**Material Didático sobre a etapa inicial (genérica) do backend – padrão MVC**

**Índice**

1. [Introdução](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#introdu%C3%A7%C3%A3o)
2. [Estrutura de Pastas](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#estrutura-de-pastas)
3. [Componentes Principais](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#componentes-principais)
   * [Arquivos de Entrada](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#arquivos-de-entrada)
   * [Configurações](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#configura%C3%A7%C3%B5es)
   * [Middlewares](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#middlewares)
   * [Utilitários](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#utilit%C3%A1rios)
   * [Rotas e Controladores](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#rotas-e-controladores)
4. [Fluxos de Execução](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#fluxos-de-execu%C3%A7%C3%A3o)
   * [Inicialização do Servidor](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#inicializa%C3%A7%C3%A3o-do-servidor)
   * [Autenticação](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#autentica%C3%A7%C3%A3o)
   * [Proteção de Rotas](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#prote%C3%A7%C3%A3o-de-rotas)
5. [Conceitos Importantes](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#conceitos-importantes)
6. [Próximos Passos](https://claude.ai/chat/485789d4-7e8b-4592-9796-aa936cb7c606#pr%C3%B3ximos-passos)

**Introdução**

Este material explica a primeira etapa do desenvolvimento do backend, que estabelece a estrutura básica e o sistema de autenticação.

**O que você aprenderá:**

* Como está organizado o backend
* O que cada arquivo faz
* Como os componentes se conectam
* Como funciona o fluxo de autenticação
* Conceitos importantes de desenvolvimento web

**Estrutura de Pastas**

A organização das pastas segue um padrão modular, onde cada tipo de funcionalidade tem seu próprio diretório:

backend/

├── config/ # Configurações da aplicação

│ ├── database.js # Conexão com o banco de dados

│ └── jwt.js # Configuração de tokens JWT

├── controllers/ # Controladores (lógica de negócio)

│ └── authController.js # Controlador de autenticação

├── middleware/ # Middlewares (interceptadores)

│ ├── auth.js # Middleware de autenticação

│ ├── config.js # Configuração de middlewares

│ └── cors.js # Configuração de CORS

├── routes/ # Rotas da API

│ └── authRoutes.js # Rotas de autenticação

├── utils/ # Utilitários

│ ├── bcrypt.js # Funções para hash de senhas

│ ├── errorHandler.js # Tratamento de erros

│ └── health.js # Monitoramento de saúde

├── app.js # Configuração da aplicação

└── server.js # Ponto de entrada principal

**📝 Dica para Iniciantes:** Esta estrutura usa o padrão MVC (Model-View-Controller), que separa as responsabilidades do código. No backend, temos principalmente os Models (banco de dados) e Controllers (lógica de negócio).

**Componentes Principais**

**Arquivos de Entrada**

**server.js**

Este é o ponto de entrada principal da aplicação, o primeiro arquivo a ser executado.

**O que ele faz:**

* Importa a aplicação configurada do app.js
* Configura o monitoramento de saúde
* Inicia o servidor na porta 5000
* Trata sinais de encerramento do processo

**Trecho de código importante:**

// Inicialização do servidor

app.listen(PORT, () => {

console.log(`Servidor rodando na porta ${PORT}`);

});

**app.js**

Configura a aplicação Express e conecta todos os componentes.

**O que ele faz:**

* Importa dependências e configurações
* Aplica middlewares globais (CORS, rate limit)
* Configura e monta as rotas
* Aplica tratamento de erros

**Trecho de código importante:**

// Montagem das rotas

app.use('/api/auth', authRoutes);

app.use('/api/professor', professorRoutes);

app.use('/api/aluno', alunoRoutes);

app.use('/api/admin', adminRoutes);

**Configurações**

**config/database.js**

Estabelece a conexão com o banco de dados PostgreSQL.

**O que ele faz:**

* Cria um pool de conexões com o banco
* Configura o timezone para 'America/Sao\_Paulo'
* Verifica se a conexão está funcionando
* Exporta o pool para uso em outros arquivos

**Por que é importante:** Centraliza a configuração do banco de dados, permitindo que toda a aplicação compartilhe a mesma conexão.

**config/jwt.js**

Gerencia tokens JWT (JSON Web Tokens) usados para autenticação.

**O que ele faz:**

* Gera tokens com informações do usuário
* Define tempos de expiração diferentes por tipo de usuário
  + Administradores e professores: 24 horas
  + Alunos: 4 horas
* Verifica a validade de tokens

**Por que usamos JWT:** Os tokens JWT permitem autenticação stateless (sem estado), onde o servidor não precisa armazenar sessões. Toda informação necessária está no próprio token.

**Middlewares**

**middleware/auth.js**

Controla o acesso às rotas protegidas.

**O que ele faz:**

* Verifica se o token JWT é válido (authMiddleware)
* Verifica o tipo de usuário para autorização:
  + authAdmin: apenas administradores
  + authProfessor: apenas professores
  + authAluno: apenas alunos
  + authProfessorOrAdmin: professores ou administradores

**Como funciona na prática:**

// Exemplo de como proteger uma rota

router.get('/questionarios',

authMiddleware, // Primeiro verifica se o usuário está autenticado

authProfessor, // Depois verifica se é um professor

controller.listarQuestionarios // Se passar, executa o controlador

);

**middleware/config.js**

Configura middlewares básicos da aplicação.

**O que ele faz:**

* Configura parsing de JSON e formulários
* Aplica proteções de segurança com Helmet
* Configura logging com Morgan
* Disponibiliza a conexão do banco em todas as requisições

**middleware/cors.js**

Gerencia o Cross-Origin Resource Sharing, permitindo ou bloqueando requisições de diferentes origens.

**O que ele faz:**

* Define quais origens podem acessar a API
* Em desenvolvimento: permite localhost:3000
* Em produção: permite apenas origens específicas
* Registra quando bloqueia acessos não autorizados

**Por que o CORS é importante:** É uma medida de segurança que impede que sites maliciosos façam requisições à sua API sem permissão.

**Utilitários**

**utils/bcrypt.js**

Lida com a segurança das senhas.

**O que ele faz:**

* Gera hashes seguros para senhas
* Compara senhas fornecidas com hashes armazenados
* Trata erros relacionados a operações de hash

**Por que não armazenamos senhas em texto puro:** Hashes são unidirecionais, o que significa que mesmo se o banco de dados for comprometido, as senhas reais não podem ser descobertas.

**utils/errorHandler.js**

Gerencia erros na aplicação.

**O que ele faz:**

* Configura captura de erros não tratados
* Fornece um middleware para tratamento uniforme de erros
* Formata as respostas de erro de forma consistente

**utils/health.js**

Monitora a saúde da aplicação.

**O que ele faz:**

* Cria um endpoint /health para verificação de status
* Verifica a conexão com o banco de dados
* Retorna métricas como uso de memória e tempo de atividade

**Por que é importante:** Permite que sistemas de monitoramento verifiquem se a aplicação está funcionando corretamente.

**Rotas e Controladores**

**routes/authRoutes.js**

Define as rotas relacionadas à autenticação.

**Rotas implementadas:**

* POST /api/auth/login: Autenticação de usuários
* POST /api/auth/register: Registro de novos usuários
* GET /api/auth/verify: Verificação da validade de tokens

**Como as rotas são definidas:**

router.post('/login', authController.login);

router.post('/register', authController.register);

router.get('/verify', authController.verify);

**controllers/authController.js**

Contém a lógica de negócio para autenticação.

**O que ele faz:**

* login: Verifica credenciais e gera token
* register: Cria novos usuários com senha segura
* verify: Verifica se um token é válido

**Trecho de código importante:**

// Verificação de senha com bcrypt

const passwordMatches = await bcrypt.comparePassword(senha, user.senha\_hash);

if (!passwordMatches) {

return res.status(401).json({ error: 'Senha incorreta' });

}

**Fluxos de Execução**

**Inicialização do Servidor**

Quando você inicia a aplicação com node server.js, ocorre o seguinte fluxo:

1. server.js é carregado e executado
2. app.js é importado, configurando a aplicação
3. Middlewares globais são aplicados
4. Rotas são montadas
5. O servidor é iniciado na porta 5000
6. O endpoint de health check é configurado

**Diagrama de Inicialização:**

server.js → app.js → middlewares → rotas → servidor ativo

**Autenticação**

Quando um usuário tenta fazer login:

1. Cliente envia requisição para POST /api/auth/login
2. A requisição chega em routes/authRoutes.js
3. A rota chama authController.login
4. O controlador:
   * Busca o usuário no banco pelo email
   * Verifica a senha usando bcrypt.comparePassword
   * Gera um token JWT usando jwt.generateToken
   * Retorna o token para o cliente

**Diagrama de Autenticação:**

Cliente → authRoutes.js → authController.login → database.js → bcrypt.js → jwt.js → Cliente

**Proteção de Rotas**

Quando um usuário tenta acessar uma rota protegida:

1. Cliente envia requisição com token no header Authorization
2. Middlewares de autenticação são executados:
   * authMiddleware verifica se o token é válido
   * Middlewares específicos (authProfessor, authAdmin, etc.) verificam o tipo de usuário
3. Se autorizado, a requisição segue para o controlador
4. Se não autorizado, retorna erro 401 ou 403

**Diagrama de Acesso a Rota Protegida:**

Cliente → auth.js → (se autorizado) → controlador → resposta

→ (se não autorizado) → erro

**Conceitos Importantes**

**Middleware**

Funções que têm acesso à requisição (req), à resposta (res) e à próxima função no ciclo requisição-resposta. Podem:

* Executar código
* Modificar requisição/resposta
* Encerrar o ciclo
* Chamar a próxima função

**Autenticação vs Autorização**

* **Autenticação**: Verificar se o usuário é quem diz ser (login)
* **Autorização**: Verificar se o usuário tem permissão para acessar um recurso (auth.js)

**Rate Limiting**

Limita o número de requisições que um cliente pode fazer em um período de tempo, protegendo contra abusos e ataques.

**CORS (Cross-Origin Resource Sharing)**

Mecanismo de segurança do navegador que impede que sites façam requisições para APIs em domínios diferentes sem permissão explícita.

**JWT (JSON Web Token)**

Método para transmitir informações com segurança entre partes como um objeto JSON. Útil para autenticação e troca de informações.

**Bcrypt**

Algoritmo de hash projetado especificamente para senhas, incluindo "salt" para proteção contra ataques de tabela arco-íris.

**Próximos Passos**

Com a estrutura básica e o sistema de autenticação implementados, os próximos passos seriam:

1. **Implementar rotas específicas** para professores, alunos e administradores
2. **Criar controladores** para as entidades principais (questionários, questões, etc.)
3. **Implementar validadores** para cada entidade
4. **Desenvolver serviços** para lógica de negócio complexa
5. **Integrar com o frontend:** Na próxima etapa iremos construir backend (arquivos de validators, routes, controllers e services) e frontend (pages, css, e js) por etapas para cada fluxo.

**Resumo**

O backend desenvolvido até agora estabelece uma base sólida para qualquer aplicação. Com uma estrutura modular, segurança incorporada e fluxos bem definidos, está pronto para receber as funcionalidades específicas do domínio da aplicação.

A separação clara de responsabilidades entre arquivos e componentes facilita a manutenção e extensão do sistema, seguindo boas práticas de desenvolvimento de software.

**Relação entre auth.js e authController.js**

Sim, o middleware/auth.js continua sendo muito útil! Os dois arquivos têm funções complementares no sistema de autenticação:

Funções Diferentes

* authController.js: Lida com operações de autenticação (login, registro, verificação)
* auth.js: Protege rotas após a autenticação (middleware de autorização)

Fluxo de Autenticação Completo

1. Inicialização:
   * app.js importa as rotas, incluindo authRoutes.js
   * authRoutes.js importa o authController.js
2. Login (não protegido):
   * Requisição chega em /api/auth/login
   * Não passa por middleware de autenticação
   * routes/authRoutes.js chama authController.login
   * authController.js verifica credenciais no banco
   * utils/bcrypt.js compara a senha
   * config/jwt.js gera o token
   * Token retornado ao usuário
3. Acesso a Rotas Protegidas:
   * Requisição chega em /api/professor/questionarios (exemplo)
   * Middleware de auth.js é executado primeiro
   * authMiddleware verifica o token JWT
   * authProfessor verifica se o usuário é professor
   * Se autorizado, a requisição segue para o controlador da rota

Diagrama do Fluxo

Cliente → app.js → routes → authRoutes.js → authController.js → bcrypt.js → jwt.js

↑

Cliente → app.js → auth.js (middleware) → routes → controladores →|

Por que os Dois São Necessários

1. Separação de Responsabilidades:
   * authController: Lógica de negócio da autenticação
   * auth.js: Controle de acesso às rotas
2. Diferentes Momentos de Uso:
   * authController: Usado apenas nas rotas de autenticação
   * auth.js: Usado em todas as rotas protegidas
3. Reutilização:
   * O middleware auth.js pode ser aplicado a qualquer rota
   * Não precisamos duplicar a lógica de verificação de token

Esta separação segue o padrão MVC (Model-View-Controller) e o princípio de responsabilidade única, tornando o código mais organizado e manutenível.

**Esclarecendo a Autenticação e Autorização**

Vou esclarecer os conceitos de rotas protegidas e o funcionamento dos middlewares de autenticação:

**Conceitos Importantes**

1. **Rota não protegida**: Rota que pode ser acessada sem autenticação prévia
   * Exemplo: login, registro, página inicial pública
   * Não passa pelo middleware de verificação de token
2. **Rota protegida**: Rota que requer autenticação prévia
   * Exemplo: dashboard do professor, criação de questionários
   * Passa pelo middleware auth.js que verifica o token

**Por que Login é "Não Protegido"**

O login é "não protegido" porque seria ilógico exigir que um usuário já esteja autenticado para poder se autenticar. É como dizer "você precisa já estar dentro da casa para poder entrar na casa".

javascript

*// Rota não protegida - qualquer pessoa pode acessar*

router.post('/login', authController.login);

*// Rota protegida - apenas usuários autenticados podem acessar*

router.get('/professor/questionarios', authMiddleware, authProfessor, questionarioController.listar);

**Todos os Middlewares de Autorização**

Na minha explicação anterior, usei apenas authProfessor como exemplo, mas temos vários:

1. **authMiddleware**: Verifica se existe um token válido (autenticação)
2. **authAdmin**: Verifica se o usuário é administrador
3. **authProfessor**: Verifica se o usuário é professor
4. **authAluno**: Verifica se o usuário é aluno
5. **authProfessorOrAdmin**: Verifica se o usuário é professor OU administrador

**Exemplo de Uso Completo**

javascript

*// Rota para criar questionário - somente professores podem acessar*

router.post('/questionarios',

authMiddleware, *// Primeiro verifica se há um token válido*

authProfessor, *// Depois verifica se o usuário é professor*

questionarioController.criar

);

*// Rota para listar todos os usuários - somente admin pode acessar*

router.get('/usuarios',

authMiddleware, *// Primeiro verifica se há um token válido*

authAdmin, *// Depois verifica se o usuário é administrador*

usuarioController.listar

);

**Fluxo de Autenticação por Arquivos**

1. **Sem autenticação ainda**:
   * Usuário acessa /api/auth/login
   * routes/authRoutes.js → authController.login
   * Sistema verifica credenciais e gera token
2. **Com token obtido**:
   * Usuário acessa rota protegida com token no header
   * Requisição passa por middleware/auth.js
   * authMiddleware verifica validade do token
   * authAdmin ou authProfessor ou authAluno verifica tipo de usuário
   * Se autorizado, acessa o controlador da rota

Esta é a separação clássica entre **autenticação** (verificar identidade) e **autorização** (verificar permissões), seguindo as melhores práticas de segurança.

=================================================================

O PROJETO BÁSICO DE BACKEND

**Reorganização do código em módulos separados**

Organizei o código do servidor em arquivos separados, cada um com uma responsabilidade específica:

**1. server.js (simplificado)**

Um arquivo limpo e simplificado que apenas:

* Importa as dependências e módulos necessários
* Configura os middlewares básicos
* Define as rotas
* Inicia o servidor

**2. config/database.js**

Responsável pela configuração do banco de dados:

* Cria o pool de conexões PostgreSQL
* Verifica a conexão com o banco
* Configura o timezone para São Paulo

**3. middleware/config.js**

Configura todos os middlewares básicos da aplicação:

* Parsing de JSON e formulários
* Segurança com Helmet
* Logging com Morgan
* Configuração de CORS
* Rate limiting para proteção contra abusos

**4. middleware/auth.js**

Implementa os middlewares de autenticação e autorização:

* Verificação de tokens JWT
* Verificação de tipos de usuário (admin, professor, aluno)
* Controle de acesso a rotas protegidas

**5. config/jwt.js**

Lida com a geração de tokens JWT:

* Gera tokens com tempo de expiração diferente por tipo de usuário
* Inclui informações relevantes no payload do token

**6. utils/errorHandler.js**

Configuração para tratamento de erros:

* Captura erros não tratados
* Captura promessas rejeitadas
* Middleware para tratamento uniforme de erros

**7. routes/authRoutes.js**

Implementa as rotas de autenticação:

* Rota de login
* Rota para verificação de token

Esta estrutura segue o princípio de responsabilidade única, tornando o código mais organizado, modular e fácil de manter. Cada arquivo tem um propósito claro e didático, cumprindo a regra de ouro do projeto de ser simples, básico e funcional.

**Criação do app.js**

Criei o arquivo app.js que servirá como ponto de entrada da aplicação. Este arquivo:

1. **Importa as dependências necessárias**: Express, dotenv, e outras configurações.
2. **Importa os módulos de configuração**:
   * database.js para conexão com o banco de dados
   * middlewareConfig.js para configuração de middlewares
   * errorHandler.js para tratamento de erros
3. **Importa os arquivos de rotas**:
   * authRoutes.js para autenticação
   * professorRoutes.js para rotas específicas de professores
   * alunoRoutes.js para rotas específicas de alunos
   * adminRoutes.js para rotas específicas de administradores
4. **Define uma rota básica** para testar se o servidor está funcionando.
5. **Monta as rotas** com os prefixos apropriados:
   * /api/auth para autenticação
   * /api/professor para professores
   * /api/aluno para alunos
   * /api/admin para administradores
6. **Aplica o middleware de tratamento de erros** que captura e trata erros nas rotas.
7. **Inicia o servidor na porta 5000** conforme solicitado.

Este arquivo serve como o ponto central da aplicação, conectando todos os diferentes componentes (rotas, middlewares, configurações) de maneira organizada e modular.

**Observações importantes:**

* Os arquivos de rotas específicos (professorRoutes.js, alunoRoutes.js, adminRoutes.js) precisarão ser criados para que a aplicação funcione corretamente.
* Este arquivo segue o princípio de "preocupação única", onde ele apenas inicializa a aplicação e conecta os componentes, delegando as responsabilidades específicas aos módulos correspondentes.
* A aplicação está configurada para escutar na porta 5000, conforme solicitado.

**Conexão entre app.js e server.js com Monitoramento**

Reorganizei a estrutura do projeto para conectar o app.js com o server.js, criando também o módulo de monitoramento de saúde. Todos os arquivos agora possuem console.log no início e fim.

**1. app.js (atualizado)**

* Removida a inicialização do servidor (app.listen)
* Adicionados console.log no início e fim
* Configuração completa da aplicação mantida
* Exporta a aplicação Express configurada

**2. server.js (atualizado)**

* Importa a aplicação configurada do app.js
* Inicializa o servidor na porta 5000
* Configura o módulo de health check
* Adiciona tratamento de sinais para encerramento gracioso
* Console.log no início e fim

**3. utils/health.js (novo)**

* Implementa endpoint /health para monitoramento
* Verifica a conexão com o banco de dados
* Retorna métricas úteis como memória e uptime
* Console.log no início e fim

**4. Todos os outros arquivos**

* Adicionados console.log no início e fim

**Fluxo de inicialização**

1. Quando executamos node server.js:
   * server.js é carregado
   * app.js é importado e configurado
   * O módulo de saúde (health.js) é configurado
   * O servidor é iniciado na porta 5000
   * O endpoint /health fica disponível para monitoramento

**Benefícios desta estrutura**

1. **Separação de preocupações**:
   * app.js cuida da configuração da aplicação
   * server.js cuida da inicialização do servidor
   * health.js cuida do monitoramento
2. **Testabilidade**:
   * É possível testar app.js sem iniciar o servidor
   * É possível simular condições de falha no health check
3. **Monitoramento**:
   * O endpoint /health permite verificar o estado da aplicação
   * Facilita a integração com ferramentas de monitoramento
4. **Logs de inicialização**:
   * Os console.log permitem visualizar a ordem de carregamento dos módulos
   * Facilita a depuração em caso de problemas de inicialização

O código está pronto para ser expandido com as funcionalidades específicas do sistema de atividades avaliativas.

**Rotas Implementadas até o Momento**

Você está correto. Até agora, implementamos apenas as rotas básicas de autenticação e alguns endpoints essenciais:

**1. Rotas de Autenticação (/api/auth)**

* POST /api/auth/login - Para fazer login (admin, professor ou aluno)
* POST /api/auth/register - Para registrar novos usuários (admin ou professor)
* GET /api/auth/verify - Para verificar a validade de um token JWT

**2. Rota de Monitoramento**

* GET /health - Endpoint para verificar a saúde da aplicação (status do DB, memória, etc.)

**3. Rota Básica**

* GET / - Rota básica para verificar se a API está funcionando

**Rotas Referenciadas (Mas Não Implementadas)**

No arquivo app.js, montamos as seguintes rotas, mas ainda não implementamos os arquivos correspondentes:

javascript

app.use('/api/professor', professorRoutes);

app.use('/api/aluno', alunoRoutes);

app.use('/api/admin', adminRoutes);

Estas rotas foram apenas estruturadas no arquivo principal, mas ainda precisaríamos criar:

* routes/professorRoutes.js
* routes/alunoRoutes.js
* routes/adminRoutes.js

E seus respectivos controladores para implementar a funcionalidade completa.

O foco até agora foi estabelecer a estrutura básica e o sistema de autenticação, que são os fundamentos sobre os quais as demais funcionalidades serão construídas.

**Análise da Estrutura Básica do Backend**

Sim, podemos considerar que concluímos a estrutura básica do backend. Todos os componentes essenciais estão implementados e conectados corretamente.

**Componentes Implementados**

1. **Arquivos Core**:
   * server.js - Inicializa o servidor e configura o health check
   * app.js - Configura a aplicação e conecta os componentes
2. **Configurações**:
   * config/database.js - Conexão com PostgreSQL
   * config/jwt.js - Geração e verificação de tokens JWT
3. **Middlewares**:
   * middleware/config.js - Middlewares básicos (parsing, segurança)
   * middleware/auth.js - Autenticação e controle de acesso
   * middleware/cors.js - Proteção CORS configurável
4. **Utilitários**:
   * utils/bcrypt.js - Hash e verificação de senhas
   * utils/errorHandler.js - Tratamento centralizado de erros
   * utils/health.js - Monitoramento de saúde da aplicação
5. **Autenticação**:
   * controllers/authController.js - Lógica de login/registro
   * routes/authRoutes.js - Rotas de autenticação

**Conexões Corretas**

Todas as conexões estão corretamente estabelecidas:

* server.js importa e inicializa app.js
* app.js aplica middlewares e configura rotas
* As rotas usam os controladores
* Os controladores usam os utilitários (bcrypt) e configurações (jwt)
* O middleware de autenticação é aplicável às rotas protegidas

**Estrutura Sólida e Extensível**

Esta estrutura básica:

1. **Segue boas práticas**: Separação de responsabilidades, modularidade
2. **É facilmente extensível**: Novos controladores/rotas podem ser adicionados
3. **Tem segurança incorporada**: Bcrypt, JWT, CORS, rate limiting
4. **É didática**: Código organizado e comentado
5. **É monitorável**: Endpoint de health check
6. **É manutenível**: Componentes isolados e bem definidos

O próximo passo seria adicionar:

* Rotas específicas (professor, aluno, admin)
* Controllers para entidades de negócio (disciplinas, questionários, etc.)
* Validadores de entrada
* Serviços para lógica de negócio

Mas a base essencial para tudo isso já está implementada e conectada corretamente, seguindo as regras de ouro do projeto: simplicidade, funcionalidade e didatismo.

**Implementação do bcrypt e Reorganização da Autenticação**

Adicionei o bcrypt ao projeto e reorganizei a autenticação em uma arquitetura mais robusta e modular. Segue o resumo das implementações:

**1. utils/bcrypt.js (NOVO)**

* Implementei funções para hash e verificação de senhas usando bcrypt
* Configurei o número de rounds de salt para 10 (padrão seguro)
* Adicionei tratamento de erros e logging

**Importante:** A biblioteca bcrypt original de fato tem um método chamado compare(), não comparePassword(). No entanto, no nosso projeto, nós criamos intencionalmente um wrapper personalizado no arquivo utils/bcrypt.js.

**2. controllers/authController.js (NOVO)**

* Criei um controlador dedicado para autenticação
* Implementei três funções principais:
  + **register**: Para criar novos usuários (admin e professores)
  + **login**: Para autenticar usuários
  + **verify**: Para verificar a validade de tokens

**3. routes/authRoutes.js (ATUALIZADO)**

* Migrei a lógica de autenticação para o controlador
* Mantive as mesmas rotas, mas simplificadas
* Adicionei uma nova rota para registro de usuários

**4. config/jwt.js (ATUALIZADO)**

* Adicionei uma função verify para verificar tokens JWT
* Isso centraliza a lógica de validação de tokens

**Fluxo de Autenticação**

O fluxo de autenticação agora funciona da seguinte forma:

1. **Registro (novo)**:
   * O usuário envia nome, email, senha e tipo
   * A senha é hash'eada com bcrypt
   * O usuário é inserido no banco de dados
   * Um token JWT é gerado e retornado
2. **Login**:
   * O usuário envia email, senha e tipo
   * O sistema busca o usuário no banco de dados
   * A senha é verificada com bcrypt
   * Um token JWT é gerado e retornado
3. **Verificação**:
   * O token é extraído do cabeçalho Authorization
   * O token é verificado usando jwt.verify
   * A validade do token é retornada

**Benefícios desta Implementação**

1. **Segurança aprimorada**:
   * Senhas armazenadas com hash bcrypt
   * Proteção contra ataques de força bruta
   * Validação robusta de tokens
2. **Código organizado**:
   * Separação de responsabilidades
   * Controladores para lógica de negócios
   * Rotas simplificadas
3. **Manutenibilidade**:
   * Funções reutilizáveis
   * Código modular e didático
   * Tratamento de erros consistente

Esta implementação fornece uma base sólida para o sistema de autenticação do aplicativo de atividades avaliativas, seguindo boas práticas de segurança e organização de código.

**Implementação do Middleware CORS**

Criei o arquivo middleware/cors.js com a configuração CORS mais detalhada e modifiquei o app.js para usar esse middleware. Também removi a configuração CORS simples que estava no middleware/config.js.

**Arquivo middleware/cors.js**

Este arquivo implementa uma configuração CORS mais avançada e segura:

1. **Lista de origens permitidas**:
   * Em desenvolvimento: localhost:3000 e 127.0.0.1:3000
   * Em produção: a URL definida na variável de ambiente FRONTEND\_URL
2. **Verificação dinâmica de origem**:
   * Verifica se a origem da requisição está na lista de origens permitidas
   * Permite requisições sem origem (como as feitas pelo Postman ou curl)
   * Registra avisos quando bloqueia requisições de origens não permitidas
3. **Suporte a credenciais**:
   * Habilita o envio de cookies e cabeçalhos de autenticação entre origens

**Modificações em app.js**

1. **Importação do middleware CORS**:
   * Adicionei a importação do novo middleware corsMiddleware
2. **Aplicação do middleware**:
   * Apliquei o middleware CORS após o rate limiter e antes dos outros middlewares
   * Isso garante que a verificação CORS seja feita em todas as requisições

**Modificações em middleware/config.js**

1. **Remoção da configuração CORS**:
   * Removi a configuração CORS simples que estava neste arquivo
   * Isso evita configurações duplicadas e conflitantes

**Benefícios desta abordagem**

1. **Segurança aprimorada**:
   * Lista explícita de origens permitidas
   * Registro de tentativas de acesso bloqueadas
   * Melhor controle sobre as políticas de compartilhamento entre origens
2. **Flexibilidade**:
   * Configurações diferentes para ambientes de desenvolvimento e produção
   * Suporte a credenciais para autenticação
3. **Maior organização**:
   * Middleware CORS isolado em seu próprio arquivo
   * Responsabilidade única para cada componente
4. **Código didático**:
   * Comentários explicativos
   * Estrutura clara e simples

Esta configuração CORS é especialmente importante para um sistema educacional, pois garante que apenas o frontend autorizado possa acessar as APIs do backend, protegendo os dados sensíveis dos alunos e professores.

**Inclusão do Rate Limit Global no app.js**

Adicionei o rate limit global ao arquivo app.js e removi a versão que estava aplicada apenas às rotas /api/ no middleware/config.js. Essa mudança garante que todas as rotas da aplicação estejam protegidas contra abuso.

**Alterações realizadas:**

1. **No app.js**:
   * Importei o módulo express-rate-limit
   * Criei um limitador que permite no máximo 100 requisições por IP a cada 15 minutos
   * Apliquei o limitador globalmente à aplicação com app.use(limiter)
   * Esta configuração é aplicada antes de outros middlewares para garantir que todas as requisições sejam limitadas
2. **No middleware/config.js**:
   * Removi o rate limiter que estava aplicado apenas às rotas /api/
   * Isso evita redundância, já que agora temos um limitador global

**Benefícios desta abordagem:**

1. **Segurança abrangente**: Todas as rotas da aplicação estão protegidas contra ataques de força bruta ou scraping
2. **Clareza no código**: A configuração do rate limit está visível no ponto de entrada da aplicação, tornando mais fácil para desenvolvedores entenderem as proteções aplicadas
3. **Configuração centralizada**: Ao colocar o rate limit no app.js, centralizamos as configurações importantes de segurança num único lugar
4. **Aplicação consistente**: O mesmo limite é aplicado a todas as rotas, simplificando o entendimento e manutenção do código

Esta configuração é particularmente importante para sistemas educacionais como este, onde precisamos garantir que o servidor não será sobrecarregado durante períodos de pico, como quando muitos alunos estão realizando provas simultaneamente.

=========================================================

**Código principal do cors (simplificado e comentado)**

**CORS eh um codigo aberto do Express**

// Função para configurar as opções de CORS

function configureCors(options) {

const defaults = {

origin: '\*',

methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',

preflightContinue: false,

optionsSuccessStatus: 204

};

// Merge das opções padrão com as opções do usuário

const corsOptions = Object.assign({}, defaults, options);

// Função middleware que será usada pelo Express

return function corsMiddleware(req, res, next) {

const requestOrigin = req.headers.origin;

// Se não houver origin, não é uma requisição CORS → segue o fluxo

if (!requestOrigin) {

return next();

}

// Se origin for função (para lógica dinâmica), executa

if (typeof corsOptions.origin === 'function') {

corsOptions.origin(requestOrigin, function (err, allow) {

if (err || !allow) {

return next(err || new Error('Not allowed by CORS'));

}

setCorsHeaders(res, requestOrigin, corsOptions);

handlePreflight(req, res, next, corsOptions);

});

} else {

const allowed = checkOrigin(requestOrigin, corsOptions.origin);

if (allowed) {

setCorsHeaders(res, requestOrigin, corsOptions);

}

handlePreflight(req, res, next, corsOptions);

}

};

}

// Função para lidar com requisições OPTIONS (preflight)

function handlePreflight(req, res, next, options) {

if (req.method === 'OPTIONS') {

if (options.preflightContinue) {

return next();

}

res.statusCode = options.optionsSuccessStatus;

res.end();

} else {

next();

}

}

// Verifica se a origem está permitida

function checkOrigin(origin, allowedOrigins) {

if (allowedOrigins === '\*') return true;

if (Array.isArray(allowedOrigins)) {

return allowedOrigins.includes(origin);

}

return allowedOrigins === origin;

}

// Define os cabeçalhos CORS na resposta

function setCorsHeaders(res, origin, options) {

res.setHeader('Access-Control-Allow-Origin', origin);

res.setHeader('Access-Control-Allow-Methods', options.methods);

if (options.credentials) {

res.setHeader('Access-Control-Allow-Credentials', 'true');

}

if (options.allowedHeaders) {

res.setHeader('Access-Control-Allow-Headers', options.allowedHeaders);

}

if (options.exposedHeaders) {

res.setHeader('Access-Control-Expose-Headers', options.exposedHeaders);

}

}

// Exporta a função principal

module.exports = configureCors;

**📌 Explicação geral**

* configureCors(options) → configura o middleware com base nas opções fornecidas
* A função retornada é o middleware que o Express usa com app.use(cors(...))
* O middleware:
  + Verifica se existe Origin no cabeçalho
  + Valida se a origem está autorizada
  + Define os cabeçalhos de CORS
  + Lida com requisições OPTIONS (preflight)

**✅ Exemplo de uso no Express**

const express = require('express');

const cors = require('cors');

const app = express();

const corsOptions = {

origin: ['https://meusite.com'],

credentials: true

};

app.use(cors(corsOptions));

**==========================**

**Erros**

fs.readFile('arquivo.txt', (err, data) => {

if (err) {

console.error('Erro ao ler o arquivo:', err); // ← aqui está o erro real

return;

}

console.log(data.toString());

});

app.use((err, req, res, next) => {

console.error('Erro capturado:', err); // ← err contém o erro que foi "lançado" com next(err)

res.status(500).json({ message: 'Erro interno' });

});

err é a variável arbitraria (poderia ser e, error, ou qualquer outro nome) que contém o erro retornado pela operação. Se não houver erro, err === null

console.error(), error faz parte de console.error() e não pode ser substituido por outro nome.

**app.use((err, req, res, next) => { ... })**

Essa é a **assinatura de um middleware de erro no Express**.  
O que importa aqui **não são os nomes dos parâmetros**, mas **a ordem deles**.

**📌 Ordem correta dos parâmetros (e seus papéis):**

| **Ordem** | **Nome comum** | **Papel / conteúdo** |
| --- | --- | --- |
| 1️⃣ | err | O **erro capturado** em alguma parte da aplicação |
| 2️⃣ | req | O objeto de **requisição** do cliente (Request) |
| 3️⃣ | res | O objeto de **resposta** que você envia (Response) |
| 4️⃣ | next | Uma função para **passar o controle adiante** |

app.use((a, b, c, d) => {

console.log(a); // é o erro

console.log(b.url); // é req.url

console.log(c.status); // é res.status

});

Ou seja, são nomes arbitrários e somente a ordem importa para a função de cada parâmetro. Porém,

app.use((req, res, next, err) => {

// ERRADO: aqui o Express acha que isso é um middleware comum, não de erro

});

**Comparando dois tipos de middleware**

**🔹 Middleware normal (sem erro):**

app.use((req, res, next) => {

console.log('Requisição recebida:', req.url);

next();

});

* Padrão: (req, res, next)
* Usado para autenticação, logging, CORS etc.

**🔹 Middleware de erro:**

app.use((err, req, res, next) => {

console.error('Erro:', err.message);

res.status(500).json({ error: 'Erro interno' });

});

* Padrão: (err, req, res, next)
* Usado **exclusivamente** para capturar erros em rotas, middlewares ou async

**======================================**

**Assinaturas**

**✅ 1. (req, res, next) — Middleware padrão**

Essa assinatura é usada quando a função está **intermediando** o fluxo da requisição, ou seja, é um **middleware**.

**📌 Papel do next:**

* next é uma função que você chama para **passar o controle adiante**
  + Para o próximo middleware
  + Para a próxima rota
  + Para um tratador de erro (caso next(err))

**✅ Exemplo:**

app.use((req, res, next) => {

console.log('Middleware foi chamado');

next(); // passa para o próximo middleware ou rota

});

**🧠 Quando usar:**

* Autenticação
* Logging
* Validações
* Injeção de dados no req
* CORS
* Qualquer processamento **antes da rota final**

**====================================**

**Cabeçalho Authorization no headers e token JWT**

O cabeçalho Authorization em uma requisição HTTP **pode — e frequentemente deve — conter um JWT (JSON Web Token)** quando você está usando autenticação baseada em tokens.

**🔐 Como funciona isso na prática?**

**📌 1. O cliente (ex: frontend) envia o JWT no cabeçalho:**

GET /perfil HTTP/1.1

Host: api.exemplo.com

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6...

* O valor do header é:

Authorization: Bearer <seu\_token\_jwt>

O prefixo Bearer é uma convenção usada para indicar que o valor do cabeçalho é um **token portador**, como o JWT.

**📌 2. No backend (Express, por exemplo), você acessa assim:**

const token = req.headers.authorization?.split(' ')[1];

* Isso separa 'Bearer' e o token em duas partes
* token vai conter só o JWT

**📌 3. Verificando o JWT:**

Você normalmente usa uma biblioteca como jsonwebtoken para verificar o token:

const jwt = require('jsonwebtoken');

function autenticarToken(req, res, next) {

const authHeader = req.headers.authorization;

const token = authHeader && authHeader.split(' ')[1];

if (!token) return res.sendStatus(401);

jwt.verify(token, process.env.JWT\_SECRET, (err, usuario) => {

if (err) return res.sendStatus(403); // token inválido

req.usuario = usuario; // coloca os dados do token na requisição

next(); // passa para a próxima rota

});

}

========================

**Operador spread (…)**

É uma forma **elegante e concisa** de incluir uma propriedade **condicionalmente** em um objeto.

Sem spread, você faria algo assim:

const response = { error: err.message };

if (process.env.NODE\_ENV === 'development') {

response.stack = err.stack;

}

res.json(response);

Com spread, fica mais direto:

res.json({

error: err.message,

...(process.env.NODE\_ENV === 'development' && { stack: err.stack })

});

Interpretando:

1. process.env.NODE\_ENV === 'development'
   * Verifica se o ambiente atual é de desenvolvimento
2. && { stack: err.stack }
   * Se for, retorna o objeto { stack: err.stack }
   * Se não for, retorna false

======================================

**✅ 2. (req, res) — Manipulador de rota final (handler)**

Essa assinatura é usada quando a função **responde diretamente à requisição**, ou seja, **é o fim da linha**.  
Aqui, **não precisa de next**, pois a resposta será enviada direto com res.send(), res.json(), etc.

**✅ Exemplo:**

app.get('/usuarios', (req, res) => {

res.json({ nome: 'Caio' });

});

Se você **não vai passar para outro middleware**, não precisa do next.

**🔁 Comparando os dois:**

| **Assinatura** | **Papel principal** | **Chama next()?** | **Deve enviar resposta (res.send())?** |
| --- | --- | --- | --- |
| (req, res, next) | Middleware intermediário | Sim (normalmente) | Não obrigatoriamente |
| (req, res) | Handler final (rota) | Não | Sim |

**🎯 Exemplo didático com os dois juntos:**

// Middleware de autenticação

function autenticar(req, res, next) {

if (req.headers.authorization !== '123') {

return res.status(401).json({ error: 'Não autorizado' });

}

next(); // passou, chama a próxima função

}

// Rota protegida

app.get('/dados', autenticar, (req, res) => {

res.json({ segredo: '42' });

});

* autenticar usa (req, res, next) → é um middleware
* A função final usa (req, res) → é o **handler da rota**

**Entendendo o package.json**

O arquivo package.json é o coração de qualquer projeto Node.js. Ele funciona como um "documento de identidade" do seu projeto, contendo informações essenciais e gerenciando as dependências.

**O que é o package.json?**

O package.json é um arquivo de manifesto que:

1. **Define o projeto**: Nome, versão, descrição, autor, licença, etc.
2. **Gerencia dependências**: Lista todas as bibliotecas externas que seu projeto utiliza
3. **Configura scripts**: Define comandos personalizados para executar tarefas no projeto
4. **Estabelece configurações**: Versão do Node.js, tipo de repositório e outras configurações

**Como o npm usa o package.json**

Quando você executa npm install:

1. O npm lê o arquivo package.json
2. Verifica as seções dependencies e devDependencies
3. Baixa todas as bibliotecas listadas (e suas próprias dependências)
4. Armazena os arquivos baixados na pasta node\_modules
5. Cria ou atualiza o arquivo package-lock.json para garantir versões exatas

**Principais seções do nosso package.json**

**Metadados do Projeto**

{

"name": "sistema-atividades-avaliativas",

"version": "1.0.0",

"description": "Sistema para gerenciamento de atividades avaliativas escolares",

"main": "server.js",

// ...

}

Estes campos definem as informações básicas do projeto, incluindo qual arquivo é o ponto de entrada (main).

**Scripts**

"scripts": {

"start": "node server.js",

"dev": "nodemon server.js",

"test": "node tests/run-all-tests.js",

"test:db": "node tests/db-test.js",

"test:auth": "node tests/auth-test.js",

"test:server": "node tests/server-test.js"

}

Esta seção define comandos que podem ser executados com npm run [nome-do-script]:

* npm start: Inicia o servidor com Node.js
* npm run dev: Inicia o servidor com Nodemon (reinicia automaticamente quando há mudanças)
* npm test: Executa todos os testes
* npm run test:db: Executa apenas os testes de banco de dados

**Dependências**

"dependencies": {

"bcrypt": "^5.1.1",

"cors": "^2.8.5",

"dotenv": "^16.3.1",

"express": "^4.18.2",

"express-rate-limit": "^7.1.5",

"helmet": "^7.1.0",

"jsonwebtoken": "^9.0.2",

"morgan": "^1.10.0",

"pg": "^8.11.3"

}

Esta seção lista todas as bibliotecas necessárias para o projeto **funcionar em produção**:

* express: Framework web
* pg: Cliente PostgreSQL
* bcrypt: Biblioteca para hash de senhas
* jsonwebtoken: Para autenticação JWT
* dotenv: Carrega variáveis de ambiente
* helmet: Middleware de segurança
* morgan: Logger de requisições HTTP
* cors: Controle de acesso Cross-Origin
* express-rate-limit: Limitação de requisições

O ^ antes da versão significa "compatível com esta versão ou mais recente na mesma versão maior". Por exemplo, ^5.1.1 aceita 5.1.2 ou 5.2.0, mas não 6.0.0.

**Dependências de Desenvolvimento**

"devDependencies": {

"nodemon": "^3.0.2"

}

Bibliotecas necessárias apenas durante o desenvolvimento, não em produção:

* nodemon: Reinicia automaticamente o servidor durante o desenvolvimento

Para instalar apenas as dependências de produção (sem as de desenvolvimento), use npm install --production.

**Como instalar o projeto**

Para instalar todas as dependências listadas no package.json:

npm install

Este comando:

1. Lê o arquivo package.json
2. Cria a pasta node\_modules (se não existir)
3. Baixa e instala todas as bibliotecas listadas
4. Cria/atualiza o arquivo package-lock.json

**Adicionando novas dependências**

Para adicionar uma nova biblioteca ao projeto:

npm install nome-da-biblioteca # Adiciona às dependências

npm install nome-da-biblioteca --save-dev # Adiciona às devDependencies

Estes comandos instalam a biblioteca **e** a adicionam automaticamente ao package.json.

**O arquivo package-lock.json**

Ao executar npm install, também é criado um arquivo package-lock.json. Este arquivo:

* Registra as versões exatas de todas as dependências instaladas (incluindo dependências indiretas)
* Garante que todos os desenvolvedores usem exatamente as mesmas versões
* Torna as instalações mais rápidas e previsíveis

É importante incluir este arquivo no controle de versão junto com o package.json.

**A pasta node\_modules**

A pasta node\_modules contém todas as dependências instaladas. Ela:

* Pode ser muito grande (centenas de MB)
* **Não** deve ser incluída no controle de versão (adicione ao .gitignore)
* Pode ser facilmente recriada com npm install

**Resumo**

O package.json é essencial para qualquer projeto Node.js:

1. Define metadados do projeto
2. Lista todas as dependências
3. Configura scripts para automação
4. Permite compartilhar e reproduzir facilmente o ambiente de desenvolvimento

Com este arquivo configurado corretamente, qualquer pessoa pode clonar o repositório e executar npm install para ter um ambiente de desenvolvimento idêntico ao seu.

-----------------------------------------------------------------------------------------

npm run dev

> app-escola-atividades-backend@1.0.0 dev

> nodemon server.js

[nodemon] 3.1.10

[nodemon] to restart at any time, enter `rs`

[nodemon] watching path(s): \*.\*

[nodemon] watching extensions: js,mjs,cjs,json

[nodemon] starting `node server.js`

Início do arquivo server.js

Início do arquivo app.js

Início do arquivo config/database.js

Fim do arquivo config/database.js

Início do arquivo middleware/config.js

Fim do arquivo middleware/config.js

Início do arquivo middleware/cors.js

Fim do arquivo middleware/cors.js

Início do arquivo utils/errorHandler.js

Fim do arquivo utils/errorHandler.js

Início do arquivo routes/authRoutes.js

Início do arquivo controllers/authController.js

Início do arquivo config/jwt.js

Fim do arquivo config/jwt.js

Início do arquivo utils/bcrypt.js

Fim do arquivo utils/bcrypt.js

Fim do arquivo controllers/authController.js

Fim do arquivo routes/authRoutes.js

Fim do arquivo app.js

Início do arquivo utils/health.js

Fim do arquivo utils/health.js

Health check configurado em: /health

Fim do arquivo server.js

Servidor rodando na porta 5000

Ambiente: development

Timezone configurado para: America/Sao\_Paulo

Health check disponível em: /health

Banco de dados conectado com sucesso, timestamp: 2025-07-24T16:48:59.708Z

IMPORTANTE: Você está observando algo importante! O arquivo middleware/auth.js não aparece nos logs de inicialização porque ele não está sendo carregado durante a inicialização do servidor. Isso é normal e não indica um problema.

**Por que middleware/auth.js não aparece nos logs:**

**Carregamento sob demanda**: Diferente dos outros arquivos, o middleware/auth.js não é importado diretamente no app.js ou server.js

**Arquivos nos logs**: Os arquivos que você vê nos logs são aqueles que:

São importados diretamente no início da aplicação

São necessários para a configuração inicial

São usados para montar as rotas básicas

**middleware/auth.js é diferente**: Ele:

Contém middlewares que são aplicados apenas em rotas específicas

Não é necessário durante a inicialização

Será importado apenas dentro dos arquivos de rotas que precisam de proteção

**Como o middleware/auth.js será usado:**

O arquivo middleware/auth.js será importado e utilizado dentro dos arquivos de rotas que implementaremos na próxima etapa, como por exemplo:

*// Em um futuro arquivo routes/professorRoutes.js*

const { authMiddleware, authProfessor } = require('../middleware/auth');

*// Aqui a rota usa o middleware de autenticação*

router.get('/questionarios', authMiddleware, authProfessor, controller.listar);

Como ainda não implementamos as rotas protegidas para professor, aluno e admin, o arquivo middleware/auth.js não está sendo carregado durante a inicialização. Ele só será carregado quando essas rotas forem acessadas.

O middleware está corretamente implementado, mas só é usado quando necessário, o que é uma boa prática para otimização de recursos. Isso explica por que você não o vê nos logs de inicialização.

**package.json**

|  |
| --- |
| {    "name": "app-escola-atividades-backend",    "version": "1.0.0",    "description": "Backend do Sistema de Atividades Avaliativas para Escola",    "main": "server.js",    "engines": {      "node": ">=14.0.0",      "npm": ">=6.0.0"    },  "scripts": {    "start": "node server.js",    "dev": "nodemon server.js",    "test": "node tests/run-all-tests.js",    "test:db": "node tests/db-test.js",    "test:auth": "node tests/auth-test.js",    "test:server": "node tests/server-test.js"    },    "dependencies": {    "bcrypt": "^5.1.1",    "cors": "^2.8.5",    "dotenv": "^16.3.1",    "express": "^4.18.2",    "express-rate-limit": "^7.1.5",    "helmet": "^7.1.0",    "jsonwebtoken": "^9.0.2",    "morgan": "^1.10.0",    "pg": "^8.11.3"  },    "devDependencies": {      "nodemon": "^3.0.2"  },    "author": "Desenvolvedor Sistema Escola",    "license": "MIT",    "repository": {      "type": "git",      "url": "https://github.com/usuario/app-escola-atividades-backend.git"    },    "bugs": {      "url": "https://github.com/usuario/app-escola-atividades-backend/issues"    },    "homepage": "https://github.com/usuario/app-escola-atividades-backend#readme"  } |

**Arquivos do backend basico de acordo com o fluxo de conexões**

**server.js**

|  |
| --- |
| console.log('Início do arquivo server.js');  // server.js - Arquivo principal para inicialização do servidor  // Importação da aplicação configurada  const app = require('./app');  // Importação do módulo de monitoramento de saúde  const healthCheck = require('./src/utils/health');  // Porta do servidor  const PORT = process.env.PORT || 5000;  // Configuração do health check  healthCheck.setup(app);  // Inicialização do servidor  app.listen(PORT, () => {    console.log(`Servidor rodando na porta ${PORT}`);    console.log(`Ambiente: ${process.env.NODE\_ENV || 'desenvolvimento'}`);    console.log('Timezone configurado para: America/Sao\_Paulo');    console.log('Health check disponível em: /health');  });  // Tratamento de sinais para encerramento gracioso  process.on('SIGTERM', () => {    console.info('SIGTERM recebido. Encerrando servidor...');    process.exit(0);  });  process.on('SIGINT', () => {    console.info('SIGINT recebido. Encerrando servidor...');    process.exit(0);  });  console.log('Fim do arquivo server.js'); |

**app.js**

|  |
| --- |
| console.log('Início do arquivo app.js');  // app.js - Configuração da aplicação e inicialização do servidor  // Importação de dependências  const express = require('express');  const rateLimit = require('express-rate-limit');  require('dotenv').config(); // carrega as variáveis do .env para process.env.  console.log(process.env.PORT); // 5000  console.log(process.env.NODE\_ENV); // development  // Importação de configurações e middlewares  const db = require('./src/config/database');  const middlewareConfig = require('./src/middleware/config');  const corsMiddleware = require('./src/middleware/cors');  const errorHandler = require('./src/utils/errorHandler');  // Importação das rotas  const authRoutes = require('./src/routes/authRoutes');  // const professorRoutes = require('./src/routes/professorRoutes');  // const alunoRoutes = require('./src/routes/alunoRoutes');  // const adminRoutes = require('./src/routes/adminRoutes');  // Criação da aplicação Express  const app = express();  // Limita a 100 requisições por IP a cada 15 minutos  const limiter = rateLimit({    windowMs: 15 \* 60 \* 1000, // 15 minutos    max: 100, // máximo de 100 requisições por IP    message: 'Você excedeu o número de requisições permitidas. Tente novamente mais tarde.'  });  // Aplica o limitador globalmente  app.use(limiter);  // Aplicação do middleware CORS  app.use(corsMiddleware);  // Aplicação de middlewares  middlewareConfig(app); |

**config/database.js**

|  |
| --- |
| console.log('Início do arquivo config/database.js');  // config/database.js - Configuração da conexão com o banco de dados  const { Pool } = require('pg');  require('dotenv').config();  console.log('Tipo do DATABASE\_URL:', typeof process.env.DATABASE\_URL);  // Configuração do banco de dados PostgreSQL  const pool = new Pool({    connectionString: process.env.DATABASE\_URL,    ssl: process.env.NODE\_ENV === 'production' ? { rejectUnauthorized: false } : false,      // Configurações de conexão    max: 20, // Máximo de conexões no pool    idleTimeoutMillis: 30000, // Tempo limite para conexões ociosas    connectionTimeoutMillis: 10000, // Tempo limite para estabelecer conexão      // Configurações específicas para o timezone    timezone: 'America/Sao\_Paulo'  });  // Verificação de conexão com o banco  pool.query('SELECT NOW()', (err, res) => {    if (err) {      console.error('Erro ao conectar ao banco de dados:', err);    } else {      console.log('Banco de dados conectado com sucesso, timestamp:', res.rows[0].now);      // Configurar timezone no banco      pool.query("SET timezone = 'America/Sao\_Paulo'");    }  });  // Exporta o pool para ser usado em outros módulos  module.exports = pool;  console.log('Fim do arquivo config/database.js'); |

**middleware/config.js**

|  |
| --- |
| console.log('Início do arquivo middleware/config.js');  // middleware/config.js - Configuração dos middlewares básicos da aplicação  const express = require('express');  // Ambas importam bibliotecas externas, helmet e morgan (não são nativas do Node.js).  const helmet = require('helmet'); // Biblioteca externa de segurança para aplicações Express  const morgan = require('morgan'); // Biblioteca externa de logging de requisições  const db = require('../config/database');  // O método app.use() é usado para registrar middlewares na sua aplicação Express  /\*Essa arrow function espera receber o objeto app da aplicação Express,    e aplica vários middlewares a ele com app.use().\*/  const setupMiddlewares = (app) => {    // Middlewares para parsing de dados    app.use(express.json()); // Para parsing de JSON    // Se você enviar uma requisição com corpo { "nome": "Caio" }, o Express vai transformar isso em: req.body = { nome: 'Caio' }    app.use(express.urlencoded({ extended: true })); // Para parsing de dados de formulário    // Se você enviar um formulário tipo nome=Caio&idade=30, isso vira: req.body = { nome: 'Caio', idade: '30' }    // Middlewares de segurança    app.use(helmet()); // Proteção básica com headers HTTP    // proteger sua aplicação contra vulnerabilidades conhecidas (como clickjacking, sniffing etc).    app.use(morgan('dev')); // Logging de requisições    // Exibe logs como: GET /alunos 200 15ms    // Middleware para disponibilizar o pool de conexão em todas as requisições    app.use((req, res, next) => {         req.db = db;         next();       });    };  // Isso evita que você tenha que importar o banco de dados em cada rota. Em vez disso, basta acessar: req.db.query(...)  module.exports = setupMiddlewares;  console.log('Fim do arquivo middleware/config.js'); |

**middleware/cors.js**

|  |
| --- |
| console.log('Início do arquivo middleware/cors.js');  // middleware/cors.js - Configuração de CORS (Cross-Origin Resource Sharing)  const cors = require('cors'); // importa a biblioteca externa chamada cors  /\*  É um middleware para o Express que permite habilitar o CORS (Cross-Origin Resource Sharing)  — ou seja, controlar quais domínios podem fazer requisições à sua API.  O Node.js puro não tem cors embutido.  \*/  // Lista de origens permitidas  const getAllowedOrigins = () => {    const devOrigins = [      'http://localhost:3000',      'http://127.0.0.1:3000',  'http://127.0.0.1:5500'    ];    if (process.env.NODE\_ENV === 'production') {      return [process.env.FRONTEND\_URL].filter(Boolean);    }    return devOrigins;  };  /\*  1) Requisição do navegador: fetch('https://api.exemplo.com/dados')  O navegador (Chrome, Firefox, etc.) adiciona automaticamente:  - Origin: https://seusite.com  Exemplo do cabeçalho da requisição:  - GET /dados HTTP/1.1  - Host: api.exemplo.com  - Origin: https://seusite.com  2) Requisição SEM Origin (Postman, cURL, backend)  Exemplo usando Postman ou curl: curl https://api.exemplo.com/dados  Cabeçalho da requisição:  - GET /dados HTTP/1.1  - Host: api.exemplo.com  - # (sem Origin!)  \*/  // origin e credentials são chaves reconhecidas pela função cors().  // Postman e cURL fazem requisições que não têm cabeçalho Origin  const corsOptions = {    origin: function (origin, callback) {      const allowedOrigins = getAllowedOrigins();      // Permitir chamadas sem origin (ex: Postman, curl)      if (!origin) return callback(null, true); // null → sem erro, true → permitir o acesso      if (allowedOrigins.includes(origin)) {        return callback(null, true);      }      console.warn(`❌ CORS bloqueado para origem: ${origin}`);      return callback(new Error('Origem não permitida pelo CORS'));    },    credentials: true // permite cookies, headers como Authorization  };  module.exports = cors(corsOptions);  console.log('Fim do arquivo middleware/cors.js'); |

**utils/errorHandler.js**

|  |
| --- |
| console.log('Início do arquivo utils/errorHandler.js');  // utils/errorHandler.js - Configuração de tratamento de erros globais  /\*\*   \* Configura handlers para erros não tratados na aplicação   \* O process é um objeto global do Node.js que representa o processo em execução da aplicação Node.   \* process não precisa ser importado — ele está sempre disponível   \*/  const setup = () => {    // Trata exceções não capturadas    process.on('uncaughtException', (error) => {      console.error('Erro não tratado:', error);      // Em produção, aqui poderia enviar o erro para um serviço de monitoramento    });    /\*    \* 'uncaughtException' é disparado quando um erro é lançado e não tratado com try/catch    \* 'unhandledRejection' é disparado quando uma promessa (Promise) é rejeitada e ninguém trata com .catch()    \*/    /\*    Esses eventos são úteis para:  - Capturar erros graves que escaparam do seu controle  - Evitar que o processo "morra silenciosamente"  - Logar o erro  - Notificar um sistema de monitoramento (ex: Sentry, LogRocket, Datadog)    \*/    // Trata promessas rejeitadas não capturadas    process.on('unhandledRejection', (reason, promise) => {      console.error('Promessa rejeitada não tratada:', reason);      // Em produção, aqui poderia enviar o erro para um serviço de monitoramento    });  };  /\*  \* Apesar de úteis, não se deve depender exclusivamente desses handlers. Eles são como "última linha de defesa".  \* Sempre prefira: try/catch para blocos de código síncronos  \*/  /\*\*   \* Middleware para tratamento de erros em rotas Express   \* Express reconhece como middleware de erro não é pelo nome da função, mas sim a assinatura dela:     (err, req, res, next) => { ... }  \* Se um middleware tiver quatro parâmetros, nessa ordem:  function meuTratadorDeErros(err, req, res, next) {    // ...  }  O Express sabe automaticamente que ele serve para capturar e tratar erros.  É um middleware especial do Express para erros, identificado por ter 4 parâmetros: err, req, res, next.  O Express reconhece automaticamente esse tipo de middleware como um tratador global de erros.   \*/  const errorMiddleware = (err, req, res, next) => {    console.error('Erro na requisição:', err);      // Determina o código de status HTTP adequado    // Verifica se o erro tem um statusCode definido. Caso não tenha, usa o código 500 (Erro interno do servidor).    const statusCode = err.statusCode || 500;      // Resposta formatada para o cliente    res.status(statusCode).json({      error: err.message || 'Erro interno no servidor', // error: é uma chave (de nome arbitrario) de objeto JS do JSON.      // Incluir stack trace apenas em ambiente de desenvolvimento      ...(process.env.NODE\_ENV === 'development' && { stack: err.stack })    });  };  /\*  ... são o operador spread (ou "operador de espalhamento") do JavaScript.  No contexto de objetos, ele “espalha” as propriedades de outro objeto dentro de um novo objeto  \*/  module.exports = {    setup,    errorMiddleware  };  console.log('Fim do arquivo utils/errorHandler.js'); |

**middleware/auth.js**

|  |
| --- |
| console.log('Início do arquivo middleware/auth.js');  // middleware/auth.js - Middlewares de autenticação e autorização  const jwt = require('jsonwebtoken');  require('dotenv').config();  // Middleware para verificação de JWT (autenticação)  const authMiddleware = (req, res, next) => {    const authHeader = req.headers.authorization;      if (!authHeader || !authHeader.startsWith('Bearer ')) {      return res.status(401).json({ error: 'Token não fornecido' });    }    const token = authHeader.split(' ')[1];      try {      const decoded = jwt.verify(token, process.env.JWT\_SECRET);      req.user = decoded;      next();    } catch (error) {      return res.status(401).json({ error: 'Token inválido ou expirado' });    }  };  // Middleware para verificar se é administrador  const authAdmin = (req, res, next) => {    if (req.user && req.user.tipo === 'admin') {      next();    } else {      res.status(403).json({ error: 'Acesso negado. Somente administradores.' });    }  };  // Middleware para verificar se é professor  const authProfessor = (req, res, next) => {    if (req.user && req.user.tipo === 'professor') {      next();    } else {      res.status(403).json({ error: 'Acesso negado. Somente professores.' });    }  };  // Middleware para verificar se é aluno  const authAluno = (req, res, next) => {    if (req.user && req.user.tipo === 'aluno') {      next();    } else {      res.status(403).json({ error: 'Acesso negado. Somente alunos.' });    }  };  // Middleware para verificar se é professor ou admin  const authProfessorOrAdmin = (req, res, next) => {    if (req.user && (req.user.tipo === 'professor' || req.user.tipo === 'admin')) {      next();    } else {      res.status(403).json({ error: 'Acesso negado. Somente professores ou administradores.' });    }  };  // Exporta todos os middlewares de autenticação  module.exports = {    authMiddleware,    authAdmin,    authProfessor,    authAluno,    authProfessorOrAdmin  };  console.log('Fim do arquivo middleware/auth.js'); |

**Importante: Como e quando o middleware/auth.js será usado?**

O arquivo middleware/auth.js será importado e utilizado dentro dos arquivos de rotas que implementaremos na próxima etapa, como por exemplo:

*// Em um futuro arquivo routes/professorRoutes.js*

const { authMiddleware, authProfessor } = require('../middleware/auth');

*// Aqui a rota usa o middleware de autenticação*

router.get('/questionarios', authMiddleware, authProfessor, controller.listar);

Como ainda não implementamos as rotas protegidas para professor, aluno e admin, o arquivo middleware/auth.js não está sendo carregado durante a inicialização. Ele só será carregado quando essas rotas forem acessadas.

**routes/authRoutes.js**

|  |
| --- |
| console.log('Início do arquivo routes/authRoutes.js');  // routes/authRoutes.js - Rotas relacionadas à autenticação  const express = require('express');  const router = express.Router();  const authController = require('../controllers/authController');  /\*\*   \* Rota de login para todos os tipos de usuário   \* POST /api/auth/login   \*/  router.post('/login', authController.login);  /\*\*   \* Rota para verificar se o token é válido   \* GET /api/auth/verify   \*/  router.get('/verify', authController.verify);  module.exports = router;  console.log('Fim do arquivo routes/authRoutes.js'); |

**controllers/authController.js**

|  |
| --- |
| console.log('Início do arquivo controllers/authController.js');  // controllers/authController.js - Controlador para funções de autenticação  const jwt = require('../config/jwt');  const bcrypt = require('../utils/bcrypt');  /\*\*   \* Função para autenticar usuário (login)   \* @param {Object} req - Requisição Express   \* @param {Object} res - Resposta Express   \*/  const login = async (req, res) => {    try {      const { email, senha, tipo } = req.body;        // Validação mais flexível      if (!email || !tipo) {        return res.status(400).json({ error: 'Email e tipo são obrigatórios' });      }        // Para admin e professor, senha é obrigatória      if ((tipo === 'admin' || tipo === 'professor') && !senha) {        return res.status(400).json({ error: 'Senha é obrigatória para administradores e professores' });      }        let user = null;      let query = '';        // Verifica o tipo de usuário para buscar na tabela correta      switch (tipo) {        case 'admin':          query = 'SELECT id\_admin as id, nome\_admin as nome, email\_admin as email, senha\_admin as senha\_hash FROM administrador WHERE email\_admin = $1 AND ativo = TRUE';          break;        case 'professor':          query = 'SELECT id\_professor as id, nome\_professor as nome, email\_professor as email, senha\_hash, id\_disciplina FROM professor WHERE email\_professor = $1 AND ativo = TRUE';          break;        case 'aluno':          query = 'SELECT a.id\_aluno as id, a.nome\_aluno as nome, a.email\_aluno as email, m.numero\_matricula as matricula FROM aluno a JOIN matricula m ON a.id\_aluno = m.id\_aluno WHERE a.email\_aluno = $1 AND a.ativo = TRUE AND m.ativa = TRUE';          break;        default:          return res.status(400).json({ error: 'Tipo de usuário inválido' });      }        const result = await req.db.query(query, [email]);        if (result.rows.length === 0) {        return res.status(401).json({ error: 'Usuário não encontrado ou inativo' });      }        user = result.rows[0];        // Para alunos, não verificamos senha (seria implementado diferentemente)      // Para admin e professor, verificamos a senha com bcrypt      if (tipo !== 'aluno') {        // Verificar a senha usando bcrypt        const passwordMatches = await bcrypt.comparePassword(senha, user.senha\_hash);        if (!passwordMatches) {          return res.status(401).json({ error: 'Senha incorreta' });        }      }        // Gera o token JWT usando o módulo jwt      const token = jwt.generateToken(user, tipo);        return res.json({        token,        user: {          id: user.id,          nome: user.nome,          email: user.email,          tipo        }      });      } catch (error) {      console.error('Erro no login:', error);      return res.status(500).json({ error: 'Erro interno no servidor' });    }  };  /\*\*   \* Verifica se o token é válido   \* @param {Object} req - Requisição Express   \* @param {Object} res - Resposta Express   \*/  const verify = (req, res) => {    const authHeader = req.headers.authorization;      if (!authHeader || !authHeader.startsWith('Bearer ')) {      return res.status(401).json({ valid: false });    }    const token = authHeader.split(' ')[1];      try {      const decoded = jwt.verify(token, process.env.JWT\_SECRET);      return res.json({ valid: true, user: decoded });    } catch (error) {      return res.status(401).json({ valid: false });    }  };  module.exports = {    login,    verify  };  console.log('Fim do arquivo controllers/authController.js'); |

**config/jwt.js**

|  |
| --- |
| console.log('Início do arquivo config/jwt.js');  // config/jwt.js - Configuração e funções relacionadas a JWT  const jwt = require('jsonwebtoken');  require('dotenv').config();  /\*\*   \* Gera um token JWT baseado no tipo de usuário e suas informações   \* @param {Object} user - Objeto com dados do usuário   \* @param {string} tipo - Tipo de usuário ('admin', 'professor', 'aluno')   \* @returns {string} Token JWT gerado   \*/  const generateToken = (user, tipo) => {    // Define o tempo de expiração do token com base no tipo de usuário    let expiresIn = '4h'; // Padrão para alunos (4 horas)    if (tipo === 'admin' || tipo === 'professor') {      expiresIn = '24h'; // 24 horas para professores e administradores    }      // Dados a serem incluídos no token    const payload = {      id: user.id,      tipo,      ...(tipo === 'professor' ? { disciplina: user.id\_disciplina } : {}),      ...(tipo === 'aluno' ? { matricula: user.matricula } : {})    };      // Gera o token JWT    return jwt.sign(payload, process.env.JWT\_SECRET, { expiresIn });  };  /\*\*   \* Verifica se um token JWT é válido   \* @param {string} token - Token JWT a ser verificado   \* @param {string} secret - Chave secreta para verificação   \* @returns {Object} Payload decodificado do token   \*/  const verify = (token, secret = process.env.JWT\_SECRET) => {    return jwt.verify(token, secret);  };  module.exports = {    generateToken,    verify  };  console.log('Fim do arquivo config/jwt.js'); |

**utils/bcrypt.js**

|  |
| --- |
| console.log('Início do arquivo utils/bcrypt.js');  // utils/bcrypt.js - Funções para hash e verificação de senhas  const bcrypt = require('bcrypt');  // Número de rounds para o salt (quanto maior, mais seguro, mas mais lento)  const SALT\_ROUNDS = 10;  /\*\*   \* Gera um hash para uma senha   \* @param {string} password - Senha em texto puro   \* @returns {Promise<string>} Hash da senha   \*/  const hashPassword = async (password) => {    try {      const salt = await bcrypt.genSalt(SALT\_ROUNDS);      const hash = await bcrypt.hash(password, salt);      return hash;    } catch (error) {      console.error('Erro ao gerar hash da senha:', error);      throw new Error('Falha ao processar senha');    }  };  /\*\*   \* Verifica se uma senha corresponde a um hash   \* @param {string} password - Senha em texto puro   \* @param {string} hash - Hash da senha armazenado   \* @returns {Promise<boolean>} Verdadeiro se a senha corresponder ao hash   \*/  const comparePassword = async (password, hash) => {    try {      return await bcrypt.compare(password, hash);    } catch (error) {      console.error('Erro ao comparar senha:', error);      throw new Error('Falha ao verificar senha');    }  };  module.exports = {    hashPassword,    comparePassword  };  console.log('Fim do arquivo utils/bcrypt.js'); |

**Importante:** A biblioteca bcrypt original de fato tem um método chamado compare(), não comparePassword(). No entanto, no nosso projeto, nós criamos intencionalmente um wrapper personalizado no arquivo utils/bcrypt.js.

**utils/health.js**

|  |
| --- |
| console.log('Início do arquivo utils/health.js');  // utils/health.js - Módulo para monitoramento de saúde da aplicação  // Importação do pool de conexão com o banco de dados  const db = require('../config/database');  /\*\*   \* Configuração da rota de health check da aplicação   \* @param {Object} app - Instância do Express   \*/  const setup = (app) => {    // Rota de health check    app.get('/health', async (req, res) => {      try {        // Verificação de conexão com o banco de dados        const dbCheck = await checkDatabase();          // Status geral da aplicação        const status = {          status: dbCheck.connected ? 'UP' : 'DEGRADED',          timestamp: new Date().toISOString(),          uptime: process.uptime(),          memory: process.memoryUsage(),          database: dbCheck,          environment: process.env.NODE\_ENV || 'development'        };          // Código de status HTTP baseado no estado do banco        const statusCode = dbCheck.connected ? 200 : 503;          res.status(statusCode).json(status);      } catch (error) {        console.error('Erro ao verificar saúde da aplicação:', error);          res.status(500).json({          status: 'DOWN',          timestamp: new Date().toISOString(),          error: error.message        });      }    });      console.log('Health check configurado em: /health');  };  /\*\*   \* Verifica o estado da conexão com o banco de dados   \* @returns {Object} Objeto com informações sobre a conexão   \*/  const checkDatabase = async () => {    try {      // Tenta executar uma query simples para verificar conexão      const startTime = Date.now();      const result = await db.query('SELECT NOW()');      const responseTime = Date.now() - startTime;        return {        connected: true,        responseTime: `${responseTime}ms`,        timestamp: result.rows[0].now      };    } catch (error) {      console.error('Erro na verificação do banco de dados:', error);        return {        connected: false,        error: error.message      };    }  };  module.exports = {    setup,    checkDatabase  };  console.log('Fim do arquivo utils/health.js'); |

**TESTES**

**auth-test.js**

|  |
| --- |
| // tests/auth-test.js - Testes simples para verificar a autenticação  // Execução: node src/tests/auth-test.js  // Importações  const db = require('../config/database');  const bcrypt = require('../utils/bcrypt');  const jwt = require('../config/jwt');  require('dotenv').config();  // Cores para console  const colors = {    reset: "\x1b[0m",    bright: "\x1b[1m",    red: "\x1b[31m",    green: "\x1b[32m",    yellow: "\x1b[33m",    blue: "\x1b[34m"  };  // Contador de testes  let passedTests = 0;  let failedTests = 0;  let totalTests = 0;  // Função para registrar resultados de testes  function logTest(name, success, error = null) {    totalTests++;    if (success) {      console.log(`${colors.green}✓ PASSOU: ${name}${colors.reset}`);      passedTests++;    } else {      console.log(`${colors.red}✗ FALHOU: ${name}${colors.reset}`);      if (error) {        console.log(`  ${colors.red}Erro: ${error.message}${colors.reset}`);      }      failedTests++;    }  }  // Função para resumo final  function printSummary() {    console.log("\n" + "=".repeat(50));    console.log(`${colors.bright}RESUMO DOS TESTES:${colors.reset}`);    console.log(`${colors.green}Passaram: ${passedTests}${colors.reset}`);    console.log(`${colors.red}Falharam: ${failedTests}${colors.reset}`);    console.log(`${colors.blue}Total: ${totalTests}${colors.reset}`);    console.log("=".repeat(50) + "\n");  }  // ========== TESTES ==========  async function runTests() {    console.log(`${colors.bright}${colors.blue}TESTES DE AUTENTICAÇÃO${colors.reset}`);    console.log("=".repeat(50));    console.log("Executando testes...\n");    try {      // TESTE 1: Verificar geração de hash com bcrypt      try {        const password = "senha123";        const hash = await bcrypt.hashPassword(password);          // Hash deve ser uma string longa        const isValidHash = typeof hash === 'string' && hash.length > 50;        logTest("Geração de hash bcrypt", isValidHash);          if (isValidHash) {          console.log(`  Hash gerado: ${hash.substring(0, 20)}...`);        }      } catch (error) {        logTest("Geração de hash bcrypt", false, error);      }      // TESTE 2: Verificar comparação de senhas com bcrypt      try {        const password = "senha123";        const hash = await bcrypt.hashPassword(password);          // Comparação com senha correta        const validComparison = await bcrypt.comparePassword(password, hash);        logTest("Comparação de senha correta", validComparison);          // Comparação com senha incorreta        const invalidComparison = await bcrypt.comparePassword("senha\_errada", hash);        logTest("Rejeição de senha incorreta", !invalidComparison);        } catch (error) {        logTest("Comparação de senhas", false, error);      }      // TESTE 3: Verificar geração de JWT      try {        // Usuário de teste        const testUser = {          id: 1,          nome: "Usuário Teste",          email: "teste@escola.com"        };          // Gerar tokens para diferentes tipos de usuário        const adminToken = jwt.generateToken(testUser, 'admin');        const professorToken = jwt.generateToken(testUser, 'professor');        const alunoToken = jwt.generateToken(testUser, 'aluno');          logTest("Geração de token para admin", typeof adminToken === 'string' && adminToken.length > 0);        logTest("Geração de token para professor", typeof professorToken === 'string' && professorToken.length > 0);        logTest("Geração de token para aluno", typeof alunoToken === 'string' && alunoToken.length > 0);          console.log(`  Token de admin: ${adminToken.substring(0, 20)}...`);        console.log(`  Token de professor: ${professorToken.substring(0, 20)}...`);        console.log(`  Token de aluno: ${alunoToken.substring(0, 20)}...`);        } catch (error) {        logTest("Geração de tokens JWT", false, error);      }      // TESTE 4: Verificar se JWT\_SECRET está configurado      try {        const jwtSecret = process.env.JWT\_SECRET;        const isConfigured = jwtSecret && jwtSecret.length > 0;          logTest("JWT\_SECRET configurado", isConfigured);          if (!isConfigured) {          console.log(`  ${colors.yellow}Aviso: JWT\_SECRET não está configurado no arquivo .env${colors.reset}`);        }      } catch (error) {        logTest("Verificação de JWT\_SECRET", false, error);      }        // TESTE 5: Verificar usuários no banco (se existirem)      try {        // Verificar se a tabela professor existe        const tableResult = await db.query(`          SELECT EXISTS (            SELECT FROM information\_schema.tables            WHERE table\_schema = 'public'            AND table\_name = 'professor'          ) as exists        `);          if (tableResult.rows[0].exists) {          // Contar professores ativos          const countResult = await db.query(`            SELECT COUNT(\*) as count FROM professor WHERE ativo = TRUE          `);            const professorCount = parseInt(countResult.rows[0].count);          logTest("Existem professores ativos no sistema", professorCount > 0);          console.log(`  Total de professores ativos: ${professorCount}`);            if (professorCount === 0) {            console.log(`  ${colors.yellow}Aviso: Não existem professores ativos cadastrados${colors.reset}`);          }        } else {          logTest("Tabela de professores existe", false);          console.log(`  ${colors.yellow}Aviso: A tabela 'professor' não existe no banco${colors.reset}`);        }      } catch (error) {        logTest("Verificação de usuários", false, error);      }      // TESTE 6: Testar a verificação de JWT      try {        // Teste com um token válido        const user = { id: 1, nome: "Teste" };        const token = jwt.generateToken(user, 'admin');          try {          const decoded = jwt.verify(token);          const isValid = decoded && decoded.id === 1;            logTest("Verificação de token válido", isValid);          console.log(`  Conteúdo decodificado: ${JSON.stringify(decoded)}`);        } catch (err) {          logTest("Verificação de token válido", false, err);        }          // Teste com um token inválido        try {          const invalidToken = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzNCwiaWF0IjoxNTE2MjM5MDIyfQ.invalid\_signature";          jwt.verify(invalidToken);            // Se chegar aqui, o teste falhou          logTest("Rejeição de token inválido", false);        } catch (err) {          // Erro é esperado para token inválido          logTest("Rejeição de token inválido", true);          console.log(`  Erro corretamente gerado: ${err.message}`);        }      } catch (error) {        logTest("Verificação de JWT", false, error);      }    } catch (error) {      console.error(`${colors.red}Erro geral nos testes: ${error.message}${colors.reset}`);    } finally {      // Imprimir resumo      printSummary();        // Fechar conexão com o banco      console.log("Fechando conexão com o banco de dados...");      await db.end();      console.log("Testes concluídos.");    }  }  // Executar os testes  runTests(); |

**db-test.js**

|  |
| --- |
| // tests/db-test.js - Testes simples para verificar a integração com o banco de dados  // Execução: node src/tests/db-test.js  // Importações  const db = require('../config/database');  require('dotenv').config();  // Cores para console  const colors = {    reset: "\x1b[0m",    bright: "\x1b[1m",    red: "\x1b[31m",    green: "\x1b[32m",    yellow: "\x1b[33m",    blue: "\x1b[34m"  };  // Contador de testes  let passedTests = 0;  let failedTests = 0;  let totalTests = 0;  // Função para registrar resultados de testes  function logTest(name, success, error = null) {    totalTests++;    if (success) {      console.log(`${colors.green}✓ PASSOU: ${name}${colors.reset}`);      passedTests++;    } else {      console.log(`${colors.red}✗ FALHOU: ${name}${colors.reset}`);      if (error) {        console.log(`  ${colors.red}Erro: ${error.message}${colors.reset}`);      }      failedTests++;    }  }  // Função para resumo final  function printSummary() {    console.log("\n" + "=".repeat(50));    console.log(`${colors.bright}RESUMO DOS TESTES:${colors.reset}`);    console.log(`${colors.green}Passaram: ${passedTests}${colors.reset}`);    console.log(`${colors.red}Falharam: ${failedTests}${colors.reset}`);    console.log(`${colors.blue}Total: ${totalTests}${colors.reset}`);    console.log("=".repeat(50) + "\n");  }  // ========== TESTES ==========  async function runTests() {    console.log(`${colors.bright}${colors.blue}TESTES DE INTEGRAÇÃO COM BANCO DE DADOS${colors.reset}`);    console.log("=".repeat(50));    console.log("Executando testes...\n");    try {      // TESTE 1: Verificar conexão com o banco      try {        const result = await db.query('SELECT NOW()');        logTest("Conexão com o banco de dados", true);        console.log(`  Timestamp do banco: ${result.rows[0].now}`);      } catch (error) {        logTest("Conexão com o banco de dados", false, error);      }      // TESTE 2: Verificar se o timezone está configurado      try {        const result = await db.query("SHOW timezone");        const timezone = result.rows[0].timezone;        const isCorrectTimezone = timezone === 'America/Sao\_Paulo';          logTest("Timezone configurado corretamente", isCorrectTimezone);        console.log(`  Timezone atual: ${timezone}`);          if (!isCorrectTimezone) {          console.log(`  ${colors.yellow}Aviso: O timezone deveria ser 'America/Sao\_Paulo'${colors.reset}`);        }      } catch (error) {        logTest("Verificação de timezone", false, error);      }      // TESTE 3: Contar tabelas do sistema      try {        const result = await db.query(`          SELECT COUNT(\*) as total          FROM information\_schema.tables          WHERE table\_schema = 'public'        `);          const tableCount = parseInt(result.rows[0].total);        logTest("Verificação de tabelas do sistema", tableCount > 0);        console.log(`  Total de tabelas encontradas: ${tableCount}`);          if (tableCount === 0) {          console.log(`  ${colors.yellow}Aviso: Nenhuma tabela encontrada. O esquema está vazio.${colors.reset}`);        }      } catch (error) {        logTest("Verificação de tabelas do sistema", false, error);      }      // TESTE 4: Testar inserção e remoção de dados temporários      try {        // Criar tabela temporária para teste        await db.query(`          CREATE TEMPORARY TABLE test\_temp (            id SERIAL PRIMARY KEY,            name VARCHAR(100) NOT NULL,            created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP          )        `);          // Inserir dados        const insertResult = await db.query(`          INSERT INTO test\_temp (name) VALUES ($1) RETURNING \*        `, ['Teste de Integração']);          const inserted = insertResult.rows[0];        const insertSuccess = inserted && inserted.name === 'Teste de Integração';          logTest("Inserção de dados", insertSuccess);          if (insertSuccess) {          console.log(`  Registro inserido com ID: ${inserted.id}`);        }          // Buscar dados        const selectResult = await db.query(`          SELECT \* FROM test\_temp WHERE id = $1        `, [inserted.id]);          const selected = selectResult.rows[0];        const selectSuccess = selected && selected.id === inserted.id;          logTest("Busca de dados", selectSuccess);          // Excluir dados        await db.query(`          DELETE FROM test\_temp WHERE id = $1        `, [inserted.id]);          const verifyResult = await db.query(`          SELECT COUNT(\*) as count FROM test\_temp WHERE id = $1        `, [inserted.id]);          const deleteSuccess = verifyResult.rows[0].count === '0';        logTest("Exclusão de dados", deleteSuccess);        } catch (error) {        logTest("Operações CRUD básicas", false, error);      }      // TESTE 5: Verificar se as tabelas principais existem      const requiredTables = [        'disciplina',        'professor',        'turma',        'aluno',        'administrador',        'questionario',        'questao'      ];        for (const table of requiredTables) {        try {          const result = await db.query(`            SELECT EXISTS (              SELECT FROM information\_schema.tables              WHERE table\_schema = 'public'              AND table\_name = $1            ) as exists          `, [table]);            const tableExists = result.rows[0].exists;          logTest(`Tabela ${table} existe`, tableExists);            if (!tableExists) {            console.log(`  ${colors.yellow}Aviso: A tabela '${table}' não foi encontrada.${colors.reset}`);          }        } catch (error) {          logTest(`Verificação da tabela ${table}`, false, error);        }      }        // TESTE 6: Verificar se a função de timezone foi executada      try {        const result = await db.query(`          SELECT current\_setting('timezone') as current\_tz        `);          const currentTz = result.rows[0].current\_tz;        const tzSuccess = currentTz === 'America/Sao\_Paulo';          logTest("Configuração de timezone na sessão atual", tzSuccess);        console.log(`  Timezone da sessão atual: ${currentTz}`);          if (!tzSuccess) {          console.log(`  ${colors.yellow}Aviso: O timezone da sessão deve ser 'America/Sao\_Paulo'${colors.reset}`);        }      } catch (error) {        logTest("Verificação de timezone da sessão", false, error);      }    } catch (error) {      console.error(`${colors.red}Erro geral nos testes: ${error.message}${colors.reset}`);    } finally {      // Imprimir resumo      printSummary();        // Fechar conexão com o banco      console.log("Fechando conexão com o banco de dados...");      await db.end();      console.log("Testes concluídos.");    }  }  // Executar os testes  runTests(); |

**server-test.js**

|  |
| --- |
| // tests/server-test.js - Testes simples para verificar se o servidor está funcionando  // Execução: node .\src\tests\server-test.js  // Importações  const http = require('http');  require('dotenv').config();  // Cores para console  const colors = {    reset: "\x1b[0m",    bright: "\x1b[1m",    red: "\x1b[31m",    green: "\x1b[32m",    yellow: "\x1b[33m",    blue: "\x1b[34m"  };  // Contador de testes  let passedTests = 0;  let failedTests = 0;  let totalTests = 0;  // Função para registrar resultados de testes  function logTest(name, success, error = null) {    totalTests++;    if (success) {      console.log(`${colors.green}✓ PASSOU: ${name}${colors.reset}`);      passedTests++;    } else {      console.log(`${colors.red}✗ FALHOU: ${name}${colors.reset}`);      if (error) {        console.log(`  ${colors.red}Erro: ${error.message || error}${colors.reset}`);      }      failedTests++;    }  }  // Função para resumo final  function printSummary() {    console.log("\n" + "=".repeat(50));    console.log(`${colors.bright}RESUMO DOS TESTES:${colors.reset}`);    console.log(`${colors.green}Passaram: ${passedTests}${colors.reset}`);    console.log(`${colors.red}Falharam: ${failedTests}${colors.reset}`);    console.log(`${colors.blue}Total: ${totalTests}${colors.reset}`);    console.log("=".repeat(50) + "\n");  }  // Função para fazer requisição HTTP  function makeRequest(path, options = {}) {    return new Promise((resolve, reject) => {      const PORT = 5000; // Porta padrão do servidor      const defaultOptions = {        hostname: 'localhost',        port: PORT,        path: path,        method: 'GET',        headers: {          'Content-Type': 'application/json'        },        timeout: 5000 // 5 segundos de timeout      };      // Mesclar opções      const requestOptions = { ...defaultOptions, ...options };        const req = http.request(requestOptions, (res) => {        let data = '';          // Receber dados        res.on('data', (chunk) => {          data += chunk;        });          // Finalizar resposta        res.on('end', () => {          try {            // Tentar fazer parse do JSON            const jsonData = data ? JSON.parse(data) : {};            resolve({              statusCode: res.statusCode,              headers: res.headers,              data: jsonData            });          } catch (error) {            // Se não for JSON, retornar como texto            resolve({              statusCode: res.statusCode,              headers: res.headers,              data: data            });          }        });      });      // Lidar com erros      req.on('error', (error) => {        if (error.code === 'ECONNREFUSED') {          reject(new Error(`Não foi possível conectar ao servidor na porta ${PORT}. Verifique se o servidor está rodando.`));        } else {          reject(error);        }      });      // Timeout      req.on('timeout', () => {        req.destroy();        reject(new Error(`Tempo limite excedido ao tentar conectar ao servidor na porta ${PORT}.`));      });      // Se houver corpo, enviar      if (options.body) {        req.write(JSON.stringify(options.body));      }        req.end();    });  }  // ========== TESTES ==========  async function runTests() {    console.log(`${colors.bright}${colors.blue}TESTES DO SERVIDOR${colors.reset}`);    console.log("=".repeat(50));    console.log("Executando testes...\n");    console.log(`${colors.yellow}IMPORTANTE: O servidor deve estar rodando na porta 5000.${colors.reset}`);    console.log(`${colors.yellow}Certifique-se de iniciar o servidor com 'node server.js' antes de executar este teste.${colors.reset}\n`);    try {      // TESTE 1: Verificar se o servidor está rodando      try {        const response = await makeRequest('/');        logTest("Servidor está rodando", response.statusCode >= 200 && response.statusCode < 500);        console.log(`  Código de status: ${response.statusCode}`);        console.log(`  Resposta: ${JSON.stringify(response.data)}`);      } catch (error) {        logTest("Servidor está rodando", false, error);        // Se o servidor não estiver rodando, interromper os testes        console.log(`${colors.red}Não foi possível conectar ao servidor. Encerrando testes.${colors.reset}`);        printSummary();        return;      }      // TESTE 2: Verificar rota de health check      try {        const response = await makeRequest('/health');        logTest("Rota de health check", response.statusCode === 200);        console.log(`  Código de status: ${response.statusCode}`);        console.log(`  Status do sistema: ${response.data.status || 'Não disponível'}`);        console.log(`  Status do banco: ${(response.data.database && response.data.database.connected) ? 'Conectado' : 'Não conectado'}`);      } catch (error) {        logTest("Rota de health check", false, error);      }      // TESTE 3: Verificar rota de autenticação (existência)      try {        // Apenas verificar se a rota existe, não se a autenticação funciona        // Tentativa com credenciais inválidas deve retornar 401, não 404        const response = await makeRequest('/api/auth/login', {          method: 'POST',          body: { email: 'teste@email.com', senha: 'senha\_invalida', tipo: 'professor' }        });          // Se retornar 404, a rota não existe        // Se retornar 401 ou qualquer outro código, a rota existe        const routeExists = response.statusCode !== 404;          logTest("Rota de autenticação existe", routeExists);        console.log(`  Código de status: ${response.statusCode}`);          if (response.statusCode === 401) {          console.log(`  Resposta esperada para credenciais inválidas (401 Unauthorized)`);        } else if (response.statusCode === 400) {          console.log(`  Validação de entrada funcionando (400 Bad Request)`);        }      } catch (error) {        // Se a rota não estiver registrada, ainda é um erro de conexão        logTest("Rota de autenticação existe", false, error);      }      // TESTE 4: Verificar CORS      try {        // Fazer uma requisição com Origin inválido        const response = await makeRequest('/', {          headers: {            'Origin': 'http://site-malicioso.com'          }        });          // Verificar se os headers CORS estão presentes        const corsHeaderExists = 'access-control-allow-origin' in response.headers;          logTest("Headers CORS configurados", corsHeaderExists);          if (corsHeaderExists) {          console.log(`  Access-Control-Allow-Origin: ${response.headers['access-control-allow-origin']}`);        } else {          console.log(`  ${colors.yellow}Aviso: Headers CORS não encontrados na resposta${colors.reset}`);        }      } catch (error) {        logTest("Verificação de CORS", false, error);      }      // TESTE 5: Verificar Rate Limiting      try {        console.log(`  ${colors.yellow}Fazendo múltiplas requisições para testar rate limiting...${colors.reset}`);          // Fazer 5 requisições rápidas        const results = await Promise.all([          makeRequest('/'),          makeRequest('/'),          makeRequest('/'),          makeRequest('/'),          makeRequest('/')        ]);          // Verificar se alguma resposta tem o header de rate limit        const hasRateLimitHeaders = results.some(result =>          'x-ratelimit-limit' in result.headers ||          'x-ratelimit-remaining' in result.headers        );          logTest("Rate limiting configurado", hasRateLimitHeaders);          if (hasRateLimitHeaders) {          // Mostrar headers da última resposta          const lastResponse = results[results.length - 1];          console.log(`  X-RateLimit-Limit: ${lastResponse.headers['x-ratelimit-limit'] || 'Não disponível'}`);          console.log(`  X-RateLimit-Remaining: ${lastResponse.headers['x-ratelimit-remaining'] || 'Não disponível'}`);        } else {          console.log(`  ${colors.yellow}Aviso: Headers de rate limit não encontrados${colors.reset}`);          console.log(`  ${colors.yellow}Isso pode ser normal se o limite for muito alto ou se não estiver configurado globalmente${colors.reset}`);        }      } catch (error) {        logTest("Verificação de rate limiting", false, error);      }    } catch (error) {      console.error(`${colors.red}Erro geral nos testes: ${error.message}${colors.reset}`);    } finally {      // Imprimir resumo      printSummary();      console.log("Testes concluídos.");    }  }  // Executar os testes  runTests(); |