src/composables/useMapRouting.js

```
import { ref, watch } from 'vue';
import { getWaypoints } from '@/services/LocationService'; // Use @ para alias de src/

export function useMapRouting(userPosition, selectedLocal, currentFloor, mapScale, mapDimensions) {
  const routeSegments = ref([]); // Array de { x, y, length, angle }
  const hasRoute = ref(false);
  const debugWaypoints = ref([]); // Para visualizar waypoints no mapa
  const routingError = ref(null);

  const waypoints = getWaypoints();
  const waypointMap = {};
  waypoints.forEach(wp => { waypointMap[wp.id] = wp; });

  // --- Funções Auxiliares Internas ---

  /** Encontra o waypoint mais próximo de um ponto (x, y) em um andar específico */
  const findNearestWaypoint = (x, y, andar) => {
    let nearest = null;
    let minDistanceSq = Infinity; // Comparar quadrados evita raiz quadrada

    for (const waypoint of waypoints) {
      if (waypoint.andar !== andar) continue;
      const dx = waypoint.x - x;
      const dy = waypoint.y - y;
      const distanceSq = dx * dx + dy * dy;

      if (distanceSq < minDistanceSq) {
        minDistanceSq = distanceSq;
        nearest = waypoint;
      }
    }
    return nearest;
  };

```

```
  /** Algoritmo de Dijkstra para encontrar o caminho mais curto entre waypoints */
  const findShortestPath = (startId, endId) => {
    const distances = {};
    const previous = {};
    const unvisited = new Set();

    // Inicialização
    waypoints.forEach(wp => {
      // Considera apenas waypoints do andar atual ou pontos de conexão entre andares
      // A lógica de Dijkstra precisa ser adaptada para multi-andares se necessário
      // Por ora, focamos em rotas no mesmo andar.
      if (wp.andar === currentFloor.value) {
          distances[wp.id] = wp.id === startId ? 0 : Infinity;
          previous[wp.id] = null;
          unvisited.add(wp.id);
      } else {
          // Inicializa distâncias para outros andares como infinito,
          // a menos que seja um ponto de conexão direto.
          // Lógica mais complexa necessária para rotas multi-andar completas.
           distances[wp.id] = Infinity;
           previous[wp.id] = null;
      }
    });


     // Garante que o ponto de partida esteja no conjunto, mesmo que seja ponto de conexão
     if(waypointMap[startId]){
        distances[startId] = 0;
        unvisited.add(startId);
     } else {
         console.error("Waypoint inicial não encontrado:", startId);
         return null; // Ponto de partida inválido
     }

     // Garante que o ponto final exista no mapa de waypoints
     if(!waypointMap[endId]){
        console.error("Waypoint final não encontrado:", endId);
```

```
        return null;
      }


    while (unvisited.size > 0) {
      // Encontrar nó não visitado com menor distância
      let currentId = null;
      let smallestDistance = Infinity;
      for (const nodeId of unvisited) {
        if (distances[nodeId] < smallestDistance) {
          smallestDistance = distances[nodeId];
          currentId = nodeId;
        }
      }

      // Se não encontrou (inalcançável) ou chegou ao destino
      if (currentId === null || currentId === endId) break;

      unvisited.delete(currentId);
      const currentWaypoint = waypointMap[currentId];

      // Se currentWaypoint for indefinido, algo está errado
      if (!currentWaypoint) {
          console.error("Waypoint atual inválido no Dijkstra:", currentId);
          continue; // Pula para a próxima iteração
      }


      // Iterar sobre vizinhos (conexões)
      if (currentWaypoint.connections) {
        for (const neighborId of currentWaypoint.connections) {
          const neighbor = waypointMap[neighborId];

          // Pula se o vizinho não existe ou não está no andar (simplificação para rota no mesmo andar)
          if (!neighbor || neighbor.andar !== currentFloor.value) continue;

          // Calcula distância euclidiana entre waypoints
```

```
          const dx = currentWaypoint.x - neighbor.x;
          const dy = currentWaypoint.y - neighbor.y;
          const distance = Math.sqrt(dx * dx + dy * dy); // Distância percentual

          const totalDistance = distances[currentId] + distance;

          if (totalDistance < distances[neighborId]) {
            distances[neighborId] = totalDistance;
            previous[neighborId] = currentId;
          }
        }
      } else {
           console.warn(`Waypoint ${currentId} não possui conexões definidas.`);
      }

    } // Fim do while

    // Reconstruir caminho
    const path = [];
    let current = endId;
    // Verifica se o destino foi alcançado (previous[endId] não será null se um caminho foi encontrado)
    // Ou se o início e fim são o mesmo waypoint
    if (previous[endId] !== undefined || startId === endId) {
        while (current !== null && current !== undefined) {
            // Adiciona ao início do array
            if(waypointMap[current]){
               path.unshift(waypointMap[current]);
            } else {
               console.error("Waypoint inválido durante reconstrução do caminho:", current);
               break; // Interrompe se encontrar waypoint inválido
            }
            // Verifica se existe um nó anterior antes de acessá-lo
            if (previous[current] !== undefined) {
               current = previous[current];
            } else {
               // Chegou ao início ou houve um erro
               break;
```

```
            }
        }
    }


    // Retorna o caminho se ele começar no waypoint inicial esperado, senão retorna null
    return path.length > 0 && path[0]?.id === startId ? path : null;
  };

  /** Cria um segmento de rota (linha) entre dois pontos */
  const createRouteSegment = (x1, y1, x2, y2) => {
      // Precisa das dimensões do mapa em pixels para calcular length/angle corretamente
      const baseWidth = mapDimensions.value?.width || 1; // Evita divisão por zero
      const baseHeight = mapDimensions.value?.height || 1;

      const dxPercent = x2 - x1;
      const dyPercent = y2 - y1;

      // Converte delta percentual para delta em pixels
      const pixelDx = (dxPercent / 100) * baseWidth;
      const pixelDy = (dyPercent / 100) * baseHeight;

      // Calcula comprimento em pixels e ângulo
      // O comprimento precisa considerar a escala atual do mapa
      const length = Math.sqrt(pixelDx * pixelDx + pixelDy * pixelDy); // * mapScale.value; // O scale é aplicado no CSS/transform
      const angle = Math.atan2(pixelDy, pixelDx) * (180 / Math.PI);

      routeSegments.value.push({
          x: x1,         // Posição X inicial do segmento (%)
          y: y1,         // Posição Y inicial do segmento (%)
          length: length,  // Comprimento do segmento em pixels (sem escala)
          angle: angle,    // Ângulo do segmento em graus
      });
  };

    /** Cria uma rota como uma linha reta simples (fallback) */
```

```
    const createSimpleRouteLine = (startPos, endPos) => {
        routeSegments.value = []; // Limpa segmentos anteriores
        if (!startPos || !endPos) {
            hasRoute.value = false;
            return;
        }
        createRouteSegment(startPos.x, startPos.y, endPos.x, endPos.y);
        hasRoute.value = routeSegments.value.length > 0;
        routingError.value = "Não foi possível calcular a rota detalhada. Mostrando linha reta."; // Informa o fallback
    };


// --- Função Principal de Criação de Rota ---

const calculateRoute = () => {
    routeSegments.value = []; // Limpa rota anterior
    hasRoute.value = false;
    routingError.value = null;
    debugWaypoints.value = []; // Limpa waypoints de debug

    const startPos = userPosition.value;
    const endLocal = selectedLocal.value;

    // Condições para calcular a rota
    if (!startPos || !endLocal || endLocal.andar !== currentFloor.value) {
      return; // Sai se não houver usuário, destino ou se estiverem em andares diferentes
    }

    // 1. Encontrar waypoints mais próximos
    const startWaypoint = findNearestWaypoint(startPos.x, startPos.y, currentFloor.value);
    const endWaypoint = findNearestWaypoint(endLocal.x, endLocal.y, currentFloor.value);

    if (!startWaypoint || !endWaypoint) {
      console.warn("Não foi possível encontrar waypoints próximos para o início ou fim. Usando linha reta.");
      routingError.value = "Waypoints próximos não encontrados.";
      createSimpleRouteLine(startPos, endLocal);
      return;
```

```
    }

    // Adiciona waypoints de início e fim para depuração visual
     debugWaypoints.value.push({ id: 'debug_start_wp', ...startWaypoint });
     debugWaypoints.value.push({ id: 'debug_end_wp', ...endWaypoint });


    // 2. Encontrar caminho mais curto entre waypoints
    let path = null;
    if (startWaypoint.id === endWaypoint.id) {
        // Se o waypoint mais próximo for o mesmo, a rota é direta
        path = [startWaypoint]; // Caminho contém apenas um waypoint
    } else {
        path = findShortestPath(startWaypoint.id, endWaypoint.id);
    }


    if (!path || path.length === 0) {
        console.warn(`Não foi possível encontrar um caminho entre ${startWaypoint.id} e ${endWaypoint.id}. Usando linha reta.`);
        routingError.value = "Não foi possível calcular a rota entre os pontos.";
        createSimpleRouteLine(startPos, endLocal);
        return;
    }

     // Adiciona os waypoints do caminho para depuração visual
     path.forEach(wp => debugWaypoints.value.push({ id: `debug_path_${wp.id}`, ...wp }));


    // 3. Criar segmentos da rota
    // Segmento: Usuário -> Primeiro Waypoint do Caminho
    createRouteSegment(startPos.x, startPos.y, path[0].x, path[0].y);

    // Segmentos: Entre Waypoints do Caminho
    for (let i = 0; i < path.length - 1; i++) {
      createRouteSegment(path[i].x, path[i].y, path[i + 1].x, path[i + 1].y);
    }
```

```
255
256     // Segmento: Último Waypoint do Caminho -> Destino Final (Local)
257     createRouteSegment(path[path.length - 1].x, path[path.length - 1].y, endLocal.x, endLocal.y);
258
259     hasRoute.value = routeSegments.value.length > 0;
260     if(!hasRoute.value) {
261         routingError.value = "Falha ao gerar segmentos da rota.";
262     }
263   };
264
265   // --- Observador ---
266   // Recalcula a rota quando a posição do usuário, o local selecionado ou o andar mudam
267   watch(
268     [userPosition, selectedLocal, currentFloor, mapDimensions], // mapScale afeta apenas a exibição, não o cálculo
269     calculateRoute,
270     { deep: true, immediate: false } // `immediate: false` para esperar dados iniciais
271                                      // `deep: true` para observar mudanças dentro de userPosition/selectedLocal
272   );
273
274   // --- Retorno ---
275   return {
276     routeSegments,  // ref(Array)
277     hasRoute,       // ref(boolean)
278     debugWaypoints, // ref(Array) - Para visualizar no mapa
279     routingError,   // ref(string | null)
280     calculateRoute, // function - Pode ser chamada manualmente se necessário
281   };
282 }
```