I-TECHNOLOGY INC.

Eventz
Python Programmer's Manual

September 03, 2024
Revision 2.3

Document Revision History

Version	Date	Description of Change	Person Responsible
1.0	2017/03/13	Initial Release	Bob Jackson
1.1	2017/03/15	Section 1.1 modified	Chris Jackson
1.2	2017/03/15	Reformat, added Requirements	Bob Jackson
1.3	2017/03/16	Added links to Requirements	Chris Jackson
1.4	2017/05/29	Cleanup	Bob Jackson
1.5	2017/06/07	Added Chapter 7 headers	Chris Jackson
1.7	2017/09/19	Updated, Added State Machine	Bob Jackson
1.8	2019/05/09	Changed name to Eventz – Updated with new API	Bob Jackson
1.9	2019/09/26	Updated with new code changes	Bob Jackson
2.0	2019/10/25	Added Utility section	Bob Jackson
2.1	2021/03/11	Documented new features	Bob Jackson
2.2	2021/05/06	Updated.Added Examples (Hello World) and Archive Services section.	Bob Jackson
2.3	2024/08/16	Updates	Bob Jackson

Table of Contents

1 Introduction 1
1.1 An Eventz Primer 1
2 Requirements 1
3 Fast Track 2
4 Publishing 2
4.1 Metadata 3
4.2 Data 4
4.3 Programming 4
4.3.1 Parameters 4
4.3.2 Publishing 8
5 Subscribing 9
5.1 Qt 9
5.2 Tkinter 10
5.3 WxPython 11
5.4 Exiting 11
6 Querying The Archive 12
7 Logging 15
8 Table Driven State Machine 16
9 Utilities 18
9.1 Refresh Archive 19
9.2 updateArchive 19
9.3 archive 20
9.4 match 20
9.5 startApplication 20
9.6 stopApplication 20
10 Archive Services 21
11 Examples 21
11.1 Hello World 21

1 Introduction

The Eventz Infrastructure provides a mechanism for inter-module communications and a common data store for synchronizing data across modules (Event sourcing). It relies on a publish and subscribe paradigm to pass data. It also archives data in one or more secure locations. The Archive is an indelible, read-only store that is exclusively written by the Eventz Archivist service. Data is given to a module through subscription and can, in addition, be obtained by querying a Librarian service. This manual will gave a Python programmer the information necessary to utilize Eventz using the Eventz API (eventzAPI.py).

1.1 An Eventz Primer

Eventz Infrastructure is a concept that involves compartmentalizing a software project development into small, logically distinct modules (microservices) which are easily modified and infinitely more manageable. Thus the completed software project is constructed in 'building-block style' by implementing microservices.

The basic idea is that the microservices are the only ones permitted to interface with the program. Each module is created, tested and maintained separately. Each is designed to be responsible for a defined task. Typically, a microservice will have a limited amount of code for easy maintenance.

Eventz returns to the data encapsulation concept from the early days of computing, by using records with prescribed fields. These we call Eventz records. They are communicated using Publish and Subscribe methods. Every Eventz record that is published can be recorded in a Write Once Read Many store (Archive), and there can be more than one archive each located far apart for safety and security. Changes result in a new read-only records instead of the constant, and potentially disastrous, record modification. Published Eventz records are indelible and cannot be altered, like being written on a granite wall onto which everything is 'cast in stone'.

Networking of the Eventz records is provided by a Broker and an Archivist. A Librarian is made available to support queries of previous Eventz records in the Archive. These services can be located all on one machine or distributed widely over a network and the internet.

Microservices have the option of creating a local archive that contains those Eventz records that the application subscribes to. The Eventz API Subscriber object will keep the local archive current.

2 Requirements

Before starting a project the following components need to be in place (download links for programs are provided):

1. RabbitMQ (https://www.rabbitmq.com/download.html) Broker running on a machine accessible through the network*
2. Depending on the strategy in the next step, download and run Docker (https://www.docker.com)
3. Get Eventz Archivist and Eventz Librarian from Docker Hub or from the GitHub rerpositories.
4. Eventz Archivist and Eventz Librarian running on a machine accessible through the network
5. Python3.12 installed (https://www.python.org/download/releases/3.12/) either globally or in a virtual environment.
6. Python IDE (PyCharm [https://www.jetbrains.com/pycharm/download/], IntelliJ IDEA [https://www.jetbrains.com/idea/download/], Eclipse [https://www.eclipse.org/downloads/eclipse-packages/] with Python plug-in [https://marketplace.eclipse.org/content/pydev-python-ide-eclipse], Idea, etc.)
7. Python Libraries (pyqt5, QT6 [http://pyqt.sourceforge.net/Docs/PyQt6/installation.html])

3 Fast Track

The eventzAPI instantiates a number of objects. In order to simplify the process an ApplicationInitializer object has been created. It has one method: “initialize()”. The ApplicationInitializer has the following parameters:

- routing_keys – a list of eventz record types that the application subscribes to.
- publications a list of eventz record types that the application publishes.
- applicationId – a UUID that uniquely identifies this application.
- applicationName – the name of the application.
- path_to_settings – the path to the settings.yaml from which the application will get initial parameters.
- user_id – a string containing a user id possibly obtained when a user logs on.

The initialize method instantiates the following objects:

- Publisher: The publisher object handles the publication of event records.
- Subscriber: The subscriber task watches for event records in routing_keys and passes them to the app.
- Logger: The logger publishes log eventz records (90000002.00)
- LibrarianClient: The librarian client submits queries to the Librarian service and returns the result set.
- Utilities: The utilities object has many methods that aid the process.
- Parameters: The parameters object contained argument values that are used by the eventzAPI objects.

```
# Initialize to get eventz objects

ai = ApplicationInitializer(subscriptions, publications, application_id, application_name,
                            path_to_settings, user_id)
a_publisher, subscriber, logger, librarian_client, utilities, parameters = ai.initialize()
```

Figure 1: Sample code for eventz object instantiation

4 Publishing

As part of the Eventz paradigm a developer defines the data that is to be shared among modules. This definition follows a defined structure that includes metadata that is pre-pended to every record. This metadata is part of the search criteria allowed by the Librarian

4.1 Metadata

Since Python does not preserve order in objects we describe data using an Enum that defines order. This Metadata Enum names the elements and their order.

```
#
# The DS_Metadata Enum provides an index into the fields of
# metadata that pre-pend a DS record
#

class DS_MetaData(Enum):
    recordType = 0
    action = 1
    recordId = 2
    link = 3
    tenant = 4
    userId = 5
    publishDateTime = 6
    applicationId = 7
    versionLink = 8
    versioned = 9
    sessionId = 10
    userMetadata1 = 11
    userMetadata2 = 12
    userMetadata3 = 13
    userMetadata4 = 14
    userMetadata5 = 15
```

While Python utilized duck typing the records are nevertheless defined in the following table.

Field	Data Type	Notes
recordType	Floating Point # (2 Decimal places)	Decimal values are for versioning e.g. 25000.01
action	Integer	0 = Insert, 1 = Update, 2 = Delete
recordId	UUID	Unique record identifier
link	UUID	recordId of updated or deleted record
tenant	UUID	Used to differentiate one entities' data from anothers. The Librarian will only reurn 1 tenants' data from the Archive.
userId	String	Name or UUID

Field	Data Type	Notes

publishDateTime	Date Time	`datetime.datetime.utcnow().isoformat(sep='T')`
		
		or
		
		`datetime.datetime.now(datime.UTC).isoformat(sep='T')`

applicationId	UUID	The module programmer gets a UUID for their Application
versionLink	UUID	A link to the original record being versioned
versioned	Boolean	True if this is generated as a version
sessionId	UUID	Provides a link between messages that belong to a session. Useful when multiple copies of an application are running as each will recogize when subscribed data resulted from their query.
userMetadata1	User defined	Data the programmer wants indexed by the Librarian for queries
userMetadata2	User defined	as above.
userMetadata3	User defined	as above.
userMetadata4	User defined	as above.
userMetadata5	User defined	as above.

4.2 Data

The programmer provides the record data in a tuple. Fields are string conversions from the underlying type.

4.3 Programming

The Eventz Framework provides message passing facilities using a publish and subscribe methodology. In order to integrate the Eventz API into a python microservice a number of things need to happen,

4.3.1 Parameters

The API works with a RabbitMQ service. In order to connect to this service a number of parameters need to be set. Some of these parameters are hard coded by the programmer and some are unique to the installation and are provided in a file (e.g. settings.yaml – the name is passed as the first argument for the application call) which is outlined below. In addition several parameters are defined in the code

The initialization procedures in eventzAPI draw on data provided in a settings.yaml file. The file is described in figure 2 below. The parameters in this file need to be set at installation when they are known. The path to this file is passed as a parameter to the initialization routines and may be hard coded in the application or passed as

a command line parameter. (see example)

```
applicationId = [UUID Unique to this application]
applicationName = [The name of the application]
applicationUserName = [The user running the app.]

routingKeys = [A list of record types subscribed to by the app.]
publications = [A list of record types published by the app.]

e.g.

applicationId = '35c87ca6-e9e6-4ae3-b10c-942d4508208a'
applicationName = 'Front Desk Microservice v1.0'
applicationUserName = 'I-Tech'
routingKeys = ['6030.00', '6040.00']# Record Types subscribed to
publications = ['6010.00']      # Record Types published
```

Figure 2: Hard Coded Parameters

```
brokerExchange: amq.topic
brokerIP: [I.P. of the RabbitMQ broker]
brokerPassword: [The RabitMQ Brokers' password]
brokerUserName: [The RabbitMQ Brokers' User Name]
brokerVirtual: [For a Broker in a remote VM the password]
librarianExchange: LIBRARIAN_RPC
librarianExchangeType: direct
librarianQueue: rpc_queue
amqpURL: amqp://ydsvxvfo:zVErILVLnKFTYi1ZOHnUAxlFJZqji-
7i@shrimp.rmq.cloudamqp.com/ydsvxvfo
deviceId: [The device ID. Usually a mac address]
deviceName: [A human readable name identifying the device]
location: [A plain text location identifier]
firstData: [Offset to the data in a message, currently 16]
qt: [If Qt is used for the gui this is true, otherwise false]
localArchivePath: [Path to a local Archive if one is needed]
master_archive: [true if there is a master archive
                 false if the service stands alone]
encrypt: [Use encryption? true or false]
pathToCertificate: [If encrypt is true this is path to cert]
pathToKey: [If encrypt is true this is path to key]
pathTocaCert: [If encrypt is true this is path to cacert]
tenant: [UUID identifying the owner of the data. (0 is:
          '00000000-0000-0000-0000-000000000000']
```

Figure 3: settings.yaml file contents

In the API, the parameters are extracted into a dsParam object by a call to DS_Init.getParams. Note that the fast track process above creates this parameters object for you.

```
dsInit = DS_Init(applicationId, applicationName)    # Create the object
parameters = dsInit.getParams('settings.yaml', routingKeys, publications, None)
```

Figure 4: Code initializing the parameters object – executed in ApplicationInitialivzer.initialize()

The `DS_Parameters object is defined below.`

```
class DS_Parameters(object):
      '''
      Parameters object to be instatiated once and passed as a parameter
      to DSAPI objects and methods
      23 Parameters
      '''

      def __init__(self, exchange, brokerUserName, brokerPassword,
                  brokerIP, sessionID, interTaskQueue, routingKeys,
                  publications, deviceId, deviceName, location,
                  applicationId, applicationName, tenant, archivePath,
                  master_archive, encrypt, pathToCertificate, pathToKey,
                  pathToCaCert, qt, brokerVirtual, thePublisher,
                  firstData = 16):

            # RabbitMQ Parameters
            self.broker_user_name = brokerUserName
            self.broker_password = brokerPassword
            self.broker_IP = brokerIP
            self.virtualhost = brokerVirtual
            self.exchange = exchange

            # Encryption Parameters
            self.pathToCaCert = pathToCaCert
            self.pathToCertificate = pathToCertificate
            self.pathToKey = pathToKey
            self.encrypt = encrypt

            # Application parameters
            self.session_id = sessionID
            self.archivePath = archivePath
            self.master_archive = master_archive
            self.deviceId = deviceId
            self.deviceName = deviceName
            self.location = location
            self.applicationId = applicationId
            self.applicationName = applicationName
            self.tenant = tenant
            self.routingKeys = self.subscriptions = routingKeys
            self.publications = publications
            self.firstData =firstData
            self.qt = qt
            self.inter_task_queue = interTaskQueue
            self.the_publisher = thePublisher
```

Figure 5: The DS_Parameters Object Definition

4.3.2 Publishing

The Eventz API provides a Publisher object in parameters. This object is instantiated once in the application. The publisher opens and maintains a connection with the Broker. The code below shows how to publish a record.

```
eg.

from eventzAPI.eventzAPI import Publisher
.
logTuple = (deviceID, deviceName, applicationID, applicationName, errorType,
            errorLevel, errorAction, errorText)
messagePublished = dsParam.thePublisher.publish(6010.00, 0, 0, self.userID,
                                                "", "", "", "", "", logTuple)
```

The messagePublished value returned by the publish method above is a string showing the actual message published including the metadata. This will be useful if you need to know the record_id or other metadata field values assigned by the api.

Table 1: publish method arguments

Field	Description
recordType	The floating point number that identifies the record.
action	0 = Insert, 2 = Update, 3 = Delete
link	UID pointing to recordId of modified record (0 will be changed to: '00000000-0000-0000-0000-000000000000')
userId	Name or UUID indicating who wrote the record.
versionLink	UID pointing to recordId of versioned record (0 will be changed to: '00000000-0000-0000-0000-000000000000')
versioned	True if this record generated by a Versioner
sessionID	UID of the current session (0 will be changed to: '00000000-0000-0000-0000-000000000000')
umd1 umd2 umd3 umd4 umd5	Data the programmer wants exposed to the Librarian for queries
dataTuple	The record data as a tuple.

Publishing a record will send a copy of the record to all subscribers for the recordType including the Archivist which will append it to the Archive.

5 Subscribing

Every module must subscribe to the data (record types) it needs to know about. If the application doesn't need to know about events it still needs a subscriber to respond to System Messages.

The Subscriber Factory is an object with a method, ‘makeSubscriber()’, that creates a thread that watches for messages from the RabbitMQ Broker. Some System Messages (Ping) are handled by the Subscriber Thread, all other messages are passed to the main thread of the module. The Subscriber Thread uses either QT signals and slots or an inter-task queue to pass information from the thread to the main application. If the application uses a Gui it will be necessary to break into the gui’s event loop to service a message from the subscriber task. If Qt is the gui it will require the programmer to accept QT licensing and to download the requisite files. We use pyqt.py and QT5.

The ApplicationInitializer.initialize() method creates the subscriber thread, starts it and returns the instantiated object:

5.1 Qt

If the GUI is Qt then, to handle incoming messages passed on from the subscriber task, we need a slot.

```
@pyqtSlot(float, str)
def processMessage(self, record_type, message):


   print('Processing Record Type: %f  %s' %(recordType, message))
   if record_type == “xxxx.xx”:
       # Process this record type here
       .
       .
   elif record_type == “yyyy.yy”;
       # Process this record type here

       .
       .
   .
   .
```

Figure 6: Qt Slot for Record Handling

We connect to the slot as follows.

```
self.subscriber.pubIn.connect(self.processMessage) # Connect to signal
```

Figure 7: Subscriber Task connection

5.2 Tkinter

If the GUI is Tkinter we need to break into the event loop to test the inter-task queue.

```
import tkinter as tk

Class Main():
  .
  .
  def run(self):
    .
    .
    self.check_queue()

  def check_queue(self):

      if not self.parameters.inter_task_queue.empty():
          message = self.parameters.inter_task_queue.get_nowait()
          print(f'Received: {message}')
          self.parameters.inter_task_queue.task_done()
          self.process_message(message[0], message)

      self.root.after(100, self.check_queue)

if __name__ == "__main__":
    root = tk.Tk()
    main = Main(root)
    root.mainloop()
    pass
```

Figure 8: Tkinter event loop interdiction

Here the root.after() function calls check_queue every 100 ms.

5.3 WxPython

If the GUI is wxPython the following code works

```
import wx


Class Main():
  .
  .
  def run(self):
    .
    .
    self.check_queue()

  def check_queue(self):

      if not self.parameters.inter_task_queue.empty():
          message = self.parameters.inter_task_queue.get_nowait()
          print(f'Received: {message}')
          self.parameters.inter_task_queue.task_done()
          self.process_message(message[0], message)

      self.CallLater(100, self.check_queue)

if __name__ == "__main__":
    app = wx.App(False)
    main = Main(None)
    app.MainLoop()
    pass
```

Figure 9: WxPython event loop interdiction

5.4 Exiting

When the application terminates it needs to close the connections to the Broker and terminate the thread. We do this by invoking atexit:

```
import atexit

atexit.register(exiting, closeEvent) # Register code to run when program
terminates

def closeEvent(self, QCloseEvent):
  # Publish a Stopping Message
  newUUID = uuid.uuid4()
  stopTime = datetime.datetime.now(datetume.UTC).isoformat(sep='T')
  stoppingTuple = (9000001.00, 0, str(newUUID),
      '00000000-0000-0000-0000-000000000000',
      '00000000-0000-0000-0000-000000000000', self.dsParam.deviceId,
       self.dsParam.deviceName, self.dsParam.location,
      self.dsParam.applicationId, self.dsParam.applicationName,stopTime)

  self.dsParam.thePublisher.publish(stoppingTuple, g.archivePath)

  self.subscriberThread.stop() # Stop the subscriber thread
  print('CLOSING!!!!')
```

Figure 10: Exiting an application

Note: atexit() publishes a 'Stopping' message to inform any monitor that this application is no longer running.

6 Querying The Archive

The Eventz Librarian serves to give Programmers access to the Eventz Archive. The EventzAPI provides a LibrarianClient class to be used in formulating and sending queries to the Librarian microservice and receiving results back. To instantiate the LibrarianClient:

The Librarian Client, when instantiated, accepts dsQuery objects, connects to the Librarian microservice, sends the query and awaits a response. When the response arrives it closes the connection and returns the response to the caller. Querys are passed using the librarianClient .call() method detailed in figure 10 below

```
Import eventzapi

librarianClient = LibrarianClient(self.parameters, logger)
```

Figure 11: Instantiating a LibrarianClient

```
librarianClient.call( userName, tenant, startDate, endDate, limit,
                      queries))
```

Figure 12: Querying a Librarian

where:

userName = A string identifying the user

tenant = UUID identifying the tenant. If none then ***'00000000-0000-0000-0000-000000000000'***

startDate = A date filter setting the record date results must follow or equal. yyyy/MM/dd format.

endDate = A date filter setting the record date the results must precede or equal. yyyy/MM/dd format.

The 'limit' argument allows the programmer to limit the number of results returned. A limit value of 0 means all results will be returned.

queries = A list of one or more QueryTerms

A Query Term is an object as defined in figure 11 below.

```
class QueryTerm(dict):
    '''
    A term in a query consisting of a field name, an operator and
    the value for the search
    '''
    def __init__(self, fieldName, operator, value):
        self.fieldName = fieldName
        self.operator = operator
        self.value = value
```

Figure 13: The QueryTerm object

An example:

```
queries = []
query = QueryTerm("recordType", "EQ", "9000000.0")
queries.append(query)
query = QueryTerm("publishDateTime", "LT",
                  datetime.datetime.utcnow().isoformat(sep='T'))
queries.append(query)

limit = 0
start = datetime.datetime.strptime('1955/01/01', '%Y-%m-%d')
end = datetime.datetime.utcnow().isoformat(sep='T')
result = librarianClient.call('Test User', testTenant, start,
                              end , limit, queries)
print('Result retrieved.')
results = result.decode("utf_8")
```

Figure 14: A Query Example

```
recordCount = 0
if results:
   recordSz = ''
   first = True
   gotCloser = False

   for r in results:
     if r != '[':
       if r == ']':
         if gotCloser == False:  # Ignore if this is the second ] in a row
           newRecordSz = recordSz.replace("'", "")
           newRecordSz = newRecordSz.replace(", ", ",")

           if first == True:
               recordList = newRecordSz.split(',')
               first = False
           else:
               recordList = newRecordSz[1:].split(',')

           print(recordList)  # Here at end of record. Print it and clear
                                  the string
           recordCount += 1
           recordList = []
           recordSz = ''
           gotCloser = True

       else:
         recordSz += r
         gotCloser = False
     else:
       gotCloser = False

           # first = True
   print('Data Transferred from Archive. Got '+str(recordCount)+'records.')
```

Figure 15: Processing a result set

The result set returned from the call method is a list of qualifying records in tuples.

Process the results by iterating through the list and detecting the closing pattern to determine when done:

Filter criteria are limited to 'and' conditions on values for metadata fields. Conditional operators are:

EQ, GE, GT, LE, LT

Metadata field names are:

1. recordType
2. action
3. recordId

4. link
5. tenant
6. userId
7. publishDateTime
8. applicationId
9. versionLink
10. versioned
11. sessionId
12. userMetadata1
13. userMetadata2
14. userMetadata3
15. userMetadata4
16. userMetadata5

Note that edits and deletions in the Archive will result in linked transactions that may need to be reconciled. This can be done by iterating through the list and applying update or delete actions to the records they are linked to. This can be accomplished by passing the results to dsUtility.updateArchive()

7 Logging

The System Messages include an error reporting record. The EventzAPI includes a Logging class (DS_Logger) that publishes log messages.

```
Import eventzapi

logger = DS_Logger(parameters)
```

parametersFigure 16: Instantiate a DS_Logger

The argument to the constructor is dsParam. Everything DS_Logger needs is in parameters (see Parameters above)

To create a log message:

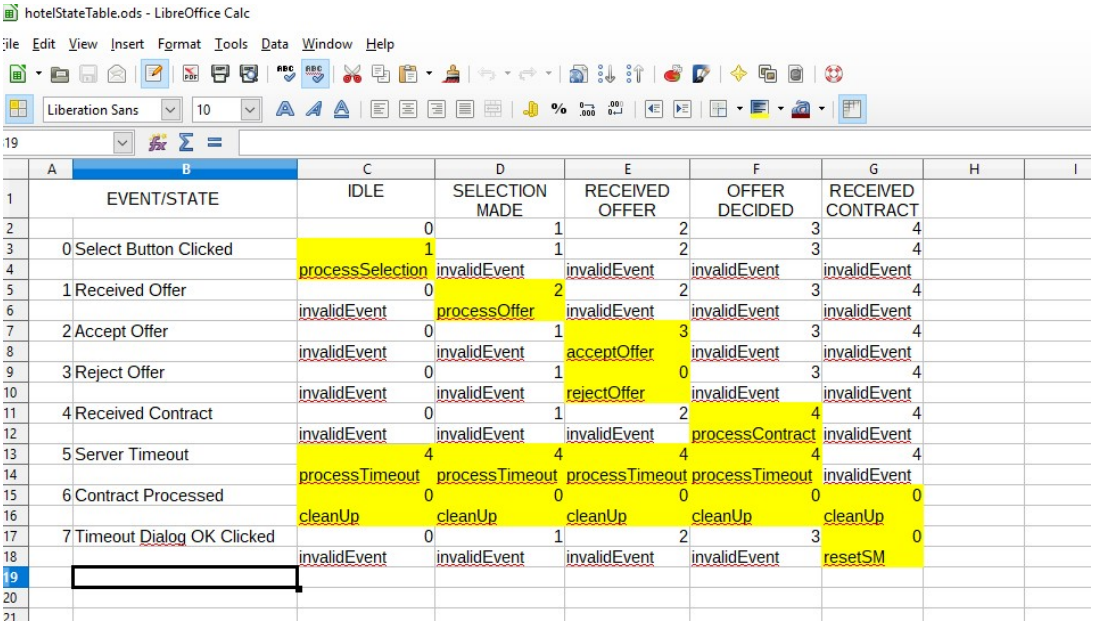
```
logger.log('I-Tech',100, 'INFO', 0, 'Not Really an Error. Just a test.')
```

The arguments to the log message are:

1. UserID — The User name or UUID identifier
2. errorType — A error type defined by the programmer (eg. 404)
3. errorLevel — Error level : INFO, WARNING, ERROR, CRITICAL
4. errorAction — Action (0 = display, 1 = Email Alert Level 1 .. 3 = Email Alert Level 3, 4 = Page Alert 5 = Syslog Alert
5. errorText — A formatted string description of the error

8 Table Driven State Machine

The Table Driven State Machine keeps track of application state and processes asynchronous events according to the dictates of a State Table.



hotelStateTable.ods - LibreOffice Calc

File Edit View Insert Format Tools Data Window Help

Liberation Sans 10

	A	B	C	D	E	F	G	H	I
1		EVENT/STATE	IDLE	SELECTION MADE	RECEIVED OFFER	OFFER DECIDED	RECEIVED CONTRACT		
2			0	1	2	3	4		
3	0	Select Button Clicked	1	1	2	3	4		
4			processSelection	invalidEvent	invalidEvent	invalidEvent	invalidEvent		
5	1	Received Offer	0	2	2	3	4		
6			invalidEvent	processOffer	invalidEvent	invalidEvent	invalidEvent		
7	2	Accept Offer	0	1	3	3	4		
8			invalidEvent	invalidEvent	acceptOffer	invalidEvent	invalidEvent		
9	3	Reject Offer	0	1	0	3	4		
10			invalidEvent	invalidEvent	rejectOffer	invalidEvent	invalidEvent		
11	4	Received Contract	0	1	2	4	4		
12			invalidEvent	invalidEvent	invalidEvent	processContract	invalidEvent		
13	5	Server Timeout	4	4	4	4	4		
14			processTimeout	processTimeout	processTimeout	processTimeout	invalidEvent		
15	6	Contract Processed	0	0	0	0	0		
16			cleanUp	cleanUp	cleanUp	cleanUp	cleanUp		
17	7	Timeout Dialog OK Clicked	0	1	2	3	0		
18			invalidEvent	invalidEvent	invalidEvent	invalidEvent	resetSM		
19									
20									

Figure 17: State Table Worksheet

The state Table is developed in a spreadsheet as shown above.

States are in columns and events are in rows. A machine in a certain state, when confronted with an event, executes the method in the corresponding cell and sets the next state to the index value in the cell. Illegal event/states are handled by the invalidEvent method.

Once completed the spreadsheet is saved as a Tab delimited csv. The EventzApi has a method (SMU class, translateTable()) that translates the csv into a stateTable used by the StateMachine class.

The StateMachine class has one method: processEvent() that is called whenever an event needs to be handled by the StateMachine.

Events may be generated by happenings in the IO loop or in other methods when they are executed. The receipt of a particular message by the SubscriberClient may trigger an event.

The sample code below initializes a state machine.

```
    # Set up State Machine

    pathToTable = s.pathToTable
    self.rt = ''
    self.message = ''
    self.transitions = {
       'processSelection'  : self.processSelection,
       'processOffer'      : self.processOffer,
       'acceptOffer'       : self.acceptOffer,
       'rejectOffer'       : self.rejectOffer,
       'processContract'   : self.processContract,
       'processTimeout'    : self.processTimeout,
       'cleanUp'           : self.cleanUp,
       'resetSM'           : self.resetSM,
       'invalidEvent'      : self.invalidEvent
    }
    self.smu = SMU()
    self.states, self.stateTable = self.smu.translateTable(pathToTable) # Acquire State Table from csv file
    self.sm = StateMachine(self.states, self.transitions, self.stateTable)   # Instantiate a State Machine
```

Figure 18: Initializing a State Machine

Note: The transitions dictionary in the above example maps the table entries to methods. Any method can be referenced from the state table

```
 #
 # Accept the Offer from Server
 #
 def acceptOffer(self):
    # Prepare message tuple
    acceptedMessage = (self.clientId, self.facilityId, self.arrival, self.departure, self.roomClass, self.roomCount,
                       'Accepted')

    # Publish the message
    dsParam.thePublisher.publish(10200.00, 0, '00000000-0000-0000-0000-000000000000',
                         '00000000-0000-0000-0000-000000000000', self.applicationUser,
                         self.applicationId, '', '', '', '', acceptedMessage)
```

Figure 19: Sample Transition Method

To han

```
    #
    # Process incoming messages
    #
    @pyqtSlot(float, str)
    def processMessage(self, recordType, message):
       print('Processing Record Type: %f  %s' % (recordType, message))

       self.rt = str(int(recordType * 100))
       self.message = message
       mtuple = literal_eval(message)
       if self.rt == '1010000':
          print('Received Room Offer.')

          self.sm.processEvent('Received Offer')

       elif self.rt == '1030000':
             .
             .
```


Figure 20: Example Passing Subscriber Event to State Machine

Similarly, any method launched as a result of a gui event (key click etc.) can/ should be processed by the state machine.

```
#
# Continue Reservation
#
def continueReservation(self):
    self.sm.processEvent('Select Button Clicked')
```

Figure 21: State Table Event Notification from a Gui invoked function

9 Utilities

The DS_Utility Class has several utility methods which can aid in application development. They are detailed in the following sections. To Instantiate the DS_Utility object:

```
from eventzapi import DS_UTILITY

dsu = DS_UTILITY(logger, librarianClient, parameters, userId)
```

Figure 22: Instantiating the DS_Utility

The arguments are:

- logger = A reference to a DS_Logger object instantiated prior to instantiating DS_Utility
- librarianClient = A reference to a LibrarianClient object instantiated prior to instantiating DS_Utility

9.1 Refresh Archive

The refreshArchive method will refresh a local data store from the remote Archive. This method will make a copy of the Archive locally in the location specified by dsParam.archivePath.

There are three parameters:

loggedInUser which is a string containing a user name provided by the programmer and typically sourced from a log-in module.

archivePath which is the path where the local archive should be stored

subscriptions an optional parameter. a list of record types that will populate the local archive. If no value is given the local archive will contain all record types stored in the remote archive.

```
from eventzapi import DS_Utility

dsu = DS_UTILITY(logger, librarianClient, parameters, userId)
dsu.refreshArchive(userId, archivePath, tenant, subscriptions)
```

Figure 23: Refreshing a Local Archive

9.2 updateArchive

The updateArchive method updates either a list or local archive. To reconcile all records that have

been updated or delected. will also remove and records that are not yours by comparing

the tenant records.

```
from eventzapi import DS_Utility

dsu = DS_UTILITY(logger, librarianClient, parameters, userId)
dsu.updateArchive(userId, tenant, recordList)
```

Figure 24: Reconciling a Local Archive

Arguments are:

UserID The User name or UUID identifier

tenantID

recordList An optional argument. A list of record tuples which may be the result set of a Query. In the absence of this argument the method will operate on the local archive, if there is one.

Events are stored in the archive as strings representing tuples. These tuples consist of metadata followed by the data payload of the record. The second parameter in the metadata is an integer representing the action of the event. Actions are:

0 = Insert a new recorded

1 = Update a prior recorded

2 = Delete a prior record

The local archive or a result set from a query will contain all the records of the type requested. To provide a local archive or a result set that contains only the latest updates and removes those records subsequently deleted, use the updateArchive() method.

9.3 archive

A function to append a new Event Record to the local Archive. The local archive contains string representations of the complete record tuples. It is a tab delimited csv file.

Arguments:

record	The string containing the record to be written to the local archived
pathToArchive	The string containing the path to the local archive

9.4 match

For use with QTableWidget. it finds a record in the table where the data in a column matches a target value.

Arguments;

table	The name of the table
column	The column number containing the data to be compared
target	The value of the data being searched for.

Returns:

The row number of a match or -1 if no match found.

9.5 startApplication

Called when an application starts. Generates and publishes a `9000000.00` system message.

```
dsu.startApplication(aPublisher, userId)
```

Figure 25: Start Application method call

Arguments:

aPublisher	A publisher object used to publish the message
userId	A string representing the user. This data is included in the record metadata

9.6 stopApplication

Called when an application starts. Generates and publishes a `9000001.00` system message.

```
dsu.stopApplication(aPublisher)
```

Figure 26: Stop Application method call

Argument:

aPublisher A publisher object used to publish the message.

10 Archive Services

To provide persistence an archive is created and used by an Archivist service and a Librarian service. The Archivist subscribes to all records published to the exchange and writes them to the archive in a tab delimited csv format. The archive is append only and once written, the records are indelible.

The Librarian service monitors a rpc exchange and responds to queries sent to it by the librarianClient object in any application. The queries are limited to the fields in the record metadata and only have AND relationships. The Librarian takes note of the tenantId in the query and will only return records with the proper tenantId. A tenantId of zero (UUID) will have the Librarian return all records that match the query regardless of tenantId.

The Archivist and Librarian are available at GitHub () as a zip file (EventzArchive.zip). Unzipping this file will create an EventzArchivist directory. The directory will contain archivist.py and librarian.py as well as ReadMe.txt, a sample settings.yaml file and EventzArchive.sh and hello_world.py scripts. The ReadMe.txt explains how to launch the Archivist and Librarian services. The hello_world.py script is a python program that can be run to show eventzAPI in action.

The EventzArchive.sh script will launch two terminals, one running the archivist.py program and the other running the librarian.py program. Each terminal will stay open and show a log of its operations. The archivist will create a master archive (archive.txt) that will contain every published eventz record in a csv format.

11 Examples

The example(s) described below are included in the examples folder in the GitHub repository.

11.1 Hello World

This Hello World application demonstrates the use of the eventzAPI library. It launches a small window with one button "Hello World". Pressing this button publishes a Hello World eventz message that is sent to the RabbitMQ broker specified in the settings.yaml file. The application also subscribes to this message so when the broker receives it it sends it back to the application. Upon receipt, the application displays the complete message in a message box. The message consists of a 16 item metadata piece followed by the "Hello World" string as the payload.

To run this program you need python3.8 or higher and eventzAPI (pip3 install eventzapi) and a running RabbitMq broker.

The command is:

python hello_world settings.yaml

```
from eventzAPI.eventzAPI import DS_Logger, RecordAction, ApplicationInitializer
import atexit
import tkinter
from tkinter import ttk, BOTH, RAISED, messagebox
from tkinter.ttk import ( Button, Frame, Style)
import time
import sys
'''
    This Hello World application demonstrates the use of the eventzAPI library. It launches a small
    window with one
    button "Hello World". Pressing this button publishes a Hello World eventz message that is sent to
    the RabbitMQ
    broker specified in the settings.yaml file. The application also subscribes to this message so
    when the broker
    receives it it sends it back to the application. Upon receipt, the application displays the
    complete message in a
    message box. The message consists of a 16 item metadata piece followed by the "Hello World"
     string as the payload.

    To run this program you need python3.12 or higher and eventzAPI (pip3 install eventzapi)'''
#
# Main Object
#
class Main(ttk.Frame):
    # Object constructor
    def __init__(self, parent, ds_param, user_Id, librarian_client, utilities, subscriber):
        # Attributes
        # Define any attributes needed for the application here
        self.style = Style()
        self.inter_taskQ = ds_param.inter_task_queue
        self.user_Id = user_Id
        # Get the Publisher object to enable publishing DS messages
        self.my_publisher = ds_param.the_publisher
        # Create a DS_Logger object to enable DS Logging which publishes log messages
        my_logger = DS_Logger(ds_param)
        # Create a Librarian Client object to send queries to the Librarian
        # librarian_client = LibrarianClient(ds_param, my_logger)
```

```
        # Create a DS_Utility object to allow access to DS specific utility methods
        self.my_utility = utilities
        self.my_utility.start_application(self.my_publisher, self.user_Id)  # Publish the fact
                                                                            # that this app is starting
        # Instantiate an Archiver (Local)
        archiver = None
        # Create and start a subscriber thread
        # a_subscriber = subscriber
        self.subscriber = subscriber
        # self.subscriber.start()
        # GUI
        ttk.Frame.__init__(self, parent)
        self.root = parent
        self.init_gui()
        self.say_hello()
        # Handle exiting the app.
        atexit.register(self.stopping)
        self.root.protocol("WM_DELETE_WINDOW", self.stopping)
    # Code to execute when the application stops
    def stopping(self):
        print('Stopping! The Subscriber')
        self.subscriber.stop()  # Stop the subscriber task too
        print('Stopping! The Application')
        self.root.destroy()
    # Subscriber initialization sample
    def init_gui(self):
        """Builds GUI."""
        self.root.title('Hello')
        self.root.geometry('300x100+10+20')
        self.style.theme_use("default")
        frame = Frame(self, relief=RAISED, borderwidth=1)
        frame.pack(fill=BOTH, expand=True)
        self.pack(fill=BOTH, expand=True)
        self.hello_button = Button(self.root, text = 'Say Hello', command=lambda: self.say_hello())
        self.hello_button.pack(padx = 5, pady = 5)
        pass
    def say_hello(self):
        message = ('Hello World',)
        messagePublished = self.my_publisher.publish(50000.00, message, RecordAction.INSERT.value,
                                                     userId=self.user_Id)
```

```
        pass
    def show_response(self, response):
        messagebox.showinfo('Response', response)
#
# Main code executed at the beginning
#
if __name__ == "__main__":
    # Set the application parameters
    application_id = '8ea2d01a-ae28-4f16-856e-aa8ccdd34b43'    # Identifies the application. Set by
                                                               # developer and only changed with
                                                               # version
    application_name = 'Hello World'
    path_to_settings = sys.argv[1]
    subscriptions = ['50000.00']   # Record Types subscribed to
    publications = ['50000.00']      # Record Types published
    user_id = 'You' # This user name is included in the metadata for any message published
    # Initialize to get eventz objects
    ai = ApplicationInitializer(subscriptions, publications, application_id, application_name,
                                path_to_settings, user_id)
    a_publisher, subscriber, logger, librarian_client, utilities, parameters = ai.initialize()
    root = tkinter.Tk()         # Set up GUI
    # Instantiate your Main object. This name changes when you refactor
    main = Main(root, parameters, user_id, librarian_client, utilities, subscriber)
    # Process Loop
    root.update()
    while len(root.children) != 0:
        root.update_idletasks()
        root.update()
        # Watch for and handle messages from the Subscriber Task
        if parameters.inter_task_queue.empty() == False:
            message = parameters.inter_task_queue.get_nowait()   # Get the message - non blocking
            print('Message Received: {}'.format(message))
            parameters.inter_task_queue.task_done()  # Let the queue know the message was handled
            # Put code here to handle the message
            if message[0] == '50000.00':
                main.show_response(message)
                pass
        else:
            time.sleep(.1)  # Wait to prevent the polling from using up cpu resources
```

The settings.yaml file shoild be in the same directory as the application. It not, specify the path to it as a parameter to the applications.

```
brokerExchange: amq.topic
brokerIP: codfish.rmq.cloudamqp.com
brokerPassword: 7ZJpQw7rSg1LyHLe0zx1rx2a3O27JktI
brokerUserName: lqcoztwn
brokerVirtual: lqcoztwn
currentUser: 1
deviceId: 58:00:E3:F6:96:50
deviceName: ITECH_TEST_PS
encrypt: false
firstData: 16
master_archive: False
localArchivePath: archive.txt
interTaskQueue: ''
librarianExchange: LIBRARIAN_RPC
librarianExchangeType: direct
librarianQueue: rpc_queue
location: Your location
loginDialog: LogIn.ui
myDBID: c8ff80d5-9b15-48b5-b987-213fa4149b3d
pathToCertificate: /home/you/client/cert.pem
pathToKey: /home/you/client/key.pem
pathTocacert: /home/you/client/all_cacert.pem
qt: false
rmqServer: true
tenant: 00000000-0000-0000-0000-000000000000
```

11.2 Seats Demo

To get the Seats Demo as a zip file:

1.Go to the Releases page:

•Users should navigate to the repository's Releases page. This can be done by going to the repository and clicking on the "Releases" tab, or by going to a URL like:

```
bash
Copy code
https://github.com/i-Technology/eventzAPI/releases
```

2.Find the release:

•The user locates the release to which the zip file is attached.

3.Download the zip file:

•Below the description of the release, there will be a section labeled Assets. Under this section, any attached files will be listed.
•The user simply clicks on the name of the zip file (seats.zip) to download it directly.

Here is an example of what the release page looks like:

```
Assets
seats.zip          <-- The user clicks here to download the zip file
Source code (zip)
Source code (tar.gz)
```

By clicking on the specific zip file name, the user can download only the zip file without interacting with the source code or cloning the repository.

Description

This python script demonstrates the use of eventzAPI. It generates a seat map simulating the client for a ticket selling application. When the user selects a seat, a message is generated that is sent to a publish and subscribe broker (RabbitMQ) The application subscribes to this message and when it receives it, it changes the state (colour) of the seat to yellow. selecting other seats will repeat the process. If the user clicks on the ‘Payment’ button within a timeout period the subsequent message turnaround will cause the seats selected to turn green indicating they have been purchased by the user. If the timeout expires messaging will be sent that causes the seats to revert to an ‘available’ state (grey) As seats are claimed they are added to a list displayed in a text box at the bottom of the map.

A second instance of this application, running on the users’ machine or another users’ machine, will also subscribe to these messages and ,when the other user selects a seat, the seat will turn to red indicating that the other user has claimed the seat and that it is not available to you. Similarly, a released seat will revert to ‘available’ (grey) . Seats claimed by the user will appear in red on the other users’ map. This will be true for any additional instances of the application wherever they run.

If an application is launched after other applications have claimed seats it will send a ‘seats request’ message and the other running applications will publish a ‘seat vector’ message which contains a list of all claimed seats. The new application receives these seat vectors and updates its’ map accordingly.

Code

The application uses tkinter for the gui. The startup code instantiates a tkinter root, an Auditorium object and calls the root.mainloop().

The Auditorium object initialization sets up attributes, initializes the eventz environment, and executes a run() method.

Eventz initialization consists of

1. creating a subscriptions list that lists the eventz record identifiers that the application subscribes to.
2. creating a publication list that lists the eventz record identifiers that the application publishes.
3. setting the application id (a uuid that uniquely identifies this application).
4. setting the application name.
5. setting a user_id.
6. instantiates an Application Initializer.
7. executes the Application Initializer initialize() method that generates the following objects:
    1. a Publisher – handles publishing eventz records
    2. a subscriber – a task that watches for subscribed to messages from the broker and passes them on to the main task
    3. a logger
    4. a librarian client – interfaces with the librarian service when one is used. (Not so here)
    5. a utility object – has useful utility methods
    6. a parameters object – holds system attributes needed for operation

The run() method:

1. creates a seat vector
2. publishes a seat request – so other running applications will publish their seat vectors
3. creates the gui (seat map)
4. checks the inter-task queue for any subscribed to messages – if a message arrives the process_message() method handles it. The check_queue method also down counts a purchase period that if zero, returns any claimed seats.

The application requires access to a RabbitMQ broker. CloudAMQP (https://www.cloudamqp.com/) provides free brokers that are useful for our purpose. The credentials for an instance of the Broker are placed in the file settings.yaml as follows:

brokerExchange: amq.topic

brokerIP: <Hosts> e.g.codfish-01.rmq.cloudamqp.com

brokerPassword: <Password>W8WyOQoRRCfAIMw_lExq0h--g38eU7Vy

brokerUserName: <Username>zuklzxqa

brokerVirtual: <Username>zuklzxqa

A complete settings.yaml file is in this repository. use it as an example.

Deployment

1. Establish a RabbitMQ broker at cloudamqp.com (The free one is fine).
2. unzip the seats.zip file creating s Seats directory with the relevant files.

3. Edit the settings.yaml to provide the broker credentials.
4. With a terminal in the seats directory:
   a) pip install eventzAPI
   b) pip install pika
   c) pip install pyYaml
   d) python seats.py
   e) Buy some seats
5. With another terminal in the seats directory:
   a) python seats.py
6. Buy seats in each running instance and see the interaction.