# What is Rabbit MQ

RabbitMQ is a reliable and mature messaging and streaming broker, which is easy to deploy on cloud environments, on premises, and on your local machine. It is currently used by millions worldwide.

* **Interoperable:** RabbitMQ supports several open standard protocols, including AMQP 1.0 (Advanced Message Queuing Protocol). and MQTT 5.0. There are multiple client libraries available, which can be used with your programming language of choice, just pick one. No vendor lock-in!
* **Flexible:** RabbitMQ provides many options you can combine to define how your messages go from the publisher to one or many consumers. Routing, filtering, streaming, federation, and so on, you name it.
* **Reliable:** With the ability to acknowledge message delivery and to replicate messages across a cluster, you can ensure your messages are safe with RabbitMQ.

## Key Concepts

* **Producer:** Sends messages.
* **Consumer:** Receives messages.
* **Queue:** Holds messages until consumed.
* **Exchange:** Routes messages to queues based on rules.
* **Routing Key:** Label/message attribute used by exchanges for routing.
* **Binding**: Links exchanges to queues using rules.

## Part #1: Let’s Start with Queues Only (No Exchanges Yet)

Here’s a very simple message flow:

***Producer -> Queue -> Consumer***

RabbitMQ has a **default exchange** which we can use for now, and it routes directly to a queue with the same name.

**Step 1: Setup RabbitMQ**

docker run -d --hostname rabbit --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management

Access UI at: <http://localhost:15672>  
Default credentials: guest / guest

**Step 2: Create a Simple Queue**

Let’s create a queue called hello-queue.

**🧑‍💻 Producer (Node.js Example):**

Install the library:

``` npm install amqplib```

// producer.js

1. const amqp = require('amqplib');

2.

3. async function sendMsg() {

4. const conn = await amqp.connect('amqp://localhost');

5. const channel = await conn.createChannel();

6.

7. const queue = 'hello-queue';

8. await channel.assertQueue(queue, { durable: false });

9.

10. const msg = 'Hello from RabbitMQ!';

11. channel.sendToQueue(queue, Buffer.from(msg));

12.

13. console.log('Sent:', msg);

14.

15. setTimeout(() => {

16. conn.close();

17. }, 500);

18. }

19.

20. sendMsg();

21.

// consumer.js

1. const amqp = require('amqplib');

2.

3. async function receiveMsg() {

4. const conn = await amqp.connect('amqp://localhost');

5. const channel = await conn.createChannel();

6.

7. const queue = 'hello-queue';

8. await channel.assertQueue(queue, { durable: false });

9.

10. console.log('Waiting for messages in:', queue);

11. channel.consume(queue, (msg) => {

12. console.log('Received:', msg.content.toString());

13. }, { noAck: true });

14. }

15.

16. receiveMsg();

17.

## Part #2: Deep Dive into Queues

### 1. Durability

By default, your queue and messages are not persistent — meaning, if RabbitMQ restarts, everything is lost.

**To make queues and messages survive restarts:**

**Queue Durability**

1. channel.assertQueue('task-queue', { durable: true });

### 2. Message Persistence

1. channel.sendToQueue('task-queue', Buffer.from(msg), { persistent: true });

Note: This is **not guaranteed persistence**, just a hint to RabbitMQ. For full reliability, you’d need to use **publisher confirms** (we’ll cover this later).

### 3. Message Acknowledgments

**Why?**

RabbitMQ will remove a message from the queue **only after the consumer acknowledges** it.

**Default behavior:**

* noAck: true means RabbitMQ deletes the message immediately after sending.
* noAck: false means you must explicitly ack the message.

1. channel.consume('task-queue', (msg) => {

2. console.log('Received:', msg.content.toString());

3.

4. // simulate work

5. setTimeout(() => {

6. console.log("Done");

7. channel.ack(msg);

8. }, 1000);

9. }, { noAck: false });

10.

### 4. Fair Dispatch (Avoid Overloading Consumers)

By default, RabbitMQ sends messages in round-robin, **without waiting for ack**. That means:

* Fast consumers may be idle
* Slow ones may get overloaded

Use prefetch(1) to make RabbitMQ wait for the ack before sending the next one to that consumer.

channel.prefetch(1); // fair dispatch

This makes message distribution smarter and fairer.

### 5. Dead-Letter Queues (DLQs)

Dead-lettering lets you reroute **expired**, **rejected**, or **unacknowledged** messages to another queue for analysis or retry.

**Example:**

**Create main queue with DLX (dead-letter exchange):**

1. channel.assertQueue('main-queue', {

2. durable: true,

3. deadLetterExchange: '',

4. deadLetterRoutingKey: 'dead-letter-queue',

5. });

6.

**Create the dead-letter queue:**

1. channel.assertQueue('dead-letter-queue', { durable: true });

2.

Now, if you channel.nack(msg, false, false) a message (reject without requeue), it’ll go to dead-letter-queue.

**5. Reject vs Nack**

* channel.ack(msg): Acknowledge successful processing.
* channel.nack(msg, false, true): Requeue the message (maybe due to error).
* channel.nack(msg, false, false): Reject the message (send to DLQ).
* channel.reject(msg, false): Same as nack with requeue=false.

**🧠 Summary So Far:**

|  |  |
| --- | --- |
| **Feature** | **Code Snippet** |
| Durable Queue | assertQueue(name, { durable: true }) |
| Persistent Message | sendToQueue(name, ..., { persistent: true }) |
| Manual Ack | channel.ack(msg) |
| Fair Dispatch | channel.prefetch(1) |
| Dead Letter Queue | deadLetterExchange, deadLetterRoutingKey |
| Reject Message | channel.nack(msg, false, false) |

### Task Queue System Implementation

1. producer.js — Sends tasks
2. consumer.js — Processes tasks, handles acks and DLQ
3. setup.js — (optional) Sets up the queues, including DLQ

Step 1: Setup (DLQ + Main Queue)

1. // setup.js

2. const amqp = require('amqplib');

3.

4. async function setup() {

5. const conn = await amqp.connect('amqp://localhost');

6. const channel = await conn.createChannel();

7.

8. const dlq = 'dead-letter-queue';

9. const mainQueue = 'task-queue';

10.

11. // Create Dead Letter Queue

12. await channel.assertQueue(dlq, { durable: true });

13.

14. // Create Main Queue with DLQ binding

15. await channel.assertQueue(mainQueue, {

16. durable: true,

17. deadLetterExchange: '', // default exchange

18. deadLetterRoutingKey: dlq,

19. });

20.

21. console.log('Queues created successfully');

22.

23. await channel.close();

24. await conn.close();

25. }

26.

27. setup();

28.

Step 2: Producer — Sends Tasks

// producer.js

1. const amqp = require('amqplib');

2.

3. async function produce() {

4. const conn = await amqp.connect('amqp://localhost');

5. const channel = await conn.createChannel();

6.

7. const queue = 'task-queue';

8. await channel.assertQueue(queue, { durable: true });

9.

10. for (let i = 1; i <= 5; i++) {

11. const task = `Task #${i}`;

12. channel.sendToQueue(queue, Buffer.from(task), {

13. persistent: true, // ensure message survives broker restarts

14. });

15. console.log('Sent:', task);

16. }

17.

18. setTimeout(() => conn.close(), 500);

19. }

20.

21. produce();

22.

🔧 Step 3: Consumer — Processes Tasks

// consumer.js

1. const amqp = require('amqplib');

2.

3. async function consume() {

4. const conn = await amqp.connect('amqp://localhost');

5. const channel = await conn.createChannel();

6.

7. const queue = 'task-queue';

8. await channel.assertQueue(queue, { durable: true });

9.

10. // Fair dispatch

11. channel.prefetch(1);

12.

13. console.log('Waiting for tasks...');

14.

15. channel.consume(queue, async (msg) => {

16. const task = msg.content.toString();

17. console.log('Received:', task);

18.

19. await new Promise(resolve => setTimeout(resolve, 2000)); // simulate processing

20.

21. if (task === 'Task #3') {

22. console.log('Rejecting:', task);

23. channel.nack(msg, false, false); // send to DLQ

24. } else {

25. console.log('Done:', task);

26. channel.ack(msg); // acknowledge

27. }

28. }, { noAck: false });

29. }

30.

31. consume();

32.

Bonus: See the Dead Letter Queue Messages

1. // dlqConsumer.js

2. const amqp = require('amqplib');

3.

4. async function consumeDLQ() {

5. const conn = await amqp.connect('amqp://localhost');

6. const channel = await conn.createChannel();

7.

8. const dlq = 'dead-letter-queue';

9. await channel.assertQueue(dlq, { durable: true });

10.

11. console.log('Listening to DLQ...');

12.

13. channel.consume(dlq, (msg) => {

14. console.log('DLQ Received:', msg.content.toString());

15. channel.ack(msg);

16. }, { noAck: false });

17. }

18.

19. consumeDLQ();

20.

**🧪 Run it all**

* Run: node setup.js (only once)
* Run: node consumer.js
* In another terminal, run: node producer.js
* Optionally run: node dlqConsumer.js to monitor rejected messages

## Part #3 What is an Exchange?

In RabbitMQ, an **Exchange** receives messages from producers and **routes** them to one or more queues based on type and rules (bindings).

### Type of exchanges:

| **Type** | **Description** |
| --- | --- |
| direct | Routes messages to queues **with exact matching routing key** |
| fanout | **Broadcasts** messages to **all bound queues**, routing key is ignored |
| topic | Routes based on **pattern matching** in routing keys (wildcards like \*) |
| headers | Routes based on **message headers**, not routing keys |

### Setup Shared Constants (constants.js)

Create a shared file for reuse:

1. // constants.js

2. module.exports = {

3. EXCHANGES: {

4. DIRECT: 'direct\_logs',

5. FANOUT: 'broadcast\_logs',

6. TOPIC: 'topic\_logs',

7. HEADERS: 'header\_logs'

8. }

9. };

10.

### Direct Exchange — Exact Match Routing

* Routing by exact match.
* Messages go to queues with a binding key that matches the routing key exactly.

// directProducer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.DIRECT, 'direct', { durable: false });

9.

10. const levels = ['info', 'warn', 'error'];

11. for (const level of levels) {

12. const msg = `Log: ${level}`;

13. ch.publish(EXCHANGES.DIRECT, level, Buffer.from(msg));

14. console.log('Sent:', msg);

15. }

16.

17. setTimeout(() => conn.close(), 500);

18. })();

19.

// directConsumer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.DIRECT, 'direct', {durable: false});

9.

10. const q = await ch.assertQueue('', {exclusive: true});

11.

12. const bindingKey = 'error'; // only receive 'error' logs

13. await ch.bindQueue(q.queue, EXCHANGES.DIRECT, bindingKey);

14.

15. console.log('Waiting for error logs...');

16.

17. ch.consume(q.queue, (msg) => {

18. console.log('Received:', msg.content.toString());

19. }, { noAck: true });

20. })();

21.

### Fanout Exchange — Broadcast to All Queues

* Broadcasts messages to all queues bound to it.
* Ignores routing key.

// fanoutProducer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.FANOUT, 'fanout', { durable: false });

9.

10. ch.publish(EXCHANGES.FANOUT, '', Buffer.from('Broadcast to everyone!'));

11. console.log('Broadcast sent');

12. setTimeout(() => conn.close(), 500);

13. })();

14.

// fanoutConsumer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.FANOUT, 'fanout', { durable: false });

9.

10. const q = await ch.assertQueue('', { exclusive: true });

11. await ch.bindQueue(q.queue, EXCHANGES.FANOUT, '');

12.

13. console.log('Listening to broadcast...');

14.

15. ch.consume(q.queue, (msg) => {

16. console.log('Received broadcast:', msg.content.toString());

17. }, { noAck: true });

18. })();

19.

### Topic Exchange — Wildcard Routing

* Uses wildcard pattern matching.
* matches one word, # matches zero or more.

Symbol meaning

* One word
* # Zero or more words

// topicProducer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.TOPIC, 'topic', { durable: false });

9.

10. const topics = ['user.signup', 'user.login', 'admin.login'];

11. for (const key of topics) {

12. ch.publish(EXCHANGES.TOPIC, key, Buffer.from(`Event: ${key}`));

13. console.log('Sent:', key);

14. }

15.

16. setTimeout(() => conn.close(), 500);

17. })();

18.

// topicConsumer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.TOPIC, 'topic', { durable: false });

9.

10. const q = await ch.assertQueue('', { exclusive: true });

11.

12. const pattern = 'user.\*'; // match both 'user.signup' and 'user.login'

13. await ch.bindQueue(q.queue, EXCHANGES.TOPIC, pattern);

14.

15. console.log(`Waiting for messages matching "${pattern}"`);

16.

17. ch.consume(q.queue, (msg) => {

18. console.log('Received:', msg.fields.routingKey, msg.content.toString());

19. }, { noAck: true });

20. })();

21.

### Headers Exchange — Match on Headers Instead of Routing Key

* Routes based on message headers, not routing keys.
* More complex but flexible.

// headersProducer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.HEADERS, 'headers', { durable: false });

9.

10. const headers = { type: 'report', format: 'pdf' };

11. ch.publish(EXCHANGES.HEADERS, '', Buffer.from('PDF Report!'), {

12. headers,

13. });

14.

15. console.log('Header-based message sent');

16. setTimeout(() => conn.close(), 500);

17. })();

18.

// headersConsumer.js

1. const amqp = require('amqplib');

2. const { EXCHANGES } = require('./constants');

3.

4. (async () => {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. await ch.assertExchange(EXCHANGES.HEADERS, 'headers', { durable: false });

9.

10. const q = await ch.assertQueue('', { exclusive: true });

11.

12. await ch.bindQueue(q.queue, EXCHANGES.HEADERS, '', {

13. 'type': 'report',

14. 'format': 'pdf',

15. 'x-match': 'all' // can be 'any' or 'all'

16. });

17.

18. console.log('Listening to header-based messages...');

19.

20. ch.consume(q.queue, (msg) => {

21. console.log('Received (headers’), msg.content.toString());

22. }, { noAck: true });

23. })();

24.

## Part #4 What is RPC in RabbitMQ?

RPC allows one service (client) to send a request and wait for a response from another service (server), over RabbitMQ.

It uses:

* A request queue where clients send messages.
* A response queue where the server sends back results.
* A correlationId to match requests and response

**Graph**

[Client]

│

├─► (request + correlationId + replyTo queue)

│

▼

[RPC Queue] ──► [Server]

└─► process → respond to replyTo queue with same correlationId

### 1. Server: rpc\_server.js

The server listens for requests and responds back.

1. const amqp = require('amqplib');

2.

3. async function startServer() {

4. const conn = await amqp.connect('amqp://localhost');

5. const ch = await conn.createChannel();

6.

7. const queue = 'rpc\_queue';

8. await ch.assertQueue(queue, { durable: false });

9.

10. ch.consume(queue, async (msg) => {

11. const input = parseInt(msg.content.toString());

12. console.log(`Received: ${input}`);

13.

14. const result = fib(input); // simulate processing

15.

16. ch.sendToQueue(

17. msg.properties.replyTo,

18. Buffer.from(result.toString()),

19. { correlationId: msg.properties.correlationId }

20. );

21.

22. ch.ack(msg);

23. });

24.

25. console.log('RPC Server is ready...');

26.

27. function fib(n) {

28. return n <= 1 ? n : fib(n - 1) + fib(n - 2);

29. }

30. }

31.

32. startServer();

33.

### 2. Client: rpc\_client.js

The client sends a request and waits for the response.

1. const amqp = require('amqplib');

2. const { v4: uuidv4 } = require('uuid');

3.

4. async function requestFibonacci(n) {

5. const conn = await amqp.connect('amqp://localhost');

6. const ch = await conn.createChannel();

7.

8. const correlationId = uuidv4();

9.

10. const q = await ch.assertQueue('', { exclusive: true }); // reply queue

11.

12. ch.consume(q.queue, (msg) => {

13. if (msg.properties.correlationId === correlationId) {

14. console.log('Got response:', msg.content.toString());

15. setTimeout(() => conn.close(), 500);

16. }

17. }, { noAck: true });

18.

19. ch.sendToQueue('rpc\_queue', Buffer.from(n.toString()), {

20. correlationId,

21. replyTo: q.queue

22. });

23.

24. console.log(`Requested fib(${n})`);

25. }

26.

27. requestFibonacci(7);

28.

## Part #5 What Are RabbitMQ Streams?

RabbitMQ Streams are a durable, scalable, and log-based messaging model designed for:

* **High-volume** message ingestion (millions of messages/sec)
* **Persistent** storage
* **Replayability** — consumers can rewind to old messages
* **Message** **ordering** guarantee

Think of them like Kafka-style streams — but native to RabbitMQ.

|  |  |  |
| --- | --- | --- |
| **Feature** | **Queues** | **Streams** |
| Model | Push-based | Pull-based |
| Replay messages | ❌ No | ✅ Yes |
| Ordered messages | Not guaranteed | Guaranteed |
| High throughput | Moderate | Very high |
| Durable storage | Optional | Built-in |
| Partitioning | ❌ No | ✅ Yes (via routing key) |

**🚀 How Streams Work**

* Producers publish messages to a stream.
* Consumers pull messages from any offset (position), even from the beginning.
* Streams are append-only logs and support parallel partitioned consumption.

**Setup Requirements**

* RabbitMQ must be v3.9+
* Enable stream plugin:

**rabbitmq-plugins enable rabbitmq\_stream**

**Example: Stream Producer and Consumer (Node.js)**

npm install rabbitmq-stream-js-client

**Producer**

1. const { StreamClient } = require('rabbitmq-stream-js-client');

2.

3. async function produce() {

4. const client = await StreamClient.connect({ host: 'localhost' });

5.

6. const stream = 'logs-stream';

7. await client.createStream(stream, { maxLengthBytes: 10\_000\_000 });

8.

9. const producer = await client.producer(stream);

10.

11. for (let i = 0; i < 10; i++) {

12. await producer.send(Buffer.from(`Message ${i}`));

13. }

14.

15. console.log('✅ Messages sent to stream');

16. await producer.close();

17. await client.close();

18. }

19.

20. produce();

21.

**Consumer**

1. const { StreamClient, OffsetSpecification } = require('rabbitmq-stream-js-client');

2.

3. async function consume() {

4. const client = await StreamClient.connect({ host: 'localhost' });

5. const stream = 'logs-stream';

6.

7. await client.consumer({

8. stream,

9. offset: OffsetSpecification.first(), // Start from beginning

10. messageHandler: async (msg) => {

11. const content = msg.payload.toString();

12. console.log('📥 Received:', content);

13. }

14. });

15.

16. console.log('✅ Consumer started');

17. }

18.

19. consume();

20.

**Offsets in Streams**

* OffsetSpecification.first() → From beginning
* OffsetSpecification.last() → Only new messages
* OffsetSpecification.offset(n) → From a specific index
* OffsetSpecification.timestamp(Date) → From a specific time

Use Cases for Streams

* Event sourcing
* Audit logs
* Real-time analytics
* Time-travel debugging
* Replaying message history

## Part #6 Interview Questions

**10. What is a publisher confirm?**

It's a lightweight **acknowledgment** from RabbitMQ to a producer to confirm that a message has been **successfully received and saved**.

**11. What are quorum queues in RabbitMQ?**

**Quorum queues** use the **Raft consensus algorithm** for replication, providing **strong consistency** and **high availability** (recommended for modern RabbitMQ deployments).

**12. Explain the difference between a regular queue and a stream in RabbitMQ.**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Regular Queue** | **Stream** |
| Access | Push to consumer | Pull by consumer |
| Ordering | Not strict | Strict |
| Replay | No | Yes |
| Throughput | Moderate | Very High |

13. How can you handle backpressure in RabbitMQ?

* Set prefetch limit.
* Scale consumers horizontally.
* Use flow control.
* Use lazy queues.

1. Explain how RabbitMQ clustering works.

A **cluster** connects multiple RabbitMQ nodes to **share queues** (metadata replication).  
**Messages**, however, are **stored locally** unless mirrored/quorum

**2. What is a mirrored (classic mirrored) queue?**

A **classic mirrored queue** replicates its contents across multiple nodes to provide **failover**.  
(Being replaced by quorum queues.)

What is a quorum queue? Why is it recommended over mirrored queues now?

* Based on Raft protocol.
* No split brain issues.
* Better for large queue sizes.
* Automatic leader election.

4. How does RabbitMQ ensure message consistency across nodes?

By **replicating** queue metadata and messages (using **quorum queues** or **classic mirrored queues**).

**5# Explain shovel and federation plugins in RabbitMQ.**

* **Shovel**: Moves messages from one broker to another (even different data centers).
* **Federation**: Connects two RabbitMQ clusters allowing exchange/queue sharing.

What happens if the connection between the publisher and RabbitMQ breaks while publishing a message?

If **publisher confirms** are enabled, the producer will **know** the message is not delivered and **retry**.

**What are lazy queues in RabbitMQ?**

Lazy queues **keep messages on disk** as much as possible to save memory.

**What is the difference between lazy queues and quorum queues?**

* **Lazy**: Optimized for memory efficiency.
* **Quorum**: Optimized for **consistency and durability** using Raft.

**How do you scale RabbitMQ horizontally?**

* Create **clusters**.
* Use **multiple queues**.
* Use **sharding**.
* Use **stream partitions** (for streams).

10. Explain the use of the rabbitmq\_stream plugin.

Enables **stream-native support** (high-throughput, log-based messaging) in RabbitMQ.

How would you tune RabbitMQ for high throughput?

* Increase prefetch count.
* Use lazy queues.
* Batch publishes.
* Use Streams.
* Tune TCP settings (e.g., buffer sizes).

**12. How can you secure RabbitMQ (authentication, authorization, SSL)?**

* **Enable TLS/SSL** for network encryption.
* Use **user authentication** (username/password, LDAP).
* Define **permissions** per user/vhost.

**13. How do you monitor RabbitMQ performance?**

* Use the **RabbitMQ Management Plugin**.
* Use **Prometheus/Grafana** dashboards.
* Monitor **queue lengths, memory usage, message rates**.

Topics to remember

* Exchanges (direct, fanout, topic, headers)
* Queues (durable, exclusive, auto-delete)
* Message acknowledgments (manual, automatic)
* Dead Letter Exchanges (DLX)
* RPC over RabbitMQ
* Streams vs Queue
* High availability setups (cluster, quorum queues)
* Monitoring with Prometheus, Grafana, RabbitMQ Management UI