**Keypath Education**

**Exercise 1 Design Document**

**Message Sending Service**

Mike Sullivan

Applicant

September 14, 2020

Draft Revision 0.1

1. **Purpose**The purpose of this document is to describe the implementation of a message sending service at a level appropriate for a software development team to build out the application.
2. **Scope**The scope of the work includes implementation for a Window Service that processes queued text messages using a third-party REST API. It includes the appropriate status and delivery updates to the database and Windows Event Log.  
     
   A system for monitoring / notification of abnormal performance is described. Delivery of that work is out of scope for this design document. That is an important task and should merit development consideration.
3. **Design Overview**
   1. **Draft Status**This document is in draft status. Please review the Pending and Future Design consideration section at the end of the document. Some items may need resolution to advance this design from draft status.
   2. **Description**The Message Sending Service reads queued text message requests from a database and delivers SMS text messages through a third-party Web API.
   3. **Technologies**
      1. **.NET Framework (4.5+) Windows Service**The Message Sending Service application will be a Windows Service written in C# designed to run concurrently across on a minimum of two Windows Servers.
      2. **On Premise Windows Servers**The target platform to host the application is Microsoft Windows Server.
      3. **Windows Application Event Log**The service will log status, warning, and error information the Windows Application Event Log. It’s expected that the log can help trouble-shoot issues the Message Sending Service and its performance.
      4. **On Premise Microsoft SQL Server**The database hosting the message requests resides on Microsoft SQL Server. The database will contain artifacts to determine if performance goals are met.
      5. **REST API**The service to construct and send the messages is a third-party REST API. The endpoint is not yet known. The Message Sending Service will perform an http POST to the endpoint. The message body will contain JSON containing the “From” and “To” numbers as well as the message text.
      6. **Multi-threaded**In light of the Third-Party REST API performance constraints, our desired performance goals, and the resource stack dedicated to the application, we must make the process that interfaces with the third-party API multi-threaded.  
           
         HttpClient is thread safe in .NET 4.5+
   4. **Performance Goals**
      1. Concurrent instances of the service running on two service able to accommodate and deliver:
         1. 10,000 messages a day.
         2. An expected peak load of 3,000 messages in an hour.
      2. 10 Minute Delivery: Messages delivered within 10 minutes of creation.
   5. **Third-Party REST API** 
      1. **Constraints**We are unable to directly influence these constraints. We can review our design should our business partner address them.
         1. Poor Up-Time: 95%. The service is unavailable for an average of 72 minutes/day.
         2. 3 Second Turnaround: The service takes about 3 seconds to process each message.
      2. **Performance**No Bandwidth limitations (when up and running)
   6. **Architecture**
      1. **Windows Service**The Windows Service application contains and controls the execution of the end-to-end process.
      2. **Data Model**The classes need to organize the messages, their statuses, and the logging of message delivery attempts.
      3. **Data Storage**The interface for reading, writing, and updating the database.
      4. **Resiliency / Self-Healing**The service ought to be able to continue and catch-up on its work after an outage or failure.
      5. **Monitoring / Diagnostics (note: out of scope)**A subsystem to analyze and produce performance reports. It should also understand abnormal performance and initiate notification of key technical and operational personnel when detected.  
           
         This work is out of scope for purposes of this design document.  
           
         The work to log appropriate information for these reports and decisions to be make is in scope for this design.
4. **Ambiguity**The Third-Party API returns a Failed Status labeled ‘Not valid by Time Zone’. As of this writing, it is not understood why that is a state or what it means. These requirements treat this condition as temporary and specify attempts to re-deliver the message. That appears to bring less harm that considering the message dead.  
     
   If we are not able to ascertain its meaning, the data logged by the application should prove or disprove our assumption. Once a better meaning is known or derived, we can adjust the behavior of the application.
5. **Data Model**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Message** | |  | **DeliveryStatus** | |  | **DeliveryLog** | |
| ID | numeric(18,0) identity |  | ID | numeric(18,0) PK |  | ID | numeric(18,0) identity |
| To | numeric(18,0) |  | StatusDesc | nvarchar(50) |  | MessageID | numeric(18,0) FK |
| From | numeric(18,0) |  |  |  |  | DeliveryStatusID | numeric(18,0) FK |
| Message | nvarchar(100) |  |  |  |  | API\_Message | nvarchar(50) |
| DeliveryStatusID | numeric(18,0) FK and Index |  |  |  |  | AdditionalInfo | Nvarchar(1024) |
| DeliveryAttempts | int |  |  |  |  | CreationDT | DateTime |
| CreationDT | DateTime |  |  |  |  |  |  |
| UpdateDT | DateTIme |  |  |  |  |  |  |
| DeliveryDT | DateTIme |  |  |  |  |  |  |

* 1. **Message Table**Records are added to this table from other sources not described here. A secondary on index DeliveryStatusID/ID may serve to speed data retrieval as the table grows.

|  |  |
| --- | --- |
| Column Name | Purpose |
| ID | Identity Column |
| To | Phone number of message recipient |
| From | Phone number of message sender |
| Message | The message to deliver |
| DeliveryStatusID | Current delivery status (FK to DeliveryStatus) |
| DeliveryAttempts | The number of attempts to deliver this message |
| CreationDT | Timestamp when row was added to table |
| UpdateDT | Timestamp when row was last updated |
| DeliveryDT | Timestamp when row was successfully delivered |

* 1. **DeliveryStatus Table**

The **DeliveryStatus** table is populated with these row values (and these row values only):

|  |  |
| --- | --- |
| ID | StatusDesc |
| 0 | Successfully Delivered |
| 1 | Queued |
| 2 | In Progress |
| 3 | Retry – Invalid Time Zone |
| 4 | Retry – API Unresponsive |
| 5 | Retry – Http Error |
| 6 | Retry – Network Error |
| 7 | Dead – Invalid Phone Number |

0: Happy path: The message was successfully delivered

1: The message awaits delivery.

2: The Message Sending Service is processing this row. This status allows 2 or more Message Sending Service instances to run concurrently and operate on separate sets of messages.

3: The Third-Party API returned this failure status for the delivery attempt. Our current understanding of this failure dictates that a delivery retry is appropriate.

4: The Third-Party API was not responding when the delivery attempt occurred. The Third-Party API is down for an average of 72 minutes / Day. A delivery retry should be attempted in subsequent

5: The Third-Party API return this failure status for the delivery attempt. Since the phone number is invalid, message delivery is not possible. Consider the request dead.

* 1. **DeliveryLog Table**

|  |  |
| --- | --- |
| Column Name | Purpose |
| ID | Identity Column |
| MessageID | Foreign Key to Message Table |
| DeliveryStatusID | Delivery status for this attempt (FK to DeliveryStatus) |
| API\_Message | Verbatim response returned by the Third-Party API |
| CreationDT | Timestamp when row was added to table. It is not expected that these rows get updated. |

1. **Data Transfer Objects**
   1. QueuedMessageRequestDTO

|  |  |  |
| --- | --- | --- |
| Column Name | Type | Description |
| DeliveryAttempts | Int | Number of attempts made to deliver message |
| MessageID | int | ID of Message Row |
| To | int | Phone number of message recipient |
| From | int | Phone number of message sender |
| Message | string | Message to send |

* 1. LogDeliveryDTO

|  |  |  |
| --- | --- | --- |
| Column Name | Type | Description |
| MessageID | int | ID of Message Row |
| DeliveryStatusID | int | DeliveryStatusID of delivery attempt |
| API\_Message | string | Verbatim status message returned by API |
| AdditionalInfo | string | Additional Info (http error info) |

* 1. MessageDTO

|  |  |  |
| --- | --- | --- |
| Column Name | Type | Description |
| To | int | Phone number of message recipient |
| From | int | Phone number of message sender |
| Message | string | Message to send |

1. **Message Sending Service Application**
   1. Configuration Items & Default Values

|  |  |  |
| --- | --- | --- |
| Configuration Item | Description | Suggested Default |
| EventSource | Source name for writing application event log rows | KeyPathMessageService |
| LogName | Name of the log the event source’s entries are written to | KeyPathMessageLog |
| Threads | Number of threads to use for HttpClient POSTs to Third-Party API | 2 |
| PollingPeriodSeconds | Poll Message Queue every x seconds and process any message requests returned | 60 |
| PollingRowFetchMax | Maximum number of queued requests to process | 30 |
| Third-Party API Endpoint | Endpoint for the third-party message sending API | Cannot be defaulted |

With these defaults and an expected instance count of two services running across two servers, the service should be able to meet performance expectations. Of course, the defaults can be adjusted if necessary.

Configuration can be done with app.config values, registry entries, or command-line (startup) parameters. It’s suggested to start with app.config for simplicity and visibility.

* 1. Initialization / Startup / Installer
     1. Service should start automatically
     2. Service should run under an account that allows minimum permissions to execute, write to the event log, and consume the database for reads, inserts, and updates.
     3. Service should use and adhere to configuration parameters.
     4. Upon successful start, write to Informational Message to Event Log.
  2. Service Events: Pause, Stop, Continue, Shutdown  
     Write appropriate Informational Message to Event Log.
  3. DataAccess   
     Construct the DataAccess class(es) with an injectable DatabaseContext that can provide in-memory data for testing. Log any exceptions to the Application Event Log.  
       
     The following methods should support data access needs
     1. *public void RequeueStaleInProgressRows()*To assist with the self-healing need of the application, this method updates all Message rows to ‘Queued’ that correspond to this filter
        + ‘In Progress’ status and more than 10 minutes difference between CreationDT and UpdateDT [Note: These likely are records left in ‘In Progress’ status by a failed process],   
          or
        + ‘Retry – API Unresponsive, or
        + ‘Retry – Invalid Time Zone’, or
        + ‘Retry – Http Error’, or
        + ‘Retry – Nez Error’
     2. *public List<QueuedMessageRequestDTO> GetQueuedMessageRequests(int maxRows)*
        1. Selects up to maxRows rows with a DeliveryStatusID of ‘Queued’ (1)
        2. Retrieves maxRows rows in DeliveryAttempts, MessageID order which makes it chronologically favor the new message requests. Note: May need to reconsider retrieval order if failed attempts cause large delays in delivery of those messages
        3. Updates returned rows as follows
           1. DeliveryStatusID => ‘In Progress’
           2. Increments DeliveryAttempts
           3. UpdateDT refreshed with current DateTime
        4. Runs in a SQL transaction to support concurrent access from multiple service instances. That way everyone gets the correct unique set to process.
     3. *void UpdateMessageDeliveryStatus(LogDeliveryDTO logDeliveryDTO)*Updates DeliveryStatus of Message row identified by MessageID with newDeliveryStatus. The UpdateDT is also updated. If the newDeliveryStatus indices ‘Successfully Delivered’ the DeliveryDT is updated.  
          
        This method must insert a DeliveryLog row also.
  4. Message Delivery   
     This class manages the integration with the Third-Party REST API that sends the message. Because the throughput of the API will not allow us to remain within our performance target, we will take a multi-threaded batch approach to POST our Message Requests. Batch size is configured and provided to the service startup.  
       
     The HttpClient.PostAsync() method in .NET Framework 4.5 or greater is thread-safe. That’s the expected method to use to integrate with the third-party service.
     1. Class instantiation should inject a DataAccess object. Other items needed to configure the service are (Set them through the constructor and setter methods):
        1. Number of Threads
        2. PollingRowFetchMax
        3. Third-Party API Endpoint

The class should throw a ConfigurationException if a method is used before these values are set. Log as an error to the Application Event Log as well.

* + 1. The following methods should support the objectives of the class.
       1. *public List<QueuedMessageRequestDTO> AssembleNextMessageList()*Use the GetQueuedMessageRequests(maxRows) method of the DataAccess class to assemble a list of queued messages to process. Feed the instance variable PollingRowFetchMax as the argument
       2. *public void SendMessages(List<QueuedMessageRequestDTO> messages)*This function uses the PostAsync() method of an HttpClient object in a multi-threaded manner that POSTS parallel batches (Number of Threads instance variable) of MessageDTO rows until the list exhausts. Construct MessageDTO objects, a requirement of the Third-party API, from the QueuedMessageRequestDTO objects.  
            
          If a network timeout or related exception occurs:
          1. call the *UpdateMessageDeliveryStatus*() to update database Message row and log the attempt. Send ‘Retry - Network Error’ as the status. The API\_Message will be empty. Try to capture some info about the outage in the AdditionalInfo property.
          2. Log the error to the Application Event Log.

Else If the API returns return an http error response (status 400/500 level):

* + - * 1. call the *UpdateMessageDeliveryStatus*() to update database Message row and log the attempt. Send ‘Retry - Http Error’ as the status. Use API\_Message and AdditionalInfo to record more info on the problem.
        2. Log the error to the Application Event Log.

Else

1. Examine the message returned by the API and make the following resolution for the DeliveryStatusID:  
   - ‘Success’ ==> ‘Successfully Delivered’  
   - ‘Failed – Not a valid phone’ ==> ‘Dead – Invalid Phone Number’  
   - ‘Failed – Not valid by Time zone ==> ‘Retry – Invalid Time’  
     
   call the *UpdateMessageDeliveryStatus*() to update database Message row and log the attempt. Send the resolved DeliveryStatusID, the message returned by the API, and any Additional info in the LogDeliveryDTO
2. Log the Information (Success), Warning (Failed-Time Zone), or Error (Failed-Invalid Phone) to the Application Event Log.

f) Message Delivery Service   
The Message Deliver Service manages and coordinates the end-to-end message delivery process. It should execute on a timer dictated by the PollingPeriodSeconds configuration parameter.  
  
The first iteration (and every 10th thereafter) executes the RequeueStaleInProgressRows() process described in section 7d above (DataAccess). This will requeue failed, but recoverable messages. This provides resiliency to the process.  
  
All of the other iterations (90%) execute the SendMessages() process described in section 7e above (Message Delivery Service). This is the primary activity of our application.  
  
10% of the iterations provide resiliency. The remaining 90% perform the intended work.

1. **Performance KPIs**We should work to glean performance data from the artifacts. For each statistic we should seek to determine averages, means, standard deviations, and warning thresholds.
   1. Messages delivered within 10 minutes / Total Messages for various time periods.
   2. Baseline counts of messages queued per day, hour, 10-minute period.
   3. Current count of messages older than 10 minutes awaiting delivery.
   4. Statistics by Failure Type. Maybe we can determine that ‘Failed - Not valid by Time zone’ is a final state and the message cannot be delivered.
   5. Third-party API outage patterns. Is it consistent? If so, can we change to take advantage? Can we use it to drive improvement by the third-party?
   6. Average delivery attempts
   7. Diagnostic to determine that service may be down. Ex. A message attempt hasn’t been tried in x minutes during an appropriate time range established by baselines.
2. **Pending and Future Design Considerations.**
   1. Third-party API Response: ’Failed – Not valid by Time zone’.   
      We need a better understanding. Should a message with this response be considered dead? Our current design treats it as recoverable. If that’s not the case, and this occurs with regularity, we risk our performance goals due to endless cycles wasted on undeliverable content.
   2. Our design prioritizes new messages over requeued failed messages. We took this stance with our 10-minute message delivery goal in mind. Is this the right strategy? Should we take steps to treat requeued messages with higher priority. It’s hard to say. Real-world execution, a review its performance data, and conversations with stakeholders may drive change to this strategy.
   3. Real-time monitoring and incident response. This design encourages the creation of artifacts to measure performance, establish baselines, and tweak performance. It’s wise to consider a small project to put real-time monitoring in place along with a strategy to determine unacceptable performance and notify responsible parties.
   4. Database archiving: The database may grow at a rapid pace. We expect to add over 3.5 million messages a year to the database. The Application Event Logs on the application servers will also grow rapidly. We need to consider an archive plan to keep our environment clean.
   5. Is there some time threshold where we should consider and queued message too old to send? And maybe convert it to a ‘dead’ status.