

5조

# 창문 이용료 징수단

Service Robot Project

2

# 목차

1

## 팀원 소개

- 팀 구성
- 개인 역할

2

## 프로그램 개요

- 개발 목적
- 주요 기능 요약

3

## 전체 시스템 구성

- 시스템 구성도
- 주요 시스템 및 기능 소개

4

## 시스템 동작 및 통신

- QoS 설정
- 메세지 통신 방법
- 주요 기능 코드 설명

5

## 마무리

- 기대 효과
- 앞으로의 계획

3

# 팀원 소개

1

## 류재준

- 시스템 전체 통합
- 통계 시스템 개발
- 데이터베이스 설계

2

## 양준혁

- 주방 모니터링 개발
- 로봇 컨트롤 시스템 개발

3

## 이상우

- 테이블 오더 시스템 개발

4

## 김주원

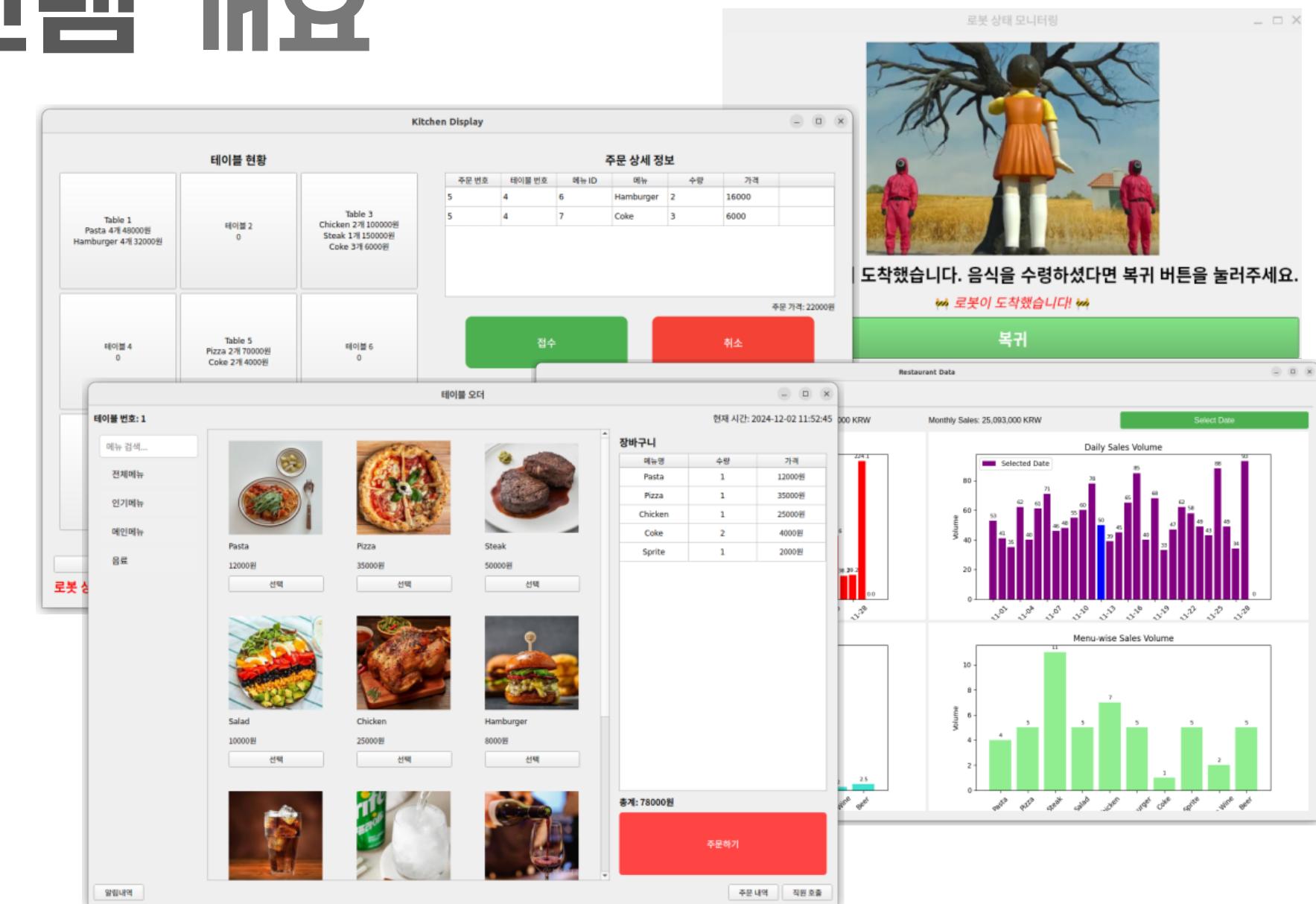
- 발표 자료 제작

# 프로그램 개요

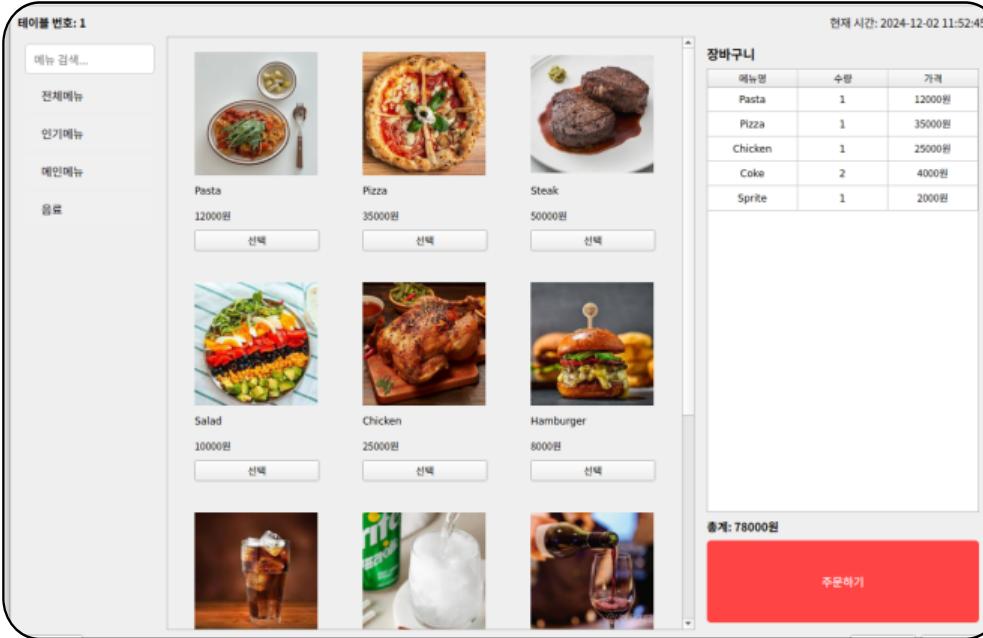
## 프로그램 개발 목적

이 프로그램은 식당에서의 대기 시간 증가, 주문 실수, 높은 인건비 문제를 해결하기 위해 개발하였습니다. 기존 방식에서는 주문 접수와 전달 과정에서 많은 시간이 소요되고 문제가 발생하기 쉬웠습니다. 이를 개선하기 위해 우리는 테이블 오더, 주방 모니터링, 서빙 로봇 시스템을 식당에 통합했습니다.

고객은 테이블에서 메뉴를 직접 선택하고 주문할 수 있으며, 고객의 주문 데이터는 실시간으로 주방에 전달됩니다. 메뉴 조리 완료 후, 서빙 로봇을 이용하여 자동으로 주문한 테이블에 음식을 전달합니다. 식당은 이 프로그램을 통해 고객에게 더 빠르고 편리한 서비스를 제공하고, 운영 효율성을 높이며 비용을 절감할 수 있습니다.

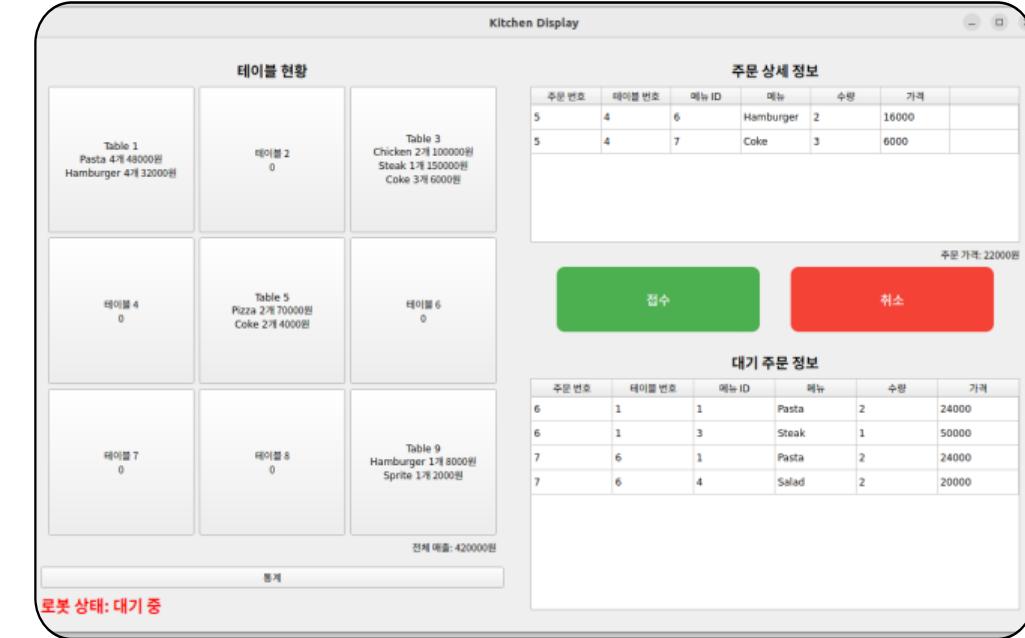


# 주요 기능 요약



## 테이블 오더

- 테이블 비치 태블릿으로 메뉴 선택 및 주문
- 주문 진행 상황 실시간 알림



## 주방 모니터링

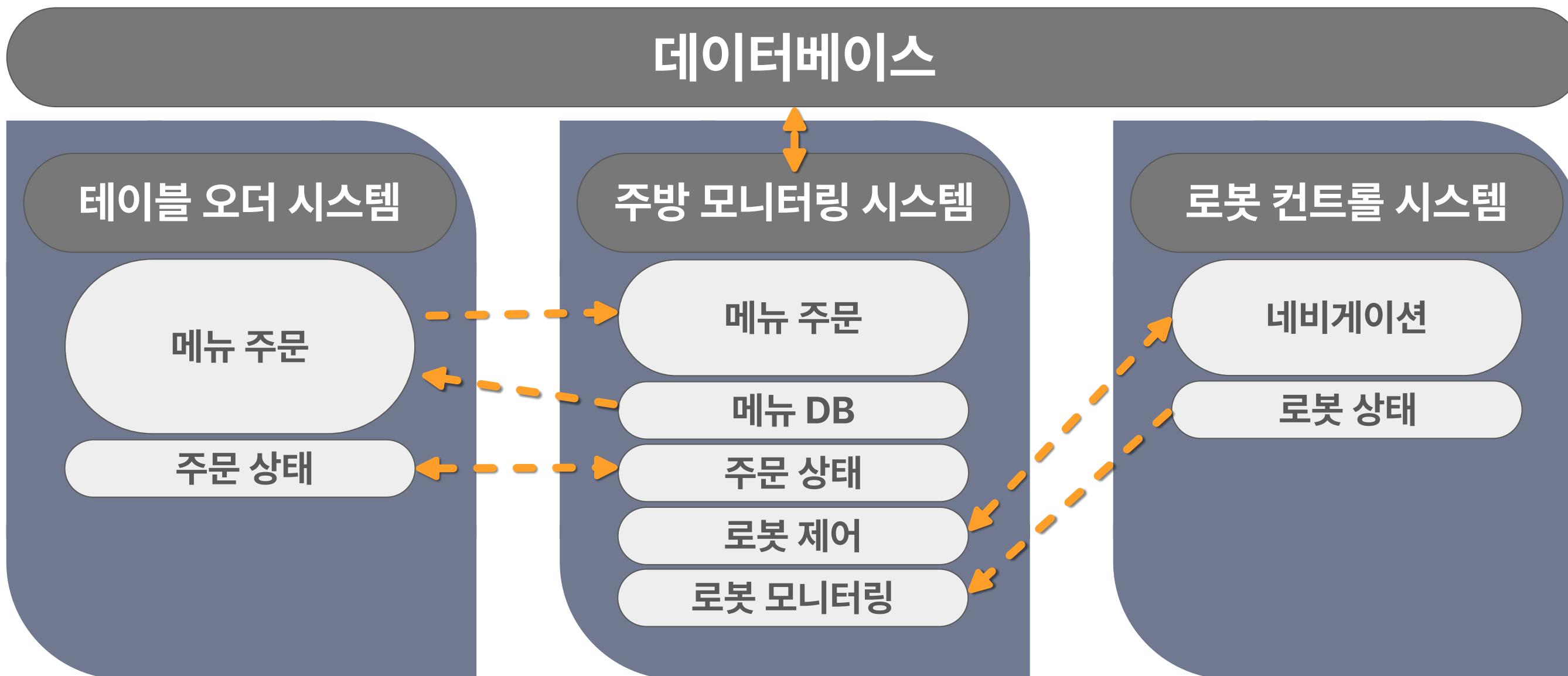
- 태블릿으로 접수된 주문 실시간 확인 및 취소
- 테이블별 주문 내역 확인 및 상태 관리



## 통계 시스템

- 주문 데이터 자동 저장
- 메뉴별 판매량 및 주문 빈도 분석
- 일별, 월별, 메뉴별 매출 데이터 제공

# 시스템 구성도



# 테이블 오더 시스템

기능 1

## 사용자 중심 인터페이스

GUI기반의 고객 친화적인 인터페이스를 제공, 카테고리별 정렬, 메뉴 검색 기능, 직관적인 메뉴 구성으로 고객의 메뉴 탐색 시간을 단축

기능 2

## 메뉴 통합 관리

모든 메뉴 데이터는 서버로 통합 관리되며 각 테이블의 기기는 서버를 통해 메뉴 데이터를 받음. 서버의 변경 사항은 즉시 각 테이블 오더 시스템에 반영됨

기능 3

## 주문 내역 확인

주문 완료 후 태블릿 화면에서 메뉴 이름, 수량, 금액 등을 확인할 수 있음. 이를 통해 주문 내용을 검토하거나, 메뉴 취소, 직원 호출 할 수 있음

기능 4

## 문자 기반 메뉴 검색

메뉴 이름의 일부 문자를 입력해 관련 메뉴를 빠르게 검색 가능

# 주방 모니터링 시스템

기능 1

## 실시간 주문 관리

테이블 오더 시스템에서 접수된 주문을 실시간으로 확인하고 관리, 주문 상태를 실시간으로 업데이트 하여 작업 효율성을 높임

기능 2

## 서빙 로봇 호출

주방에서 간단한 조작으로 서빙 로봇을 호출 할 수 있으며 서빙 로봇의 상태를 실시간으로 확인

기능 3

## 테이블 주문 관리

각 테이블의 주문 내역과 상태를 한눈에 확인할 수 있음, 접수된 주문은 우선순위에 따라 자동으로 정렬되며, 작업 흐름을 최적화

기능 4

## 통계 확인

메뉴별 판매량, 주문 빈도, 매출 데이터를 시각적으로 표시하여 운영 효율성을 개선. 다양한 통계 데이터 제공으로 식당 운영에 활용 할 수 있음

# 서빙 로봇 시스템

기능 1

## 서빙 로봇 호출

주방의 모니터링 시스템에서 버튼 하나로 간단하게 서빙 로봇을 호출, 원하는 대기 지점으로 이동

기능 2

## 경로 최적화 및 안전한 이동

서빙 로봇은 저장된 맵을 활용해 최적 경로를 계산하고, 이동 중 장애물을 자동 회피하여 안전하며 효율적인 경로 이동 가능

기능 3

## 테이블 자동 매칭 및 상태 추적

주문 데이터를 기반으로 지정된 테이블로 정확히 서빙 하며, 로봇의 상태를 실시간으로 추적

기능 4

## 고객 맞춤 인터페이스

서빙 로봇 목표 도착 시각 알림을 통해 고객에게 메뉴가 전달되었음을 표시

# 시스템 동작 및 통신

프로그램 시연

# QoS 설정

## QoS 목표



- 데이터 손실 방지

주문과 상태 등 중요한 데이터는 반드시 전달

- 시스템 안정성 확보

네트워크 부하나 자연 상황에서도 정상 작동

## QoS 설정



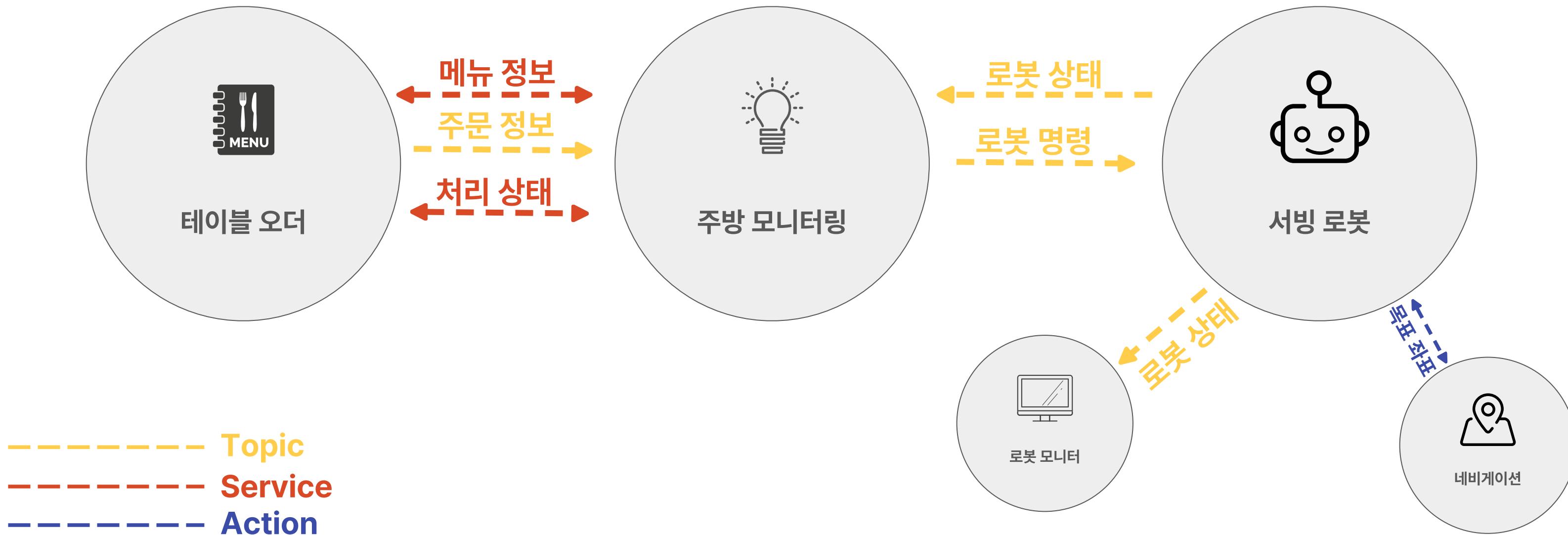
- Reliability = Reliable

주문과 상태 데이터를 반드시 전달

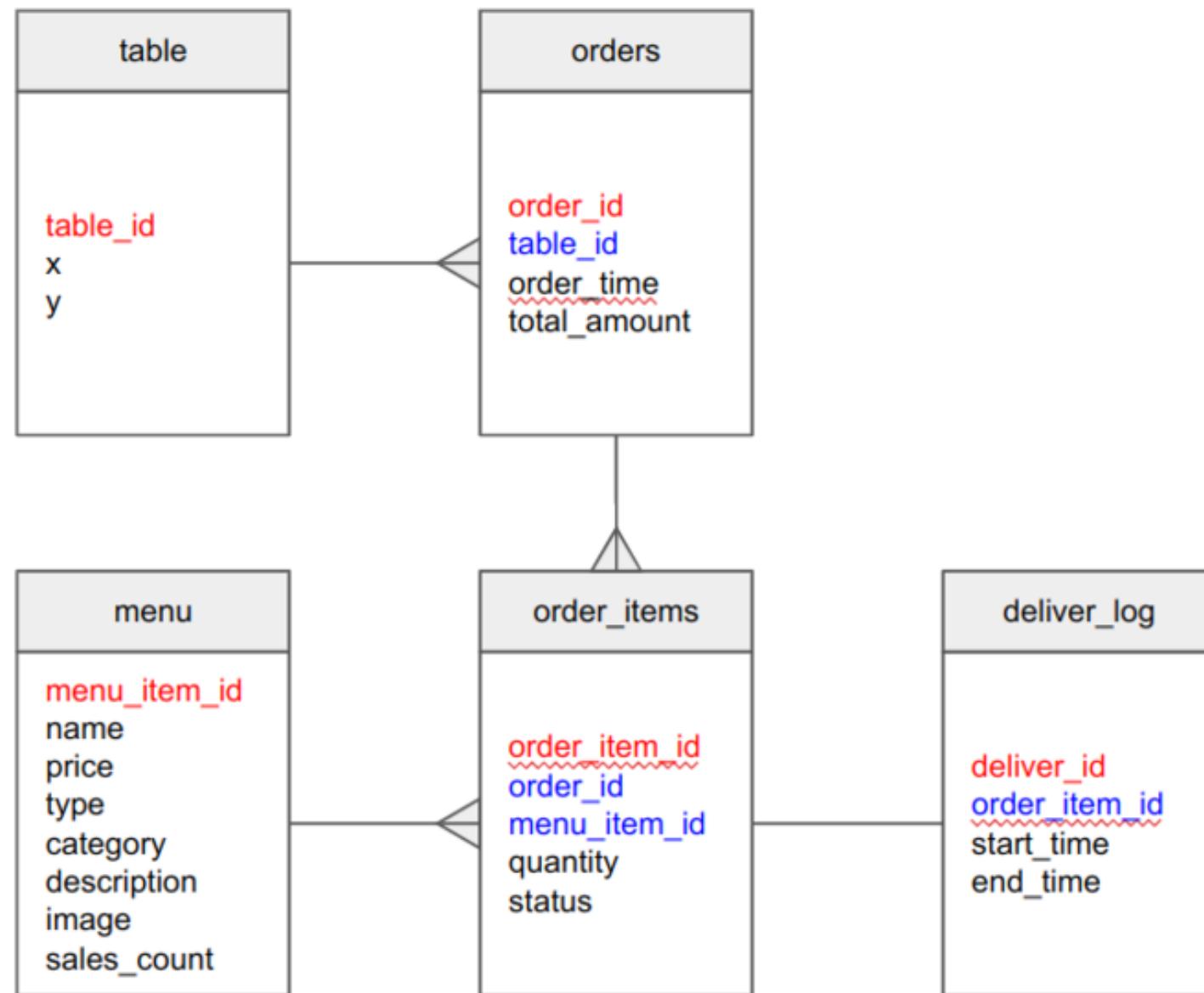
- Depth=10

메시지 전송 실패 시 재전송 대기열로 저장

# 노드별 메세지 통신 방법



# 데이터베이스 소개



# 1. 메뉴 데이터 요청 및 응답

## 테이블 오더 시스템에서 메뉴 데이터 요청

- NODE 클래스의 초기화에서 서비스 클라이언트를 생성하여 메뉴 데이터를 요청합니다.
- `request_table()` 메서드를 호출하여 메뉴 데이터를 요청합니다.

테이블 오더 시스템은 `menu_table_service` 서비스에 `get_menu_table` 요청을 보냅니다.

```
# 메뉴 정보를 요청하기 위한 서비스 클라이언트 생성
self.cli = self.create_client(MenuTable, 'menu_table_service')
```

```
def request_table(self):
    self.req.request_type = 'get_menu_table' # 요청 유형 설정
    future = self.cli.call_async(self.req) # 비동기 서비스 요청
    future.add_done_callback(self.table_response_callback)
```

# 1. 메뉴 데이터 요청 및 응답

## 주방 모니터 시스템에서 메뉴 데이터 요청 처리

- KitchenSubscriber 클래스에서 서비스 서버를 생성하여 메뉴 데이터 요청을 처리합니다.
- handle\_table\_request 메서드에서 요청을 처리합니다.

주방 모니터 시스템은 menu\_table\_service 요청을 수신하면 데이터베이스에서 메뉴 데이터를 조회합니다.

조회한 데이터를 JSON 형태로 직렬화하여 응답으로 보냅니다.

```
#####
# 메뉴 테이블을 제공하기 위한 서비스 서버 생성 #####
self.menu_service = self.create_service(
    MenuTable, # 서비스 타입
    'menu_table_service', # 서비스 이름
    self.handle_table_request # 요청 처리 콜백 함수
)

def handle_table_request(self, request, response):
    """
    메뉴 테이블 요청 처리
    """
    if request.request_type == 'get_menu_table':
        try:
            # 데이터베이스 연결 및 메뉴 데이터 가져오기
            conn = db.db_connection()
            cursor = conn.cursor()

            cursor.execute('SELECT * FROM menu')
            table_rows = cursor.fetchall()

            # 컬럼 이름 가져오기
            columns = [desc[0] for desc in cursor.description]

            conn.close()

            # 테이블 데이터를 딕셔너리로 변환하여 직렬화
            response.table_data = [json.dumps(dict(zip(columns, row))) for row in table_rows]
            self.get_logger().info(f"Sending full table data with {len(table_rows)} rows.")
        except Exception as e:
            self.get_logger().error(f"Failed to fetch table data: {e}")
            # 에러 메시지를 문자열 리스트로 설정
            response.table_data = ["Error: Unable to fetch table data"]

        else:
            # 잘못된 요청 유형 처리
            response.table_data = ["Error: Invalid request type"]

    return response
```

 You have Docker installed on

# 1. 메뉴 데이터 요청 및 응답

## 테이블 오더 시스템에서 메뉴 데이터 응답 처리

- `table_response_callback` 메서드에서 응답을 처리합니다.

응답으로 받은 메뉴 데이터를 파싱하여 `menu_db`에 저장합니다.

GUI를 업데이트하여 사용자에게 메뉴를 표시합니다.

```
def table_response_callback(self, future):
    response = future.result()
    menu_items = []
    for item in response.table_data:
        item = json.loads(item)
        # 메뉴 아이템 딕셔너리 생성
        menu_item_dict = {
            'id': int(item.get('menu_item_id')),
            'name': item.get('name', ''),
            'price': int(item.get('price')),
            'category': item.get('category', ''),
            'description': item.get('description', ''),
            'image_path': item.get('image', ''),
            'sales_count': item.get('sales_count', '')
        }
        menu_items.append(menu_item_dict)
    # 메뉴 데이터베이스 로드 및 GUI 업데이트
    self.menu_db = MenuDatabase(menu_items)
    if self.gui is not None:
        self.gui.menu_db = self.menu_db
        self.gui.update_menu_display()
```

## 2. 주문 발생 및 처리

### 테이블 오더 시스템에서 주문 메시지 발행

- `place_order` 메서드에서 주문 메시지를 생성하고 퍼블리시합니다.

주문 정보(테이블 ID와 주문 아이템들)를 JSON으로 직렬화하여 `order_topic` 토픽에 퍼블리시합니다.

```
def place_order(self):
    order_message = {
        "table_id": 1,
        "orders": [
            {
                "menu_item_id": item["id"],
                "item": item["name"],
                "quantity": item["quantity"],
                "price": int(item["price"]) * item["quantity"]
            }
            for item in self.order_items
        ]
    }
    # ROS 퍼블리셔로 메시지 전송
    self.node.queue.put(json.dumps(order_message))
```

## 2. 주문 발생 및 처리

### 주방 모니터 시스템에서 주문 메시지 수신 및 처리

- KitchenSubscriber 클래스의 `order_callback` 메서드에서 주문 메시지를 수신합니다.
- KitchenMonitoring 클래스의 `update_order_details` 메서드에서 주문 정보를 업데이트합니다.

주방 모니터 시스템은 `order_topic` 토픽에서 주문 메시지를 수신합니다.

수신한 주문 정보를 파싱하여 주문 번호를 부여하고, 주문 대기열이나 현재 주문으로 분류합니다.

GUI를 통해 주방 직원에게 주문 상세 정보를 표시합니다.

```
def order_callback(self, msg):
    # 메시지를 파싱하여 주문 정보를 추출
    data = json.loads(msg.data)
    table_id = data["table_id"]
    orders = data["orders"]
    # GUI로 주문 정보를 전달하기 위해 시그널을 사용
    signaler.order_received.emit(msg.data)

def update_order_details(self, message):
    """ROS 토픽으로부터 받은 주문 정보를 화면에 업데이트"""
    try:
        # 받은 메시지를 JSON 형식으로 파싱
        data = json.loads(message)
        table_id = data["table_id"] # 테이블 번호
        orders = data["orders"] # 주문 목록

        ##### 주문 번호 할당 : 한 번에 들어온 주문은 주문 번호가 같아야 함#####
        order_number = self.order_counter
        self.order_counter += 1 # 주문 번호 증가
        ##### ##### ##### ##### ##### ##### ##### ##### #####
        # 주문에 'order_number'와 상태 필드 추가
        for order in orders:
            order['order_number'] = order_number
            order['checked'] = False
            order['disabled'] = False
```

## 2. 주문 발생 및 처리

### 주방에서 주문 수락 또는 거절 처리

주방 직원이 주문을 수락하거나 거절할 수 있습니다.

- handle\_accept 메서드에서 주문 수락 시 처리

```
# 대기 주문 정보에서 다음 주문을 가져와 주문 상세 정보에 표시
if self.cumulative_table.rowCount() > 0:
    first_order_number = self.cumulative_table.item(0, 0).text()

    rows_to_move = []
    for row in range(self.cumulative_table.rowCount()):
        if self.cumulative_table.item(row, 0).text() == first_order_number:
            rows_to_move.append(row)

    for row in rows_to_move:
        row_count = self.order_table.rowCount()
        self.order_table.insertRow(row_count)
        for col in range(self.cumulative_table.columnCount()):
            item = self.cumulative_table.item(row, col)
            if item:
                self.order_table.setItem(row_count, col, QTableWidgetItem(item.text()))

# 대기 주문 정보에서 해당 행 삭제
for row in reversed(rows_to_move):
    self.cumulative_table.removeRow(row)

# 총 가격 업데이트
self.update_total_price()
self.update_order_total_price()

# 주문 결과를 서비스로 전송
response_message = f"Order Accepted for Table {table_id}"
self.send_order_result(response_message)
```

## 2. 주문 발생 및 처리

### 주방에서 주문 수락 또는 거절 처리

- handle\_cancel 메서드에서 주문 거절 시 처리

```
def handle_cancel(self):
    """'Cancel' 버튼 클릭 시 현재 주문을 취소하고 화면을 업데이트"""
    if self.order_table.rowCount() == 0:
        return

    # 테이블 번호 가져오기
    table_id = int(self.order_table.item(0, 1).text())

    # 주문 상세 정보 테이블 초기화
    self.order_table.setRowCount(0)

    # 대기 주문 정보에서 다음 주문 가져오기
    if self.cumulative_table.rowCount() > 0:
        next_order_numbers = []
        first_order_number = self.cumulative_table.item(0, 0).text()

        # 다음 주문 번호와 동일한 주문 모두 가져오기
        for row in range(self.cumulative_table.rowCount()):
            order_number = self.cumulative_table.item(row, 0).text()
            if order_number == first_order_number:
                next_order_numbers.append(row)

        for row in next_order_numbers:
            order_table: QTableWidget = self.order_table
            row_count = self.order_table.rowCount()
            self.order_table.insertRow(row_count)
            for col in range(self.cumulative_table.columnCount()):
                item = self.cumulative_table.item(row, col)
                if item:
                    self.order_table.setItem(row_count, col, QTableWidgetItem(item.text()))

        # 대기 주문에서 해당 주문 삭제
        for row in reversed(next_order_numbers):
            self.cumulative_table.removeRow(row)

        # 주문 가격 총합 업데이트
        self.update_order_total_price()

        # 서비스 요청 전송
        response_message = f"Order Canceled for Table {table_id}"
        self.send_order_result(response_message)
```

## 2. 주문 발생 및 처리

### 주방에서 주문 수락 또는 거절 처리

- send\_order\_result 메서드에서 주문 결과를 전송

주문 수락 또는 거절에 따라 주문 결과 메시지를 생성합니다.

order\_result\_service를 통해 테이블 오더 시스템에 주문 결과를 전송합니다.

```
def send_order_result(self, message):
    """주문 결과 메시지를 ROS 노드로 전송"""
    self.subscriber_node.send_order_result(message)
```

## 2. 주문 발생 및 처리

### 테이블 오더 시스템에서 주문 결과 수신 및 처리

- NODE 클래스의 `order_result_callback` 메서드에서 주문 결과를 수신합니다.
- GUI 클래스의 `check_notifications` 메서드에서 알림을 처리합니다.
- `show_notification_popup` 메서드에서 알림 팝업을 표시합니다.

테이블 오더 시스템은 `order_result_service`를 통해 주문 결과를 수신합니다.

수신한 결과를 GUI에 표시하고, 주문 내역을 업데이트합니다.

```
# 알림 팝업 표시
def show_notification_popup(self, message):
    QMessageBox.information(self, "알림", message)
```

```
def order_result_callback(self, request, response):
    """주문 처리 결과를 주방 디스플레이 노드로부터 수신"""
    self.get_logger().info(f"Received order result: {request.result_message}")
    self.order_result_received = True
    self.last_order_result = request.result_message
    # 주문 결과가 "Order Accepted"일 경우만 처리
    if "Order Accepted" in request.result_message:
        self.get_logger().info("Order accepted, adding to order history.")

        # GUI 클래스에서 저장된 pending_order 목록을 가져와서 추가
        if hasattr(self.gui, 'pending_order') and self.gui.pending_order:
            self.get_logger().info(f"Pending orders to be added: {self.gui.pending_order}")
            self.gui.add_to_order_history(self.gui.pending_order) # GUI 클래스의 메소드 호출
            self.gui.pending_order.clear() # 목록을 추가한 후 초기화

    elif "Order Canceled" in request.result_message:
        # GUI 클래스에서 저장된 pending_order 목록을 가져와서 추가
        self.get_logger().info("Order canceled, clearing pending orders.")
        if hasattr(self.gui, 'pending_order') and self.gui.pending_order:
            self.gui.pending_order.clear() # 목록을 초기화

    else:
        self.get_logger().error(f"Order failed: {request.result_message}")

    self.notification_queue.put(request.result_message)
    response.success = True # 응답 확인
    return response

def check_notifications(self):
    """알림 큐에서 메시지를 가져와 알림 리스트에 추가"""
    while not self.node.notification_queue.empty():
        notification = self.node.notification_queue.get()
        self.notifications.append(notification)
        # 시그널을 통해 GUI 스레드에서 팝업 표시
        self.notification_received.emit(notification)
```

# 3. 로봇 제어 및 상태 업데이트

## 주방 모니터 시스템에서 로봇 제어 명령

- 주방 직원이 로봇을 제어하기 위해 팝업 창에서 명령을 실행합니다.
- ControlPopup 클래스의 start\_robot 메서드에서 로봇 이동 명령을 퍼블리시합니다.

주방 모니터 시스템은 robot\_command 토픽으로 로봇 이동 명령을 퍼블리시합니다.

명령에는 이동할 위치와 배송할 주문 아이템 ID가 포함됩니다.

```
def start_robot(self):
    """해당 테이블로 이동 명령 퍼블리시"""
    if isinstance(self.table_id, int):
        position_key = f"table_{self.table_id}"

        # 선택된 주문의 order_item_id 수집
        selected_order_item_ids = []
        for idx in range(self.order_table.rowCount()):
            checkbox = self.order_table.cellWidget(idx, 0)
            if checkbox.isChecked() and checkbox.isEnabled(): # 체크되어 있고 활성화된 항목만 선택
                # 주문 아이템 ID 가져오기
                order_item_id = int(self.order_table.item(idx, 2).text())

                # "Delivered" 상태인 경우는 제외
                order_status = self.order_table.item(idx, 7).text() # 주문 상태가 "Delivered"인지 확인
                if order_status != "Delivered":
                    selected_order_item_ids.append(order_item_id)

            # 상태 업데이트
            self.order_table.setItem(idx, 7, QTableWidgetItem("Delivering"))
            checkbox.setEnabled(False) # 비활성화

            # 정확한 주문 항목 찾기 및 상태 업데이트
            for order in self.orders:
                for item in order['items']:
                    if item['order_item_id'] == order_item_id:
                        item['checked'] = True
                        item['disabled'] = True

            # 데이터베이스에 배달 로그 기록 (배달 시작)
            db.insert_delivery_log(order_item_id, end=False)
            break

        if not selected_order_item_ids:
            QMessageBox.warning(self, "Robot Control", "No orders selected for delivery.")
            return

    # 로봇 명령 생성 및 퍼블리시
    command = {
        "command": "move",
        "position": position_key,
        "order_item_ids": selected_order_item_ids # 추가된 필드
    }
    self.subscriber_node.publish_robot_command(command)
```

# 3. 로봇 제어 및 상태 업데이트

## 로봇 컨트롤러에서 로봇 명령 수신 및 처리

- RobotController 클래스의 command\_callback 메서드에서 명령을 수신합니다.
- move\_to\_position 메서드에서 로봇을 이동시킵니다.

로봇 컨트롤러는 robot\_command 토픽에서 명령을 수신하고, 로봇을 해당 위치로 이동시킵니다.

이동 시작 시 robot\_status 토픽으로 상태를 퍼블리시합니다.

```
def move_to_position(self, position, position_key, order_item_ids):
    """로봇을 지정된 위치로 이동시키는 함수"""
    # NavigateToPose 목표 메시지 생성
    goal_msg = NavigateToPose.Goal()

    # 목표 위치 설정
    goal_msg.pose.pose.position.x = position['x']
    goal_msg.pose.pose.position.y = position['y']

    # 오일러 각도를 쿼터니언으로 변환하여 방향 설정
    q = self.euler_to_quaternion(0, 0, position['theta'])
    goal_msg.pose.pose.orientation = q

    # 좌표 프레임 설정
    goal_msg.pose.header.frame_id = 'map'
    goal_msg.pose.header.stamp = self.get_clock().now().to_msg()

    # 액션 서버가 실행될 때까지 대기
    while not self.action_client.wait_for_server(timeout_sec=1.0):
        self.get_logger().info('Waiting for action server...')

    # 목표를 액션 서버로 보내기
    self._send_goal_future = self.action_client.send_goal_async(goal_msg, feedback_callback=self.feedback_callback)
    self._send_goal_future.add_done_callback(lambda future: self.goal_response_callback(future, position_key, order_item_ids))
```

```
def command_callback(self, msg):
    """로봇 명령을 수신하여 처리하는 콜백 함수"""
    try:
        data = json.loads(msg.data)
        command = data.get('command')
        position_key = data.get('position')
        order_item_ids = data.get('order_item_ids', [])

        if command == 'move' and position_key in self.positions:
            position = self.positions[position_key]

            # 로봇 이동 시작 상태 퍼블리시
            status_msg = String()

            # Standardizing message format with JSON
            status_data = {
                "status": "",
                "position": position_key,
                "order_item_ids": order_item_ids
            }

            # Set the appropriate status based on the position
            if position_key == 'waiting':
                status_data["status"] = "로봇이 대기 위치로 이동 중입니다."
            elif position_key == 'kitchen':
                status_data["status"] = "로봇이 주방 위치로 이동 중입니다."
            else:
                status_data["status"] = f"로봇이 {order_item_ids}을 배송하기 위해 {position_key}으로 이동 중입니다."

            # Publish the message as a JSON string
            status_msg.data = json.dumps(status_data)
            self.status_publisher.publish(status_msg)
            self.get_logger().info(f"Published robot status: {status_msg.data}") # 추가된 로그

            # Move to the target position
            self.move_to_position(position, position_key, order_item_ids)
        else:
            self.get_logger().warn(f"Unknown command or position: {command}, {position_key}")

    except json.JSONDecodeError:
        self.get_logger().error("Failed to decode JSON from robot_command")
```

## 3. 로봇 제어 및 상태 업데이트

### 로봇 상태 모니터에서 로봇 상태 수신 및 표시

- RobotStatusMonitor 클래스의 robot\_status\_callback 메서드에서 상태 메시지를 수신합니다.

```
def robot_status_callback(self, msg):  
    try:  
        # Log the raw message  
        self.get_logger().info(f"Received raw message: {msg.data}")  
  
        # Check if the message looks like a valid JSON string  
        if msg.data.startswith('{') and msg.data.endswith('}'):  
            # Attempt to decode using unicode_escape for potential issues with encoding  
            decoded_data = msg.data.encode('utf-8').decode('unicode_escape')  
            self.get_logger().info(f"Decoded message: {decoded_data}")  
  
            # Try parsing the JSON message  
            status_msg = json.loads(decoded_data)  
  
            # Extract relevant fields  
            status = status_msg.get("status", "")  
            position = status_msg.get("position", "")  
            order_item_ids = status_msg.get("order_item_ids", [])  
  
            # Update GUI with the extracted data  
            self.gui.update_status(status, position, order_item_ids)  
        else:  
            self.get_logger().error(f"Message is not a valid JSON: {msg.data}")  
    except json.JSONDecodeError as e:  
        self.get_logger().error(f"Failed to parse robot status message: {str(e)}")  
    except UnicodeDecodeError as e:  
        self.get_logger().error(f"Unicode decode error: {str(e)}")
```

### 3. 로봇 제어 및 상태 업데이트

#### 로봇 상태 모니터에서 로봇 상태 수신 및 표시

- RobotMonitorGUI 클래스의 update\_status 메서드에서 GUI를 업데이트합니다.

로봇 컨트롤러는 이동 상태 및 완료 상태를 robot\_status 토픽으로 퍼블리시합니다.

로봇 상태 모니터는 해당 메시지를 수신하여 GUI를 업데이트하고, 사용자에게 로봇의 현재 상태를 표시합니다.

```
def update_status(self, status, position, order_item_ids):
    """로봇 상태 메시지를 받아 GUI를 업데이트하는 함수"""
    self.status_label.setText(f"로봇 상태: {status}")

    # 상태에 따라 추가 메시지와 스타일 변경
    if "이동 중입니다" in status:
        self.warning_label.setText("⚠️ 로봇이 이동 중입니다. 주변을 조심해 주세요! ⚠️")
        self.status_background.setStyleSheet("background-color: #FFDAB9;") # 복숭아색 배경
    ##### 로봇이 테이블에 도착했을때 알림 뜨도록 설정 #####
    elif "음식이 도착했습니다" in status:
        self.warning_label.setText("⚠️ 로봇이 도착했습니다! ⚠️")
        self.return_button.setVisible(True) # 복귀 버튼 활성화
        self.status_background.setStyleSheet("background-color: #FFDAB9;") # 복숭아색 배경
    ##### 대기 위치입니다 #####
    elif "대기 위치입니다" in status or "주방 위치입니다" in status:
        self.warning_label.setText("") # 추가 메시지 숨기기
        #### 복귀 버튼을 숨기고 있어야함 #####
        self.return_button.setVisible(False) # 복귀 버튼 숨김
        self.status_background.setStyleSheet("background-color: #90EE90;") # 연두색 배경
    else:
        self.warning_label.setText("") # 기본 상태일 때 추가 메시지 숨기기
    ##### 복귀 버튼은 로봇이 테이블에 도착했을 때만 나오도록 설정 #####
        self.return_button.setVisible(False) # 복귀 버튼 숨김
        self.status_background.setStyleSheet("background-color: #F0F0F0;") # 기본 배경색
```

# 3. 로봇 제어 및 상태 업데이트

## 로봇이 테이블에 도착했을 때 처리

- RobotController 클래스의 get\_result\_callback 메서드에서 이동 완료 시 상태를 퍼블리시합니다.

로봇 컨트롤러는 로봇이 테이블에 도착했음을 알리는 상태 메시지를 퍼블리시합니다.

```
def get_result_callback(self, future, position_key, order_item_ids):
    """액션 수행 결과를 처리하는 콜백 함수"""
    result = future.result().result
    status = future.result().status

    if status == GoalStatus.STATUS_SUCCEEDED:
        self.get_logger().info('Goal succeeded!')

        # 목표 도착 상태 퍼블리시
        status_msg = {
            "status": "",
            "position": position_key,
            "order_item_ids": order_item_ids
        }

        if position_key == 'waiting':
            status_msg["status"] = "대기 위치입니다."
        elif position_key == 'kitchen':
            status_msg["status"] = "주방 위치입니다."
        elif "table_" in position_key: # 테이블에 도착했을 경우
            status_msg["status"] = (
                "음식이 도착했습니다. "
                "음식을 수령하셨다면 복귀 버튼을 눌러주세요."
            )
        else:
            status_msg["status"] = f"{position_key} 위치입니다."

        # JSON 직렬화 후 퍼블리시
        self.status_publisher.publish(String(data=json.dumps(status_msg)))
        self.get_logger().info(f"Published robot status: {status_msg}")
    else:
        self.get_logger().info(f'Goal failed with status: {status}')
```

## 3. 로봇 제어 및 상태 업데이트

### 테이블에서 로봇 복귀 명령

- 사용자가 "복귀" 버튼을 클릭하면 로봇에게 복귀 명령을 보냅니다.
- RobotMonitorGUI 클래스의 send\_return\_command 메서드에서 복귀 명령을 퍼블리시합니다.

```
##### 복귀 명령을 퍼블리시 하기 위한 메서드 #####
def send_return_command(self):
    """복귀 명령을 퍼블리시"""
    command_msg = String()
    command_msg.data = json.dumps({"command": "move", "position": "waiting"})
    self.command_publisher.publish(command_msg)

    ##### 손님이 복귀 버튼을 누르면 복귀 버튼을 다시 숨겨야함 #####
    self.return_button.setVisible(False)
#####
```

# 3. 로봇 제어 및 상태 업데이트

## 로봇 컨트롤러에서 복귀 명령 수신 및 처리

- RobotController 클래스의 command\_callback 메서드에서 복귀 명령을 수신하고 처리합니다.

로봇 컨트롤러는 복귀 명령을 수신하여 로봇을 대기 위치로 이동시킵니다.  
이동 시작 및 완료 시점에 상태를 퍼블리시하여 시스템에 알립니다.

```
def command_callback(self, msg):
    """로봇 명령을 수신하여 처리하는 콜백 함수"""
    try:
        data = json.loads(msg.data)
        command = data.get('command')
        position_key = data.get('position')
        order_item_ids = data.get('order_item_ids', [])
        
        if command == 'move' and position_key in self.positions:
            position = self.positions[position_key]
            
            # 로봇 이동 시작 상태 퍼블리시
            status_msg = String()
            
            # Standardizing message format with JSON
            status_data = {
                "status": "",
                "position": position_key,
                "order_item_ids": order_item_ids
            }
            
            # Set the appropriate status based on the position
            if position_key == 'waiting':
                status_data["status"] = "로봇이 대기 위치로 이동 중입니다."
            elif position_key == 'kitchen':
                status_data["status"] = "로봇이 주방 위치로 이동 중입니다."
            else:
                status_data["status"] = f"로봇이 {order_item_ids}을 배송하기 위해 {position_key}으로 이동 중입니다."
                
            # Publish the message as a JSON string
            status_msg.data = json.dumps(status_data)
            self.status_publisher.publish(status_msg)
            self.get_logger().info(f"Published robot status: {status_msg.data}") # 추가된 로그
            
            # Move to the target position
            self.move_to_position(position, position_key, order_item_ids)
        else:
            self.get_logger().warn(f"Unknown command or position: {command}, {position_key}")
    except json.JSONDecodeError:
        self.get_logger().error("Failed to decode JSON from robot_command")
```

# 4. 배달 로그 및 주문 상태 업데이트

## 로봇이 테이블에 도착했을 때 배달 로그 업데이트

- KitchenSubscriber 클래스의 robot\_status\_callback 메서드에서 로봇 상태를 수신합니다.

로봇이 테이블에 도착했다는 상태 메시지를 수신하면, 해당 주문 아이템의 배달 로그를 업데이트하고 상태를 "Delivered"로 변경합니다.

```
def robot_status_callback(self, msg):
    """로봇 상태 메시지를 받으면 호출되는 콜백 함수"""
    print(msg)
    try:
        status_msg = json.loads(msg.data)
        status = status_msg.get("status", "")
        position = status_msg.get("position", "")
        order_item_ids = status_msg.get("order_item_ids", [])

        self.get_logger().info(f"Received robot status: {status}")
        self.get_logger().info(f"Position: {position}, Order IDs: {order_item_ids}")

        if "음식이 도착" in status:
            # order_item_ids를 정수형으로 변환
            for order_item_id in map(int, order_item_ids):
                print(order_item_id, type(order_item_id))
                db.insert_delivery_log(order_item_id, end=True)

            # 시그널을 통해 GUI로 상태 메시지를 전달
            signaler.robot_status_updated.emit(status)
    except json.JSONDecodeError:
        self.get_logger().error("Failed to parse robot status message")
```

## 4. 배달 로그 및 주문 상태 업데이트

### 데이터베이스에 배달 로그 업데이트

- `insert_delivery_log` 메서드에서 배달 완료 처리를 합니다.

주문 아이템의 배달 완료 시간을 기록하고 상태를 업데이트하여 배달이 완료되었음을 데이터베이스에 반영합니다.

```
def insert_delivery_log(order_item_id, end=False):
    with db_connection() as conn:
        cursor = conn.cursor()
        if end:
            end_time = get_current_timestamp()
            cursor.execute('''
                UPDATE deliver_log
                SET end_time = ?
                WHERE order_item_id = ? AND end_time IS NULL
            ''', (end_time, order_item_id))
            cursor.execute('''
                UPDATE order_items
                SET status = "Delivered"
                WHERE order_item_id = ?
            ''', (order_item_id,))
        # ...
```

# 기대 효과

- 효율성**
- 고객이 직접 주문을 입력, 주방으로 실시간 전달
  - 서빙 로봇이 조리된 메뉴를 자동으로 테이블에 서빙
- 고객 경험 개선**
- 고객이 직접 주문할 수 있어 대기 시간이 단축
  - 서빙 로봇으로 고객에게 신선한 경험과 재미를 제공
  - 자동화된 서비스로 고객 만족도 향상



- 비용 절감**
- 주문과 서빙이 자동화되어 식당의 인건비 감소
  - 고객 응대, 청소 등 다른 업무에 집중 가능
- 스마트 운영**
- 데이터 기반 운영으로 직원 개입을 최소화
  - 메뉴 판매 데이터, 시간대별 매출 등 데이터를 분석하여 운영 전략을 최적화

# 앞으로의 계획

비동기 처리에 대한 꼼꼼한 검증 필요.  
QA를 하다 보면 더 많은 문제가 있을 것으로 예상됨.

다중 작업 처리에 대한 검증이 필요.  
만약 여러 테이블에서 동시에 주문이 들어올 경우 문제가 발생할 수 있음.

주방 입장에서 더 좋은 정보를 제공하는지에 의문

로봇과 더 다채롭게 상호작용 할 수 있는 방법



# Thank You!

경청해주셔서 감사합니다.