Write code that uses Service Bus topics

5 minutes

In a distributed application, some messages need to be sent to a single recipient component. Other messages need to reach more than one destination.

Let's discuss what happens when a user cancels a pizza order. This is a little different than placing the initial order. In that case, we wanted to wait until the order cleared payment processing before sending the order on to the next steps (which is having it prepared and cooked at the local storefront). But for the cancel operation, we are going to notify both the storefront *and* the payment processor at the same time. This approach minimizes the chances that we waste ingredients or delivery driver time.

To allow multiple components to receive the same message, we'll use an Azure Service Bus topic.

Code with topics vs. code with queues

If you want every message sent to be delivered to all subscribing components, you'll use topics. Writing code that uses topics is a way to replace queues. You will use the same **Microsoft.Azure.ServiceBus** NuGet package, configure connection strings, and use asynchronous programming patterns.

However, you'll use the `TopicClient` class instead of the `QueueClient` class to send messages and the `SubscriptionClient` class to receive messages.

Setting filters on subscriptions

If you want to control that specific messages sent to the topic are delivered to particular subscriptions, you can place filters on each subscription in the topic. In the pizza application, for instance, our storefronts are running Universal Windows Platform (UWP) applications. Each store can subscribe to the "OrderCancellation" topic but filter for its own StoreId. We save internet bandwidth because we are not sending unnecessary messages to distant store locations. Meanwhile, the payment processing component subscribes to all our cancellation messages.

Filters can be one of three types:

- **Boolean Filters.** The `TrueFilter` ensures that all messages sent to the topic are delivered to the current subscription. The `FalseFilter` ensures that none of the messages are delivered to the current subscription. (This effectively blocks or switches off the subscription.)
- **SQL Filters.** A SQL filter specifies a condition by using the same syntax as a `WHERE` clause in a SQL query. Only messages that return `True` when evaluated against this subscription will be delivered to the subscribers.
- **Correlation Filters.** A correlation filter holds a set of conditions that are matched against the properties of each message. If the property in the filter and the property on the message have the same value, it is considered a match.

For our StoreId filter, we *could* use a SQL filter. SQL filters are the most flexible, but they're also the most computationally expensive and could slow down our Service Bus throughput. In this case, we choose a correlation filter instead.

TopicClient example

In any sending or receiving component, you should add the following `using` statements to any code file that calls a Service Bus topic:

C# Copy

```
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.ServiceBus;
```

To send a message, start by creating a new `TopicClient` object and passing it the connection string and the name of the topic:

C# Copy

```
topicClient = new TopicClient(TextAppConnectionString, "GroupMessageTopic");
```

You can send a message to the topic by calling the `TopicClient.SendAsync()` method and passing the message. As with queues, the message must be in the form of a UTF-8 encoded string:

C# Copy

```
string message = "Cancel! I can't believe you use canned mushrooms!";
var encodedMessage = new Message(Encoding.UTF8.GetBytes(message));
await topicClient.SendAsync(encodedMessage);
```

To receive messages, you must create a `SubscriptionClient` object, not a `TopicClient` object, and pass it the connection string, the name of the topic, **and** the name of the subscription:

C# Copy

```
subscriptionClient = new SubscriptionClient(ServiceBusConnectionString,
"GroupMessageTopic", "NorthAmerica");
```

Then register a message handler - this is the asynchronous method in your code that processes the retrieved message.

C# Copy

```
subscriptionClient.RegisterMessageHandler(MessageHandler, messageHandlerOptions);
```

Within the message handler, call the `SubscriptionClient.CompleteAsync()` method to remove the message from the queue:

C# Copy

```
await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);
```

Next unit: Exercise - Write code that uses Service Bus topics

Continue >