I-TECHNOLOGY INC.

Eventz
Python Programmer's Manual

May 06, 2021
Revision 2.2

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

Table of Contents

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 is 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 record 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.

Microservices have the option of creating a local archive that contain 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. Download and run Docker (https://www.docker.com)
3. Get Eventz Archivist and Eventz Librarian from Docker Hub
4. Eventz Archivist and Eventz Librarian running on a machine accessible through the network
5. Python3.9 installed (https://www.python.org/download/releases/3.9/) 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, QT5 [http://pyqt.sourceforge.net/Docs/PyQt5/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
- publications
- applicationId
- applicationName
- path_to_settings
- user_id = user_id

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
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
userId	String	Name or UUID
publishDateTime	Date Time	`datetime.datetime.utcnow().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

Field	Data Type	Notes
sessionId	UUID	Provides a link between messages that belong to a session.
userMetadata1	User defined	Data the programmer wants indexed by the Librarian for queries
userMetadata2	User defined	
userMetadata3	User defined	
userMetadata4	User defined	
userMetadata5	User defined	

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 identfier]
firstData: [Offset to the data in a message, currently 16]
qt: [If Qt is used for 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.

```
dsInit = DS_Init(applicationId, applicationName)    # Create the object
dsParam = dsInit.getParams('settingsRemote1.yaml', routingKeys, publications, None)
```

Figure 4: Code initializing the parameters object

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 dsParam. 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. Tgis will be useful if you need to know the record_id or other metadata fielf values assigned by the api.

Table 1: publish method arguments

Field	Description
recordType	The floating point number that identifies the record.
dataTuple	The record data as a tuple.
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 versionedrecord (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

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.

To create the subscriber thread and start it:

```
from EventzAPI import *


# Create and start a subscriber thread
aSubscriber = SubscriberFactory()
subscriber = aSubscriber.makeSubscriber(dsParam, self.userID,
                                        archiver, self.myUtility)
subscriber.start()
```

Figure 6: Code to create and start a subscriber thread

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, recordType, message):

   postMessage(recordType, message)
   print('Processing Record Type: %f  %s' %(recordType, message))

   m = re.sub('[()]', '', message)
   record = tuple(m.split(','))
   aDevice = record[g.firstData]
   deviceName = record[g.firstData + 1]
   location = record[g.firstData + 2]
   applicationId = record[g.firstData + 3]
   applicationName = record[g.firstData + 4]
   aTime = record[g.firstData + 5]
      .
      .
```

Figure 7: Qt Slot for Record Handling

We connect to the slot as follows.

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

Figure 8: 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.

```
root = tkinter.Tk()


while len(root.children) != 0:
    root.update_idletasks()
    root.update()
    if dsParam.interTaskQueue.empty() == False:
        message = dsParam.interTaskQueue.get_nowait()
        print('Message Received: {}'.format(message))
        dsParam.interTaskQueue.task_done()

        # Process the each record type subscribed to here

        if message[0] == '6030.00':
            print (f'Received room cleaned: {message[13]}')
            frontDesk.addRoomToList(message[13])
    else:
        time.sleep(.1) # Wait to prevent the polling from using up cpu
```

Figure 9: Tkinter event loop interdiction

5.3 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.utcnow().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

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 and receiving results back. To instantiate the LibrarianClient:

```
Import eventzapi

librarianClient = LibrarianClient(dsParam, logger)
```

Figure 11: Instantiating a LibrarianClient

Querying is accomplished using the librarianClient .call() method detailed in figure 10 below

```
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(',')
The result set returned from the call method is a list of qualifying records in tuples.
           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

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

EQ, GE, GT, LE, LT

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(dsParam)
```

Figure 16: Instantiate a DS_Logger

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

To create a log message:

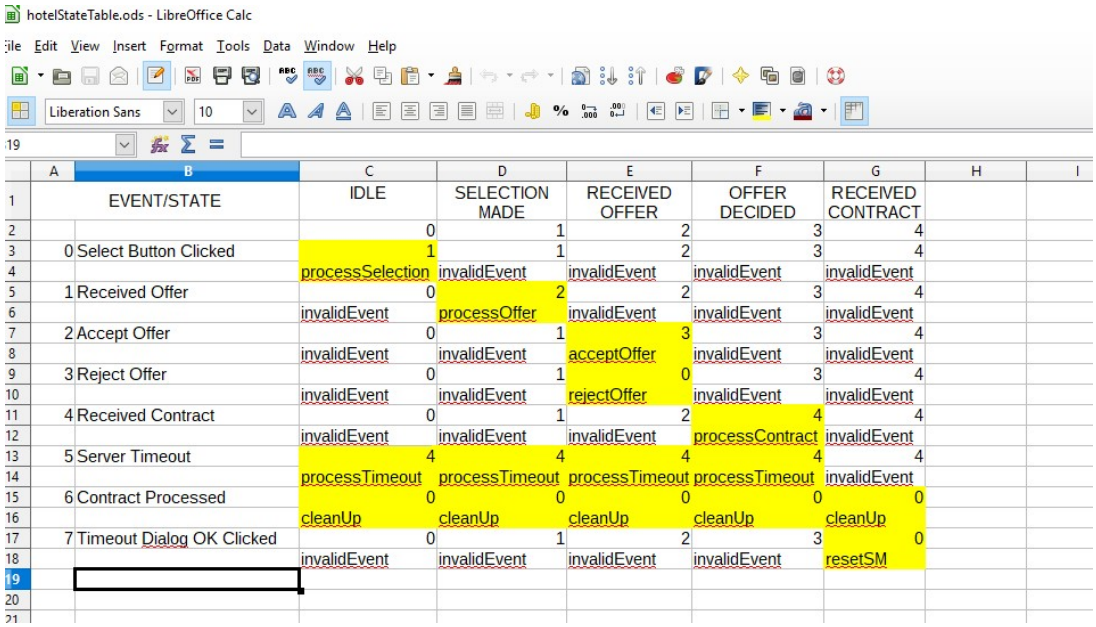
```
logger.log('I-Tech',100, 'INFO', 0, 'Not Really an Error. Just a test.')
```

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 handle messages subscribed to, a processMessage method can call processEvent()

```
 #
# 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, dsParam, 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, dsParam, 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, dsParam, 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 nd Librarian are each available in Docker containers in Docker Hub. Docker needs to be present and running on the machine where the services are to be mounted. Below are commands to be run in a terminal that has navigated to the directory where you want the Archive to reside. The docker.run command has as a parameter the path to the directory where the Archive resides. An example is shown in italics.

```
On all OS's:
docker pull eventz/eventzapi_tools:archivist
docker pull eventz/eventzapi_tools:librarian

On Linux or OSX:    (separate terminal for each line)
docker run --rm -v $(pwd):/var/lib/ eventz/eventzapi_tools:archivist
docker run --rm -v $(pwd):/var/lib/ eventz/eventzapi_tools:librarian

On Windows:(separate terminal for each line)
docker run --rm -v C:\Users\sbj31\Desktop:/var/lib/ eventz/eventzapi_tools:archivist
docker run --rm -v C:\Users\sbj31\Desktop:/var/lib/ eventz/eventzapi_tools:librarian
```

Figure 27: Docker Commands for Archivist & Librarian

Each terminal will stay open and show a log of its operating service.

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)