To design an Order system with CQRS, event sourcing, and eventual consistency using Azure Service Bus and Azure SQL Database, here's a best practices flow:

**1. CQRS Pattern**

* **Command Side**:
  + **Service Bus**: Commands are sent to a specific queue in Azure Service Bus.
  + **Command Handlers**: Handle the commands, perform business logic, and persist changes in the write model (Azure SQL Database).
  + **Event Generation**: After processing a command, an event is generated to reflect the change.
* **Query Side**:
  + **Read Models**: Separate read models optimized for querying are maintained in Azure SQL Database. These are updated asynchronously through events.

**2. Event Sourcing**

* **Event Store**:
  + Store all events in a durable storage (e.g., Azure Blob Storage or a dedicated table in Azure SQL Database).
  + Events represent state changes and are used to reconstruct the current state of an entity.
* **Event Publishing**:
  + After events are stored, they are published to the Service Bus to notify interested systems.

**3. Eventual Consistency**

* **Message Delivery**:
  + Use Azure Service Bus for reliable, decoupled communication between components.
  + Ensure idempotency in event handlers to handle potential duplicate events.
* **Read Model Update**:
  + As events are processed, the read models are updated asynchronously.
  + Use a separate process to handle eventual consistency, ensuring read models eventually reflect the latest state.

**4. Azure SQL Database:**

* **Write Model**:
  + Optimized for transactional operations and consistency.
  + Use stored procedures or transactions to ensure atomicity.
* **Read Model**:
  + Denormalized for performance, optimized for queries.
  + Regularly synchronized with the latest events.

**5. Best Practices:**

* **Idempotency**: Ensure command handlers, event handlers, and processes are idempotent to handle retries and duplicates.
* **Message Ordering**: Use Azure Service Bus sessions or partitions to maintain the order of events.
* **Error Handling**: Implement robust error handling, retries, and dead-letter queues for failed messages.
* **Versioning**: Plan for event and command versioning to support future changes.
* **Monitoring**: Use Azure Monitor, Application Insights, and logging frameworks to monitor the system's health and performance.

To implement an Event Store in .NET Core with SQL Server, you'll model the event store using entities that capture the essential information about each event. Below is an example of how

you might structure this in terms of .NET Core classes and SQL Server schema.

**Event Store .NET Core Modeling**

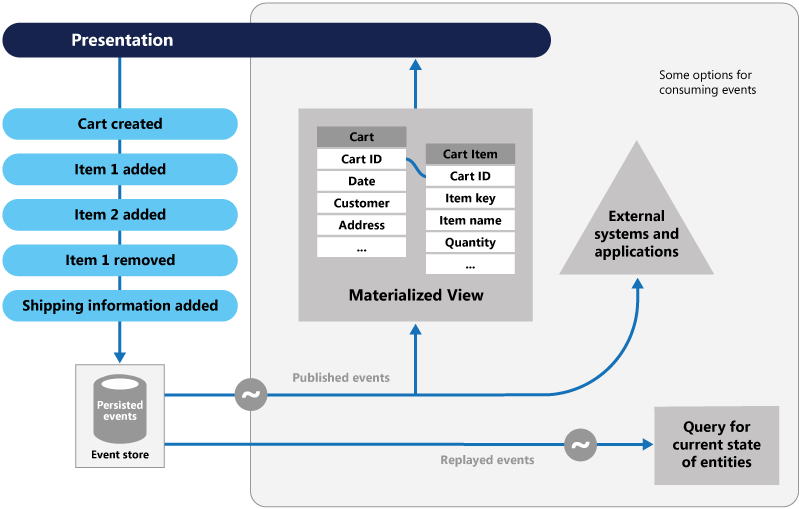


Diagram of a process

Description automatically generated

 **Write Side (Command)**:

* Commands are sent to the system (e.g., CreateOrder).
* These commands are processed, and events are generated and stored in the Event Store.
* Events are then published to an Event Bus (e.g., Azure Service Bus).

 **Event Bus**:

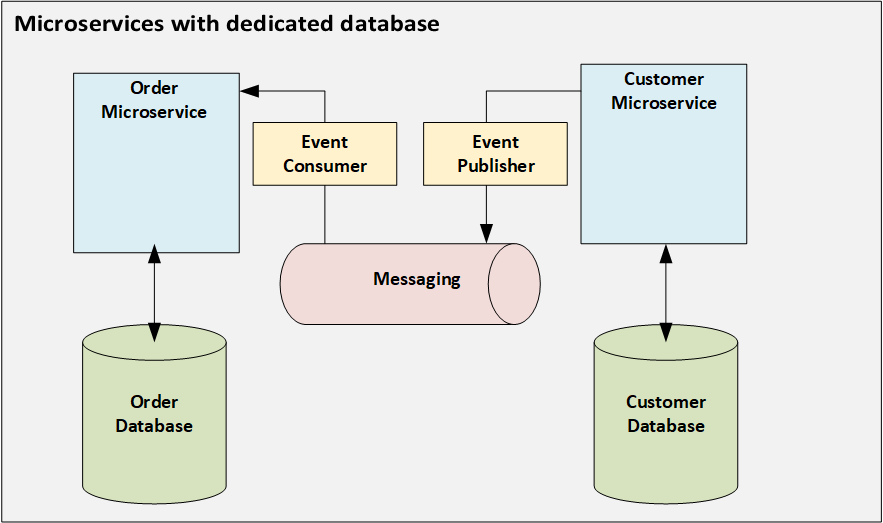
* Events are sent to Azure Service Bus, which acts as a message broker to distribute events to interested subscribers.

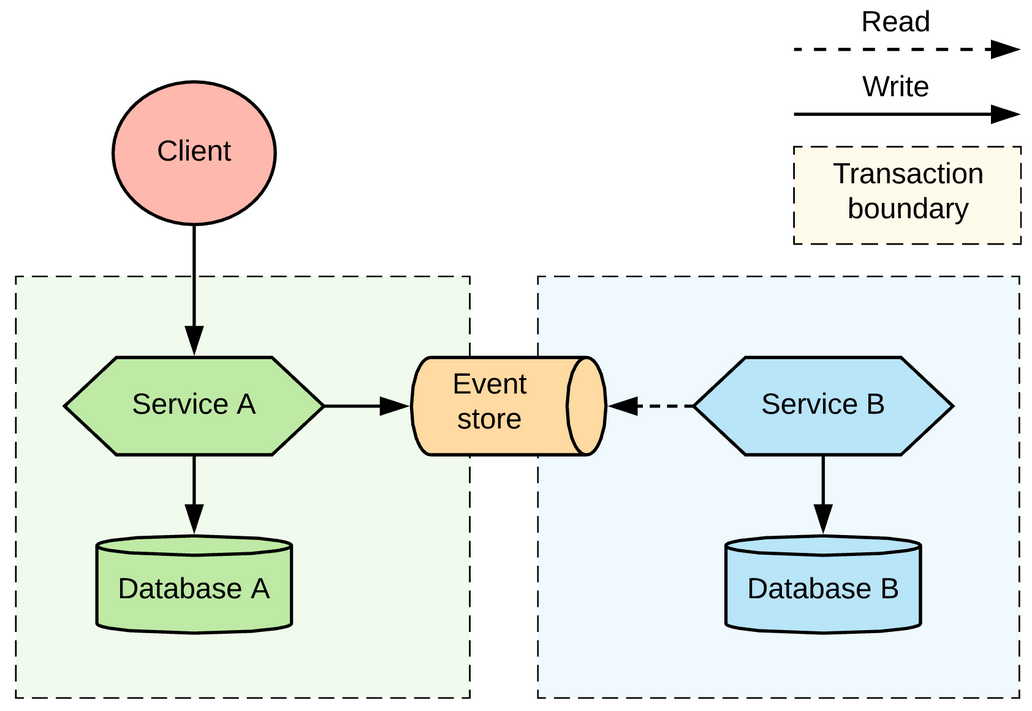
 **Read Side (Query)**:

* Subscribers listen to events from Azure Service Bus.

Events are processed to update the Read Database (Materialized View), providing eventual consistency.

|  |
| --- |
| public abstract class Event  {  public Guid Id { get; private set; } = Guid.NewGuid();  public string AggregateId { get; private set; }  public DateTime OccurredOn { get; private set; } = DateTime.UtcNow;  public int Version { get; private set; }  protected Event(string aggregateId, int version)  {  AggregateId = aggregateId;  Version = version;  }  } |
| public class OrderCreatedEvent : Event  {  public string OrderId { get; private set; }  public string CustomerId { get; private set; }  public decimal TotalAmount { get; private set; }  public OrderCreatedEvent(string aggregateId, string orderId, string customerId, decimal totalAmount, int version)  : base(aggregateId, version)  {  OrderId = orderId;  CustomerId = customerId;  TotalAmount = totalAmount;  }  } |
| public class AzureServiceBusEventPublisher  {  private readonly ITopicClient \_topicClient;  public AzureServiceBusEventPublisher(string connectionString, string topicName)  {  \_topicClient = new TopicClient(connectionString, topicName);  }  public async Task PublishEventAsync(Event @event)  {  var message = new Message(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(@event)))  {  Label = @event.GetType().Name  };  await \_topicClient.SendAsync(message);  }  } |
| public class OrderEventHandler  {  private readonly string \_connectionString;    public OrderEventHandler(string connectionString)  {  \_connectionString = connectionString;  }  public async Task HandleAsync(Message message)  {  var eventType = message.Label;  var eventData = Encoding.UTF8.GetString(message.Body);  if (eventType == nameof(OrderCreatedEvent))  {  var orderCreatedEvent = JsonConvert.DeserializeObject<OrderCreatedEvent>(eventData);  await UpdateMaterializedViewAsync(orderCreatedEvent);  }  // Handle other event types as needed  }  private async Task UpdateMaterializedViewAsync(OrderCreatedEvent orderCreatedEvent)  {  using (var connection = new SqlConnection(\_connectionString))  {  var command = new SqlCommand("INSERT INTO OrdersReadModel (OrderId, CustomerId, TotalAmount) VALUES (@OrderId, @CustomerId, @TotalAmount)", connection);  command.Parameters.AddWithValue("@OrderId", orderCreatedEvent.OrderId);  command.Parameters.AddWithValue("@CustomerId", orderCreatedEvent.CustomerId);  command.Parameters.AddWithValue("@TotalAmount", orderCreatedEvent.TotalAmount);  connection.Open();  await command.ExecuteNonQueryAsync();  }  }  } |
| CREATE TABLE OrdersReadModel  (  OrderId NVARCHAR(255) PRIMARY KEY,  CustomerId NVARCHAR(255),  TotalAmount DECIMAL(18, 2)  ); |





A diagram of a business process

Description automatically generated

A diagram of a software source

Description automatically generated

**a. Event Base Class**

|  |
| --- |
| public abstract class Event  {  public Guid Id { get; private set; } = Guid.NewGuid(); // Unique event ID  public DateTime OccurredOn { get; private set; } = DateTime.UtcNow; // Event timestamp  public string EventType { get; private set; } // Name of the event type  public string AggregateId { get; private set; } // ID of the aggregate root  public int Version { get; private set; } // Version of the event  protected Event(string aggregateId, string eventType, int version)  {  AggregateId = aggregateId;  EventType = eventType;  Version = version;  }  } |

**b. Specific Event Example**

|  |
| --- |
| public class OrderCreatedEvent : Event  {  public string OrderId { get; private set; }  public string CustomerId { get; private set; }  public decimal TotalAmount { get; private set; }  public OrderCreatedEvent(string aggregateId, string orderId, string customerId, decimal totalAmount, int version)  : base(aggregateId, nameof(OrderCreatedEvent), version)  {  OrderId = orderId;  CustomerId = customerId;  TotalAmount = totalAmount;  }  } |

**c. Event Store Repository**

|  |
| --- |
| public interface IEventStore  {  Task SaveEventAsync(Event @event);  Task<IEnumerable<Event>> GetEventsAsync(string aggregateId);  }  public class SqlEventStore : IEventStore  {  private readonly string \_connectionString;  public SqlEventStore(string connectionString)  {  \_connectionString = connectionString;  }  public async Task SaveEventAsync(Event @event)  {  using var connection = new SqlConnection(\_connectionString);  var command = new SqlCommand("INSERT INTO Events (Id, AggregateId, EventType, Data, Version, OccurredOn) VALUES (@Id, @AggregateId, @EventType, @Data, @Version, @OccurredOn)", connection);  command.Parameters.AddWithValue("@Id", @event.Id);  command.Parameters.AddWithValue("@AggregateId", @event.AggregateId);  command.Parameters.AddWithValue("@EventType", @event.EventType);  command.Parameters.AddWithValue("@Data", JsonConvert.SerializeObject(@event));  command.Parameters.AddWithValue("@Version", @event.Version);  command.Parameters.AddWithValue("@OccurredOn", @event.OccurredOn);  connection.Open();  await command.ExecuteNonQueryAsync();  }  public async Task<IEnumerable<Event>> GetEventsAsync(string aggregateId)  {  var events = new List<Event>();  using var connection = new SqlConnection(\_connectionString);  var command = new SqlCommand("SELECT \* FROM Events WHERE AggregateId = @AggregateId ORDER BY Version ASC", connection);  command.Parameters.AddWithValue("@AggregateId", aggregateId);  connection.Open();  using var reader = await command.ExecuteReaderAsync();  while (await reader.ReadAsync())  {  var eventType = reader.GetString(reader.GetOrdinal("EventType"));  var data = reader.GetString(reader.GetOrdinal("Data"));  // Assuming you have a way to deserialize the events based on type  var @event = JsonConvert.DeserializeObject(data, Type.GetType(eventType));  events.Add(@event as Event);  }  return events;  }  } |

**2. SQL Server Schema Example**

|  |
| --- |
| CREATE TABLE Events  (  Id UNIQUEIDENTIFIER PRIMARY KEY,  AggregateId NVARCHAR(255) NOT NULL,  EventType NVARCHAR(255) NOT NULL,  Data NVARCHAR(MAX) NOT NULL,  Version INT NOT NULL,  OccurredOn DATETIME2 NOT NULL  ); |

**a. Events Table**

* **Id**: A unique identifier for each event.
* **AggregateId**: The ID of the aggregate root (e.g., Order ID).
* **EventType**: The type of the event (e.g., OrderCreatedEvent).
* **Data**: JSON-serialized event data.
* **Version**: The version of the aggregate when the event occurred (used for optimistic concurrency).
* **OccurredOn**: The timestamp when the event occurred.

**3. Example Usage**

When an order is created, the following steps occur:

1. **Create Event**: An OrderCreatedEvent is instantiated with the necessary details.
2. **Save Event**: The event is saved to the Events table via the SqlEventStore.
3. **Publish Event**: The event can then be published to Azure Service Bus or handled by other components of the system.

**4. Rebuilding Aggregate from Events**

To rebuild the current state of an order, you would:

* Retrieve all events for a given AggregateId.
* Replay them in order (based on the Version) to reconstruct the aggregate's state.

This setup ensures that all changes are recorded as immutable events, allowing for powerful auditing, debugging, and the ability to rebuild the system's state from the event store.