Sending and receiving information between two systems in order to integrate the systems usually has different solutions. Each solution has its strengths and weaknesses and the right solution should be chosen according to the needs and conditions. In the following, two solutions for Stage 4 have been examined.

**Solution 1: Direct API Integration**

**Description:** In this solution, we directly integrate with Mechanic Supplies' API by making HTTP requests from our Part Store API. We use the Mechanic Supplies client in our service layer to send and receive orders.

**Advantages:**

* Simple and straightforward implementation.
* Real-time communication between systems.
* No additional infrastructure required.

**Disadvantages:**

* Dependent on Mechanic Supplies' API availability and performance.
* May introduce latency if Mechanic Supplies' API is slow and danger of request timeout.
* Requires Mechanic Supplies to expose their API publicly.

**Solution 2: Asynchronous Messaging**

**Description:** We introduce a message broker, such as RabbitMQ or Azure Service Bus, to handle asynchronous communication between Part Store and Mechanic Supplies. Orders are sent as messages to the broker, which Mechanic Supplies consumes.

**Advantages:**

* Decouples our system from Mechanic Supplies, reducing direct dependencies.
* Provides scalability and fault tolerance through asynchronous messaging.
* Allows for guaranteed delivery and load balancing.

**Disadvantages:**

* Requires additional infrastructure and setup for the message broker.
* Adds complexity to the system architecture.
* May introduce delays in order processing due to asynchronous nature.

Considering the flexibility and decoupling benefits, Solution 2 (Asynchronous Messaging) could be the preferred option. It strikes a balance between system independence and scalability, ensuring reliable communication between Parts Store and Mechanic Supplies. However, the final decision should consider the specific requirements, constraints, and capabilities of both systems.

In follow code I consider sample class for sending and receiving Order between 2 system using RabbitMQ:

1. **Sample Order Consumer class:**

public class MechanicSuppliesOrderConsumer

{

private readonly IModel \_channel;

private readonly IOrderRepository \_orderRepository;

public MechanicSuppliesOrderConsumer(IModel channel, IOrderRepository orderRepository)

{

\_channel = channel;

\_orderRepository = orderRepository;

}

public async Task Consume(BasicDeliverEventArgs eventArgs)

{

var body = eventArgs.Body;

var message = Encoding.UTF8.GetString(body);

var mechanicSuppliesOrder = JsonSerializer.Deserialize<MechanicSuppliesOrder>(message);

var order = mechanicSuppliesOrder.ToPartStoreOrder();

await \_orderRepository.CreateOrderAsync(order);

\_channel.BasicAck(eventArgs.DeliveryTag, false);

}

public async Task StartConsuming()

{

var consumer = new EventingBasicConsumer(\_channel, "parts-store-queue");

consumer.Received += async (model, eventArgs) => await Consume(eventArgs);

await \_channel.BasicConsume(consumer, true, "parts- store -queue");

}

}

1. **Sample Order sender Class which modifies OrderService.cs class:**

public class OrderService : IOrderService

{

private readonly IModel \_channel;

private readonly RabbitMQConfig \_rabbitMQConfig;

public OrderService(IModel channel, RabbitMQConfig rabbitMQConfig)

{

\_channel = channel;

\_rabbitMQConfig = rabbitMQConfig;

}

public async Task SendOrderToMechanicSuppliesAsync(Order order)

{

var mechanicSuppliesOrder = order.ToMechanicSuppliesOrder();

var message = JsonSerializer.Serialize(mechanicSuppliesOrder);

var properties = \_channel.CreateBasicProperties();

properties.Persistent = true;

var body = Encoding.UTF8.GetBytes(message);

\_channel.BasicPublish(\_rabbitMQConfig.Exchange, "parts-store-queue", properties, body);

}

}