















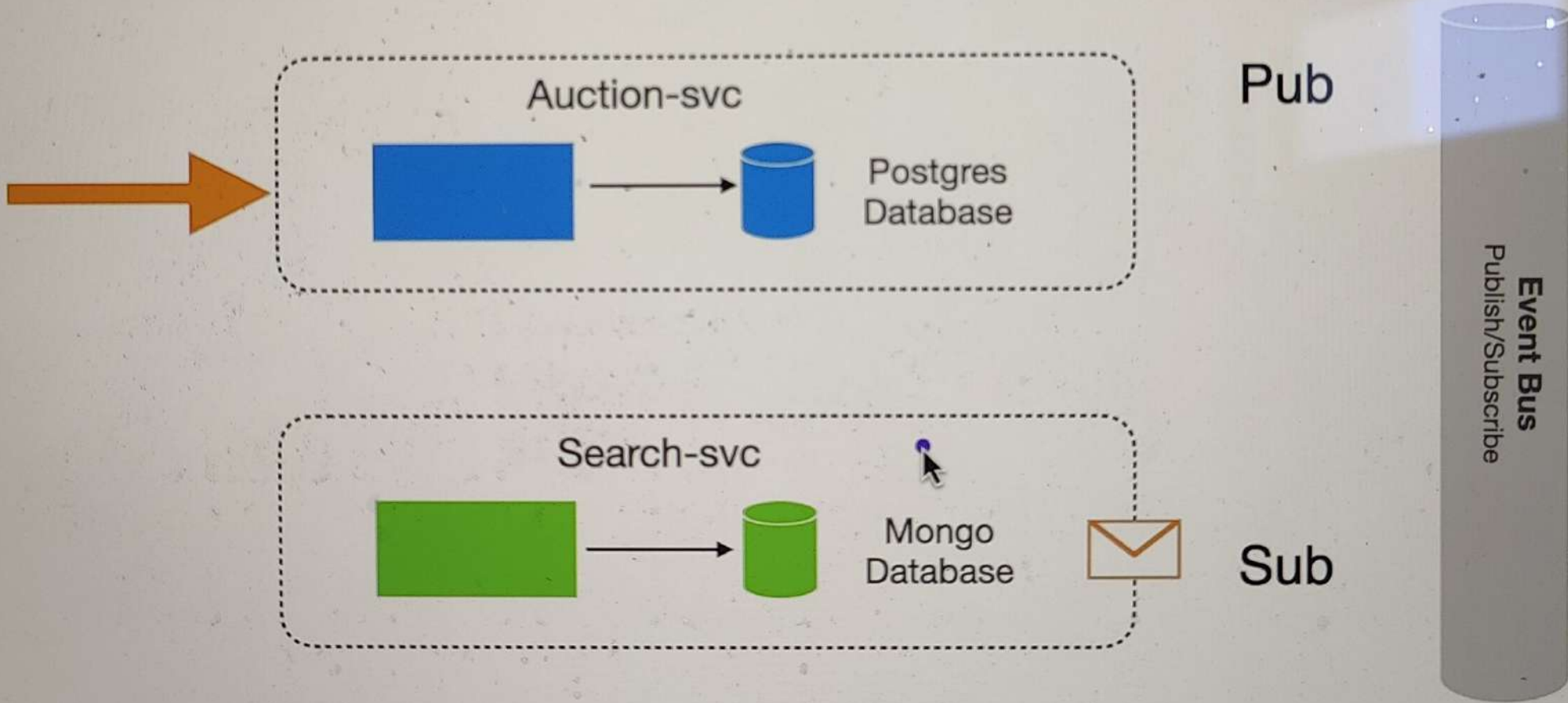
## Section 4: RabbitMQ



0 / 14 | 1hr 43min

- ☐ 34. Introduction to Section 4  
 6min
- ☐ 35. What is RabbitMQ  
 6min
- ☐ 36. Installing RabbitMQ  
 5min
- ☐ 37. Adding and configuring mass transit  
 10min
- ☐ 38. Adding the contracts  
 5min
- ☐ 39. Adding a consumer to consume a message from the Service bus  
 11min
- ☐ 40. Publishing the Auction Created event to the bus  
 9min
- ☐ 41. What could go wrong?  
 11min
- ☐ 42. Adding a message outbox  
 13min
- ☐ 43. Using message retries  
 7min
- ☐ 44. Consuming fault queues  
 9min
- ☐ 45. Challenge: Adding the update and delete consumers  
 5min
- ☐ 46. Challenge solution  
 7min
- ☐ 47. Summary of section 4  
 1min

POST /api/auctions

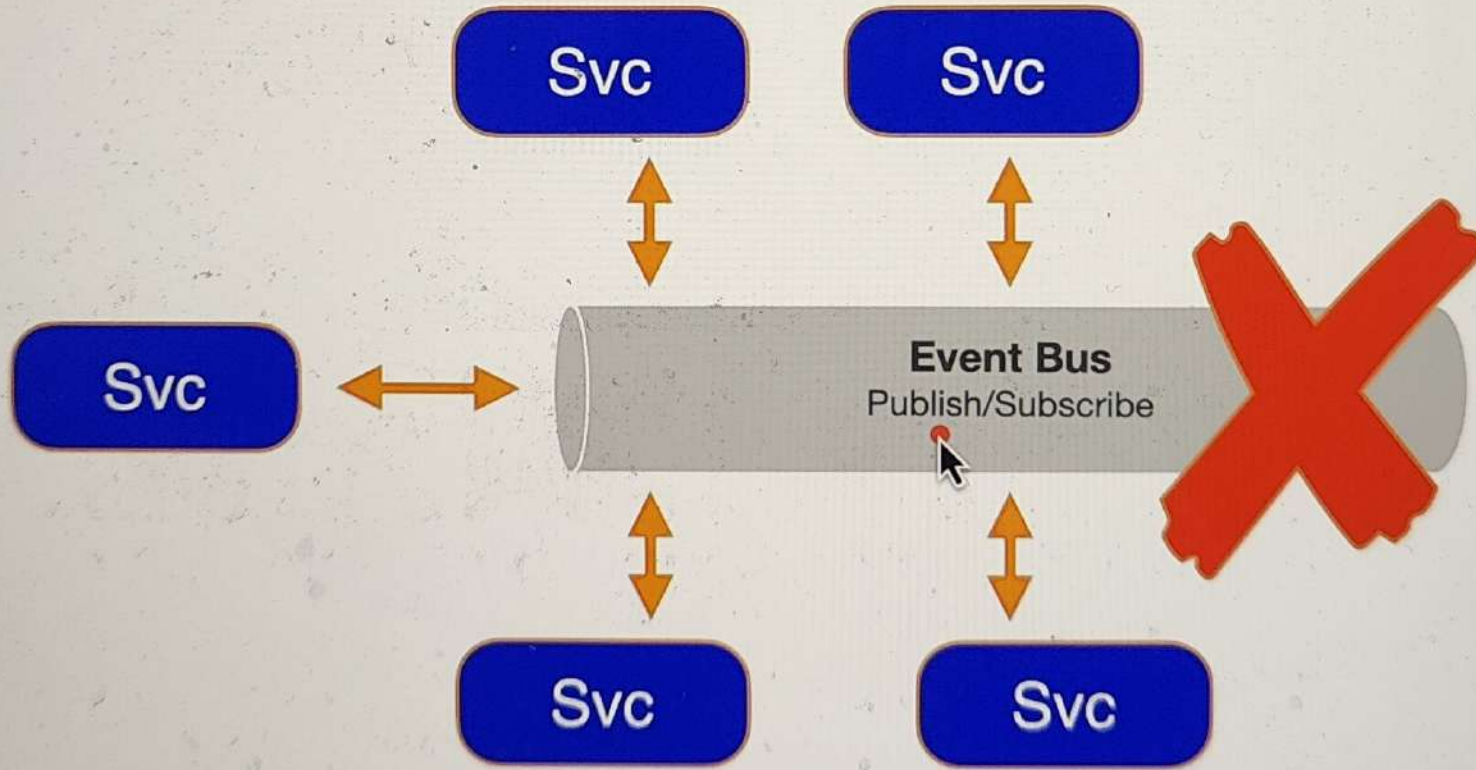


# Asynchronous Messaging

---

- No Request / Response
- Fire and forget
- Event model (publish / subscribe)
- Typically used for service to service messaging
- Transports (RabbitMQ, Azure Service Bus, Amazon SQS)
- Services only need to know about the bus
- More complex than sync messaging

# What if?





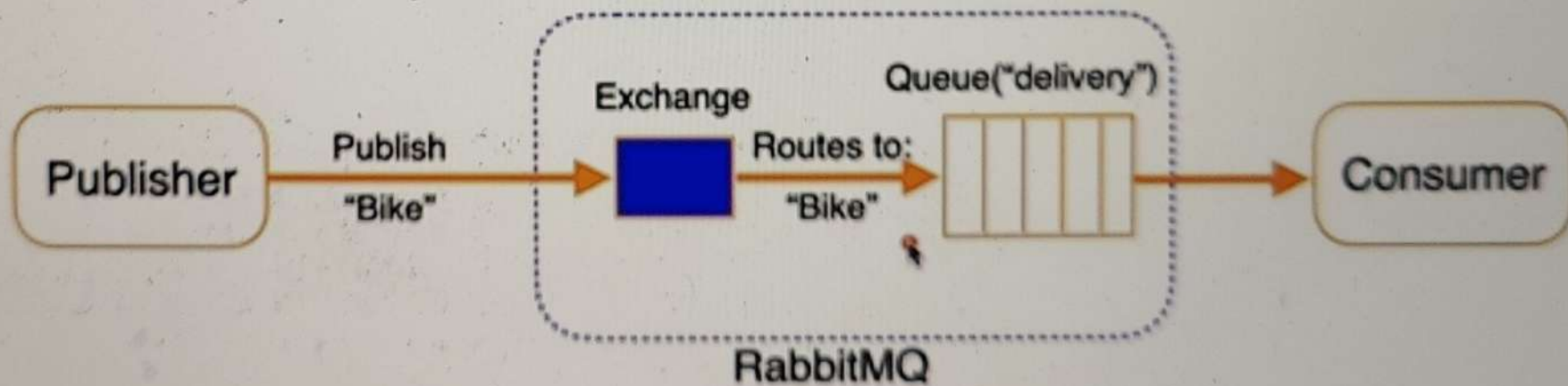
# RabbitMQ - What is it?

---

- Message Broker - accepts and forwards messages
- Producer/Consumer model (Pub/Sub)
- Messages are stored on queues (message buffer)
- Can use persistent storage
- Exchanges can be used for “routing” functionality
- Uses AMQP

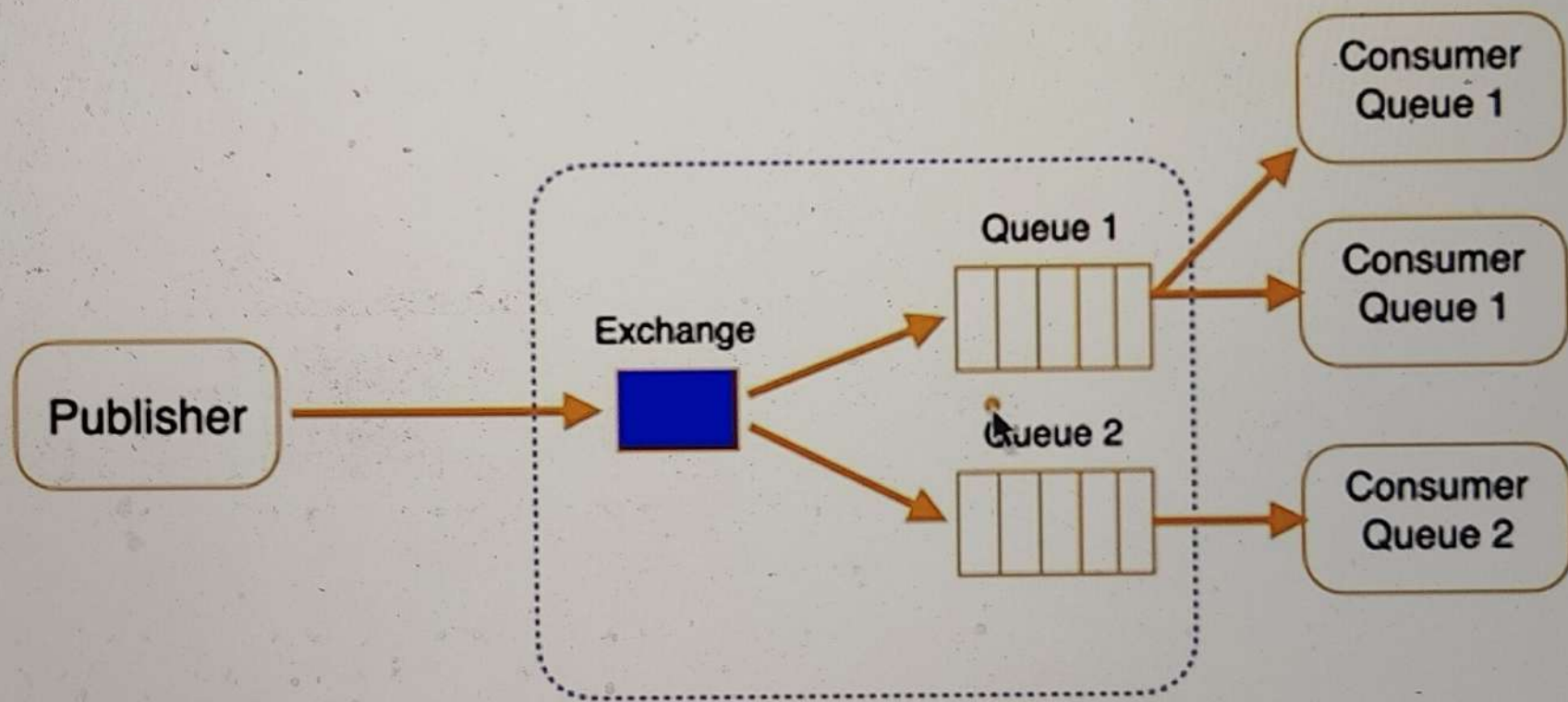
# Exchanges

- Direct
- Fanout
- Topic
- Header



# Exchanges

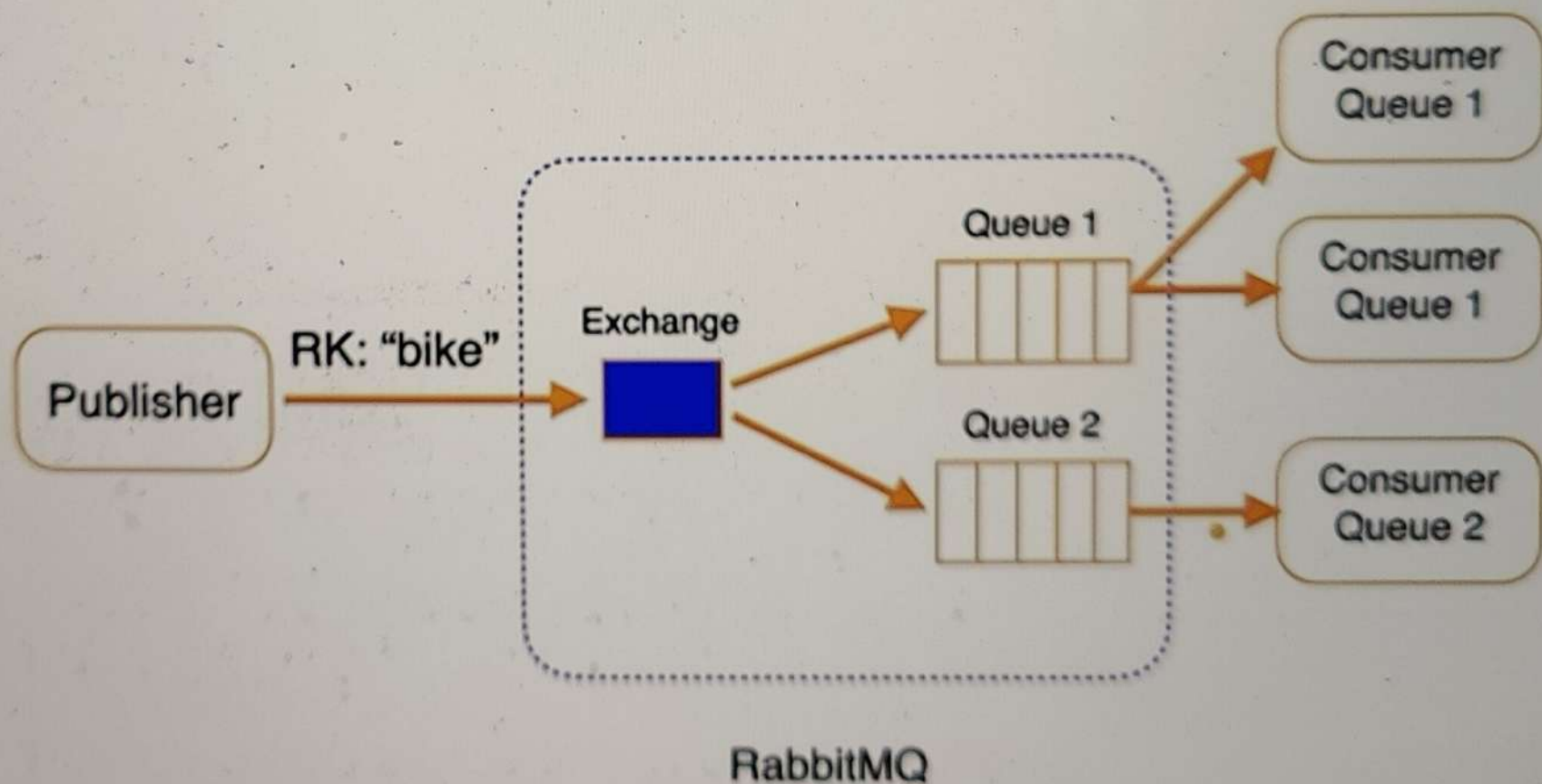
- Direct
- Fanout
- Topic
- Header



RabbitMQ

# Exchanges

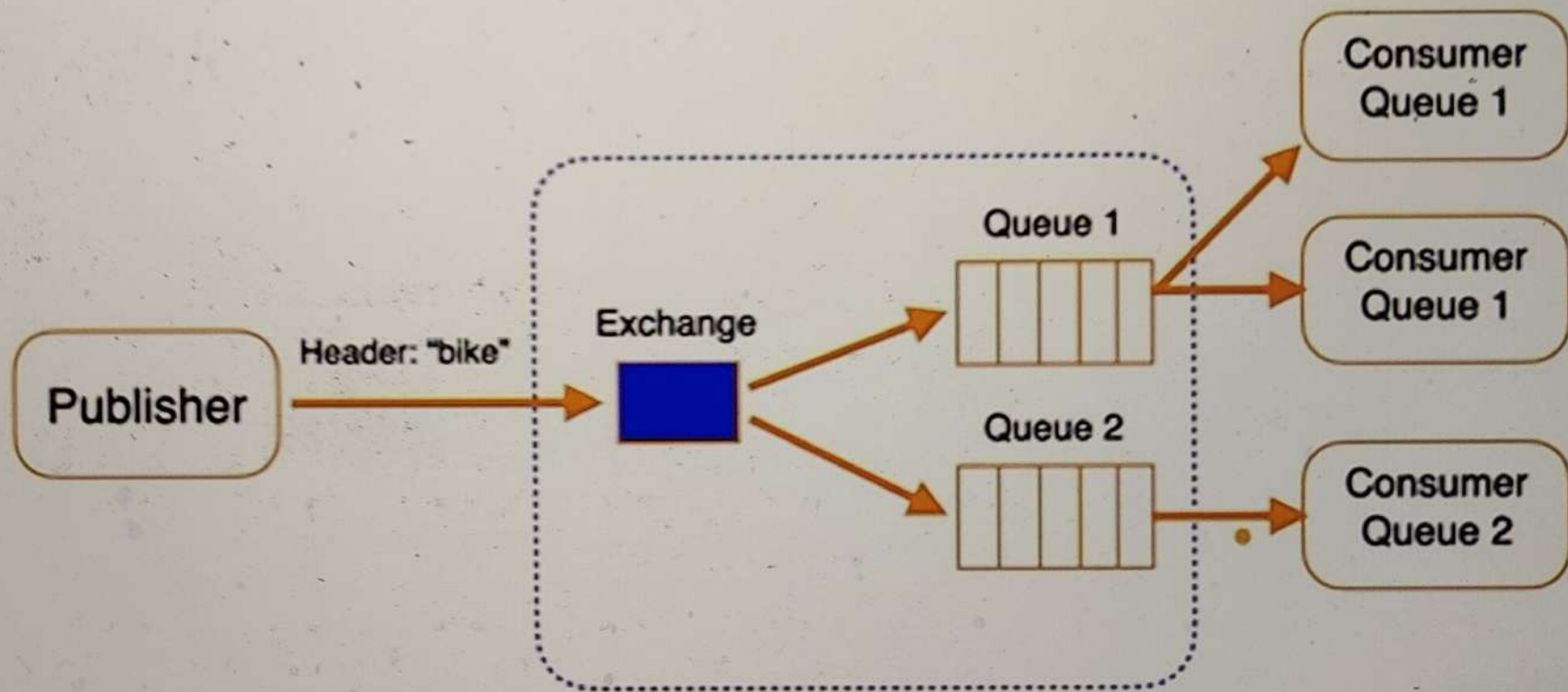
- Direct
- Fanout
- **Topic**
- Header





# Exchanges

- Direct
- Fanout
- Topic
- Header



RabbitMQ

# MassTransit



## Message Routing

Type-based publish/subscribe and automatic broker topology configuration



## Exception Handling

When an exception is thrown, messages can be retried, redelivered, or moved to an error queue



## Test Harness

Fast, in-memory unit tests with consumed, published, and sent message observers



## Observability

Native Open Telemetry (OTEL) support for end-to-end activity tracing



## Dependency Injection

Service collection configuration and scope service provider management



## Scheduling

Schedule message delivery using transport delay, Quartz.NET, and Hangfire



## Sagas, State Machines

Reliable, durable, event-driven workflow orchestration



## Routing Slip Activities

Distributed, fault-tolerant transaction choreography with compensation



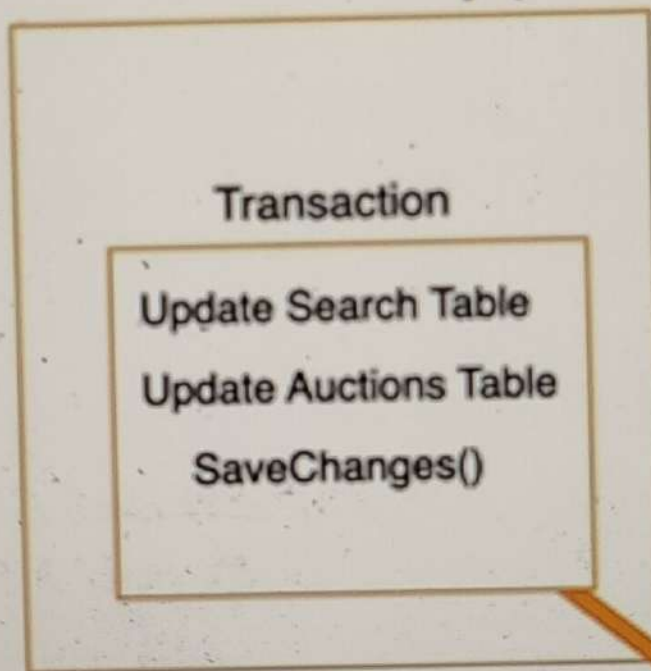
## Request, Response

Handle requests with fast, automatic response routing

POST api/auctions

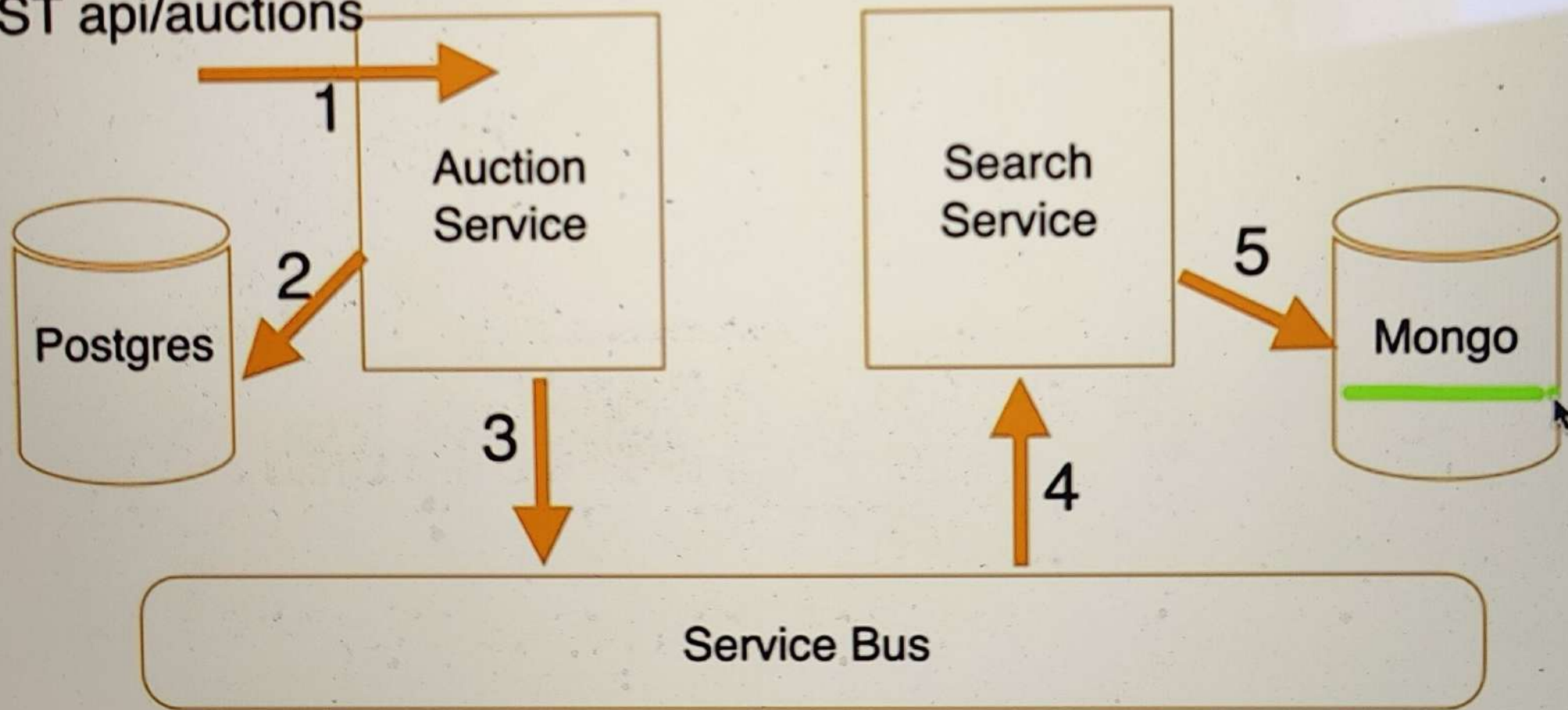


# AuctionApp



Atomicity  
Consistency  
Isolation  
Durability

POST api/auctions





# A small thought experiment...

If one of our services fails, and a user attempts to create an auction whilst that service is down, will the Auction Service DB and the Search Service DB be consistent?

					Data Consistency?
<del>Auction Service</del>	PostgresDB	Search Svc	Mongo DB	RabbitMQ	✓
Auction Service	<del>PostgresDB</del>	Search Svc	Mongo DB	RabbitMQ	✓
Auction Service	PostgresDB	<del>Search Svc</del>	Mongo DB	RabbitMQ	✓
Auction Service	PostgresDB	Search Svc	<del>Mongo DB</del>	RabbitMQ	<del>✗</del>
Auction Service	PostgresDB	Search Svc	Mongo DB	<del>RabbitMQ</del>	<del>✗</del>

# So what are the options?

---

Outbox

Retry