Hotel Application Tutorial

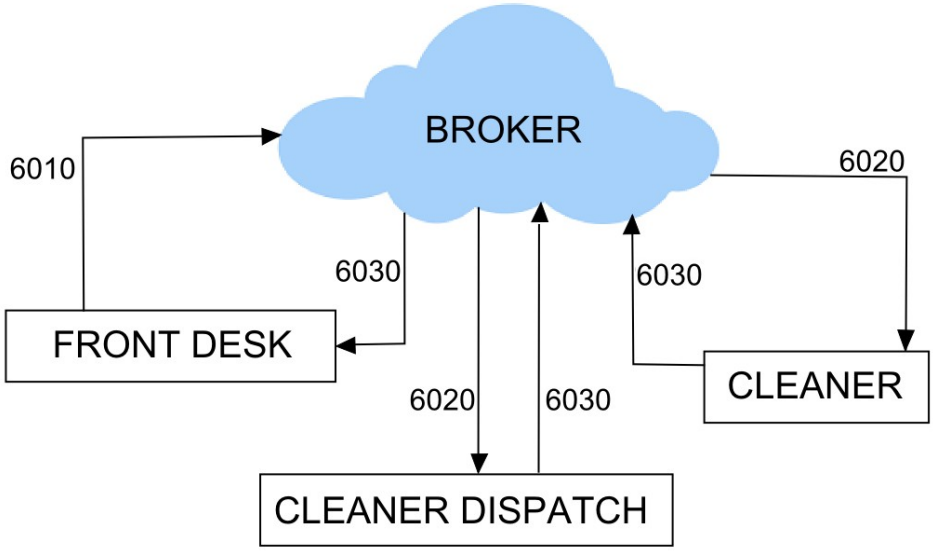
The hotel application simulates a hotel cleaning service. It showcases the unique abilities of the Eventz framework.

Overview

All processing is done with 3 microservices (i.e. Front Desk, Cleaner Dispatch and Cleaner) These microservices communicate using publish and subscribe messaging though a broker. Each microservice in this example has a Graphical User Interface (GUI)

In this example, when a guest checks out, the Front Desk publishes a “Room Checked out” message to the broker. In this example the message number is 6010 (could be any number) saying that a room has checked out. Within this message is the room number that has been checked out.

The Cleaner Dispatch microservice subscribes to and receives this message (6010) from the broker. The Cleaner dispatch microservice receives and decodes the 6010 message to get the room number. It appends it to a list that is exposed through a drop down control to the user. The user selects a room from the list amd it publishes a message containing the selected room number (6020) notifying that there is a room to clean. The Cleaner microservice subscribes to and receives the 6020 message. Once the room is cleaned, the Cleaner user selects that room and the Cleaner microservice publishes a message (6030) reporting that the room has been cleaned. The Front Desk subscribes to and receives this message and adds the cleaned room to its list of rooms available for use.



Coding the Microservices

A template program, Stub.py, is provided that lays the groundwork for each microservice. It sets up the environment for a DSI microservice.

The basic overview layout of Stub.py is shown below. Colour coding separates the various functional parts of the program

```
imports

class Main(ttk.Frame):                    # Rename this to an appropriate name e.g. FrontDesk(ttk.Frame)
        Initialize parameters
        Instantiate a Publisher
        Instantiate a Logger
        Instantiate a LibrarianClient (if you are going to query the main archive)
        Instantiate a Utility and publish a starting record
        Instantiate a local  Archiver (or not)
        Instantiate a subscriber and start it up
        Initialize the Subscriber
        Establish exit handling
        Methods to implement the application


# Execution starts here

if __name__ == "__main__":
        Get the operating parameters
        Instantiate the Main object
            Infinite loop servicing the GUI and the subscriber
```

Where:

`Initialize parameters` acquires the settings needed by the program to connect to the Broker and perform other functions in the api.

`Instantiate a Publisher` A publisher is an object that provides methods for publishing a message. It adds metadata to the message and sends it to the Broker

`Instantiate a Logger` The Logger object is used to publish logged messages. These messages are system messages that the programmer can use to alert system managers that there is a problem with the microservice that needs attending to. This logging facility is separate from other Python logging.

`Instantiate a LibrarianClient` The Librarian supports querying a central archive for previously published messages. It is a utility not used in the Hotel application but is included for other users of the template who need this service.

`Instantiate a Utility and publish a starting record` The Utility object provides utility methods for use by the programmer. One utility publishes a startup message. This message can be subscribed to by a system monitor which can keep track of which microservices are running. Similarly a stopping message can be published.

`Instantiate a local Archiver (or not)` The programmer has the ability to create a local data store (archive) for the messages it is subscribing to. It is not used in the Hotel application.

`Instantiate a subscriber and start it up` The Subscriber launches a separate thread that establishes subscriptions with the Broker. When a message that the microservice is subscribing to is published, The Broker inserts it into the queue that the subscriber is watching. The subscriber takes the message and passes it to the main thread where the programmer can deal with it. There is a subset of system messages that the Subscriber deals with without main thread involvement. An example of this would be a ping message.

`Initialize the Subscriber` This code creates the Subscriber and connects its components to methods in the program.

`Process Loop` There is a process loop that services the gui and the subscriber. Execution ultimately ends up in this loop.

The complete code for Stub.py follows. The colours coordinate with the previous discussion.

Stub.py

```
from dsapi import Publisher, Gui, DS_Logger, DS_Utility,
                 DS_Init, LibrarianClient                    # Where the magic happens

import atexit                                                # Handles graceful exit
import tkinter                                               # Gui
from tkinter import ttk, Label, StringVar, messagebox        # Gui components
import uuid

#
# Main Object
#
# Refactor this object to your custom object for your application
#
# The parameters needed to instantiate this object could come from a global file, command line
parameters or Inputs.
#
class Main(ttk.Frame):
    # Object constructor
    def __init__(self, parent, dsParam, userID,):
        # Attributes
        # Define any attributes needed for the application here.
        self.interTaskQ = dsParam.interTaskQueue
        self.userID = userID
        # Get the Publisher object to enable publishing DS messages
        self.myPublisher = dsParam.publisher
        # Create a DS_Logger object to enable DS Logging which sends log messages to SystemMonitor
        myLogger = DS_Logger(dsParam)
        # Create a Librarian Client object to enable sending of queries to the Librarian
        librarianClient = LibrarianClient(dsParam, myLogger)
        # Create a DS_Utility object to allow access to DS specific utility methods
        self.myUtility = DS_Utility(myLogger, librarianClient, dsParam, self.userID)
         # Create a DS_Utility object to allow access to DS specific utility methods
        self.myUtility.startApplication(self.myPublisher, self.userID)
        # Instantiate an Archiver (Local)
        archiver = None
        # Create and start a subscriber
        subscriberFactory = SubscriberFactory()
        self.subscriber = subscriberFactory.makeSubscriber(dsParam, self.userID, archiver) #Create a
Subscriber Thread
        self.subscriber.start()
        atexit.register(self.stopping)
        # Subscriber code here
        ttk.Frame.__init__(self, parent)
        self.root = parent
        self.init_gui()

    # Code to execute when the application stops
    def stopping(self):
        print('Stopping!')
        self.subscriber.stop()          # Stop the subscriber task
        self.root.destroy()

    # Subscriber initialization sample
    def init_gui(self):
```

```
        """Builds GUI."""
        # self.root.title('Front Desk')
        # self.root.geometry('300x300')
        # mLabel = Label(self.root, text='Room Checked Out').pack()
        # mEntry = Entry(self.root, textvariable=self.rmCO).pack()
        # mButton = Button(self.root, text='Room C.O.',
                    command = self.buttonMethod).pack()
        pass

    # Button method
    def buttonMethod(self):
        pass
#
# Main code executed at the beginning
#
if __name__ == "__main__":

    # Set the application parameters

    Qt = False              # True if we are using QT Tasks
    firstData = 13

    applicationId = ' unique uuid here'
    applicationName = 'Application name and version'
    applicationUserName = 'A default user name'
    routingKeys = ['Record Types subscribed to']      # Record Types subscribed to
    publications = ['Record Types published']     # Record Types published

    # Initialize DS ( Get dsParams from settings.yaml)
    dsInit = DS_Init(applicationId, applicationName)    # Create the object
    dsParam = dsInit.getParams('settings.yaml', routingKeys, publications, None)  # Get the parameters
    dsParam.myDBID = str(uuid.uuid4())      # Overwrite myDBID

    # Instantiate Subscriber
    root = tkinter.Tk()
    # Instantiate your Main object. This name changes when you refactor
    main = Main(root, dsParam, 'Default User')

    # Process Loop
    while True:
        # Handle the gui
        root.update_idletasks()
        root.update()
        # Watch for and handle messages from the Subscriber Task
        if dsParam.interTaskQueue.empty()==False:
            message = dsParam.interTaskQueue.get_nowait()   # Get the message - non blocking
            print('Message Received: {}'.format(message))
            # Put code here to handle the message
            dsParam.interTaskQueue.task_done()  # Let the queue know the message was handled if
                    message [0] == 'Message Type':
                        pass
```

Notes:

The a applicationId variable is hard coded since it is not to change. This value is put in metadata prepended to published messages from this application.

The a applicationName variable is similarly hard coded but it can contain a version number. This data in not part of the metadata but it is included in the Ping system message which will be discussed later.

The applicationUserName is also in the message metadata. It is provided here as a default. If an application has user security (Log in/out) then the application will modify this value depending upon who it has identified as using it. It is a string and could be a name or an id number.

routingKeys is a list of record identifiers that the Subscriber uses in order to subscribe to messages from the message broker.

publications is a list of record identifiers that this application may publish. This data is included in Ping system messages so that a system monitor can determine which applications are producing and subscribing to which record types.

DS_Init is an object that contains a method, dsInit.getParams, that gets application parameters from a yaml file and internal data to create a list of parameters, dsParam, that are used by the api to effect connection to the broker and other tasks.

The contents of an example yaml file is shown below:

settings.yaml

```yaml
brokerExchange: amq.topic
brokerIP: 192.168.0.14
brokerUser: alpha1
password: walLy34
deviceId:
deviceName: ITECH_12
location: London Ontario
encrypt: false
firstData: 13
librarianExchange: LIBRARIAN_RPC
librarianExchangeType: direct
librarianQueue: rpc_queue
localArchivePath: ''
myDBID: e5a0d9ef-e1b2-4c1f-8283-ced6208ae7e9
pathToCertificate: ''
pathToKey: ''
qt: false
rmqServer: false
tenant: 00000000-0000-0000-0000-000000000000
userName: i-tech
```

Where:

brokerExchange, brokerIP, brokerUser and password are used by the api to connect to the RabbitMQ server.

deviceId If this parameter is blank the api will get the mac address of the machine the application is running on.

deviceName is used to identify the device the application has been installed in.

location is a string identifying the location of the device that the application is installed on and is set by the installer.

encrypt true or false depending upon whether the messages are to be encrypted or not

firstData is the offset to the first non-metadata field in the message.

librarianExchange, librarianExchangeType and librarianQueue are used by the api to establish a connection to the Librarian which is used to query the Archive.

localArchivePath tells the api that it is to maintain a local archive. This archive is a file located by the path provided in the parameter. The subscriber will put all messages subscribed to into the local archive. This includes the messages the application has published itself. The developer is responsible for code to access this archive and extract data. This file should only be written to by the subscriber. Records are appended as they are received.

myDBID (deprecated) is a uuid identifying the local data store used as a local archive.

pathToCertificate and pathToKey are used by the api to find certificates if encryption is enabled.

qt determines if you are using Qt for the gui. Qt uses signals and slots to communicate events to the main thread. Other guis use an inter task queue to pass events from the gui or subscriber to the main thread.

rmqServer ????

tenant In multi-tenanted systems the tenant setting provides a uuid that identifies which tenant this instance of the application belongs to. The tenant data is part of the ecord metadata and provides a mechanism where the user will not receive nother tenants data.

userName is also in the message metadata. It is provided here as a default. If an application has user security (Log in/out) then the application will modify this value depending upon who it has identified as using it. It is a string and could be a name or an id number.

The stub is used in FrontDesk.py below. This code implements the front desk microservice. We have highlighted code that has been added to the stub code and also modifications to that code.

FrontDesk.py

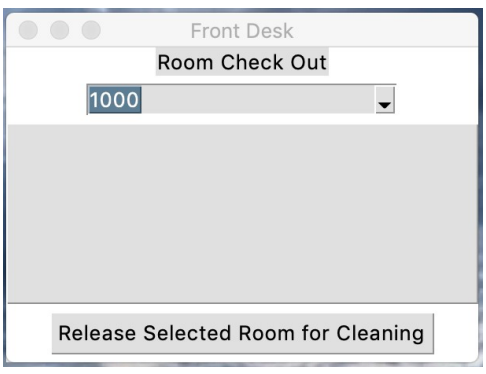
```
from dsapi import Publisher, Gui, DS_Logger, DS_Utility, DS_Init, LibrarianClient

import atexit
import tkinter
from tkinter import ttk, RIGHT, BOTH, RAISED, StringVar, messagebox
from tkinter.ttk import Label, Button, Frame, Style
from queue import Queue
import uuid

"""Front Desk"""
class FrontDesk(ttk.Frame):
    def __init__(self, parent, dsParam, userID):
        self.interTaskQ =dsParam.interTaskQueue
        self.userID = userID
        self.availableRooms = []
        # Create a Publisher object to enable publishing DS messages
        self.myPublisher = Publisher(dsParam)
        # Create a DS_Logger object to enable DS Logging which sends log messages to SystemMonitor
        myLogger = DS_Logger(dsParam)
        # Create a Librarian Client object to enable sending of queries to the Librarian
        librarianClient = LibrarianClient(dsParam, myLogger)
        # Create a DS_Utility object to allow access to DS specific utility methods
        self.myUtility = DS_Utility(myLogger, librarianClient, dsParam, self.userID)
        # Create a DS_Utility object to allow access to DS specific utility methods
        self.myUtility.startApplication( self.myPublisher, self.userID)
        # Instantiate an Archiver (Local)
        archiver = None
        # Instantiate a logger
        self.logger = DS_Logger(dsParam)
        # Create and start a subscriber
        gui = Gui(dsParam.qt)
        self.subscriber = gui.makeSubscriber(dsParam, 'HotelFD', archiver)  # Create a Subscriber Thread
        self.subscriber.start()
        # Initialize
        self.availableRooms = self.initializeRoomLoad()
        # Gui
        ttk.Frame.__init__(self, parent)
        self.root = parent
        self.init_gui()
        # Handle exiting the app.
        atexit.register(self.stopping)
        self.root.protocol("WM_DELETE_WINDOW", self.stopping)

    def stopping(self):
        print('Stopping!')
        self.subscriber.stop()  # Stop the subscriber task too

    def init_gui(self):
        """Builds GUI."""
        self.root.title('Front Desk')
        self.root.geometry('300x200+10+20')
        self.style = Style()
        self.style.theme_use("default")
```

Refactored Main class to FrontDesk

Variable needed by the application

Create the User Interface

When the window is closed call stopping()



```
        mLabel = Label(self.root, text='Room Check Out').pack()
        self.mCombo = ttk.Combobox(self.root, state='readonly')
        self.combo_post_command()
        self.mCombo.current(0)
        self.mCombo.pack(padx = 5, pady = 5)
        frame = Frame(self, relief=RAISED, borderwidth=1)
        frame.pack(fill=BOTH, expand=True)
        self.pack(fill=BOTH, expand=True)
        self.cleanButton = Button(self.root, text = 'Release Selected Room for Cleaning',
                                  command = lambda arg=self.mCombo: self.roomRelease(arg))
        self.cleanButton.pack(padx = 5, pady = 5)

    def combo_post_command(self):
        self.mCombo['values'] = self.availableRooms
        self.mCombo.set('')

    def roomRelease(self, evt):
        txt = self.mCombo.get()  # get current text
        if len(txt) > 0:
            roomNumber = int(txt)
        else:
            roomNumber = 0
        found = False
        for index, room in enumerate(self.availableRooms):
            if room == roomNumber:
                self.availableRooms.pop(index)
                release = (roomNumber,)
                messagePublished = self.myPublisher.publish(6010.00, 0, 0, self.userID, "", "", "", "", "", release)
                print(messagePublished)
                found = True
                self.combo_post_command()
                self.mCombo.current(0)
                print('Released room: {} for cleaning'.format(roomNumber))
                break
        if found == False:
            messagebox.showerror('ERROR', 'Room: {} Not in list!'.format(roomNumber))
            self.logger.log(self.userID, 1010, 0, 0, 'Invalid Room Number: {}'.format(roomNumber))


    def initializeRoomLoad(self):
        return list(range(1000, 1020))  # 20 Rooms

    def addRoomToList(self, room):
        if room in self.availableRooms:
            messagebox.showerror('ERROR', 'Room: {} in Available Rooms list!'.format(room))
            self.logger.log(self.userID, 1020, 0, 0,
                            'Attempt to duplicate Available Rooms list member. Room: {}'.format(room))
        else:
            self.availableRooms.append(room)
            self.combo_post_command()

if __name__ == "__main__":
    # Set the application parameters
    applicationId = '35c87ca6-e9e6-4ae3-b10c-942d4508208a'
    applicationName = 'Front Desk Microservice v1.0'
    applicationUserName = 'I-Tech'
    routingKeys = ['6030.00']      # Record Types subscribed to
    publications = ['6010.00']     # Record Types published

    # Initialize DS ( Get dsParams from settings.yaml)
    dsInit = DS_Init(applicationId, applicationName)    # Create the object
    dsParam = dsInit.getParams('settings.yaml', routingKeys, publications)  # Get the parameters
    dsParam.myDBID = str(uuid.uuid4())      # Overwrite myDBID
    root = tkinter.Tk()
    frontDesk = FrontDesk(root, dsParam, 'Default User')

    # Process Loop
    root.update()
    while True:
        root.update_idletasks()
        root.update()
        if dsParam.interTaskQueue.empty() == False:
            message = dsParam.interTaskQueue.get_nowait()
```

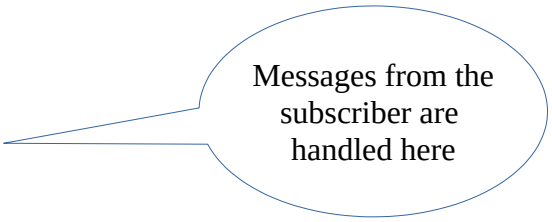
The message data is formed here

The message is published here

This is the metadata part

The api sets the value of myDBID but 3 programs use the api so we must set it uniquely in each program

```
print('Message Received: {}'.format(message))
dsParam.interTaskQueue.task_done()
if message[0] == '6030.00':
    print ('Received room cleaned: {}'.format(message[13]))
    frontDesk.addRoomToList(message[13])
```




```
self.availableRooms = [ ]
```
An empty list that will hold the room inventory

```
self.rmCO = StringVar
```
creates a Publisher object to enable publishing DS messages.

```
self.logger = DS_Logger(dsParam)
```
creates and starts a subscriber.

```
self.availableRooms = self.initializeRoomLoad()
```
means...

```
self.root.title('Front Desk')
```
means...

```
self.root.geometry('300x300')
```
means...

```
self.style =Style()
```
means...

```
self.style.theme_use("default")
```
means...

```
mLabel = Label(self.root, text='Room Check Out').pack()
```
means...

```
self.mCombo = ttk.Combobox(self.root, state='readonly')
```
means...

```
self.combo_post_command()
```
means...

```
self.mCombo.current(0)
```
means...

```
self.mCombo.pack(padx = 5, pady = 5)
```
means...

```
frame = Frame(self, relief=RAISED, borderwidth=1)
```
means...

```
frame.pack(fill=BOTH, expand=True)
```
means...

```
self.pack(fill=BOTH, expand=True)
```
means...

```
self.cleanButton = Button(self.root, text = 'Release Selected Room for
Cleaning', command = lambda arg=self.mCombo: self.roomRelease(arg))
```
means...

```
self.cleanButton.pack(padx = 5, pady = 5)
```
means...

```
def combo_post_command(self):
```
means...

```
self.mCombo['values'] = self.availableRooms
```
means...

```
self.mCombo.set('')
```
means...

```
def roomRelease(self, evt):
```
means...

```
txt = self.mCombo.get ():
```
means get the current text.

```
If len(txt)>0:
        roomNumber = int(txt)
else:
        roomNumber = 0
```
means...

```
found = False
```
means...

```
for index, room in enumerate(self.availableRooms):
```
means...

```
If room == roomNumber:
        self.availableRooms.pop(index)
        release = (roomNumber,)
        messagePublished = self.myPublisher.publish(6010.00, 0, 0, self.userID,
"", "". "", "", "", release)
                print(messagePublished)
```
means...

```
found = True
self.combo_post_command()
self.mCombo.current(0)
print('Released room: {} for cleaning'.format(roomNumber))
break
```
means...

```
        if found == False:
        messagebox.showerror('ERROR', 'Room: {} Not in list!'.format(roomNumber))
        self.logger.log(self.userID, 1010, 0, 0, 'Invalid Room Number:
{}'.format(roomNumber))
```
means...

```
        def initializeRoomLoad(self):
                return list(range(1000, 1020))
```
means 20 Rooms...

```
        def addRoomToList(self, room):
                if room in self.availableRooms:
                     messagebox.showerror('ERROR', 'Room: {} in Available Rooms
list!'.format(room))
                            self.logger.log(self.userID, 1020, 0, 0,
                                    'Attempt to duplicate Available Rooms list member.
Room: {}'. format(room))
```
means...

```
        else:
                self.availableRooms.append(room)
                       self.combo_post_command()
```
means...