professional service (BPS) share the same display/monitor(s).

Event Formatter	Description
.appendMessage(msg_name, topic)	Appends a message of the specified type to the event for the specified topic
.setElement(element_name, value)	Set the value of a named element
.pushElement(string)	Creates a new element for the EventFormatter. Pushes one level up in the stack that needs to be popped All subsequent operations apply to the element at top the stack
.appendElement()	Creates a new Array Element Pushes one level up in the stack that needs to be popped
.appendValue(value)	Adds the value to the current Array
.popElement()	Removes the current top element from the event formatter stack Each call to pushElement and appendElement requires one call to popElement

4. Local Publishing

Local Publishing responds to subscriptions and can be locally subscribed to. It is displayed in containers and can use directly in local application. It will be available globally soon and will have full control of schema, container is Restricted. This is available in Excel and uses BLPAPI. Listed below is a code example:

```
Topic topic = session->getTopic(msg);
CorrelationId cid = msg.correlationId(); // Solicited Only
CorrelationId * cidptr = (isSolicited) ? &cid : 0;
Event event = service.createPublishEvent();
EventFormatter eventFormatter(event);
eventFormatter.appendRecapMessage(topic, cidptr);
```

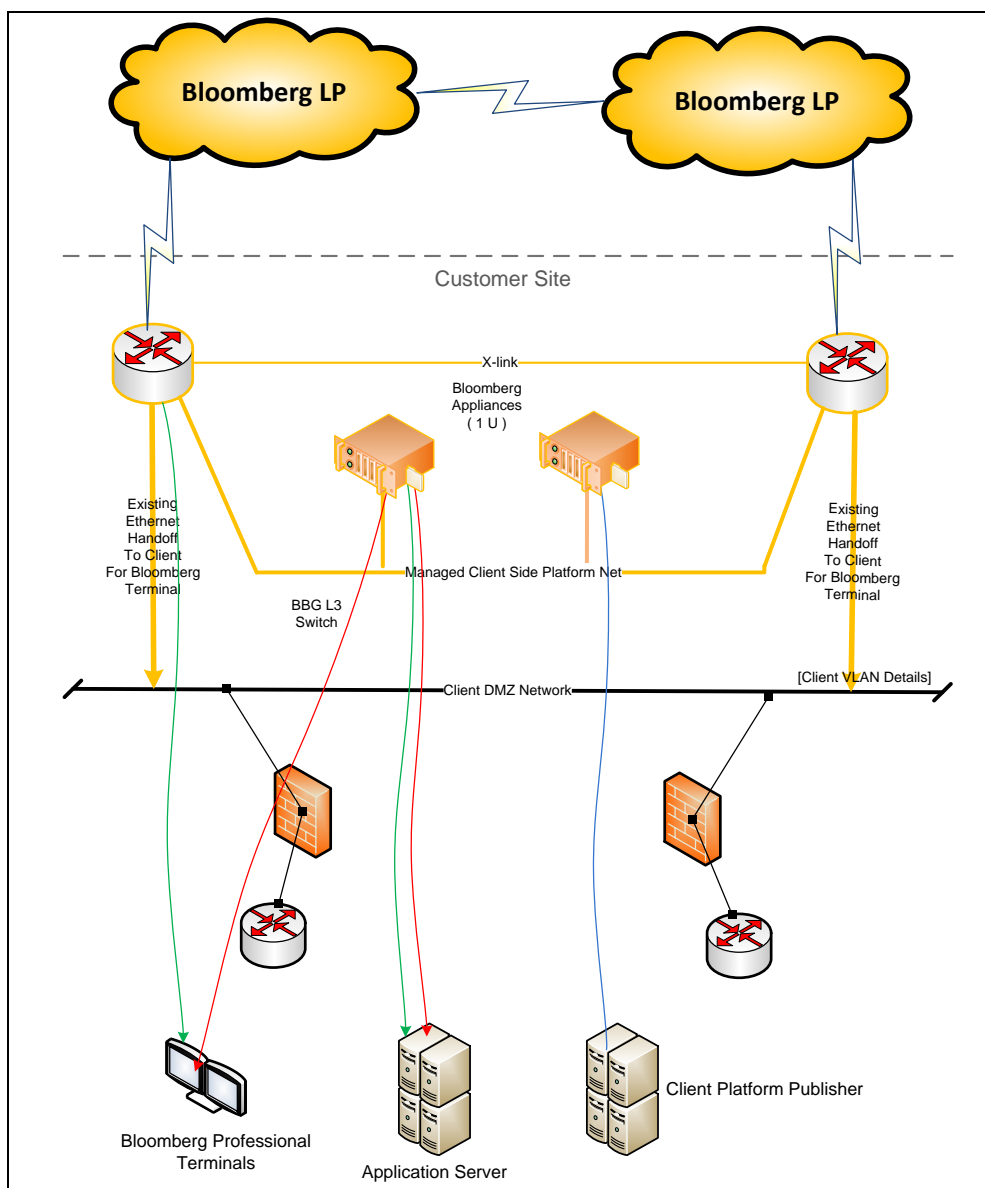


Figure 4: Local Publishing

4.1. Publisher Groups

These groups are needed if multiple publishers on the same topic and determine how the Platform behaves. The publisher will send subscriptions to publishers with the highest priority. If groups are different, multiple publishers can send to the same topic and if groups are the same, only the assigned publisher's data will be used.

Backup Pub

This is a publisher that receives no subscription requests until the primary pub fails. It is achieved by using a publisher group; the backup simply has a lower priority:

```
ServiceRegistrationOptions options;
char *groupID = "SomeGroupID";
options.setGroupId(groupID, strlen(groupID));
options.setServicePriority(Priority);
providerSession.registerService(serviceName, identity, options);
```

The present values for Priority are 'PRIORITY_LOW', PRIORITY_MEDIUM and PRIORITY_HIGH. The Platform will load balance between all active members of the group with the currently highest priority.

4.2. Launchpad Page Publishing

Page Monitor is a Launchpad component that is included as a part of (BPS) Bloomberg Professional Service. This understands a set schema on Platform for Page Publishing.

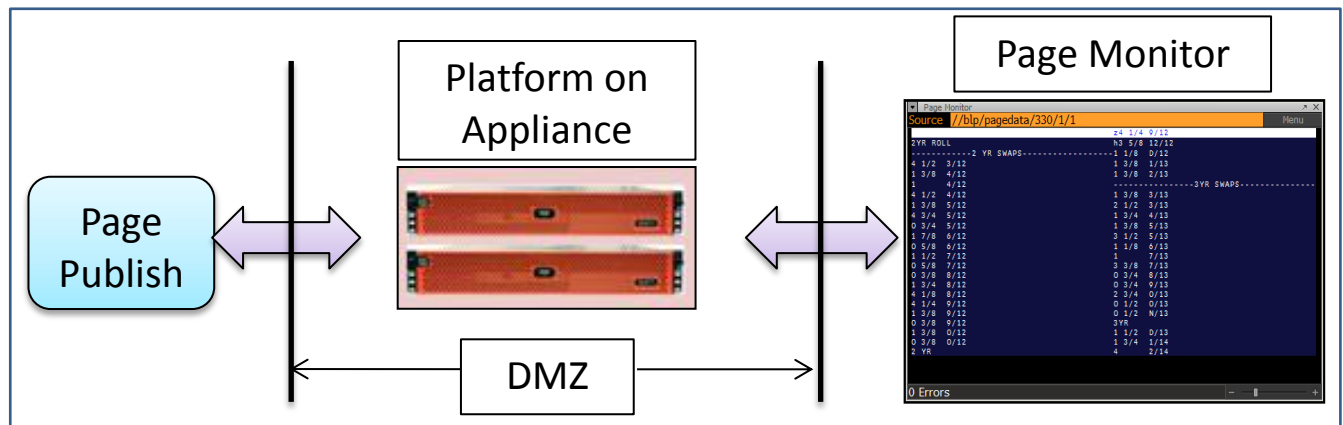


Figure 5: Page Publishing Overview

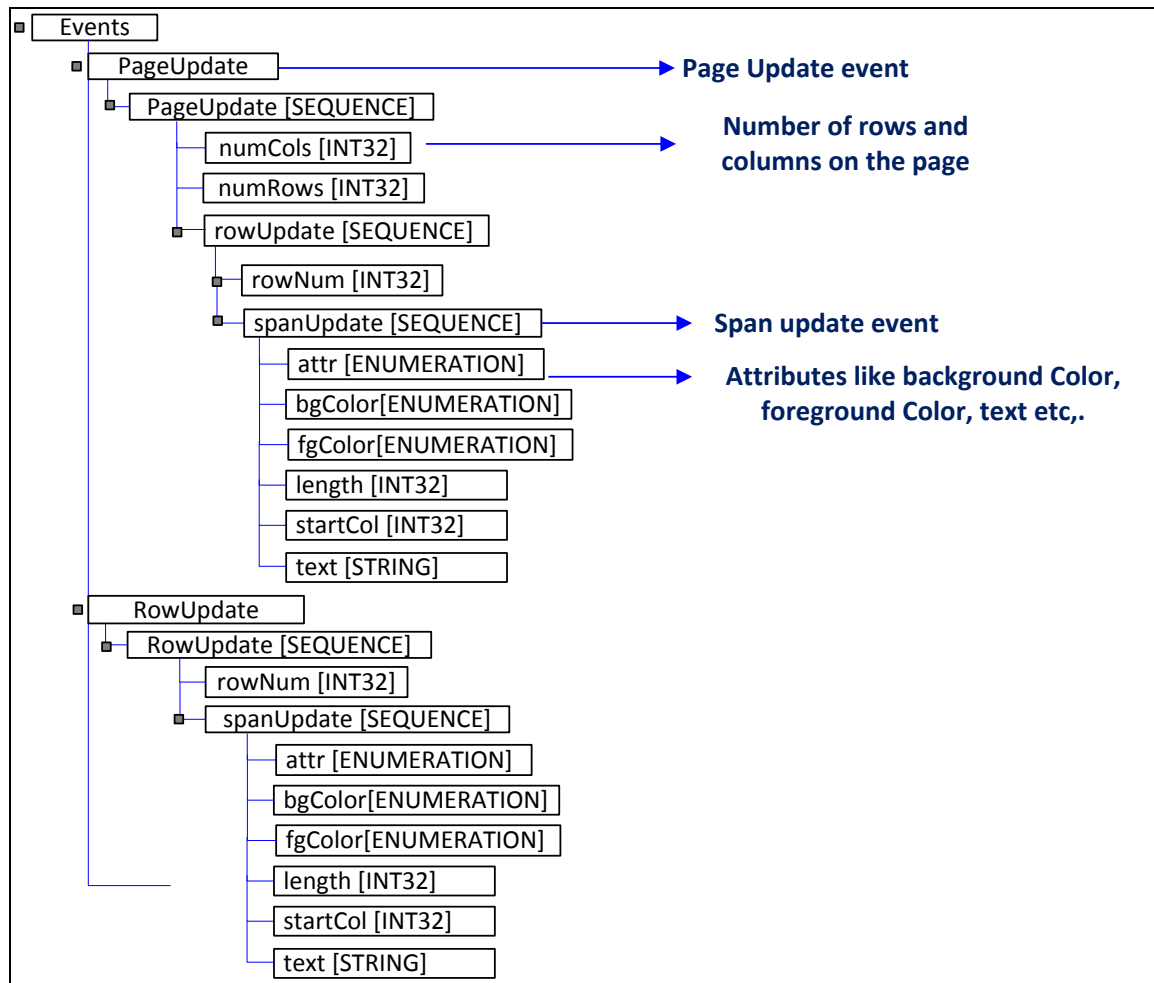


Figure 6: Publishing Schema

5. Platform Failover

Failover occurs when one of the paired platforms is unavailable for any reason – Network/Server/Router/Service:

- API gets notified of lost connection
- API will hunt through list of Appliances
 - Until it finds an active Platform

API will re-establish all subscriptions and API keeps a replay buffer (of limited size).

Platform Groups

- Any number of platforms can be grouped together.
- A service will be primary on any of the platforms in a group.
- All other platforms in the group will receive data from the primary to be distributed to any subscriber connected to that platform.
- In the event of the primary failing, a new platform will become the primary for that service.
- The new platform to become primary is guaranteed to be one with a publisher connected to it.

Platform groups are connected either via cross connect in the same site or via Bloomberg network between sites.

Publisher Groups

- Are needed if multiple publishers on the same topic.
- How the Platform behaves is determined by publisher groups and priority
- The publisher will send subscriptions to publishers with the highest priority
- If groups are different, multiple publishers can send to the same topic
- If groups are the same, only the assigned publisher's data will be used

Backup Pub

- A publisher that receives no subscription requests until the primary pub fails
- This is achieved by using a publisher group
- The backup simply has a lower priority

Load Balancing

- Giving both publishers the same priority in the same group will achieve load balancing
- Each publisher gets topic requests by round robin
- The same level of redundancy will be achieved
- This is not recommended, if this will result in publishing across sites where such network traffic is an issue

Disconnected operation

- In the event that a platform cannot connect to Bloomberg
- All local publishing will continue
- All permissioning will use the last cached copies
- Changes to EMRS will not be available

6. Contribution

Contribution is any piece of data that is published to the Bloomberg back end. It is the data that is submitted to the Bloomberg Ticker plant and provides data for distribution to other Bloomberg clients. It provides access to contribute data controlled via Enterprise Management and Reporting service (EMRS) and access to consume controlled via traditional NFPV (a contribution Entitlement function that displays the attributes for all products).

6.1. Contribution Example Code

All examples are in C++. However, only language specific differences exist for other programming languages. The structure of the code is essentially the same. For specific details, please refer to the standard examples provided with the API in each language specific directory.

a. Create the ProvideSession object

The Assumption is that there exists a sessionOptions object, and a defined event handler...

```
ProviderSession session(sessionOptions, &myEventHandler, 0);
```

b. Service registration

A service can be explicitly registered...

```
ServiceRegistrationOptions serviceOptions;
// Set of publishers for a service that support identical set of topics, but may
have different priorities.
serviceOptions.setGroupId(groupId, groupId.size());
//Integer value, the higher the value, the more preferred
serviceOptions.setServicePriority(priority);
// Assuming user already has service and identity object...
if(!providerSession->registerService(service, providerIdentity, serviceOptions))
{
    ...
}
```

...or alternatively, user can specify AUTO_REGISTER_SERVICES on Topic creation...as shown below

c. Create the TopicList object and register

```

TopicList topicList;
Topic topic;
// add topic to the resolution list
topicList.add(service + topic, CorrelationId(topic));
// createTopics() is synchronous, topicList will be updated
// with the results of topic creation (resolution will happen
// under the covers)
providerSession->createTopics(topicList,
                               ProviderSession::DONT_REGISTER_SERVICES,
                               providerIdentity);

```

d. Verify Topic resolution

```

//Parse the topicList to check resolution status of the list of topics
for (size_t i = 0; i < topicList.size(); ++i)
{
    int resolutionStatus = topicList.statusAt(i);

    if (resolutionStatus == TopicList::CREATED)
    {
        topic = providerSession->getTopic(topicList.messageAt(i));
    } else
    {
        ...
    }
}

```

e. Create the publish Event object

The Event object is the mechanism by which data is communicated to the back end

```

Service service = session.getService(service);
Event event = service.createPublishEvent();

```

f. Create the EventFormatter object

This may take place within a loop if multiple topics are being contributed...

```

EventFormatter eventFormatter(event);

```

The use of the event formatter object depends on the specific activity taking place.

g. EventFormatter for Market Data contribution

The EventFormatter object allows the developer to format an event message, in this case, Market Data...

```

eventFormatter.appendMessage("MarketData", topic);
eventFormatter.setElement("BID", bid_value);
eventFormatter.setElement("ASK", ask_value);

```

h. EventFormatter for Page Data contribution

The EventFormatter object for Page Data. The data is similar to XML in structure. Each pushElement makes that the current element. Each popElement restores the parent element as the current element..

```
eventFormatter.appendMessage("PageData", topic);
eventFormatter.pushElement("rowUpdate");
eventFormatter.appendElement();
eventFormatter.setElement("rowNum", 1);
eventFormatter.pushElement("spanUpdate");
eventFormatter.appendElement();
eventFormatter.setElement("startCol", 20);
eventFormatter.setElement("length", 4);
eventFormatter.setElement("text", "TEST");
eventFormatter.popElement();
eventFormatter.appendElement();
eventFormatter.setElement("startCol", 25);
eventFormatter.setElement("length", 4);
eventFormatter.setElement("text", "PAGE");
eventFormatter.popElement();
eventFormatter.popElement();
eventFormatter.popElement();
eventFormatter.setElement("productCode", 1);
eventFormatter.setElement("pageNumber", 1);
```

i. EventFormatter to Clear a GDCO Page

The EventFormatter object also allows for a GDCO page to be cleared. This unloads any currently loaded securities :

```
eventFormatter.appendMessage("MonitorablePageData", topic);
eventFormatter.pushElement("clearMonitorablePage");
eventFormatter.setElement("pageID", gdco); // GDCO number
eventFormatter.setElement("pageSubID", monitor); //Monitor number
eventFormatter.popElement();
```

j. EventFormatter to Clear a GPGX Page

The EventFormatter object also allows for a GPGX page to be cleared :-

```
eventFormatter.appendMessage("PageData", topic);
eventFormatter.pushElement("clearPage");
eventFormatter.setElement("contributorId", contributorID); // Contributor no.
eventFormatter.setElement("pageNumber", gpgx); // GPGX number
eventFormatter.setElement("productCode", product); //Monitor number
eventFormatter.popElement();
```

k. EventFormatter to load a security on GDCO Page

The EventFormatter object is used to load a security onto a GDCO page, at a specified row :-

```
eventFormatter.appendMessage("MonitorablePageData", topic);
eventFormatter.pushElement("loadMonitorableSecurities");
eventFormatter.setElement("pageID", gdco); // GDCO
eventFormatter.setElement("pageSubID", monitor); // Monitor
// A security is being loaded...
eventFormatter.setElement("pageOperation", "ADD");
// identifierType has to match with what is setup on the terminal for the monitor
// on the GDCO# in use.
// Contact the contribution representative and use the appropriate type.
```

```
// If there is a mismatch in the identifierType sent by the application and what is
// on the terminal, the packet will not be displayed on the GDCO.
// Possible values for identifierType are:
//     - PAGE_ACCESS_TYPE_ISIN
//     - PAGE_ACCESS_TYPE_NONE
//     - PAGE_ACCESS_TYPE_BBG_NUMBER
//     - PAGE_ACCESS_TYPE_CUSIP
//     - PAGE_ACCESS_TYPE_TICKER
//     - PAGE_ACCESS_TYPE_NEW_ISIN
eventFormatter.setElement("identifierType", identifierType);
eventFormatter.setElement("identifier", security);
// loadIndicator specify how to load the securities on the page.
// Possible values are ABSOLUTE_ORDER/MATURITY_ORDER
eventFormatter.setElement("loadIndicator", "ABSOLUTE_ORDER"); //No maturity sorting
eventFormatter.popElement();
```

I. Publishing the Event

This passes the event object to the back end to be processed :-

```
session.publish(event);
```