Dicionário do Projeto SeniorLife

Este documento descreve a estrutura de pastas e os principais componentes do projeto SeniorLife, explicando o propósito de cada um.

Visão Geral do Projeto

O SeniorLife é um sistema backend projetado para auxiliar no gerenciamento de rotinas e cuidados para idosos (dependentes) por parte de seus cuidadores (acompanhantes). Ele utiliza Node.js com o framework Express para a API REST, MongoDB para armazenar dados de rotinas e atividades, e PostgreSQL para dados de usuários (acompanhantes e dependentes) e suas relações. O sistema também emprega Socket.IO para comunicação em tempo real (notificações, atualizações de rotina) e JSON Web Tokens (JWT) para autenticação segura das APIs.

Estrutura de Pastas e Arquivos

1. Raiz do Projeto (SeniorLife**)**

Contém os arquivos de configuração principal e o ponto de entrada da aplicação.

- index.js
    - **Propósito:** É o coração da aplicação, o ponto de entrada principal que inicializa e coordena todos os outros componentes.
    - **Funcionalidades Detalhadas:**
        - **Carregamento de Variáveis de Ambiente:** Utiliza dotenv.config() para carregar configurações do arquivo .env (como segredos e strings de conexão).
        - **Inicialização do Express:** Cria uma instância do aplicativo Express (app = express()).
        - **Configuração de Middlewares Globais:**
            - morgan('dev'): Para logs detalhados das requisições HTTP no console durante o desenvolvimento.
            - cors(): Habilita o Cross-Origin Resource Sharing, permitindo que o frontend (rodando em um domínio diferente) acesse a API.
            - bodyParser.json() e bodyParser.urlencoded({ extended: true }): Para parsear o corpo das requisições HTTP (JSON e dados de formulário).
        - **Conexão com Bancos de Dados:** Chama os métodos de conexão das instâncias exportadas por instaceConn.js para MongoDB e PostgreSQL.
        - **Servidor HTTP e Socket.IO:** Cria um servidor HTTP (http.createServer(app)) a partir da instância do Express, que é então usado para inicializar o Socket.IO (new Server(server, ...)). Isso permite que o Socket.IO e o Express compartilhem a mesma porta.

        - **Gerenciador de Sockets:** Chama initializeSocketManager(io) (de socketManager.js) para configurar a lógica de autenticação e gerenciamento de salas para as conexões Socket.IO.
        - **Montagem de Rotas:** Define os prefixos base para os diferentes conjuntos de rotas da API e os associa aos respectivos arquivos de roteadores (ex: app.use('/api', routineRoutes)).
        - **Serviços de Agendamento:** Inicia o schedulerService.run() para que as tarefas agendadas (como notificações) comecem a ser executadas.
        - **Tratamento de Erros:**
            - app.all('*', ...): Define um middleware para capturar todas as requisições para rotas não definidas, criando um AppError com status 404.
            - app.use(errorHandler): Registra o middleware global de tratamento de erros (errorMiddleware.js) como o último da pilha, garantindo que ele capture todos os erros passados por next(err).
        - **Inicialização do Servidor:** Faz o servidor HTTP escutar na porta definida (ex: process.env.PORT || 3000).
        - **Exportações:** Exporta app e io para que possam ser usados em outros módulos (ex: para testes ou para os serviços que precisam emitir eventos Socket.IO).
- .env
    - **Propósito:** Armazena variáveis de ambiente que não devem ser codificadas diretamente no código-fonte, como segredos, chaves de API, e configurações específicas do ambiente (desenvolvimento, produção).
    - **Conteúdo Típico:** JWT_SECRET, MONGO_URI, PG_USER, PG_PASSWORD, PG_HOST, PG_DATABASE, PG_PORT, PORT, NODE_ENV.
- socketManager.js
    - **Propósito:** Centraliza a lógica de gerenciamento das conexões Socket.IO.
    - **Funcionalidades Detalhadas:**
        - Recebe a instância io do index.js.
        - Define um middleware de autenticação para o Socket.IO (io.use(...)). Este middleware provavelmente verifica um token JWT enviado pelo cliente durante o handshake da conexão.
        - Ao estabelecer uma nova conexão (io.on('connection', (socket) => { ... })):
            - Associa o usuário autenticado ao socket.

        - Faz o socket do usuário entrar em uma "sala" (room) com seu próprio ID (ex: socket.join(userId)). Isso permite o envio de mensagens direcionadas apenas para aquele usuário.
        - Define manipuladores para eventos de disconnect e outros eventos customizados que o cliente possa emitir.

- package.json (Implícito, mas fundamental)
    - **Propósito:** Arquivo padrão do Node.js que define:
        - **Metadados do Projeto:** Nome, versão, descrição, autor, licença.
        - **Dependências:** Lista todas as bibliotecas de terceiros necessárias para o projeto (dependencies) e para desenvolvimento (devDependencies). O npm install usa este arquivo para baixar e instalar os pacotes.
        - **Scripts:** Comandos customizados que podem ser executados com npm run <script-name> (ex: npm start para iniciar o servidor, npm test para rodar testes, npm run dev para iniciar em modo de desenvolvimento com nodemon).
        - **"type": "module" (Opcional):** Se presente, indica que o projeto usa ES Modules (import/export) por padrão. Se ausente, usa CommonJS (require/module.exports). (No seu caso, estamos padronizando para CommonJS).

2. Api

Esta pasta agrupa todos os componentes que definem a interface da API RESTful da sua aplicação.

- **Middleware/**
    - **Propósito:** Contém funções middleware. Middlewares são funções que têm acesso aos objetos de requisição (req), resposta (res), e à próxima função middleware no ciclo de requisição-resposta da aplicação (next). Eles podem executar código, fazer alterações nos objetos req e res, encerrar o ciclo de requisição-resposta, ou chamar o próximo middleware na pilha.
    - **authAcompanhante.js**
        - autenticarAcompanhante: Um middleware crucial para proteger rotas que exigem que um acompanhante (cuidador) esteja logado.
            1. Extrai o token JWT do cabeçalho Authorization da requisição (formato Bearer <token>).
            2. Se não houver token, retorna um erro 401 (Não Autorizado) usando next(new AppError(...)).
            3. Verifica a validade do token usando jwt.verify() e o JWT_SECRET do .env. Se inválido ou expirado, retorna erro 401.

4. Extrai o ID do acompanhante do payload decodificado do token.
5. Consulta o banco de dados (usando acompanhanteModel.buscarAcompanhantePorId) para garantir que o acompanhante ainda existe.
6. Verifica se o tipo de usuário no token (ou no registro do banco) é 'acompanhante'.
7. Verifica se a senha do acompanhante foi alterada após a data de emissão do token (campo password_changed_at no modelo e iat no token).
8. Se todas as verificações passarem, anexa o objeto do acompanhante (obtido do banco) a req.acompanhante e chama next() para prosseguir para a próxima função na rota.

- **errorMiddleware.js (referenciado como errorHandler no index.js)**
  - handleErrors: Atua como um "apanhador" global de erros. Qualquer erro passado para next(err) em qualquer parte da aplicação (controllers, outros middlewares) será processado aqui.
    1. Define um statusCode (padrão 500) e status (padrão 'error') para o erro, caso não existam.
    2. Em ambiente de **desenvolvimento** (process.env.NODE_ENV === 'development'), envia uma resposta JSON detalhada, incluindo a mensagem do erro, o status, e o stack trace (pilha de chamadas do erro) para facilitar a depuração.
    3. Em ambiente de **produção**, se o erro for operacional (err.isOperational === true, vindo de um new AppError(...)), envia a mensagem de erro específica para o cliente. Caso contrário (erro de programação ou desconhecido), envia uma mensagem genérica ("Algo deu muito errado!") para não expor detalhes internos, mas loga o erro completo no console do servidor.
- **permissionMiddleware.js**
  - checkCaregiverPermission: Usado em rotas onde um acompanhante tenta acessar ou modificar recursos de um dependente específico (ex: rotinas de um idoso).
    1. Obtém o id do acompanhante autenticado (de req.acompanhante.id, populado pelo autenticarAcompanhante).
    2. Obtém o id_idoso (ID do dependente) dos parâmetros da URL (req.params.id_idoso).

3. Consulta a tabela de relacionamento relacao_acompanhante_dependente no banco de dados para verificar se existe uma ligação entre esse acompanhante e esse dependente.
4. Se a relação não existir, retorna um erro 403 (Proibido) usando next(new AppError(...)).
5. Se a relação existir, chama next() para permitir o acesso.

- **validatorMiddleware.js**
  - Este arquivo exporta arrays de middlewares de validação configurados com express-validator. Cada array é uma cadeia de regras de validação seguida por handleValidationErrors.
  - validateCreateActivity, validateUpdateActivity, validateParams: São os arrays exportados. Por exemplo, validateCreateActivity pode conter body('title').notEmpty(), body('type').isIn([...]), etc.
  - handleValidationErrors (função interna usada no final de cada array de validação):
    1. Chama validationResult(req) para coletar quaisquer erros de validação detectados pelas regras anteriores na cadeia.
    2. Se houver erros (!errors.isEmpty()), formata uma mensagem de erro e cria um new AppError(...) com status 400 (Requisição Inválida), passando-o para next().
    3. Se não houver erros, chama next() para prosseguir.

- **Routes/**
  - **Propósito:** Define os endpoints (URLs) da API. Cada arquivo de rota geralmente agrupa endpoints relacionados a um recurso específico (ex: acompanhantes, dependentes, rotinas). Eles usam o express.Router() para criar módulos de rotas.
  - **acompanhanteRoutes.js**
    - Define as rotas para gerenciar os acompanhantes (cuidadores).
    - POST /Cadastro: Rota pública para registrar um novo acompanhante. Chama acompanhanteController.cadastrar.
    - POST /Login: Rota pública para login de acompanhantes. Chama acompanhanteController.login.
    - GET /: (Exemplo) Rota para listar acompanhantes (pode precisar de autenticação e autorização de admin). Chama acompanhanteController.consultar.
    - PATCH /:id: Rota para um acompanhante atualizar seu próprio perfil. Protegida por autenticarAcompanhante. Chama acompanhanteController.editar.

        - DELETE /:id: Rota para um acompanhante excluir seu próprio perfil. Protegida por autenticarAcompanhante.
          Chama acompanhanteController.excluir.
    - **dependenteRoutes.js**
        - Define as rotas para gerenciar os dependentes (idosos), geralmente acessadas por acompanhantes autenticados.
        - POST /Cadastro: Um acompanhante autenticado registra um novo dependente. Protegida por autenticarAcompanhante.
          Chama dependenteController.cadastrar.
        - GET /: Um acompanhante autenticado lista os dependentes a ele associados. Protegida por autenticarAcompanhante.
          Chama dependenteController.consultar.
        - PATCH /:id: Um acompanhante autenticado atualiza um dependente. Protegida
          por autenticarAcompanhante e checkCaregiverPermission (ou lógica similar no controller). Chama dependenteController.editar.
        - DELETE /:id: Um acompanhante autenticado exclui um dependente. Protegida por autenticarAcompanhante e checkCaregiverPermission.
          Chama dependenteController.excluir.
    - **routineRouters.js**
        - Define as rotas para gerenciar as rotinas e atividades dos dependentes.
        - As rotas são prefixadas com o ID do dependente
          (ex: /dependents/:id_idoso/activities).
        - POST /dependents/:id_idoso/activities: Cria uma atividade.
          Usa autenticarAcompanhante, validateParams (para :id_idoso), checkCaregiverPermission, validateCreateActivity (para o corpo da requisição), e finalmente routineController.createActivity.
        - GET /dependents/:id_idoso/activities: Lista atividades.
          Usa autenticarAcompanhante, validateParams, checkCaregiverPermission, routineController.getActivities.
        - PATCH /dependents/:id_idoso/activities/:activityId: Atualiza uma atividade. Usa middlewares similares, incluindo validateUpdateActivity.
        - DELETE /dependents/:id_idoso/activities/:activityId: Deleta uma atividade.

- **Validators/**
    - **Propósito:** Esta pasta parece estar vazia ou não utilizada no momento, já que as definições de validação estão localizadas em validatorMiddleware.js. Se fosse populada, poderia conter schemas de validação mais complexos ou reutilizáveis, talvez para bibliotecas como Joi ou Yup, ou simplesmente uma organização diferente para os validadores do express-validator.

3. Config

Contém arquivos de configuração, especialmente para estabelecer e gerenciar conexões com os bancos de dados.

- conn.js
    - **Propósito:** (Parece ser uma versão mais antiga ou alternativa que usa ES Modules para definir classes de conexão).
    - Define classes MongoDB e PostgresDB que encapsulam a lógica de conexão usando Mongoose (para MongoDB) e Knex (para PostgreSQL).
    - **Observação:** O arquivo instaceConn.js parece ser o que está efetivamente em uso para criar as instâncias de conexão.
- **instaceConn.js**
    - **Propósito:** É o arquivo central para criar e exportar as instâncias únicas (singletons) das conexões com os bancos de dados que serão usadas em toda a aplicação.
    - **Funcionalidades:**
        1. Importa as classes MongoDB e PostgresDB (provavelmente do conn.js ou de uma definição similar interna).
        2. Lê as configurações necessárias do arquivo .env (ex: MONGO_URI para MongoDB; PG_USER, PG_HOST, PG_DATABASE, PG_PASSWORD, PG_PORT para PostgreSQL).
        3. Cria uma instância de MongoDB com a URI e a exporta como mongoConnection. Esta instância terá um método connect().
        4. Cria uma instância de PostgresDB com o objeto de configuração do Knex e a exporta como postgresConnection. Esta instância terá um método getConnection() que retorna a instância configurada do Knex.
        - Essas instâncias exportadas (mongoConnection, postgresConnection) são então importadas e usadas no index.js para iniciar as conexões e nos arquivos de Modelo (Model) para interagir com os bancos.

4. Controller

Os controllers são responsáveis por receber as requisições HTTP (após passarem pelos middlewares), interagir com a camada de Serviço ou Modelo para processar a lógica de negócio e, finalmente, enviar uma resposta HTTP de volta ao cliente.

- acompanhanteController.js
    - **Propósito:** Lida com a lógica das requisições para o recurso "acompanhante" (cuidador).
    - **Funções Principais (convertidas para CommonJS):**
        - cadastrar (ou registerAcompanhante):

1. Recebe nome, email, senha do req.body.
2. Valida os dados de entrada.
3. Hasheia a senha usando bcrypt.hash().
4. Chama acompanhanteModel.criarAcompanhante() para salvar o novo acompanhante no PostgreSQL.
5. Gera um token JWT usando jwt.sign() com o ID do novo acompanhante e seu tipo.
6. Envia uma resposta JSON com status 201 (Criado), o token e os dados do acompanhante (sem a senha).
7. Em caso de erro (ex: email duplicado), usa next(new AppError(...)) ou next(err).

- login (ou loginAcompanhante):
  1. Recebe email, senha do req.body.
  2. Busca o acompanhante pelo email usando acompanhanteModel.buscarAcompanhantePorEmail().
  3. Se não encontrado, retorna erro 401.
  4. Compara a senha fornecida com a senha hasheada do banco usando bcrypt.compare().
  5. Se a senha estiver incorreta, retorna erro 401.
  6. Gera um token JWT.
  7. Envia resposta JSON com status 200, o token e os dados do acompanhante.
- consultar: Chama acompanhanteModel.listarAcompanhantes() e envia a lista (sem senhas).
- editar:
  1. Obtém o ID do acompanhante a ser editado de req.params.id e o ID do acompanhante autenticado de req.acompanhante.id.
  2. Verifica se o acompanhante autenticado está tentando editar seu próprio perfil. Se não, erro 403.
  3. Se req.body.senha for fornecido, hasheia a nova senha.
  4. Chama acompanhanteModel.editarAcompanhante() com o ID e os dados a serem atualizados.
  5. Envia resposta com o acompanhante atualizado.
- excluir: Similar ao editar em termos de autorização, chama acompanhanteModel.excluirAcompanhante().

- dependenteController.js
    - **Propósito:** Lida com a lógica das requisições para o recurso "dependente" (idoso). As ações são geralmente realizadas por um acompanhante autenticado.
    - **Funções Principais (convertidas para CommonJS):**
        - cadastrar:
            1. Obtém o id_acompanhante de req.acompanhante.id.
            2. Recebe os dados do dependente (nome_completo, email, senha, etc.) do req.body.
            3. Hasheia a senha do dependente.
            4. Chama dependenteModel.criarDependente() para salvar o dependente no PostgreSQL.
            5. Chama dependenteModel.criarRelacaoAcompanhanteDependente() para vincular o novo dependente ao acompanhante autenticado.
            6. Envia resposta JSON com status 201.
        - consultar:
            1. Obtém o id_acompanhante de req.acompanhante.id.
            2. Chama dependenteModel.listarDependentesPorAcompanhante() para buscar os dependentes vinculados.
            3. Envia a lista como resposta.
        - editar:
            1. Obtém id_dependente de req.params.id e id_acompanhante de req.acompanhante.id.
            2. **Importante:** Verifica se o acompanhante tem permissão sobre este dependente (usando dependenteModel.verificarRelacaoAcompanhanteDependente() ou através do permissionMiddleware.js na rota).
            3. Se req.body.senha for fornecido, hasheia.
            4. Chama dependenteModel.editarDependente().
            5. Envia resposta.
        - excluir: Similar ao editar em termos de autorização e permissão, chama dependenteModel.excluirDependente().
- **routineController.js**
    - **Propósito:** Lida com a lógica das requisições para rotinas e atividades dos dependentes.

- **Funções Principais (convertidas para CommonJS e usando catchAsync):**
  - createActivity:
    1. Obtém id_idoso de req.params e dados da atividade de req.body.
    2. Chama routineServices.createActivity() (que lida com a lógica de salvar no MongoDB e emitir evento Socket.IO).
    3. Envia resposta JSON com status 201 e a nova atividade.
  - getActivities: Chama routineServices.getActivitiesForDependent().
  - getActivityById: Chama routineServices.getActivityById().
  - updateActivity: Chama routineServices.updateActivity().
  - deleteActivity: Chama routineServices.deleteActivity().
  - deleteAllActivities: Chama routineServices.deleteAllActivities().
  - Todos os métodos usam catchAsync para envolver a lógica assíncrona e automaticamente passar erros para next(), que serão capturados pelo errorMiddleware.js.

5. Databases

Armazena arquivos relacionados aos bancos de dados, como scripts de schema ou dados de exemplo.

- **Demonstração(MongoDB).json**
  - **Propósito:** Um arquivo JSON contendo um exemplo de como os dados de rotina podem ser estruturados no MongoDB, ou um dump de uma coleção para fins de demonstração ou teste.
- **SeniorLife.sql**
  - **Propósito:** Um script SQL contendo as declarações CREATE TABLE e outras definições (chaves primárias, estrangeiras, constraints) para configurar o schema do banco de dados PostgreSQL. Define as tabelas acompanhante, dependente, relacao_acompanhante_dependente, etc.

6. Infos

Destinada a armazenar documentação de planejamento, diagramas e outras informações contextuais sobre o projeto.

- **CasosDeUso.drawio**: Um diagrama (provavelmente feito com o software diagrams.net/draw.io) ilustrando os casos de uso do sistema, mostrando como os atores (usuários) interagem com as funcionalidades.
- **DER.drawio**: Um Diagrama de Entidade-Relacionamento, também provavelmente feito com draw.io, que modela a estrutura do banco de dados relacional (PostgreSQL), mostrando entidades, seus atributos e os relacionamentos entre elas.

7. Model

A camada de Modelo é responsável pelo acesso direto e manipulação dos dados nos bancos de dados. Ela abstrai as consultas ao banco, fornecendo uma interface mais limpa para os Controllers ou Services.

- acompanhanteModel.js
    - **Propósito:** Define funções para interagir com a tabela acompanhante no PostgreSQL. Utiliza a instância do Knex.js (obtida de instaceConn.js) para construir e executar queries SQL.
    - **Funções Principais:**
        - criarAcompanhante({ nome, email, senha }): Insere um novo acompanhante.
        - listarAcompanhantes(): Retorna todos os acompanhantes (selecionando campos não sensíveis).
        - editarAcompanhante(id, fieldsToUpdate): Atualiza um acompanhante pelo ID.
        - excluirAcompanhante(id): Deleta um acompanhante pelo ID.
        - buscarAcompanhantePorEmail(email): Busca um acompanhante pelo email (usado no login, retorna com a senha).
        - buscarAcompanhantePorId(id): Busca um acompanhante pelo ID (usado para autenticação, geralmente retorna sem a senha).
- **activitySchema.js**
    - **Propósito:** Define o schema (estrutura) para os subdocumentos de "atividade" usando Mongoose. Estes não são modelos de nível superior, mas sim a estrutura dos objetos que serão armazenados como parte de um array dentro do documento DependentRoutine.
    - **Campos Típicos:** title (String), type (String, ex: 'medication'), schedule (Date), description (String, opcional), status (String, ex: 'pending'), notified_15_min_before (Boolean), notified_30_min_after (Boolean).
- dependenteModel.js
    - **Propósito:** Define funções para interagir com a tabela dependente e a tabela de junção relacao_acompanhante_dependente no PostgreSQL, usando Knex.js.
    - **Funções Principais:**
        - criarDependente(dependentData): Insere um novo dependente.
        - listarDependentes(): Lista todos os dependentes (pode precisar de refinamento para segurança).
        - buscarDependentePorId(id): Busca um dependente pelo ID.

        - editarDependente(id, fieldsToUpdate): Atualiza um dependente.
        - excluirDependente(id): Deleta um dependente.
        - criarRelacaoAcompanhanteDependente(id_acompanhante, id_dependente): Cria o vínculo na tabela de junção.
        - listarDependentesPorAcompanhante(id_acompanhante): Lista os dependentes associados a um acompanhante específico.
        - verificarRelacaoAcompanhanteDependente(id_acompanhante, id_dependente): Verifica se um vínculo específico existe (útil para permissões).

- **dependentRoutine.js**
    - **Propósito:** Define o modelo Mongoose para a coleção que armazena as rotinas dos dependentes no MongoDB (ex: coleção dependentroutines).
    - **Estrutura:**
        - id_idoso: mongoose.Schema.Types.ObjectId ou String, referenciando o ID do dependente no banco PostgreSQL. É crucial para vincular a rotina ao dependente correto.
        - activities: Um array ([activitySchema]) que armazena múltiplos subdocumentos de atividade, cada um seguindo a estrutura definida em activitySchema.js.
        - Outros campos relevantes para a rotina como um todo, se houver.
    - Exporta o modelo compilado: mongoose.model('DependentRoutine', dependentRoutineSchema).

8. Services

A camada de Serviço contém a lógica de negócio mais elaborada da aplicação. Ela atua como uma intermediária entre os Controllers e os Modelos, orquestrando operações que podem envolver múltiplos modelos ou regras de negócio complexas.

- **routineServices.js**
    - **Propósito:** Encapsula a lógica de negócio relacionada ao gerenciamento de rotinas e atividades dos dependentes.
    - **Funcionalidades Detalhadas:**
        - **Interação com MongoDB:** Usa o modelo DependentRoutine (Mongoose) para:
            - createActivity(id_idoso, activityData): Encontra a rotina do dependente pelo id_idoso (ou cria uma se não existir) e adiciona a nova activityData ao array activities.
            - getActivitiesForDependent(id_idoso): Busca e retorna as atividades para um dependente.

        - updateActivity(id_idoso, activityId, updateData): Encontra a rotina e a atividade específica e aplica as atualizações. Ao atualizar, reseta os flags de notificação (notified_15_min_before, notified_30_min_after) da atividade modificada para false, permitindo que o schedulerService reavalie e envie novas notificações se necessário.
        - Outras operações CRUD para atividades.
    - **Comunicação em Tempo Real:** Após operações de CRUD bem-sucedidas (criar, atualizar, deletar atividade), utiliza a instância io (importada do index.js) para emitir eventos Socket.IO para a "sala" do id_idoso correspondente (ex: io.to(id_idoso.toString()).emit('activity_created_realtime', newActivity)). Isso notifica o frontend do cliente em tempo real sobre as mudanças.

- **scheduleServices.js**
    - **Propósito:** Gerencia tarefas que precisam ser executadas em horários agendados ou em intervalos regulares, como o envio de notificações de lembrete para atividades da rotina.
    - **Funcionalidades Detalhadas:**
        - **Agendamento com node-cron:** Utiliza a biblioteca node-cron para definir "cron jobs". Por exemplo, cron.schedule('* * * * *', async () => { ... }) executa a função fornecida a cada minuto.
        - **Lógica de Notificação:** Dentro da função agendada:

1. Consulta o banco de dados MongoDB (usando o modelo DependentRoutine) para buscar todas as rotinas e suas atividades.
2. Itera sobre as atividades de cada rotina.
3. Verifica o schedule de cada atividade e seu status (ex: 'pending').
4. **Notificação Prévia:** Se a hora atual for X minutos (ex: 15 minutos) antes do schedule da atividade e o flag notified_15_min_before for false, envia uma notificação.
5. **Notificação de Atraso:** Se a hora atual for Y minutos (ex: 30 minutos) após o schedule, a atividade ainda estiver 'pending', e o flag notified_30_min_after for false, envia uma notificação de lembrete/atraso.
6. **Emissão de Eventos Socket.IO:** Para enviar a notificação, usa a instância io para emitir um evento (ex: 'alarm' ou 'routine_notification') para a "sala" do id_idoso correspondente, enviando os detalhes da atividade.
7. **Atualização de Flags:** Após enviar uma notificação, atualiza o respectivo flag (notified_15_min_before ou notified_30_min_after) para true no banco de dados para evitar o envio repetido da mesma notificação.

9. Utils

Contém classes e funções utilitárias que podem ser reutilizadas em diferentes partes do projeto.

- **appError.js**
    - **Propósito:** Define uma classe AppError personalizada, que estende a classe Error nativa do JavaScript.
    - **Funcionalidades:**
        - O construtor recebe uma message (mensagem de erro) e um statusCode (código de status HTTP).
        - Define a propriedade status com base no statusCode (ex: 'fail' para códigos 4xx, 'error' para 5xx).
        - Define a propriedade isOperational = true. Isso é usado pelo errorMiddleware.js para identificar erros que são "esperados" ou "operacionais" (como entrada inválida do usuário, recurso não encontrado) e podem ter suas mensagens exibidas ao cliente com segurança, em oposição a erros de programação inesperados.
        - Error.captureStackTrace(this, this.constructor): Garante que o stack trace do erro seja capturado corretamente, omitindo a própria classe AppError da pilha de chamadas.
        - Permite criar instâncias de erro de forma padronizada (ex: next(new AppError('Usuário não encontrado', 404))) que são então processadas pelo errorMiddleware.js.