RabbitMQ, Redis, and React Query Questions and Practical Exercises

This document provides a comprehensive set of questions and practical exercises for RabbitMQ and Redis, with Python-based examples alongside native commands, and a dedicated section of interview questions for React Query. The content is categorized by difficulty (Basic, Intermediate, Advanced) and topic, with detailed answers and solutions. RabbitMQ and Redis sections cover their core concepts, data structures, and integration with Python in a microservices context. React Query questions focus on interview preparation, covering its usage in React applications. The questions and exercises are designed for interviews, certifications, or practical application in distributed systems and frontend development.

RabbitMQ

Basic RabbitMQ Concepts

Theory Questions

1. **What is RabbitMQ?**
   - **Answer**: RabbitMQ is an open-source message broker that facilitates asynchronous communication between applications using the Advanced Message Queuing Protocol (AMQP). It enables producers to send messages to queues, which consumers process, decoupling services in a microservices architecture.
2. **What is a message queue, and how does RabbitMQ use it?**
   - **Answer**: A message queue is a data structure that stores messages in a FIFO (First-In, First-Out) order, enabling asynchronous communication. RabbitMQ uses queues to hold messages sent by producers until consumers process them, ensuring reliable delivery and load balancing.
3. **What are the key components of RabbitMQ?**
   - **Answer**:
     - **Producer**: Sends messages to an exchange.
     - **Exchange**: Routes messages to queues based on routing keys and bindings.
     - **Queue**: Stores messages until consumed.
     - **Consumer**: Retrieves and processes messages from queues.
     - **Binding**: Defines the relationship between an exchange and a queue.
4. **What are the types of exchanges in RabbitMQ?**
   - **Answer**:
     - **Direct**: Routes messages to queues based on an exact routing key match.
     - **Topic**: Routes messages using pattern-based routing keys (e.g., `*.log`).
     - **Fanout**: Broadcasts messages to all bound queues.
     - **Headers**: Routes based on message header attributes.
5. **What is the Single Responsibility Principle (SRP) applied to RabbitMQ?**
   - **Answer**: In RabbitMQ, SRP implies that each queue or consumer should handle a specific type of task or message. For example, a user service queue processes only user-related events, ensuring focused and maintainable message processing.
6. **What is a dead letter queue (DLQ)?**
   - **Answer**: A DLQ is a special queue for messages that cannot be processed or delivered (e.g., due to consumer failures or invalid routing). RabbitMQ supports DLQs by configuring a dead letter exchange (DLX) and binding it to a queue.
7. **What is the difference between RabbitMQ and Kafka?**
   - **Answer**: RabbitMQ is a traditional message broker focused on reliable message delivery with AMQP, suitable for task queues and small-scale systems. Kafka is a distributed streaming platform optimized for high-throughput, real-time data pipelines, with persistent event logs.
8. **What is the purpose of a routing key in RabbitMQ?**
   - **Answer**: A routing key is a string attribute in a message that determines which queue(s) it is routed to by an exchange. It's used in direct and topic exchanges to match messages with queue bindings.
9. **How does RabbitMQ ensure message durability?**

- **Answer**: RabbitMQ ensures durability by marking queues and messages as durable and persistent. Durable queues survive broker restarts, and persistent messages are stored to disk, ensuring no loss during failures.

10. **What is the time complexity of message enqueueing and dequeueing in RabbitMQ?**

    - **Answer**: Enqueueing and dequeueing messages in RabbitMQ are typically O(1) for in-memory operations, as queues operate like linked lists. Persistent messages may incur O(n) disk I/O overhead, where n is the message size.

Practical Exercises

1. **Publish a message to a RabbitMQ queue using Python**

    - **Task**: Write a Python script to publish a message to a RabbitMQ queue using the `pika` library.
    - **Solution**:

```
# publisher.py
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('rabbitmq'))
channel = connection.channel()
channel.queue_declare(queue='test_queue')

message = "Hello, RabbitMQ!"
channel.basic_publish(exchange='', routing_key='test_queue', body=message)
print(f" [x] Sent '{message}'")
connection.close()
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY publisher.py .
CMD ["python", "publisher.py"]
```

```
# requirements.txt
pika==1.3.2
```

```
# docker-compose.yml
version: '3'
services:
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"
    networks:
      - app-network
  publisher:
    build: .
    depends_on:
      - rabbitmq
    networks:
      - app-network
networks:
  app-network:
    driver: bridge
```

```
docker-compose up --build
```

    - **Explanation**: The script connects to RabbitMQ, declares a queue, and publishes a message. The Docker Compose file runs RabbitMQ and the publisher. Check the RabbitMQ management UI at `http://localhost:15672` (default credentials: guest/guest) to verify the message in

`test_queue`.

2. **Consume messages from a RabbitMQ queue**

   - **Task**: Write a Python script to consume messages from the `test_queue`.
   - **Solution**:

```
# consumer.py
import pika

def callback(ch, method, properties, body):
    print(f" [x] Received '{body.decode()}'")

connection = pika.BlockingConnection(pika.ConnectionParameters('rabbitmq'))
channel = connection.channel()
channel.queue_declare(queue='test_queue')
channel.basic_consume(queue='test_queue', on_message_callback=callback, auto_ack=True)
print(" [*] Waiting for messages. To exit press CTRL+C")
channel.start_consuming()
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY consumer.py .
CMD ["python", "consumer.py"]
```

```
# Update docker-compose.yml to include consumer
version: '3'
services:
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"
    networks:
      - app-network
  consumer:
    build: .
    depends_on:
      - rabbitmq
    networks:
      - app-network
networks:
  app-network:
    driver: bridge
```

```
docker-compose up --build
```

     - **Explanation**: The consumer listens for messages on `test_queue` and prints them. Run the publisher in a separate terminal to send messages. Output: `[x] Received 'Hello, RabbitMQ!'`.

3. **Implement a topic exchange**

   - **Task**: Create a Python script to publish and consume messages using a topic exchange with routing keys.
   - **Solution**:

```
# topic_publisher.py
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters('rabbitmq'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', exchange_type='topic')
channel.basic_publish(exchange='logs', routing_key='info.user', body='User Event')
print(" [x] Sent 'User Event' to info.user")
connection.close()
```

```
# topic_consumer.py
import pika

def callback(ch, method, properties, body):
    print(f" [x] {method.routing_key}: {body.decode()}")

connection = pika.BlockingConnection(pika.ConnectionParameters('rabbitmq'))
channel = connection.channel()
channel.exchange_declare(exchange='logs', exchange_type='topic')
result = channel.queue_declare(queue='', exclusive=True)
queue_name = result.method.queue
channel.queue_bind(exchange='logs', queue=queue_name, routing_key='info.*')
channel.basic_consume(queue=queue_name, on_message_callback=callback, auto_ack=True)
print(" [*] Waiting for logs. To exit press CTRL+C")
channel.start_consuming()
```

```
# Use same docker-compose.yml, update services for publisher and consumer
```

- **Explanation**: The publisher sends a message to a topic exchange with routing key `info.user`. The consumer binds to `info.*`, receiving matching messages. Output: `[x] info.user: User Event`.

Redis

Basic Redis Concepts

Theory Questions

1. **What is Redis?**
   - **Answer**: Redis (Remote Dictionary Server) is an open-source, in-memory data structure store used as a database, cache, or message broker. It supports data structures like strings, lists, sets, hashes, and sorted sets, offering high performance for read/write operations.
2. **What are the primary data structures in Redis?**
   - **Answer**:
     - **String**: Key-value pairs for simple data (e.g., `SET key value`).
     - **List**: Ordered, linked-list-like structure (e.g., `LPUSH`, `RPOP`).
     - **Set**: Unordered collection of unique elements (e.g., `SADD`, `SMEMBERS`).
     - **Hash**: Key-value maps for structured data (e.g., `HSET`, `HGET`).
     - **Sorted Set**: Set with scores for ranking (e.g., `ZADD`, `ZRANGE`).
     - **Bitmap**, **HyperLogLog**, **Streams**: Advanced structures for specific use cases.
3. **How does Redis differ from a traditional relational database?**
   - **Answer**: Redis is an in-memory, NoSQL key-value store optimized for speed, with flexible data structures. Relational databases (e.g., MySQL) use tables and SQL, prioritize data consistency, and are disk-based, making them slower for certain workloads but better for complex queries.
4. **What is the Single Responsibility Principle (SRP) applied to Redis?**
   - **Answer**: In Redis, SRP implies using specific data structures for distinct purposes. For example, use a list for a queue, a hash for user profiles, and a sorted set for leaderboards, ensuring each key serves a single responsibility.
5. **What is Redis persistence?**

- **Answer**: Redis supports persistence to save in-memory data to disk:
  - **RDB (Snapshotting)**: Periodically saves a snapshot of the dataset.
  - **AOF (Append-Only File)**: Logs every write operation, allowing full data reconstruction.
  - **Hybrid**: Combines RDB and AOF for durability and performance.

6. **What is the purpose of Redis Pub/Sub?**

   - **Answer**: Redis Pub/Sub enables asynchronous messaging by allowing publishers to send messages to channels and subscribers to receive them. It's lightweight but lacks message persistence, unlike RabbitMQ.

7. **What is the time complexity of Redis operations?**

   - **Answer**:
     - **String**: `SET`, `GET`: O(1).
     - **List**: `LPUSH`, `RPOP`: O(1); `LINDEX`: O(n).
     - **Set**: `SADD`, `SMEMBERS`: O(1) for add, O(n) for retrieval.
     - **Hash**: `HSET`, `HGET`: O(1).
     - **Sorted Set**: `ZADD`, `ZRANGE`: O(log n) for add/range.

8. **What is Redis Cluster?**

   - **Answer**: Redis Cluster is a distributed implementation of Redis that shards data across multiple nodes, providing scalability and high availability. It uses a hash slot mechanism (16,384 slots) to distribute keys.

9. **How does Redis handle caching?**

   - **Answer**: Redis stores frequently accessed data in memory, reducing database load. Common patterns include:
     - **Cache-Aside**: Application checks Redis; if data is missing, it queries the database and caches the result.
     - **Write-Through**: Updates are written to Redis and the database simultaneously.
     - **TTL**: Set expiration times (e.g., `EXPIRE key seconds`) to manage cache lifecycle.

10. **What are the use cases for Redis?**

    - **Answer**:
      - Caching to reduce database load.
      - Session storage for web applications.
      - Real-time analytics (e.g., leaderboards with sorted sets).
      - Task queues with lists.
      - Pub/Sub for messaging.

Practical Exercises

1. **Store and retrieve data in Redis using Python**

   - **Task**: Write a Python script to store and retrieve a user profile using Redis hashes.
   - **Solution**:

```
# user_redis.py
import redis

client = redis.Redis(host='redis', port=6379, decode_responses=True)

# Store user profile
client.hset('user:1', mapping={'id': '1', 'name': 'Alice', 'email': 'alice@example.com'})
# Retrieve user profile
user = client.hgetall('user:1')
print(user)
```

```
# Dockerfile
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY user_redis.py .
CMD ["python", "user_redis.py"]
```

```
# requirements.txt
redis==5.0.8
```

```
# docker-compose.yml
version: '3'
services:
  redis:
    image: redis:latest
    ports:
      - "6379:6379"
    networks:
      - app-network
  app:
    build: .
    depends_on:
      - redis
    networks:
      - app-network
networks:
  app-network:
    driver: bridge
```

```
docker-compose up --build
```

- **Explanation**: The script uses the `redis-py` library to store a user profile as a hash and retrieve it. Output: `{'id': '1', 'name': 'Alice', 'email': 'alice@example.com'}`.

2. **Implement a Redis list as a queue**

- **Task**: Use Redis lists to implement a task queue with a Python producer and consumer.
- **Solution**:

```
# producer.py
import redis

client = redis.Redis(host='redis', port=6379, decode_responses=True)
tasks = ['task1', 'task2', 'task3']
for task in tasks:
    client.lpush('tasks', task)
    print(f"Pushed {task}")
```

```
# consumer.py
import redis

client = redis.Redis(host='redis', port=6379, decode_responses=True)
while True:
    task = client.rpop('tasks')
    if task:
        print(f"Processed {task}")
    else:
        break
```

```
# Update docker-compose.yml for producer and consumer
version: '3'
services:
  redis:
    image: redis:latest
    ports:
      - "6379:6379"
    networks:
      - app-network
  producer:
    build:
      context: .
      dockerfile: Dockerfile.producer
    depends_on:
      - redis
    networks:
      - app-network
  consumer:
    build:
      context: .
      dockerfile: Dockerfile.consumer
    depends_on:
      - redis
    networks:
      - app-network
networks:
  app-network:
    driver: bridge
```

```
# Dockerfile.producer
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY producer.py .
CMD ["python", "producer.py"]
```

```
# Dockerfile.consumer
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY consumer.py .
CMD ["python", "consumer.py"]
```

```
docker-compose up --build
```

- **Explanation**: The producer pushes tasks to a Redis list (`LPUSH`), and the consumer pops them (`RPOP`). Output: Producer: `Pushed task1`, etc.; Consumer: `Processed task3`, etc.

3. **Implement Redis Pub/Sub**

- **Task**: Create a Python script for publishing and subscribing to a Redis channel.
- **Solution**:

```
# publisher.py
import redis

client = redis.Redis(host='redis', port=6379, decode_responses=True)
client.publish('chat', 'Hello, Redis!')
print("Published message")
```

```
# subscriber.py
import redis

client = redis.Redis(host='redis', port=6379, decode_responses=True)
pubsub = client.pubsub()
pubsub.subscribe('chat')
for message in pubsub.listen():
    if message['type'] == 'message':
        print(f"Received: {message['data']}")
        break
```

```
# Use similar docker-compose.yml, update for publisher and subscriber
```

- **Explanation**: The publisher sends a message to the `chat` channel, and the subscriber listens for it. Output: `Received: Hello, Redis!`.

React Query Interview Questions

Basic

1. **What is React Query, and why is it used?**
   - **Answer**: React Query is a data-fetching library for React that simplifies managing server-state (e.g., data from APIs). It provides hooks like `useQuery` and `useMutation` for fetching, caching, and updating data, reducing boilerplate code. It's used for its built-in caching, automatic refetching, and error handling, improving performance and developer experience.
2. **What is the difference between client-state and server-state in React Query?**
   - **Answer**: Client-state is UI-related data managed locally (e.g., form inputs, toggles), typically handled by libraries like Redux or Zustand. Server-state is data fetched from external APIs, managed by React Query, which handles caching, synchronization, and refetching to keep data consistent with the server.
3. **What is the `useQuery` hook, and how does it work?**
   - **Answer**: The `useQuery` hook fetches and caches data. It takes a unique query key (e.g., `['users', userId]`) and a fetch function (e.g., `fetchUsers`). It returns an object with properties like `data`, `isLoading`, `isError`, and `error`. Example:

     ```
     const { data, isLoading } = useQuery(['users'], () => fetch('/api/users').then(res => res.json()));
     ```

     React Query caches the result, refetches on window focus, and manages loading states.
4. **What is query caching in React Query?**
   - **Answer**: Query caching stores fetched data in memory, associated with a query key. React Query automatically caches responses, allowing reuse across components without refetching. Cache duration is controlled by `cacheTime` (default: 5 minutes), and stale data is managed with `staleTime`.
5. **What is the purpose of the `QueryClient` in React Query?**
   - **Answer**: `QueryClient` is the core instance that manages the cache and configuration for queries and mutations. It's provided to the app via `QueryClientProvider`:

     ```
     import { QueryClient, QueryClientProvider } from '@tanstack/react-query';
     const queryClient = new QueryClient();
     function App() {
       return <QueryClientProvider client={queryClient}><App /></QueryClientProvider>;
     }
     ```

Intermediate

6. **What is the `useMutation` hook, and how does it differ from `useQuery`?**

   - **Answer**: `useMutation` handles side-effect operations (e.g., POST, PUT, DELETE requests) that modify server data. Unlike `useQuery`, which is for fetching data, `useMutation` doesn't cache results but provides `mutate` and `mutateAsync` functions. Example:

     ```
     const mutation = useMutation({
       mutationFn: (data) => fetch('/api/users', { method: 'POST', body: JSON.stringify(data) }).then(res => res.json()),
       onSuccess: () => queryClient.invalidateQueries(['users'])
     });
     ```

     It integrates with `QueryClient` to invalidate queries and refresh data.

7. **How does React Query handle stale data and refetching?**

   - **Answer**: React Query uses `staleTime` to determine when data is considered stale (default: 0, immediate refetch). Stale data is served from the cache while a background refetch occurs. Refetching is triggered by events like window focus, network reconnect, or `refetch` calls. Example:

     ```
     useQuery(['users'], fetchUsers, { staleTime: 60000 }); // Stale after 1 minute
     ```

8. **What is the role of query keys in React Query?**

   - **Answer**: Query keys uniquely identify cached data. They can be strings or arrays (e.g., `['users', userId]`) to support dynamic queries. React Query uses keys to cache, invalidate, or refetch data. Consistent keys ensure data reuse across components.

9. **How do you handle errors in React Query?**

   - **Answer**: React Query provides `isError` and `error` properties in `useQuery` and `useMutation`. You can handle errors globally via `QueryClient`'s `onError` or per-query with `onError` callbacks:

     ```
     const { data, isError, error } = useQuery(['users'], fetchUsers, {
       onError: (err) => console.error('Fetch failed:', err)
     });
     if (isError) return <div>Error: {error.message}</div>;
     ```

10. **What is the difference between `cacheTime` and `staleTime` in React Query?**

    - **Answer**: `cacheTime` (default: 5 minutes) determines how long inactive query data remains in memory before garbage collection. `staleTime` (default: 0) determines how long data is considered fresh before refetching. Setting `staleTime` higher reduces refetches, while `cacheTime` affects memory usage.

Advanced

11. **How do you implement optimistic updates with React Query?**

    - **Answer**: Optimistic updates assume a mutation succeeds and update the UI immediately, rolling back if it fails. Use `useMutation` with `onMutate`, `onError`, and `onSettled`:

      ```
      const mutation = useMutation({
        mutationFn: (data) => fetch('/api/users', { method: 'POST', body: JSON.stringify(data) }),
        onMutate: async (newUser) => {
          await queryClient.cancelQueries(['users']);
          const previousUsers = queryClient.getQueryData(['users']);
          queryClient.setQueryData(['users'], (old) => [...old, newUser]);
          return { previousUsers };
        },
        onError: (err, newUser, context) => {
          queryClient.setQueryData(['users'], context.previousUsers);
        },
        onSettled: () => {
          queryClient.invalidateQueries(['users']);
        }
      });
      ```

This updates the cache optimistically and reverts on failure.

12. **What is the `useInfiniteQuery` hook, and when is it used?**

    - **Answer**: `useInfiniteQuery` fetches paginated data incrementally (e.g., infinite scrolling). It takes a `getNextPageParam` function to determine the next page's parameters:

      ```
      const { data, fetchNextPage, hasNextPage } = useInfiniteQuery(
        ['users'],
        ({ pageParam = 1 }) => fetch(`/api/users?page=${pageParam}`).then(res => res.json()),
        { getNextPageParam: (lastPage) => lastPage.nextPage || undefined }
      );
      ```

      It's used for lists with dynamic loading, like social media feeds.

13. **How do you prefetch data in React Query?**

    - **Answer**: Prefetching fetches data before it's needed, improving UX. Use `queryClient.prefetchQuery`:

      ```
      const queryClient = useQueryClient();
      const prefetchUsers = async () => {
        await queryClient.prefetchQuery(['users'], () => fetch('/api/users').then(res => res.json()));
      };
      ```

      Call `prefetchUsers` (e.g., on hover) to cache data before rendering.

14. **How does React Query integrate with TypeScript?**

    - **Answer**: React Query supports TypeScript with typed hooks and generics. Define types for query data and errors:

      ```
      interface User {
        id: number;
        name: string;
      }
      const { data } = useQuery<User[]>({
        queryKey: ['users'],
        queryFn: () => fetch('/api/users').then(res => res.json())
      });
      ```

      TypeScript ensures type safety for data, errors, and query keys.

15. **What are the best practices for using React Query in large applications?**

    - **Answer**:
      - Use unique, structured query keys (e.g., arrays for dynamic queries).
      - Set appropriate `staleTime` and `cacheTime` to balance freshness and performance.
      - Implement optimistic updates for responsive UX.
      - Use `useMutation` for side effects and invalidate queries on success.
      - Leverage `useInfiniteQuery` for paginated data.
      - Monitor performance with DevTools (`@tanstack/react-query-devtools`).
      - Handle errors globally with `QueryClient`'s `defaultOptions`.

Guidelines

- **Practice Platforms**: Use Docker Desktop or cloud-based Kubernetes for RabbitMQ and Redis deployments. Practice React Query with Create React App or Next.js on platforms like CodeSandbox.
- **Resources**: Refer to RabbitMQ and Redis documentation, Python's `pika` and `redis-py` libraries, and React Query's official docs (@tanstack/react-query). YouTube tutorials (e.g., TechWorld with Nana for RabbitMQ/Redis, Jack Herrington for React Query) are helpful.
- **Best Practices**: For RabbitMQ, use durable queues and persistent messages for reliability. For Redis, choose appropriate data structures and enable persistence for critical data. For React Query, optimize cache settings and use TypeScript for type safety.
- **Certifications**: Prepare for AWS Certified Solutions Architect or CNCF's KCNA for RabbitMQ/Redis in microservices. For React Query, focus on frontend interviews with React expertise.
- **Optimization**: Optimize RabbitMQ with proper exchange types and message batching. Use Redis for high-performance caching and minimal data structures. In React Query, minimize refetches with `staleTime` and prefetching.

- **Security**: Secure RabbitMQ with user authentication and TLS. Use Redis ACLs and encryption. For React Query, sanitize API responses to prevent XSS.