Transmitting notifications/events from PostgreSQL to web browsers via Message Oriented Middleware/Events, on Microsoft Azure

Editor	Carlo Randone, IBM Consulting
Date of last update of the article	October, 2024
Version	2.0.0
GitHub Link	https://github.com/carlornd/NotificationsAzure

Table of Contents

Introduction and Overview

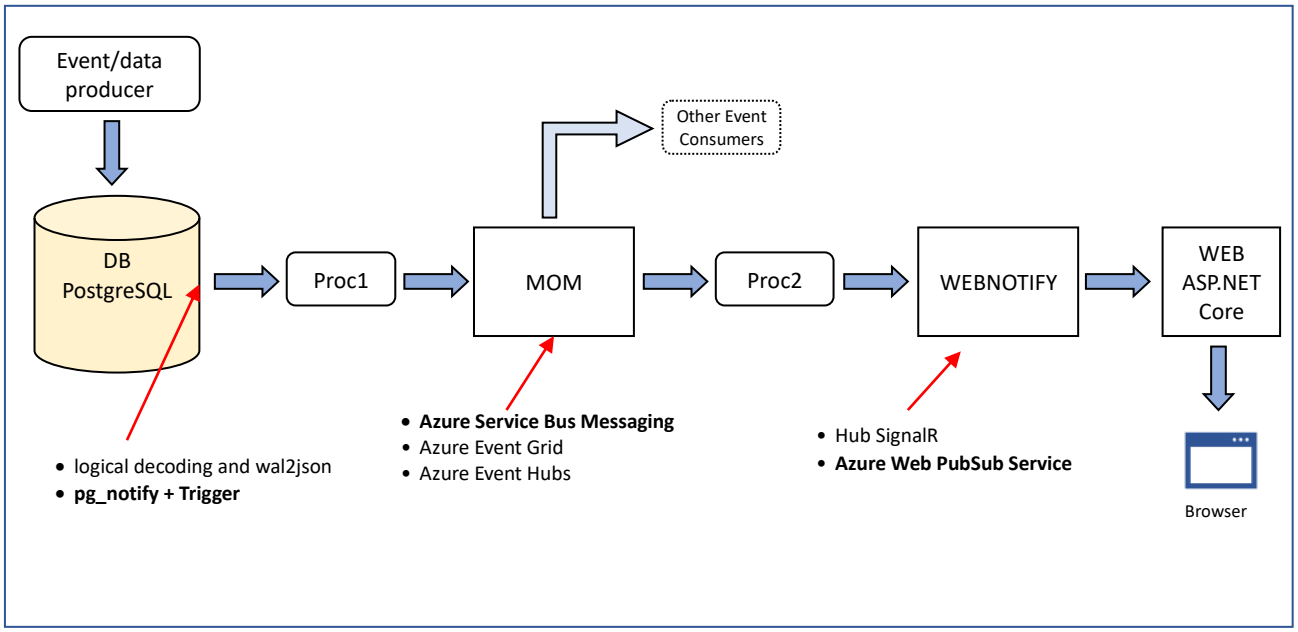
This article presents a technical summary related to the issue of near-real time propagation of events (with related data/information) from a PostgreSQL database (e.g. a PostgreSQL Flexible in PaaS on Microsoft Azure) to interactive web sessions hosted in browsers that point to a web application implemented with ASP.NET Core (in the proposed example a Blazor Server web app). This "E-2-E" (End-to-End) communication is supported by the following architectural building blocks, in chain (**Note:** it is assumed here that the primary "SOURCE" system of events/notifications writes the data via Create and/or Update operations to the PostgreSQL database):

- **DB** - PostgreSQL Database (e.g. a PostgreSQL Flexible in PaaS on Azure), with the **pg_notify** feature implemented and an active trigger.
    - **"Proc1"** - A process that collects events ("triggers") that PostgreSQL raises against data operations. This process "listens" to the PostgreSQL triggers and, when the trigger receives a set of data, inserts them into the event and/or message transmission middleware, in a Message Queuing, Publish/Subscribe or Event-based logic, depending on the pattern and the platform/technical solution adopted. This "long-running" process can be supported on the Azure platform in different ways, as described below.
- **MOM** - Centralized message-oriented middleware ("MOM") or "Broker" platform. This platform can be oriented to queues, or to "topic" logic of the pub/sub type, or natively manage events. Practical examples of such a platform in Azure are: Azure Service Bus, Azure Event Hubs, Azure Event Grid. Having such a centralized platform allows the eventual implementation of multiple "subscribers" that can subscribe to the data and receive the information "propagated" by the activities that take place on the PostgreSQL database located "upstream" in the chain.
    - **"Proc2"** - A process that "dequeues" events/messages from the adopted MOM platform, which, upon the arrival of a message/event, republishes it on a centralized notification management system to deliver the notifications to the browser sessions opened on the target web application. This "long-running" process can be supported on the Azure platform in several ways, as described below. It can also be implemented as an IHostedService/BackgroundService within the same ASP.NET Core web application used by end users.
- **WEBNOTIFY** - Centralized platform for the management of the notification to the web application of the events published by the "dequeuer" ("Proc2"). The primary technologies that can be adopted for this purpose can be the adoption of a SignalR Hub or (better) the adoption of Azure Web PubSub service.
    - **"WEB"** - Integration of data receipt logic ("event notifications") into the code of the Web ASP.NET Core application used by end-users.

So, here we have:

- an "high level" architectural proposal (or "reference architecture") to address the need to implement an asynchronous notification chain from a database to a web application, and
- an "opinionated" implementation based on Microsoft Azure services and Microsoft Visual Studio .NET code and libraries.

This is the summary scheme, with an indication of the main implementation options (in **bold** those primarily indicated for the case in question in the light of the information currently available, and adopted in the provided sample implementation):



The various steps are briefly described below, with links to support them.

Both for the "long running" processes (Proc1 and Proc2) and for the two platforms identified as "MOM" and "WEBNOTIFY", architectural and implementation options are proposed, with a "preferential" indication with respect to the specific project context in place. For example, for the capture of notifications in PostgreSQL, an implementation of the "pg_notify (NOTIFY Command) + Trigger" mechanism is preferred over an implementation of "logical decoding" (Change Data Capture) + wal2json.

As also indicated later in the section "Integrated view of the demo in operation" in this article, a "demo" implementation has been prototyped, with the following associations with respect to the architectural building blocks outlined above (the projects are on the GitHub here https://github.com/carlornd/NotificationsAzure):

Building Block Component of the Schematic	**"Demo" implementation**
DB	A PostgreSQL Flexible database in PaaS on Azure, table “Products”
Proc1	Solution .NET WPF “**EventHubPublisher.sln**” (.NET Framework 4.8), with the function of both receiving PostgreSQL notifications and loading data into the MOM (here Azure Service Bus)
MOM	Implementing Azure Service Bus, using queues, on Azure
Proc2	Solution .NET Core “**publisher.sln**” (console application), acting as both a dequeuer from the Azure Service Bus and a loader in the Azure Web PubSub service notification system
WEBNOTIFY	Implementing Azure Web PubSub Service for ASP.NET Core Web App Notification
ASP.NET Core Web App	Solution ASP.NET Core "**BlazorServerSignalRSoln.sln**"; Web App ASP.NET Core Blazor Server that receives notifications from the Azure Web PubSub service subsystem and presents them on screen.

DB - Propagating events from PostgreSQL to application code

The need is to capture operations (typically INSERT/UPDATE) on the PostgreSQL database and to transfer these "notifications" (with related data) downstream in the transmission chain described above.

To do this, there are two main "native" options in PostgreSQL:

- Using PostgreSQL "Logical Decoding" coupled with the output plugin "wal2json" (the term "WAL" refers to the PostgreSQL transaction log)
- Using the "pgnotify" PostgreSQL mechanism coupled with the use of a PostgreSQL trigger

In the following, the second option is explored, which does not necessarily require full administrative permissions on PostgreSQL, and which is lighter (and simpler) for the scenario in the field of this implementation. In any case, these are some specific references on the two options mentioned:

For Logical Decoding and wal2json:

“Change data capture in Postgres: How to use logical decoding and wal2json”

https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/change-data-capture-in-postgres-how-to-use-logical-decoding-and/ba-p/1396421

“Logical replication and logical decoding in Azure Database for PostgreSQL - Flexible Server”

https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/concepts-logical

“Logical Decoding Concepts”

https://www.postgresql.org/docs/current/logicaldecoding-explanation.html

For “pgnotify” and trigger:

“azure-pgnotify”

https://github.com/liupeirong/azure-pgnotify

“Postgres Notify for Real Time Dashboards”

https://arctype.com/blog/postgres-notify-for-real-time-dashboards/

“Real-time event processing with Azure Database for PostgreSQL and Event Grid integration”

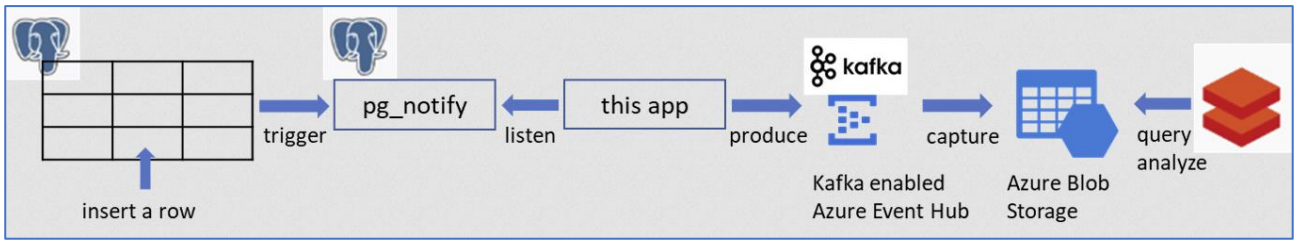
https://azure.microsoft.com/en-us/blog/event-processing-with-azure-database-for-postgresql-and-azure-event-grid-integration/

(which then refers to: https://github.com/Azure/azure-postgresql/tree/master/samples/EventGrid%20Demo)

Here are two different examples of pg_notify and trigger configuration at the PostgreSQL level, taken from the references above.

Example 1

It is taken from “azure-pgnotify”, https://github.com/liupeirong/azure-pgnotify, where this diagram is also presented, which depicts an example of message notification supported by the pg_notify mechanism and then by a Kafka-type implementation.



Define pg_notify

In PostgreSQL, define a pg_notify function. For example, the following function converts an inserted or updated row into JSON format to send to pg_notify:

```
CREATE OR REPLACE FUNCTION public."process_event_notify"()
RETURNS trigger
LANGUAGE plpgsql
COST 100
VOLATILE NOT LEAKPROOF
AS $BODY$
DECLARE
BEGIN
  PERFORM pg_notify('{channel_name}', row_to_json(NEW)::text);
  RETURN NEW;
END;
$BODY$;
```

Set trigger in PostgreSQL

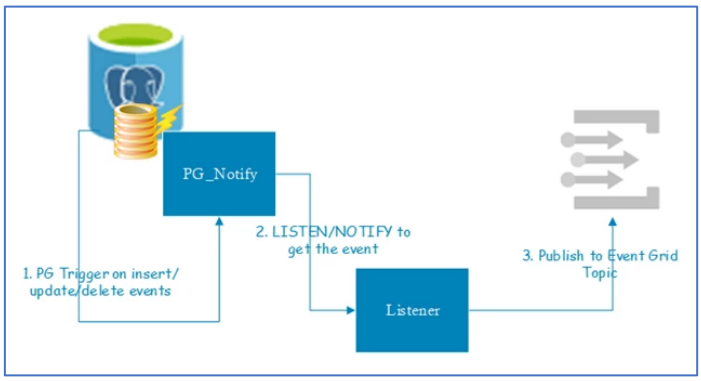
Define a trigger in the table which you want to track changes. For example, the following trigger runs pg_notify for every insert into the target table:

```
CREATE TRIGGER process_event_update
AFTER INSERT
ON public."{table_name}"
FOR EACH ROW
EXECUTE PROCEDURE public."process_event_notify"();
```

Example 2

This is a variant taken from “Real-time event processing with Azure Database for PostgreSQL and Event Grid integration”

https://azure.microsoft.com/en-us/blog/event-processing-with-azure-database-for-postgresql-and-azure-event-grid-integration/



which refers to:

“Event Grid with PostgreSQL Data update”

https://github.com/Azure/azure-postgresql/tree/master/samples/EventGrid%20Demo

Step 1: Set up PG Notify

Connect to the database and define the notification according to the table and column need to send. In our example, the function is on Product table and sending the Id as the unique identifier to receivers for correct action. The function looks like the following:

```
-- DROP FUNCTION public."Products_update_notify"();
CREATE FUNCTION public."Products_update_notify"()
        RETURNS trigger
        LANGUAGE 'plpgsql'
        COST 100
        VOLATILE NOT LEAKPROOF
AS $BODY$
DECLARE
  Id uuid;
  Name character varying(512);
BEGIN
  IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
        Id = NEW."Id";
        Name = NEW."Name";
  ELSE
        Id = OLD."Id";
        Name = OLD."Name";
  END IF;
  PERFORM pg_notify('productsnotification', TG_OP || ';' || Id || ';' || Name);
```

```
 RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public."Products_update_notify"()
   OWNER TO pgadmin;
```

[

Note: a variant I used of the above is the following (for my "Products" table with the two fields “Id” of type text (PK) and “Name” of type character varying(512):

```
-- DROP FUNCTION public."Products_update_notify"();
CREATE FUNCTION public."Products_update_notify"()
           RETURNS trigger
           LANGUAGE 'plpgsql'
           COST 100
           VOLATILE NOT LEAKPROOF
AS $BODY$
DECLARE
  Id text;
  Name character varying(512);
BEGIN
  IF TG_OP = 'INSERT' OR TG_OP = 'UPDATE' THEN
    Id = NEW."Id";
    Name = NEW."Name";
    PERFORM pg_notify('productsnotification', TG_OP || ';' || Id || ';' || Name || ';' || row_to_json(NEW)::text);
  ELSE
    Id = OLD."Id";
    Name = OLD."Name";
    PERFORM pg_notify('productsnotification', TG_OP || ';' || Id || ';' || Name || ';' || row_to_json(OLD)::text);
  END IF;

  RETURN NEW;
END;

$BODY$;
```

]

Then Create Triggers on the table that sends notification. For example, for Products table, we can create the trigger for update like the following.

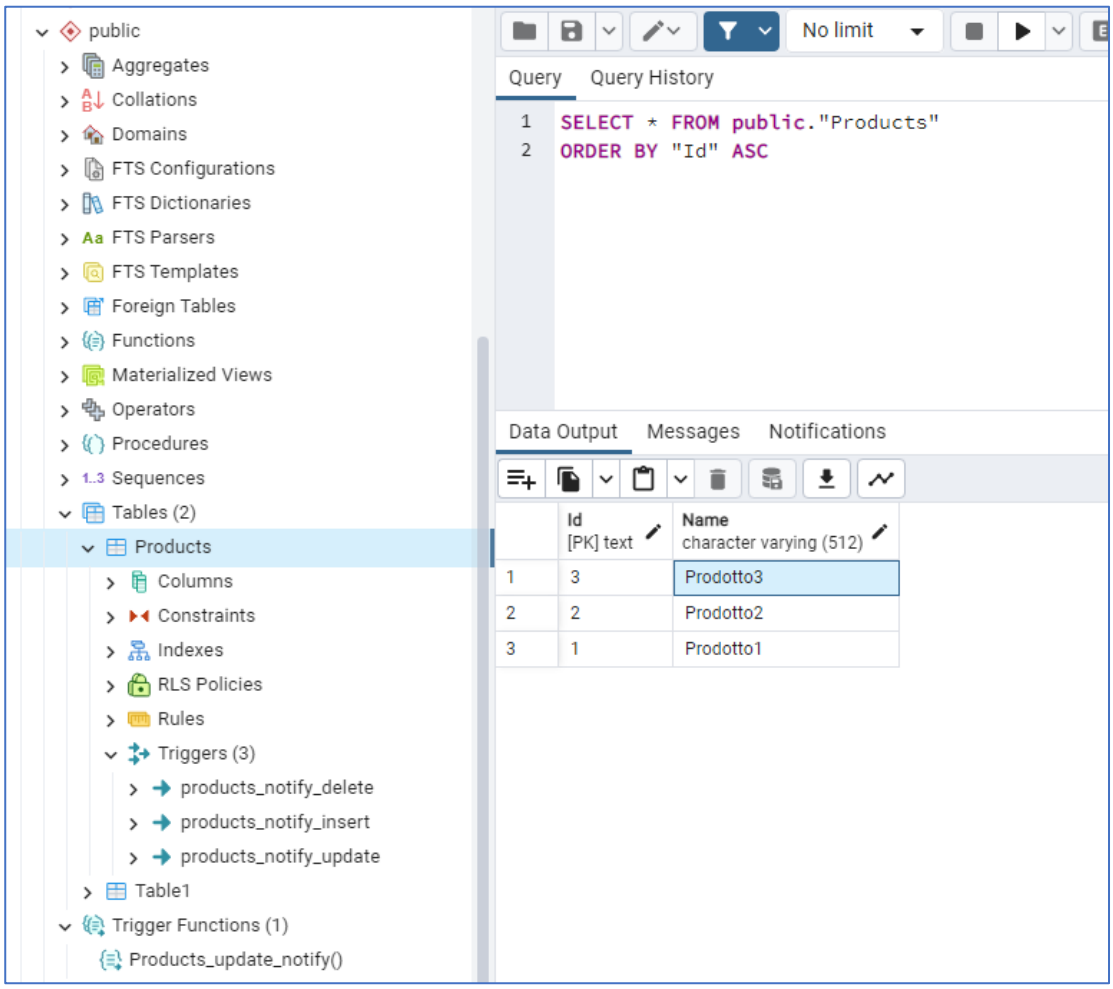
```
-- DROP TRIGGER products_notify_update ON public."Products";
CREATE TRIGGER products_notify_update
           AFTER UPDATE
           ON public."Products"
           FOR EACH ROW
           EXECUTE PROCEDURE public."Products_update_notify"();
```

We also created similar triggers for insert and delete:

```
-- DROP TRIGGER products_notify_insert ON public."Products";
CREATE TRIGGER products_notify_insert
   AFTER INSERT
   ON public."Products"
   FOR EACH ROW
   EXECUTE PROCEDURE public."Products_update_notify"();


-- DROP TRIGGER products_notify_delete ON public."Products";
CREATE TRIGGER products_notify_delete
```

AFTER DELETE
ON public."Products"
FOR EACH ROW
EXECUTE PROCEDURE public."Products_update_notify"();

This is a situation in the test environment used, in which the functions and triggers defined at the "Products" table level in the "public" schema of the test database "mydb1" on an Azure PostgreSQL Flexible instance are highlighted:



(**Note**: See the folder "..\DBeProc1\EventHub Demo\SQL Script" in the provided GitHub repository for the sql code of the example).

Proc1 - Receive events launched by PostgreSQL and propagate to the MOM

Introduction

The "**Proc1**" process serves two purposes:

- Receives events ("triggers") raised by PostgreSQL against data operations. The process "listens" for PostgreSQL triggers and when receives trigger notification, then performs its second function described below
- It inserts messages into the middleware for transmitting events and/or messages, in a Message Queuing, Publish/Subscribe or Event-based logic, depending on the pattern and the platform/technical solution adopted.

This "long-running" process can be supported on the Azure platform in different ways, as described below in the appendix "Background Jobs in Azure".

By way of example, a demo implemented as a .NET Framework WPF (Windows Presentation Foundation) application (**EventHubPublisher.sln**) is proposed, to allow easy interaction and experimentation. In a production context, the same application logics can be implemented in a daemon/long running process on Azure, or on a suitable server (or VM).

"Proc1" Process – Receiving Notifications from PostgreSQL

There are two examples available from Microsoft that implement the reception of "NOTIFY" from PostgreSQL (and implemented as described above with pg_notify and the trigger):

- An example of receiving and publishing to Event Grid: https://github.com/Azure/azure-postgresql/blob/master/samples/EventGrid%20Demo/README.md
- An example of receiving and publishing to Event Hub: https://github.com/Azure/azure-postgresql/blob/master/samples/EventHub%20Demo/README.md

In both cases, the code that RECEIVES the notifications launched by PostgreSQL is the same, and in the examples it is implemented starting from the "Start" button on the main form "MainWindow.xaml".

An event subscription is activated and the "PostgresNotification" is invoked upon receipt, which in both cases then invokes a "PublishEvent", which in one case writes to Event Grid, in the other to Event Hub. Similar code could also write to **Azure Service Bus** instead, of course.

However, the code for receiving events from PostgreSQL looks like this (note that in the sample this is a WPF desktop application; in a production context it will most likely be a console-based long running process

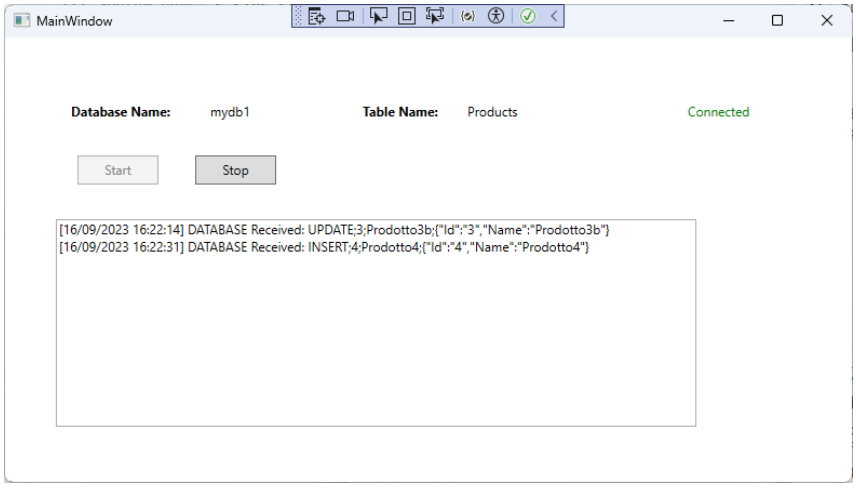
hosted in one of the modes supported by Azure for "log running processes" / "background jobs" as described in the Appendix):

```
string connectionstring = string.Empty;

            try
            {
                this.labelNotifications.Text = string.Empty;
                connectionstring = this.GetConnectionString(true);
                this.notificationConnection = new NpgsqlConnection(connectionstring);
                this.notificationConnection.Open();

                if (this.notificationConnection.State == ConnectionState.Open)
                {
                    this.labelStatus.Foreground = Brushes.Green;
                    this.labelStatus.Content = @"Connected";

                    using (var command = new NpgsqlCommand("listen " + TriggerChannelName,
this.notificationConnection))
                    {
                        command.ExecuteNonQuery();
                    }

                    this.notificationConnection.Notification += this.PostgresNotification;
                    this.startButton.IsEnabled = false;
                    this.stopButton.IsEnabled = true;
                }
                else
                {
                    this.labelStatus.Foreground = Brushes.Red;
                    this.labelStatus.Content = @"Connection failed !";
                }
            }
            catch
            {
                MessageBox.Show("Connection error: " + connectionstring);
            }
```

[…]

```
/// <summary>
        /// Postgres notification event.
        /// </summary>
        /// <param name="sender">The sender.</param>
        /// <param name="e">The event arguments.</param>
        private void PostgresNotification(object sender, NpgsqlNotificationEventArgs e)
        {
            string info = e.AdditionalInformation;
            Console.WriteLine(@"Notification -->");
            Console.WriteLine(@"  DATA {0}", info);
            Console.WriteLine(@"  CHANNEL {0}", e.Condition);
            Console.WriteLine(@"  PID {0}", e.PID);

            this.labelNotifications.Dispatcher.Invoke(new Action(() =>
            {
                labelNotifications.Text += "[" + DateTime.Now.ToString() + "] DATABASE Received: " + info +
Environment.NewLine;
            }));

            if (PublishEvent(info))
            {
                this.labelNotifications.Dispatcher.Invoke(new Action(() =>
                {
                    labelNotifications.Text += "[" + DateTime.Now.ToString() + "] Event Published! " +
Environment.NewLine;
                }));
            }
        }
```

Here's an example of receiving as part of the adapted demo (it's a .NET WPF desktop process):



In this case, the code of the "PublishEvent" will have to be implemented differently depending on the "MOM" platform adopted (e.g. Azure Service Bus).

In addition, as also clarified in the following chapter, in the specific case of Azure Service Bus, it is necessary to check whether you want to use it from a "Queue" perspective or from a "Topics and Subscriptions" perspective.

Examples of publishing (and receiving) using the Azure Service Bus can be found at the following links:

“Sending and receiving messages” (It deals with both queues and topics)

https://github.com/Azure/azure-sdk-for-net/blob/main/sdk/servicebus/Azure.Messaging.ServiceBus/samples/Sample01_SendReceive.md

“Quickstart: Send and receive messages from an Azure Service Bus queue (.NET)” (deals with the case of queues)

https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-get-started-with-queues?tabs=passwordless

“Get started with Azure Service Bus topics and subscriptions (.NET)” (deals with the case of topics)

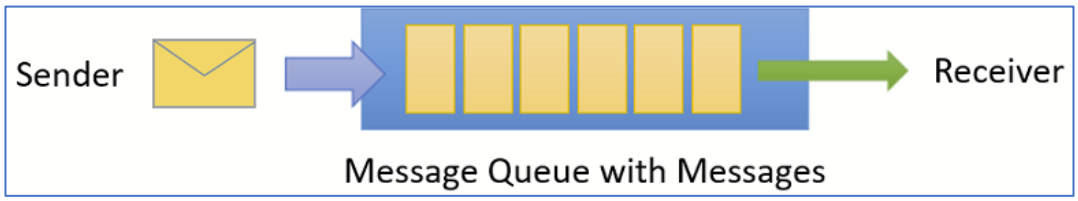
https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-how-to-use-topics-subscriptions?tabs=passwordless

"Proc1" Process – Inserting Messages into the "MOM" Middleware

In order to define the behavior of "Proc1" for the purpose of inserting the notification in the adopted MOM platform, the approach must be formalized. For example, the two examples proposed in https://github.com/Azure/azure-postgresql/blob/master/samples/EventGrid%20Demo/README.md and in https://github.com/Azure/azure-postgresql/blob/master/samples/EventHub%20Demo/README.md publish events to Azure Event Grid and Azure Event Hub, respectively.

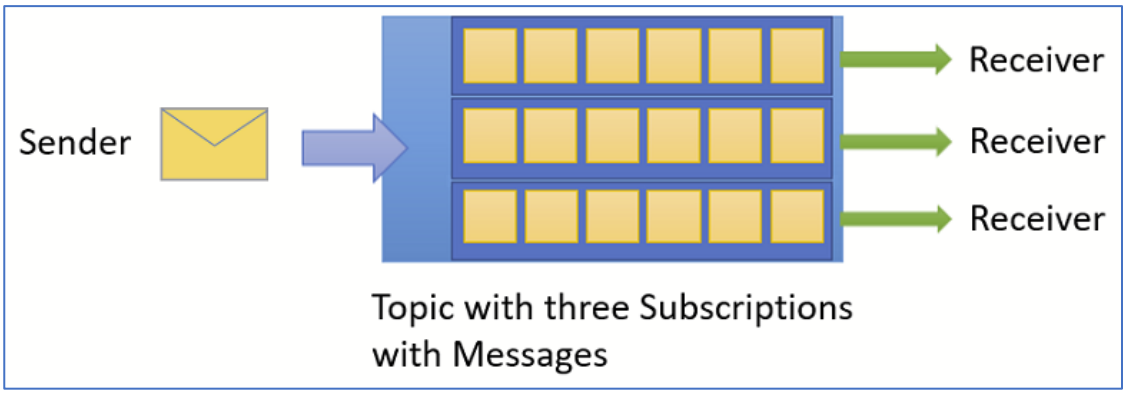
If you want to publish to the Azure Service Bus, you'll need to check how you want to do it. In fact, the Azure Service Bus offers two main modes of operation:

- **Queues** (See also: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview#queues and https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions#queues)
  << Queues allows for Sending and Receiving of messages. Often used for point-to-point communication.
  Queues offer First In, First Out (FIFO) message delivery to one or more competing consumers. That is, receivers typically receive and process messages in the order in which they were added to the queue. And only one message consumer receives and processes each message.



  A key benefit of using queues is to achieve temporal decoupling of application components. In other words, the producers (senders) and consumers (receivers) don't have to send and receive messages at the same time. That's because messages are stored durably in the queue. Furthermore, the producer doesn't have to wait for a reply from the consumer to continue to process and send messages.>>

- **Topics and subscriptions** (See also: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview#topics e https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-queues-topics-subscriptions#topics-and-subscriptions)
  << As opposed to Queues, Topics are better suited to publish/subscribe scenarios. A topic can be sent to, but requires a subscription, of which there can be multiple in parallel, to consume from. So, a queue allows processing of a message by a single consumer. In contrast to queues, topics and subscriptions provide a one-to-many form of communication in a publish and subscribe pattern. It's useful for scaling to large numbers of recipients. Each published message is made available to each subscription registered with the topic. Publisher sends a message to a topic and one or more subscribers receive a copy of the message.



The subscriptions can use additional filters to restrict the messages that they want to receive. Publishers send messages to a topic in the same way that they send messages to a queue. But consumers don't receive messages directly from the topic. Instead, consumers receive messages from subscriptions of the topic. A topic subscription resembles a virtual queue that receives copies of the messages that are sent to the topic. Consumers receive messages from a subscription identically to the way they receive messages from a queue.
A Subscription is the mechanism to consume from a Topic. Each subscription is independent, and receives a copy of each message sent to the topic. Rules and Filters can be used to tailor which messages are received by a specific subscription.
>>

The "topic and subscriptions" mode is particularly useful if the same message needs to be "consumed" by multiple subscribers.

This is a practical example of publishing on the "MOM" implemented with Azure Service Bus, in the "Queue" mode (usable with the "Basic" model/tier):

```
        // QUI PREDISPONGO UNA PUBBLICAZIONE SU AZURE SERVICE BUS (uso una QUEUE, disponibili anche sul Tier "Basic")
        private static ServiceBusClient serviceBusClient;
        private const string ServiceBusConnectionString =
"Endpoint=sb://mycrservicebusns.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=aNnxV5+rN3HiUG2
PYpJ7m6rP8jeoCHaXk+ASbBY9i3g=";
        private const string ServiceBusQueueName = "myqueue";
        // the sender used to publish messages to the queue
        private static ServiceBusSender sender;

        private bool PublishEvent2(string oneEvent)
        {
            try
            {
                PublishEventAsync2(oneEvent).GetAwaiter().GetResult();
                return true;
            }
            catch (Exception exception)
            {
                Debug.Print(exception.ToString());
                return false;
            }
        }

        private async Task PublishEventAsync2(string message)
        {
            // Creates an EventHubsConnectionStringBuilder object from the connection string, and sets the EntityPath.
            // Typically, the connection string should have the entity path in it, but this simple scenario
            // uses the connection string from the namespace.

            /*
            var connectionStringBuilder = new  EventHubsConnectionStringBuilder(EventHubConnectionString)
            {
                EntityPath = EventHubName
            };
            */
            var clientOptions = new ServiceBusClientOptions()
            {
                TransportType = ServiceBusTransportType.AmqpWebSockets
            };

            serviceBusClient = new ServiceBusClient(ServiceBusConnectionString, clientOptions);
            sender = serviceBusClient.CreateSender(ServiceBusQueueName);

            await sender.SendMessageAsync(new ServiceBusMessage(message));
            //await sender.SendMessageAsync(new EventData(Encoding.UTF8.GetBytes(message)));
```

```
            await sender.DisposeAsync();
            await serviceBusClient.DisposeAsync();
        }
        #endregion Service Bus Access
    }
```

See, for example, the implementation of logical features "equivalent" to an hypothetical "Proc1" that has been done (with .NET WPF support) in:

…\DBeProc1\EventHub Demo\EventHubPublisher

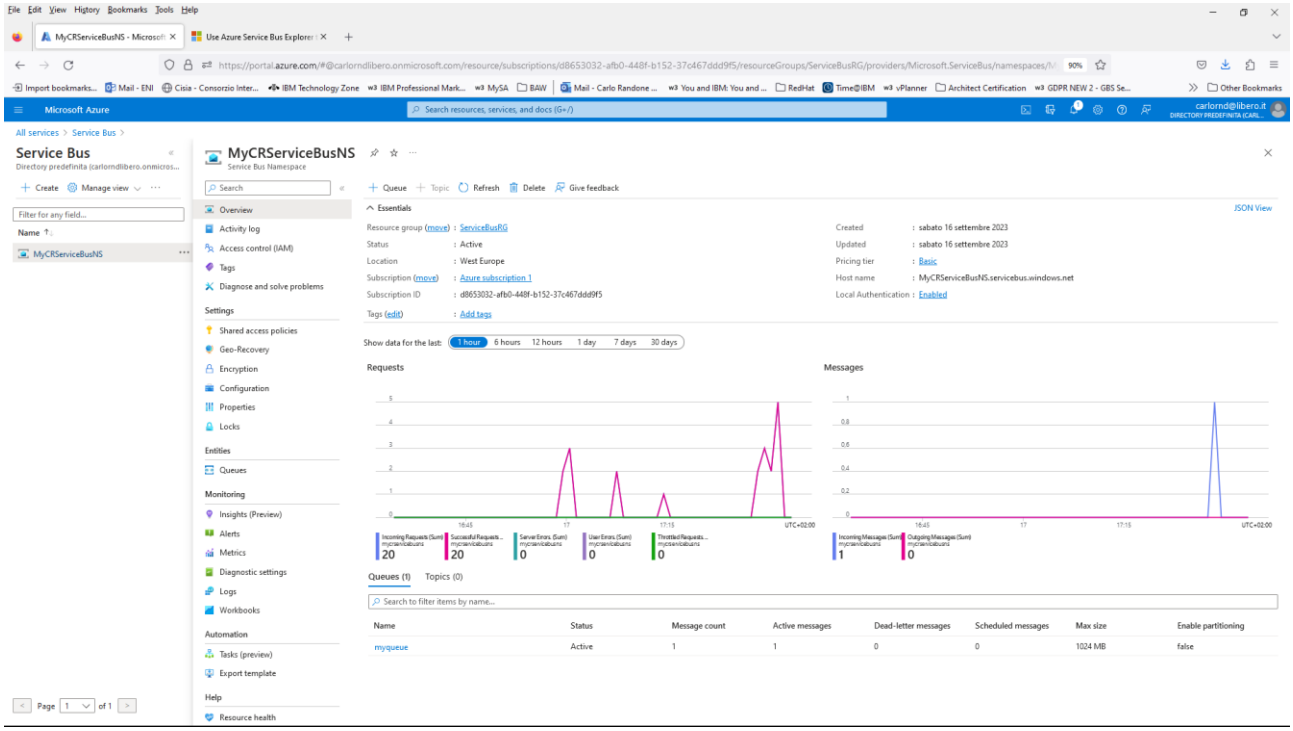
(The project name mentions "EventHub" because it was originally set up to work with Azure Event Hub. The publishing functionality on **Azure Service Bus** has also been integrated, specifically in the "Queue" mode)

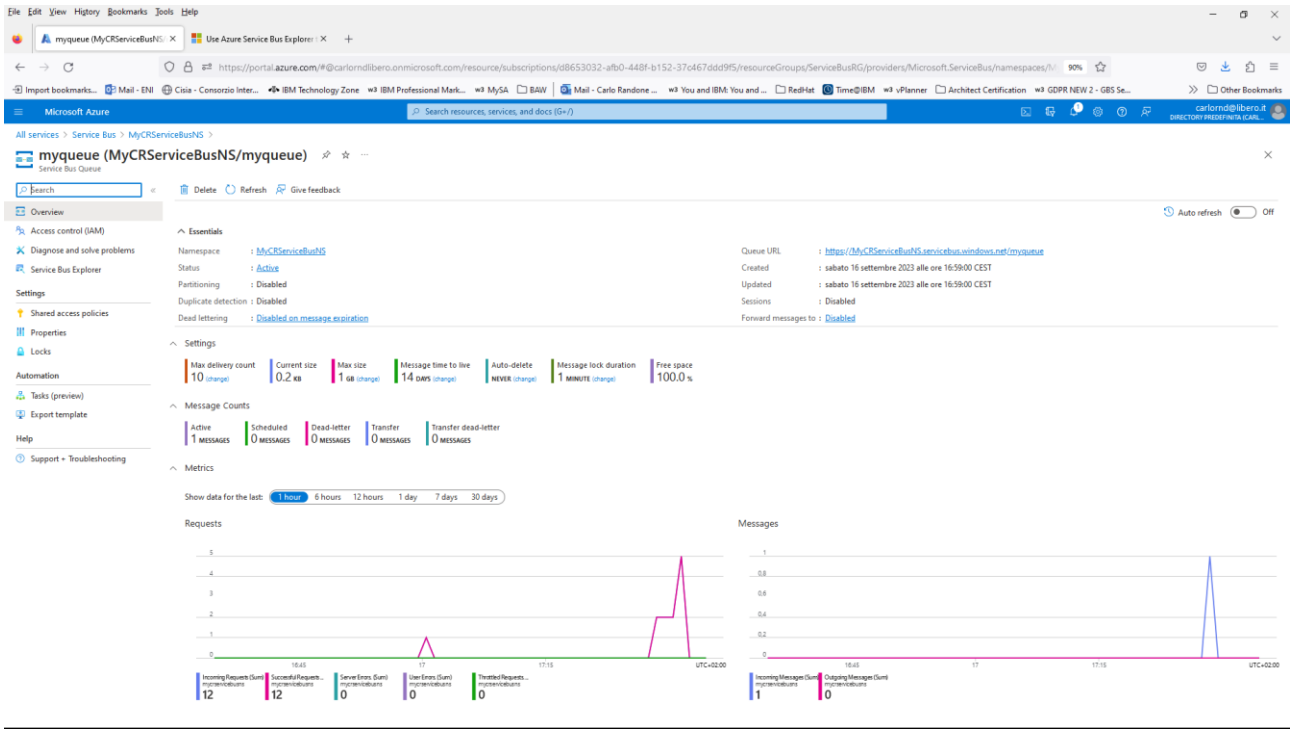
MOM – Message/Event Propagation and Management Platform Considerations

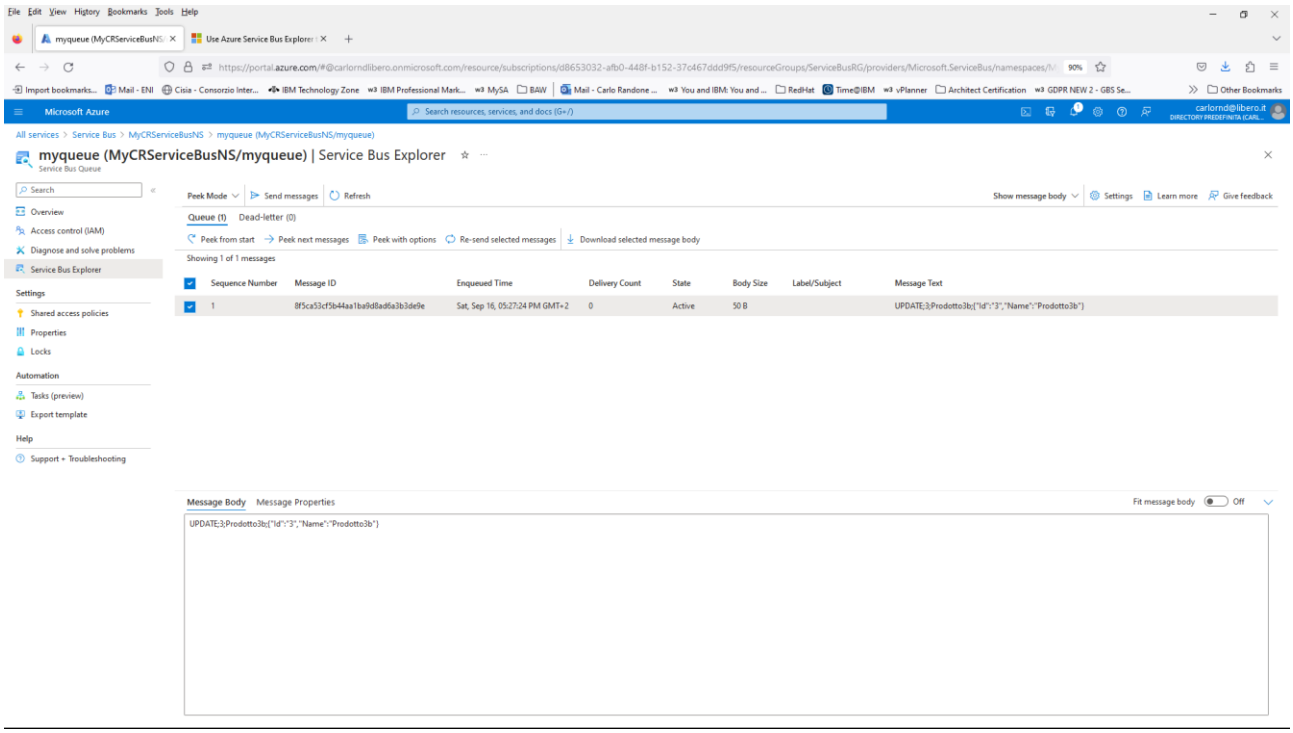
Solutions for transferring messages from PostgreSQL to the Web notification system are generically referred to in the diagram proposed at the beginning of this article as "MOM", which stands for a generic "Message-Oriented Middleware". Also according to what is introduced, for example, in the article “Asynchronous messaging options”, https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/messaging, a middleware of this type also takes the generic name of "**broker**", capable of transferring messages oriented to both "command" and "event" logics/types. In a real implementation, this middleware can be implemented with "open" solutions (such as Kafka, RabbitMQ, etc.) or with services offered within Microsoft Azure, such as Azure Service Bus, Azure Event Hubs, Azure Event Grid.

For a comparison of these Azure-based middleware, see also the appendix " Azure Service Bus, Azure Event Hub, Azure Event Grid comparison" of this article.

This is an example of a message published by "Proc1" on a "myqueue" queue of an Azure Service Bus instance (namespace "MyCRServiceBusNS"):




myqueue (MyCRServiceBusNS/myqueue)
Service Bus Queue
Delete
Refresh
Give feedback
Overview
Access control (IAM)
Diagnose and solve problems
Service Bus Explorer
Settings
Shared access policies
Properties
Locks
Automation
Tasks (preview)
Export template
Help
Support + Troubleshooting
Essentials
Namespace : MyCRServiceBusNS
Status : Active
Partitioning : Disabled
Duplicate detection : Disabled
Dead lettering : Disabled on message expiration
Queue URL : https://MyCRServiceBusNS.servicebus.windows.net/myqueue
Created : sabato 16 settembre 2023 alle ore 16:59:00 CEST
Updated : sabato 16 settembre 2023 alle ore 16:59:00 CEST
Sessions : Disabled
Forward messages to : Disabled
Settings
Message Counts
Active 1 MESSAGES
Scheduled 0 MESSAGES
Dead-letter 0 MESSAGES
Transfer 0 MESSAGES
Transfer dead-letter 0 MESSAGES
Metrics
Requests
Messages


myqueue (MyCRServiceBusNS/myqueue) | Service Bus Explorer
Service Bus Queue
Peek Mode
Send messages
Refresh
Queue (1)
Dead-letter (0)
Peek from start
Peek next messages
Peek with options
Re-send selected messages
Download selected message body
Showing 1 of 1 messages
Sequence Number
Message ID
Enqueued Time
Delivery Count
State
Body Size
Label/Subject
Message Text
1
8f5ca53cf5b44aa1ba9d8ad6a3b3de9e
Sat, Sep 16, 05:27:24 PM GMT+2
0
Active
50 B
UPDATE;3;Prodotto3b;{"Id":"3","Name":"Prodotto3b"}
Message Body
Message Properties
Fit message body
UPDATE;3;Prodotto3b;{"Id":"3","Name":"Prodotto3b"}

Proc2 – "Dequeuing" MOM messages and propagating to the Web notification solution

Introduction

This "long-running" process can be supported on the Azure platform in different ways, as described below in the appendix "Background Jobs in Azure" at the end of this article.

It can also be implemented as an IHostedService/BackgroundService within the same ASP.NET Core web application used by end users (see the appendix "Receiving events in a background task in a Web Application ASP.NET Core" in this article).

The activities carried out by this process are basically two:

- "Dequeuing" messages from the MOM platform
- Publication to the WEBNOTIFY platform

Proc2 – "Dequeuing" messages from the MOM platform

The actual implementation of this functionality, like the insertion of messages into the "MOM" platform, depends on the technology stack used. As already noted, solutions based on Azure Event Grid, Azure Event Hubs, or Azure Service Bus are available (remaining within the Azure scope). In particular, in the case of Azure Service Bus, it is then necessary to distinguish the “dequeue" logic using the "Queue" pattern from the one that is adopted if "Topics and Subscriptions" have been chosen instead.

Here are some technical and operational references:

Receiving events from Azure Event Grid:

https://github.com/Azure/azure-postgresql/tree/master/samples/EventGrid%20Demo

(see in particular the "EventGridListner" module)

Receiving events from Azure Hubs:

https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-dotnet-standard-getstarted-send?tabs=passwordless%2Croles-azure-portal

Receiving messages from Azure Service Bus (Queue):

“Quickstart: Send and receive messages from an Azure Service Bus queue (.NET)”

https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-get-started-with-queues?tabs=connection-string

Receiving messages from Azure Service Bus (Topic/Subscription):

“Get started with Azure Service Bus topics and subscriptions (.NET)”

https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-dotnet-how-to-use-topics-subscriptions?tabs=passwordless

Proc2 – Publication to the WEBNOTIFY platform

For the part of publication (by the "Proc2" process) to the WEBNOTIFY platform (centralized platform for publishing events/messages to the web app) see the (partial and simplified) sample:

…\BlazorServerSignalR3Soln\publisher

or take a direct look at the "built-in" (complete) example mentioned in the next paragraph of this chapter (a complete code that reads events from the Azure Service Bus and republishes them to Azure Web PubSub Service).

Basically, the publishing code to Azure Web PubSub looks like this:

```
using System;
using System.Threading.Tasks;
using Azure.Messaging.WebPubSub;
…
var connectionString =
"Endpoint=https://mypubsubcr.webpubsub.azure.com;AccessKey=+cVTk5RIaAekj6K7An53GKww9/0Q0GWIHaU58GHFOxI=;Version=1.
0;";
var hub = "Hub";
var message = "test message by CR";

var serviceClient = new WebPubSubServiceClient(connectionString, hub);
await serviceClient.SendToAllAsync(message);
```

The details clearly depend on how the service has been configured and how the "client"/consumer/subscriber" logic will be written within the web application.

The documentation on publishing and using notifications through Azure Web PubSub Service can be found at these links:

“Push messages from server”

https://learn.microsoft.com/en-us/azure/azure-web-pubsub/quickstarts-push-messages-from-server?tabs=csharp

Proc2 – An Integrated Example (part of the sample/demo code)

This is an integrated example of "**Proc2**", implemented as a simple .NET Core 7 "console application", which dequeues from the Azure Service Bus and inserts the message into Azure Web PubSub for later publication to browser sessions "subscribed" to the Web PubSub service

(from: …\Proc2\publisher)

```
namespace publisher
{
    internal class Program
    {
        // the client that owns the connection and can be used to create senders and receivers
        static ServiceBusClient sbclient;

        // the processor that reads and processes messages from the queue
        static ServiceBusProcessor sbprocessor;

        static WebPubSubServiceClient serviceClient;

        static async Task Main(string[] args)
        {


            Console.WriteLine("Hello, World!");

            var connectionString =
"Endpoint=https://mypubsubcr.webpubsub.azure.com;AccessKey=+cVTk5RIaAekj6K7An53GKww9/0Q0GWIHaU58GHFOxI=;Version=1.
0;";
            var hub = "Hub";
            var message = "test message by CR";

            // Either generate the token or fetch it from server or fetch a temp one from the portal
            serviceClient = new WebPubSubServiceClient(connectionString, hub);
            //await serviceClient.SendToAllAsync(message);

            // ==============================
            // Service Bus
            // ==============================

            var clientOptions = new ServiceBusClientOptions()
            {
                TransportType = ServiceBusTransportType.AmqpWebSockets
            };
            sbclient = new
ServiceBusClient("Endpoint=sb://mycrservicebusns.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAcces
sKey;SharedAccessKey=aNnxV5+rN3HiUG2PYpJ7m6rP8jeoCHaXk+ASbBY9i3g=", clientOptions);

            // create a processor that we can use to process the messages
            // TODO: Replace the <QUEUE-NAME> placeholder
            sbprocessor = sbclient.CreateProcessor("myqueue", new ServiceBusProcessorOptions());

            try
            {
                // add handler to process messages
                sbprocessor.ProcessMessageAsync += MessageHandler;

                // add handler to process any errors
                sbprocessor.ProcessErrorAsync += ErrorHandler;

                // start processing
                await sbprocessor.StartProcessingAsync();

                //Console.WriteLine("Wait for a minute and then press any key to end the processing");
                //Console.ReadKey();

                // stop processing
                //Console.WriteLine("\nStopping the receiver...");
                //await processor.StopProcessingAsync();
                //Console.WriteLine("Stopped receiving messages");
            }
            finally
            {
                // Calling DisposeAsync on client types is required to ensure that network
                // resources and other unmanaged objects are properly cleaned up.
                //await sbprocessor.DisposeAsync();
                //await sbclient.DisposeAsync();
            }

            // ==============================

            Console.WriteLine("'Enter' to terminate the program...");
            Console.ReadLine();
```

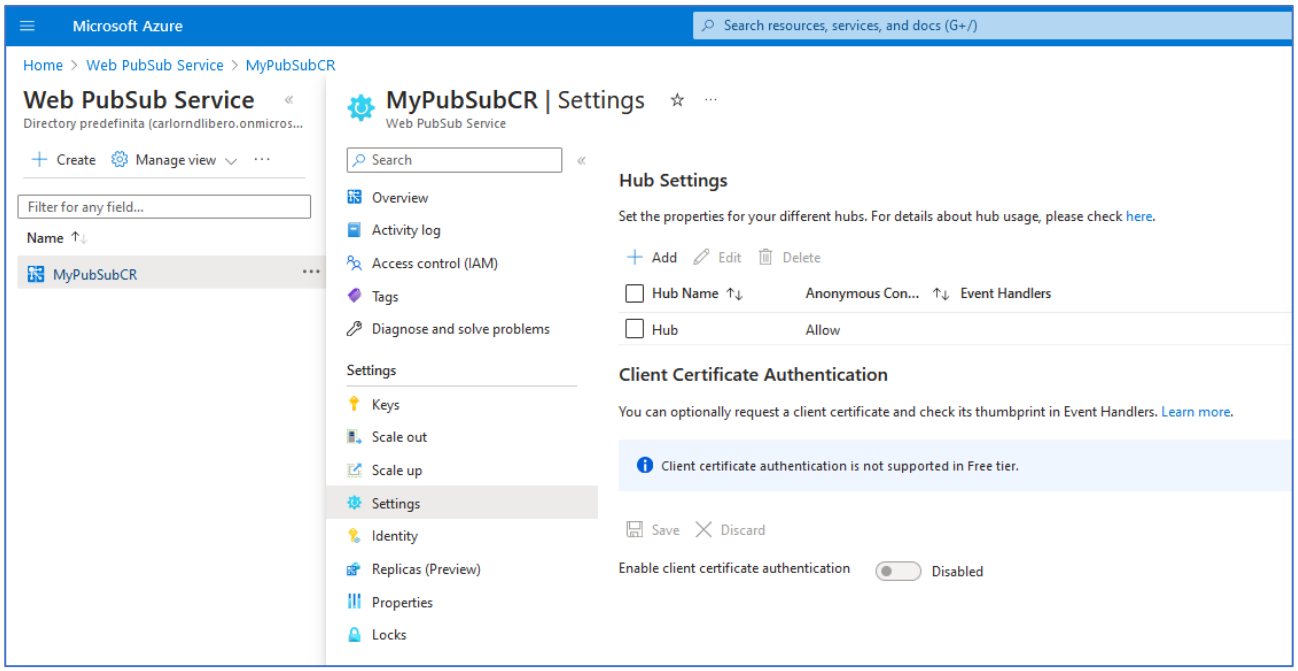
```
            // stop processing
            Console.WriteLine("\nStopping the receiver...");
            await sbprocessor.StopProcessingAsync();
            await sbprocessor.DisposeAsync();
            await sbclient.DisposeAsync();
            Console.WriteLine("Stopped receiving messages");

        }

        // handle received messages
        static async Task MessageHandler(ProcessMessageEventArgs args)
        {
            string body = args.Message.Body.ToString();
            Console.WriteLine($"Received: {body}");

            await serviceClient.SendToAllAsync(body);

            // complete the message. message is deleted from the queue.
            await args.CompleteMessageAsync(args.Message);
        }

        // handle any errors when receiving messages
        static Task ErrorHandler(ProcessErrorEventArgs args)
        {
            Console.WriteLine(args.Exception.ToString());
            return Task.CompletedTask;
        }

    }


}
```

WEBNOTIFY – Considerations on the event/message publishing platform to the web app

The centralized platform for publishing events to the discovered web app is made up of **Azure Web PubSub** service.

The Azure Web PubSub service allows you to easily build real-time messaging web applications using WebSockets and the publish-subscribe model. This real-time feature allows you to publish content updates between server and connected clients, such as a single-page web application or a mobile application. Clients should not poll for the latest updates or send new HTTP requests for updates.

This is the registration on Azure adopted:



ASP.NET Core Web APP and receiving events/messages to be presented to the user

For the part of receiving (by the ASP.NET Core web app) the notifications coming from the WEBNOTIFY platform, see the sample:

…\BlazorServerSignalR3Soln

The code was implemented using information available in the Azure Web PubSub service documentation, starting with:

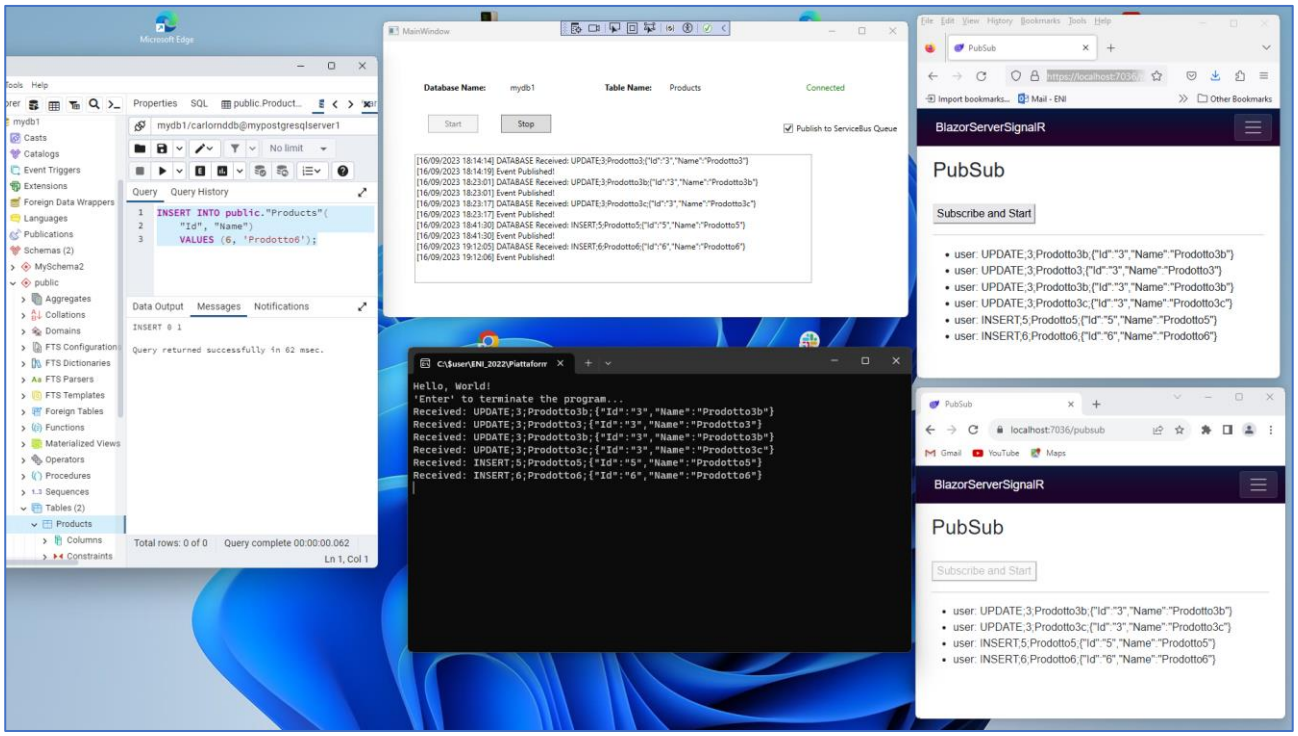
“What is Azure Web PubSub service?”

https://learn.microsoft.com/en-us/azure/azure-web-pubsub/overview

Integrated view of the demo in operation

An E2E "demo" implementation has been prototyped, with the following associations with respect to the architectural building blocks outlined in the introductory section of this article:

Building Block Component of the Schematic	**"Demo" implementation**
DB	A PostgreSQL Flexible database in PaaS on Azure, table “Products”
Proc1	.NET WPF "**EventHubPublisher.sln**" solution (.NET Framework 4.8), with the function of both receiving PostgreSQL notifications and loading data into the MOM (here Azure Service Bus)
MOM	Implementing Azure Service Bus, using queues, on Azure
Proc2	.NET Core "**publisher.sln**" solution (console application), acting as both a **dequeuer** from the Azure Service Bus and a loader in the Azure Web PubSub service notification system
WEBNOTIFY	Implementing Azure Web PubSub Service for ASP.NET Core Web App Notification
ASP.NET Core Web App	Solution ASP.NET Core "**BlazorServerSignalRSoln.sln**"; Web App ASP.NET Core Blazor Server that receives notifications from the Azure Web PubSub service subsystem and presents them on screen.



Appendix - Receiving Events in a Background Task in a Web Application ASP.NET Core

As already mentioned, the "**Proc2**" process of the proposed scheme represents a long-running process that "dequeue" events/messages from the adopted MOM platform, and which, upon the arrival of a message/event, republishes it on a centralized notification management system (in this article called "WEBNOTIFY") to the browser sessions opened on the target web application. This "long-running" process can be supported on the Azure platform in different ways, as described in a specific section of this article.

However, it can also be implemented as an IHostedService/BackgroundService within the same ASP.NET Core web application used by end users.

Note: In the following, we assume a web application ASP.NET Core Blazor Server.

The following is a solution scheme that provides (by way of example) the implementation - in the context ASP.NET Core web app - of a "long running" process that "consume" messages from the Azure Service Bus.

Note: in the case of creating a "daemon" that listens (understood here as part of a ASP.NET Core web app), I think it is better to use the "BackgroundService" abstraction, rather than the "IHostedService".

Version 1

<<

You can implement the ServiceBusReceiver globally in your ASP.NET Core Blazor Server web application rather than within a specific component. This approach allows you to create a background service or a hosted service that continuously receives and processes messages from the Azure Service Bus queue, making it available application-wide. Here's how you can do it:

Step 1: Create a Background Service

First, create a background service or hosted service that will manage the message receiving and processing. You can do this by creating a class that inherits from BackgroundService or IHostedService. Here's an example:

```
using System;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Azure.Messaging.ServiceBus;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

public class ServiceBusReceiverService : BackgroundService
{
    private readonly IServiceProvider _serviceProvider;
    private readonly CancellationTokenSource _cancellationTokenSource = new CancellationTokenSource();

    public ServiceBusReceiverService(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
```

```
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            using (var scope = _serviceProvider.CreateScope())
            {
                var messageReceiver = scope.ServiceProvider.GetRequiredService<ServiceBusReceiver>();
                try
                {
                    // Receive a message
                    ServiceBusReceivedMessage message = await messageReceiver.ReceiveMessageAsync();

                    if (message != null)
                    {
                        // Process the message
                        string messageBody = Encoding.UTF8.GetString(message.Body);

                        // Handle the message processing here
                    }
                }
                catch (Exception ex)
                {
                    // Handle exceptions here
                }
            }

            // Add a delay to control message polling frequency
            await Task.Delay(TimeSpan.FromSeconds(5), stoppingToken);
        }
    }

    public override async Task StopAsync(CancellationToken cancellationToken)
    {
        _cancellationTokenSource.Cancel();
        await base.StopAsync(cancellationToken);
    }
}
```

Step 2: Register the Background Service

In your Startup.cs or an appropriate configuration file, register the ServiceBusReceiverService as a hosted service:

```
public void ConfigureServices(IServiceCollection services)
{
    // Other services configurations

    services.AddSingleton(x =>
    {
        string connectionString = "YOUR_CONNECTION_STRING_HERE"; // Your Azure Service Bus connection string
        string queueName = "YOUR_QUEUE_NAME_HERE"; // The name of the queue you want to receive messages from

        return new ServiceBusClient(connectionString);
    });

    services.AddSingleton(x =>
    {
        var client = x.GetRequiredService<ServiceBusClient>();
        string queueName = "YOUR_QUEUE_NAME_HERE"; // The name of the queue you want to receive messages from

        return client.CreateReceiver(queueName);
    });

    services.AddHostedService<ServiceBusReceiverService>();
}
```

Replace "YOUR_CONNECTION_STRING_HERE" with your Azure Service Bus connection string and "YOUR_QUEUE_NAME_HERE" with the name of the queue you want to receive messages from.

Step 3: Configure Message Processing

In the **ServiceBusReceiverService**, you can handle the message processing logic as needed. You can also implement error handling and other custom behavior based on your application requirements.

With this approach, the ServiceBusReceiverService will run as a background service when your Blazor Server application starts and continuously receive and process messages from the Azure Service Bus queue. It's a global service available application-wide.

>>

Version 2

<<

To implement an Azure Service Bus message receiver in an ASP.NET Blazor Server web application at the application level (rather than within a specific component), you can use a background service or hosted service in ASP.NET Core. This service will run independently of any specific page or component and can handle message reception globally.

Here are the steps to achieve this:

1. **Create a Hosted Service**:
    - In your Blazor Server application, create a hosted service. This is a long-running background task that can process Service Bus messages independently.

```
// Create a new class for your hosted service
public class ServiceBusReceiverService : BackgroundService
{
    private readonly IServiceProvider _serviceProvider;

    public ServiceBusReceiverService(IServiceProvider serviceProvider)
    {
        _serviceProvider = serviceProvider;
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        // Initialize your Service Bus client here
        var serviceBusClient = new ServiceBusClient("your-connection-string");

        // Create a ServiceBusReceiver to receive messages
        var receiver = serviceBusClient.CreateReceiver("your-queue-name");

        while (!stoppingToken.IsCancellationRequested)
        {
            // Receive and process messages here
            var messages = await receiver.ReceiveMessagesAsync(1);

            foreach (var message in messages)
            {
                try
                {
                    // Process the message
                    // ...

                    // Complete the message to remove it from the queue
                    await receiver.CompleteMessageAsync(message);
                }
                catch
                {
                    // Handle exceptions
                    // ...
                }
            }
        }
    }
}
```

2. **Register the Hosted Service**:
    - In your Startup.cs file, add the hosted service to the DI container.

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    // Add your hosted service to the DI container
    services.AddHostedService<ServiceBusReceiverService>();
}
```

3. **Start and Stop the Service**:
    - ASP.NET Core will automatically start and stop the hosted service when your application starts and stops.

Now, the **ServiceBusReceiverService** will run in the background, listening for messages from Azure Service Bus. You can customize the service to handle messages as needed and add any error handling or logging as required.

Remember to replace "your-connection-string" and "your-queue-name" with your actual Azure Service Bus connection string and queue name.

This setup allows you to receive messages globally at the ASP.NET Core web application level, independently of any specific Blazor component.

>>

Note: in our scenario, the logic of the aforementioned "**ServiceBusReceiverService**" must be such that when a message/event arrives, it republishes it on a centralized notification management system (in this article called "WEBNOTIFY") to the browser sessions opened on the target web application.

Note

“ASP. NET Core - IHostedService and BackgroundService”

https://girishgodage.in/blog/customize-hostedservices

“Background tasks with hosted services in ASP.NET Core”

https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-7.0&tabs=visual-studio

Note: in the case of creating a "daemon" that listens, I think it's better to use the "BackgroundService" abstraction.

Appendix – Background Jobs in Azure (with different options)

Reference: “Background jobs”

https://learn.microsoft.com/en-us/azure/architecture/best-practices/background-jobs

For the implementation of the aforementioned "Proc1" and "Proc2" processes, since they are "long-running" processes and not primarily aimed at supporting a "GUI" (graphical user interface), their implementation can – in the context of Azure – take advantage of one of the mechanisms indicated, for example, at the link proposed here, relating to Azure "Background Jobs".

In particular, the options presented in the indicated document are the following:

<<

- **Azure Web Apps and WebJobs**. You can use WebJobs to execute custom jobs based on a range of different types of scripts or executable programs within the context of a web app.
- **Azure Functions**. You can use functions for background jobs that don't run for a long time. Another use case is if your workload is already hosted on App Service plan and is underutilized.
- **Azure Virtual Machines**. If you have a Windows service or want to use the Windows Task Scheduler, it is common to host your background tasks within a dedicated virtual machine.
- **Azure Batch**. Batch is a platform service that schedules compute-intensive work to run on a managed collection of virtual machines. It can automatically scale compute resources.
- **Azure Kubernetes Service (AKS)**. Azure Kubernetes Service provides a managed hosting environment for Kubernetes on Azure.
- **Azure Container Apps**. Azure Container Apps enables you to build serverless microservices based on containers.

>>

Appendix - Azure Service Bus, Azure Event Hub, Azure Event Grid comparison

The systems compared in the following linked documents are as follows:

- “Azure Service Bus Messaging”
- “Azure Event Grid”
- “Azure Event Hubs”

These are all potentially usable options (with their "pros" and "cons") to implement the architectural component called "MOM" in the high-level scheme initially proposed in this article.

“Asynchronous messaging options”

https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/messaging

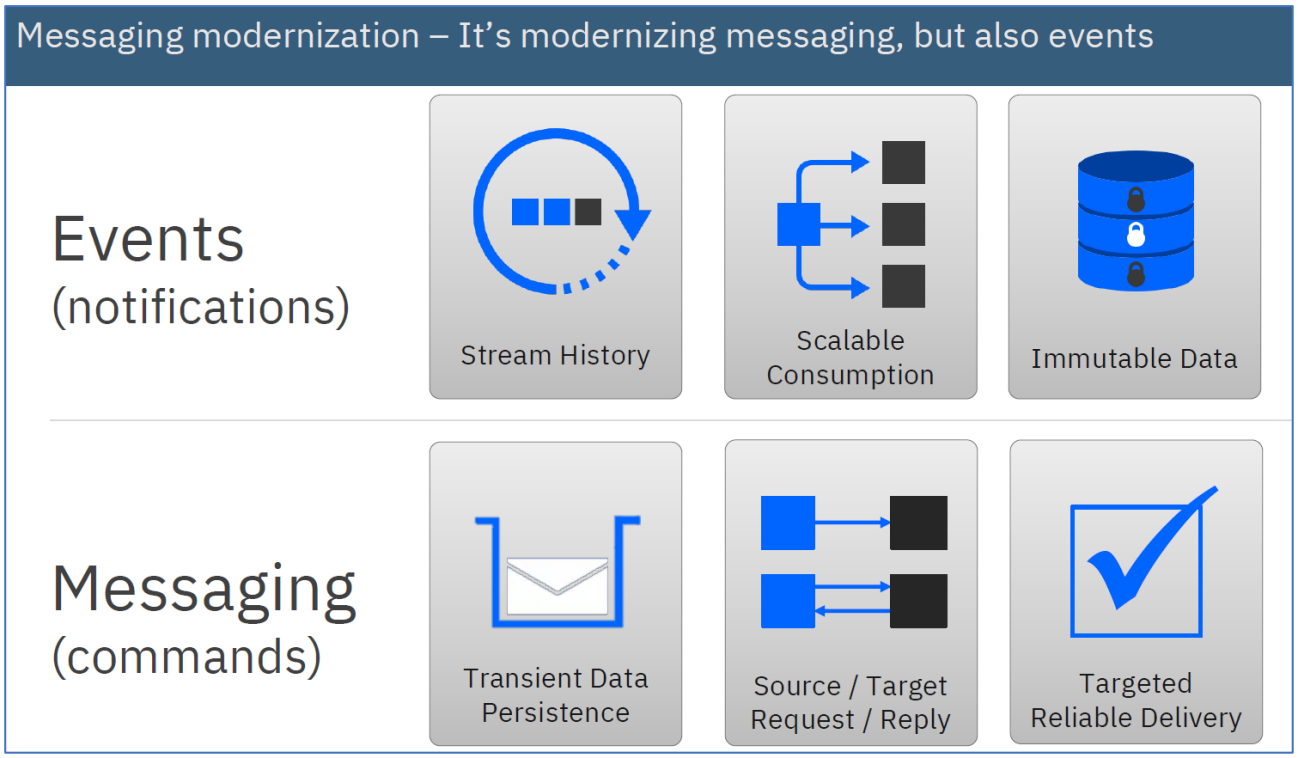
“Choose between Azure messaging services - Event Grid, Event Hubs, and Service Bus”

https://learn.microsoft.com/en-us/azure/service-bus-messaging/compare-messaging-services

Service	Purpose	Type	When to use
Event Grid	Reactive programming	Event distribution (discrete)	React to status changes
Event Hubs	Big data pipeline	Event streaming (series)	Telemetry and distributed data streaming
Service Bus	High-value enterprise messaging	Message	Order processing and financial transactions

Note: the first link above distinguishes (at an "architectural" level) between "Command" and "Event" messages; the second link distinguishes between "Event" (in line with the homonymous category of the first reference), and "Message", referring to what in the first classification are referred to as "command" messages.

The associations between the concept of "Event" and that of "Notification" and the concept of "Message" and that of "Command" (or "Operation") can also be found in this figure:



See also:

"Messaging and Event Streaming Use Cases",
https://community.ibm.com/community/user/integration/viewdocument/messaging-and-event-streaming-use-c?CommunityKey=183ec850-4947-49c8-9a2e-8e7c7fc46c64&tab=librarydocuments).