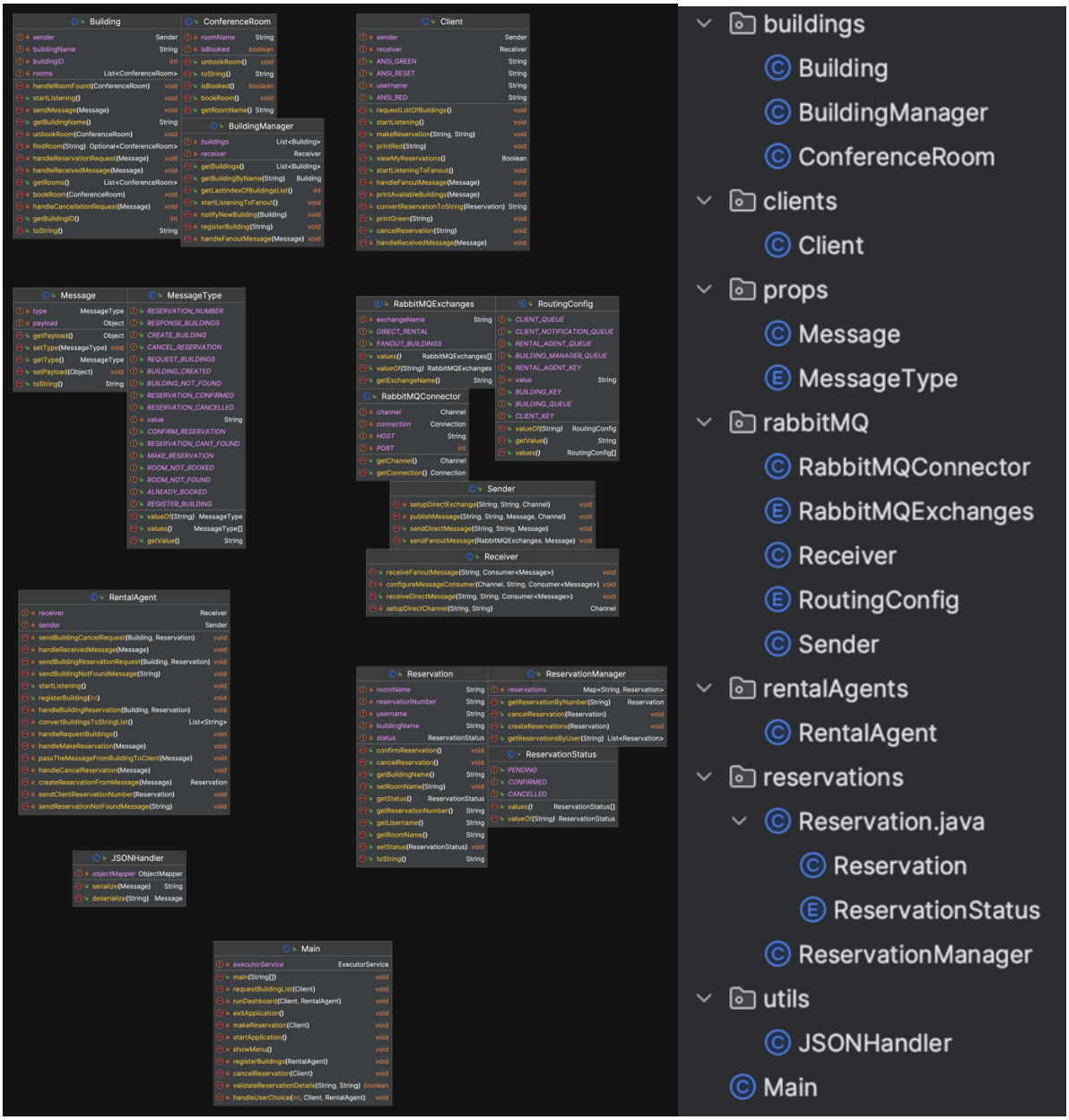
Table of Contents

Introduction

This document explains how a reservation system with microservices architecture works. Although it is not as complicated and large as real-life microservices, this project aims to ensure that the messaging between these microservices is scalable and works correctly under load. This messaging operation is done through RabbitMQ's integration. In order to run the project successfully, firstly, the dockerized RabbitMQ setup in the directory, must be run.

Packages and Classes

In this section, since there are many packages and classes on the project, they are briefly defined.



Package clients:

- **Client**:
    - Represents a user in the system.
    - Handles sending reservation requests, cancellations, and receiving messages about the status of their reservations and notifications about new buildings.
    - Interacts with the **RentalAgent** through RabbitMQ to manage reservations.

Package rentalAgents:

- **RentalAgent**:
    - Acts as an bridge between the client and the buildings.
    - Manages reservation requests, building queries, and cancellation requests from clients.
    - Communicates with **BuildingManager** to retrieve building information and with individual **Building** instances to manage reservations.

Package buildings:

- **Building**:
    - Represents a physical building that contains a number of conference rooms.
    - Handles requests to reserve or cancel a booking for a room.
    - Listens for messages on its own queue and responds to the **RentalAgent** about room availability and reservation status.
- **BuildingManager**:
    - Oversees all building instances within the system.
    - Can create new **Building** objects when notified of a new registration.
    - Broadcasts messages about new buildings being created.
- **ConferenceRoom**:
    - Represents an individual conference room within a building.
    - Manages the booking status (booked or available) of the room.

Package rabbitMQ:

- **Receiver**:
    - Manages the reception of messages from RabbitMQ exchanges.
- **Sender**:
    - Responsible for sending messages to RabbitMQ exchanges.

- **RabbitMQConnector**:
    - Handles the connection to the RabbitMQ server.
    - Provides a channel for sending and receiving messages.
- **RabbitMQExchanges**:
    - Enumerates the exchanges used in the system, such as the fanout exchange for broadcasting building creations.
- **RoutingConfig**:
    - Defines the routing keys and queue names used in RabbitMQ to route messages correctly within the system.

Package reservations:

- **Reservation**:
    - Represents a booking/reservation made by a client.
    - Contains details like reservation number, building name, room name, and status.
- **ReservationManager**:
    - Manages all reservations within the system.
    - Can create, cancel, and retrieve reservations based on user requests.

Package props:

- **Message**:
    - Represents a message that can be sent or received in the system.
    - Contains the type of message and any associated payload.
- **MessageType**:
    - Enumerates the different types of messages that can be exchanged, such as reservation requests, cancellations, or confirmations.

Package utils:

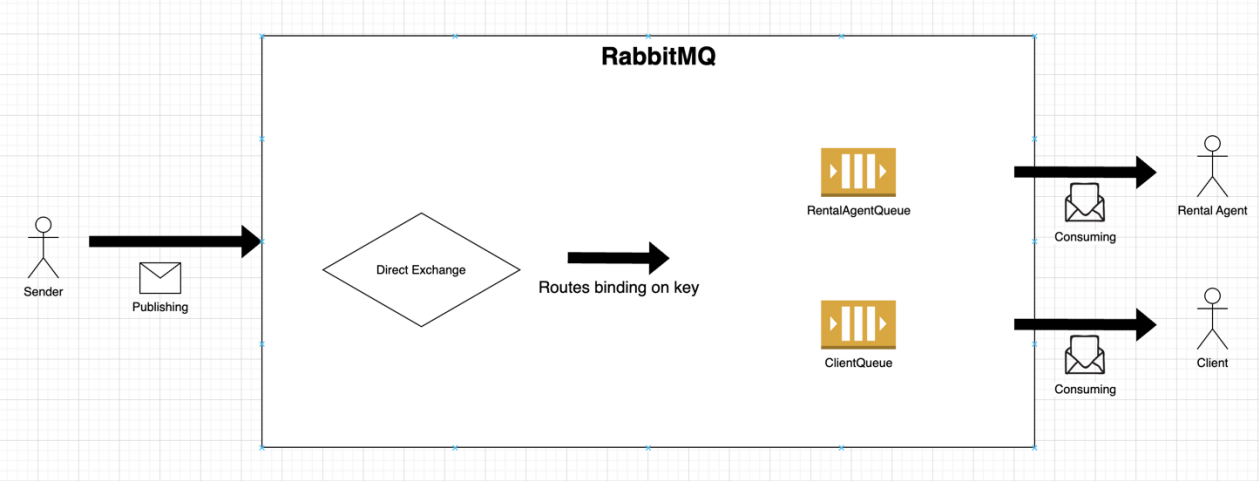
- **JSONHandler**:
    - Manages serialization and deserialization of messages to/from JSON format.

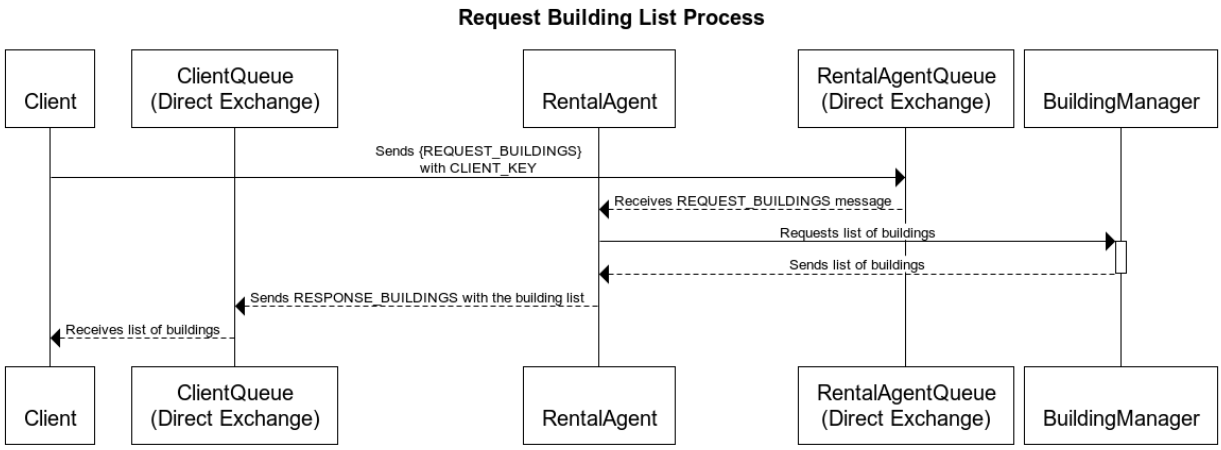
Main Application Entry (Main class):

- **Main**:
    - The entry point of the application.
    - Sets up the system, starts listening on queues, and initiates the user interface for client interactions.

The Architecture and the Sequence of the Program

Request of Buildings in the System:



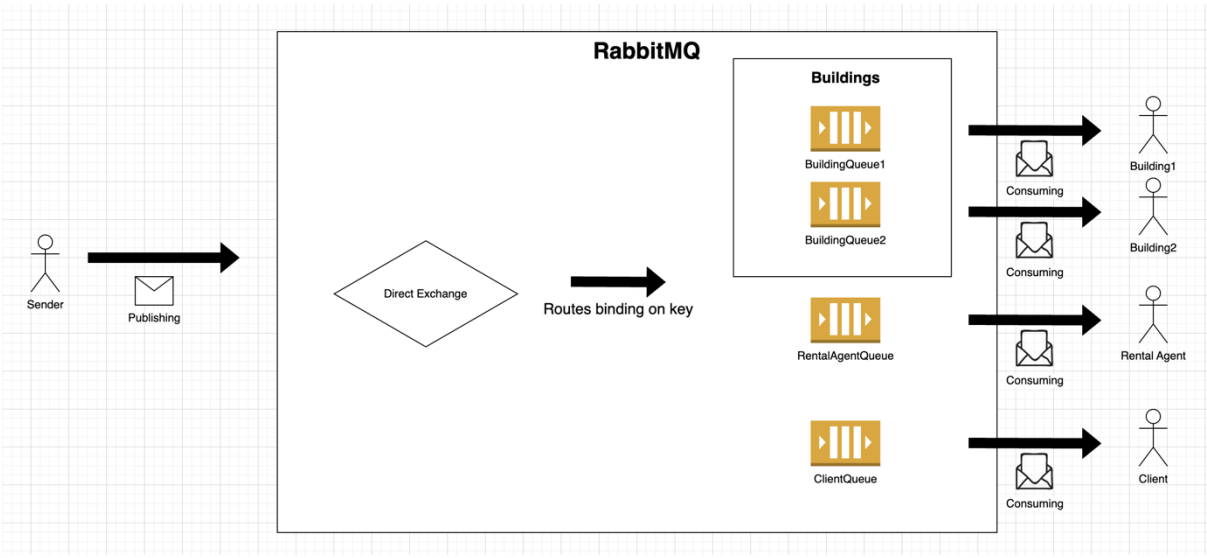


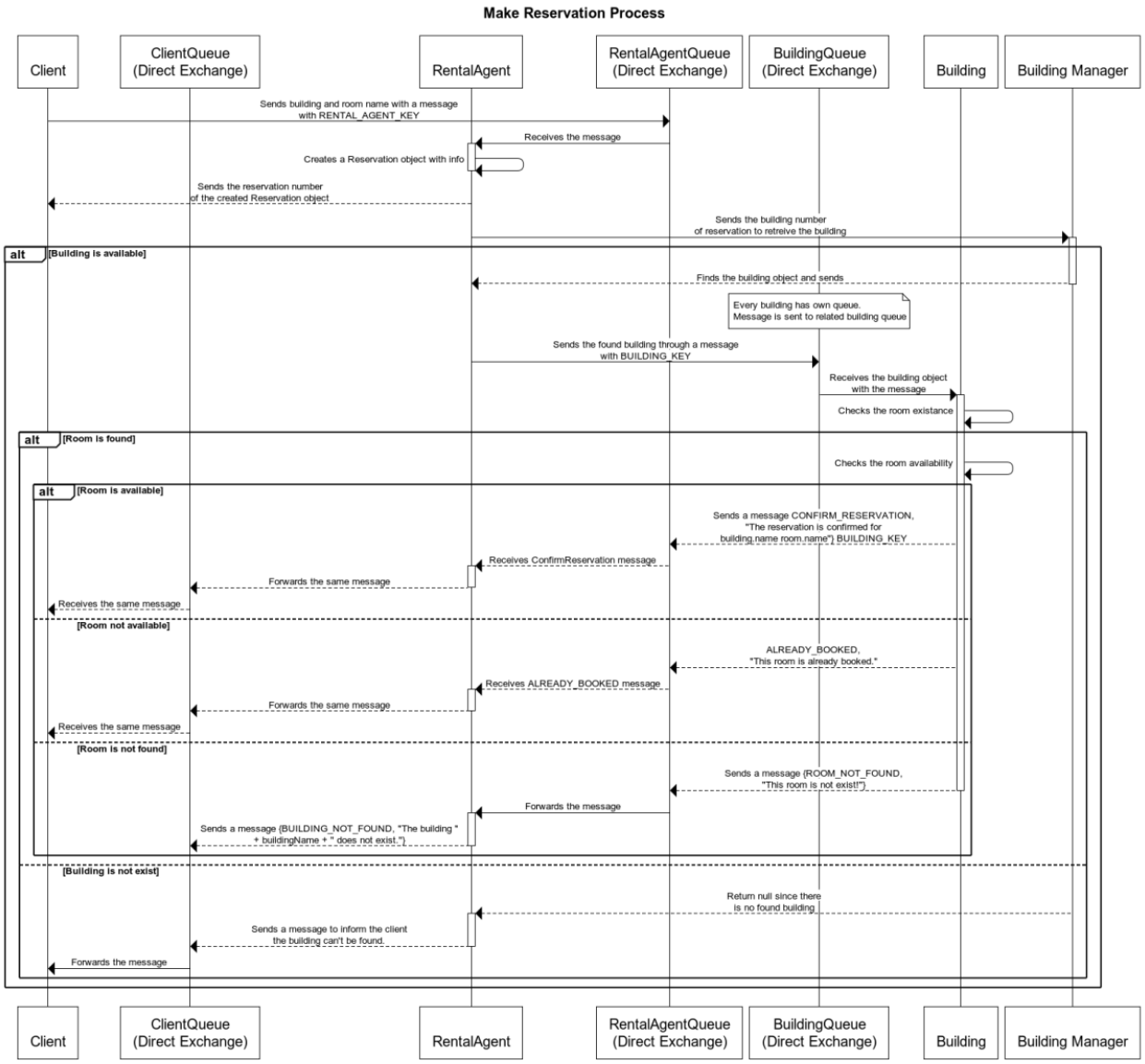
1. **Client Makes a Request**:
    - The **Client** starts the process by sending a **REQUEST_BUILDINGS** message to the **RentalAgentQueue**.
2. **Rental Agent Receives the Request**:
    - The **RentalAgentQueue** delivers the **REQUEST_BUILDINGS** message to the **RentalAgent**. The rental agent is the process responsible for handling this type of client request.
3. **Rental Agent Queries Building Manager**:
    - Upon receiving the request, the **RentalAgent** asks the **BuildingManager** for the current list of buildings.
4. **Building Manager Responds with List**:

- The **BuildingManager** finds and sends the list of buildings back to the **RentalAgent**.

5. **Rental Agent Responds to Client**:
    - After receiving the list from the **BuildingManager**, the **RentalAgent** forwards this list to the **Client**. It is sent a **RESPONSE_BUILDINGS** message with the building list through the **ClientQueue**.
6. **Client Receives Building List**:
    - Finally, the **Client** receives the **RESPONSE_BUILDINGS** message from the **ClientQueue**.

Make Reservation:



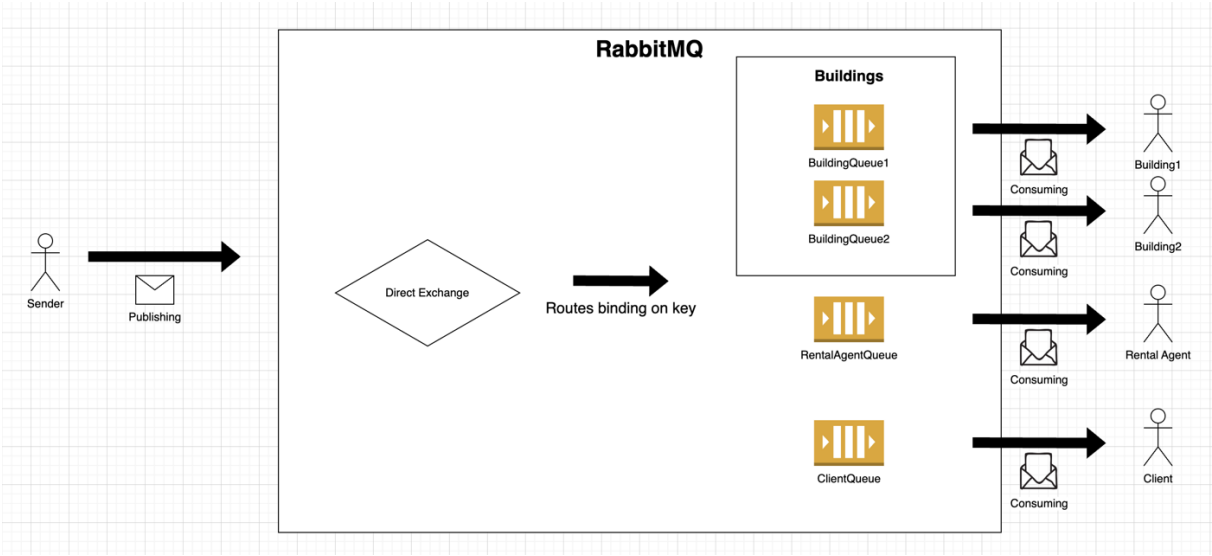


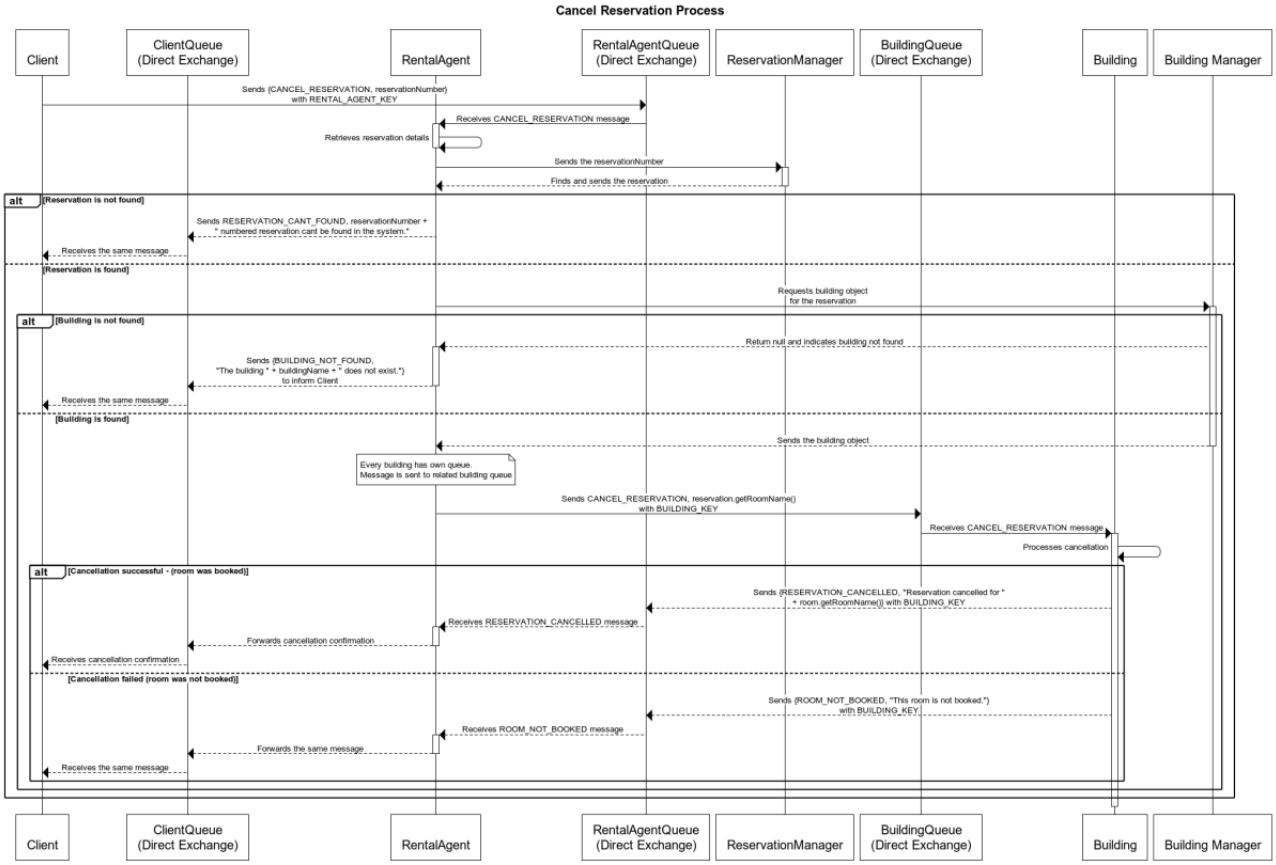
1. **Client Makes Request**:
    - The **Client** starts the reservation process by sending a message with the building and room name to the **RentalAgentQueue**.
    - This message is routed using the **RENTAL_AGENT_KEY**.
2. **Rental Agent Receives Message**:
    - The **RentalAgent** picks up the message from the **RentalAgentQueue**.
    - It creates a **Reservation** object using the building and room information contained in the message.

3. **Reservation Number Sent to Client**:
    - The **RentalAgent** sends back a reservation number to the **Client** after **Reservation** object is created. This message passes through the **ClientQueue**, indicating that a reservation has been initiated.
4. **Rental Agent Requests Building Information**:
    - The **RentalAgent** requests the **Building** object from the **Building Manager** by sending the building number from the reservation.
5. **Building Manager Finds Building**:
    - The **Building Manager** locates the appropriate **Building** object and sends it back to the **RentalAgent**.
    - If the building does not exist, the **Building Manager** returns **null**.
6. **Building Queue Message Routing**:
    - If the building exists, the **RentalAgent** sends a message to the specific **BuildingQueue** using the **BUILDING_KEY**.
    - The message contains instructions to check the room's existence and availability.
7. **Building Checks Room**:
    - The **Building** receives the message and first checks if the room exists.
    - If the room exists, it then checks if the room is available.
8. **Room Availability Scenarios**:
    - **Room Available**: The building sends a **CONFIRM_RESERVATION** message back to the **RentalAgentQueue**, which is then forwarded to the **Client** through **ClientQueue**.

- **Room Not Available**: The building sends an **ALREADY_BOOKED** message indicating the room is already booked. This message follows the same path back to the **Client**.
- **Room Not Found**: The building sends a **ROOM_NOT_FOUND** message. The **RentalAgent** processes this and sends a **BUILDING_NOT_FOUND** message to the **Client**.

Cancel Reservation:



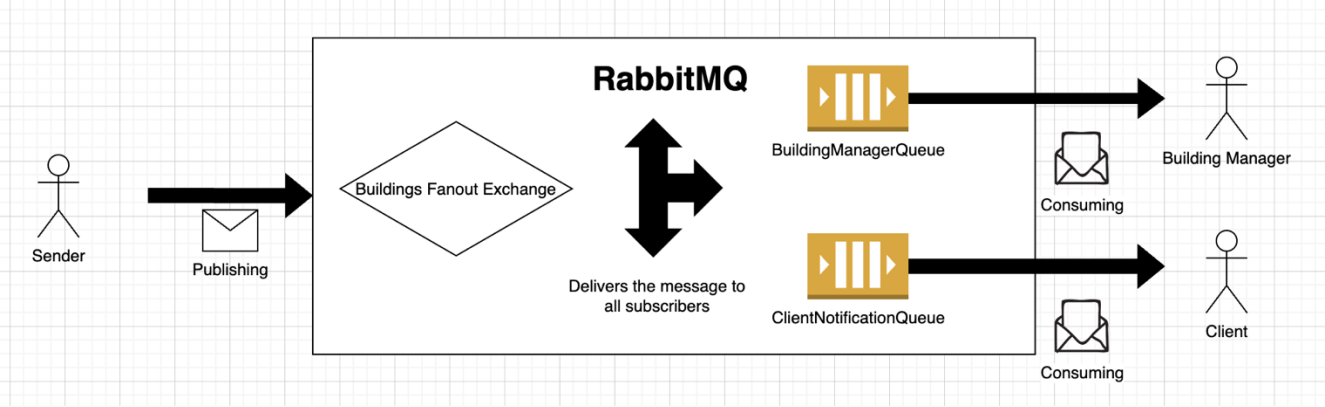


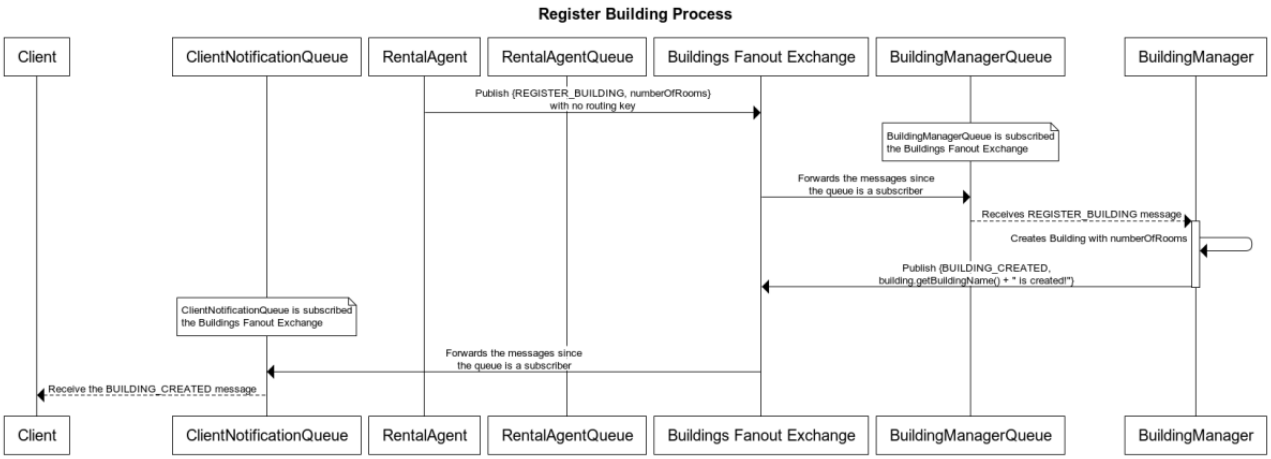
1. **Client Sends Cancellation Request**:
    - The **Client** initiates the cancellation process by sending a **CANCEL_RESERVATION** message with the reservation number to the **RentalAgentQueue** using the **RENTAL_AGENT_KEY**.
2. **Rental Agent Receives Cancellation**:
    - The **RentalAgent** picks up the cancellation message from the **RentalAgentQueue** and retrieves the **Reservation** details.

3. **Rental Agent Consults ReservationManager**:
    - The **RentalAgent** sends the reservation number to the **ReservationManager**.
4. **ReservationManager Finds Reservation**:
    - The **ReservationManager** attempts to find and send the **Reservation** back to the **RentalAgent**.
    - If the **Reservation** is not found, the **RentalAgent** sends a **RESERVATION_CANT_FOUND** message back to the **Client** via **ClientQueue**.
5. **Building Manager Consulted for Building Information**:
    - If the **Reservation** is found, the **RentalAgent** requests the **Building** object for the **Reservation** from the **Building Manager**.
6. **Building Manager Responds with Building Information**:
    - If the building is not found, the **Building Manager** returns **null** and the **RentalAgent** informs the **Client** through **ClientQueue** that the **Building** does not exist.
    - If the **Building** is found, the **Building Manager** sends the **Building** object to the **RentalAgent**.
7. **Cancellation Request Sent to Building**:
    - The **RentalAgent** sends a **CANCEL_RESERVATION** message with the room name to the specific **BuildingQueue** using the **BUILDING_KEY**.
8. **Building Processes Cancellation**:
    - The **Building** receives the **CANCEL_RESERVATION** message and attempts to cancel the booking.

- If the cancellation is successful (the room was booked), the **Building** sends a **RESERVATION_CANCELLED** message back to the **RentalAgentQueue** with the **BUILDING_KEY**.
- The **RentalAgent** then forwards the cancellation confirmation to the **Client** via **ClientQueue**.
- If the cancellation fails (the room was not booked), the **Building** sends a **ROOM_NOT_BOOKED** message back along the same path to inform the **Client** of the failure.

Building Registration





1. **Rental Agent Initiates Building Registration**:
   - The **RentalAgent** publishes a **REGISTER_BUILDING** message with the desired number of rooms to the **Buildings Fanout Exchange**. No routing key is necessary because fanout exchanges distribute messages to all subscribed queues.
2. **Message Distributed to BuildingManagerQueue**:

- The **BuildingManagerQueue** is subscribed to the **Buildings Fanout Exchange**. When the **REGISTER_BUILDING** message is published to the exchange, it is automatically forwarded to all subscribed queues, including **BuildingManagerQueue**.

3. **Building Manager Processes Registration**:
    - The **BuildingManager** receives the **REGISTER_BUILDING** message from the **BuildingManagerQueue**.
    - It then proceeds to create a new **Building** with the number of rooms specified in the message.
4. **Building Manager Notifies System**:
    - After the **Building** is created, the **BuildingManager** publishes a **BUILDING_CREATED** message to the **Buildings Fanout Exchange**. This message includes the new building's name.
5. **Notification Distributed to Subscribers**:
    - The **Buildings Fanout Exchange** forwards the **BUILDING_CREATED** message to all its subscribers.
6. **Clients Receive Building Creation Notification**:
    - The **ClientNotificationQueue** receives the **BUILDING_CREATED** message from the **Building Fanout Exchange** since this queue is a subscriber of the **Building Fanout Exchange** and then the **Client** receives the message.
    - Although it is mentioned in the assignment paper that the client should only be in communication with the rental agent, since it is a notification operation here, delivering this message to the Client via the Rental Agent would add unnecessary complexity, an implementation was made in which the Client can directly access the message.

Errors and Notifications:

- If any errors occur, such as the building not existing or the room not being found, appropriate error messages are sent back to the **Client** through the **ClientQueue**.
- The **Client** is notified at each significant step whether the action was successful or not.