1. Manual de Instalação

Para instalar e executar o sistema é preciso ter o node instalado no computador, caso não possua basta acessar esse link: https://nodejs.org/en/download, baixar o arquivo e seguir as instruções que serão dadas. Ao instalar o node basta executar o comando node --version para verificar se foi instalado corretamente na versão escolhida.
Com o node instalado, basta seguir os seguintes passos:

- Descompactar o zip
- Acessar a pasta "server" pelo terminal
- Executar o comando "npm i" para instalar as bibliotecas utilizadas
- Executar o comando "npm run start" para rodar o servidor

- Para rodar os testes basta executar o seguinte comando na pasta "server" pelo terminal: "npx jest"

Por ser um api, é possível testar os métodos através de algum programa que execute as requisições. Utilizamos o Postman para testar as nossas, mas é possível testar pelo próprio browser (apenas as funções que não requerem inserção de dados). Por exemplo, ao colocar o endereço "localhost:3000/api/cliente/retornaTodosClientes" no browser, o sistema retornara uma lista com todos os sistemas cadastrados. Todas as funções do tipo "GET" podem ser testadas pelo browser.

2. Implementação

a. Arquitetura

Para garantir uma implementação parcial eficiente do sistema de gerenciamento da oficina, concentramos nossos esforços no desenvolvimento do backend, que foi implementado como uma API REST. Essa abordagem permite que o frontend se conecte facilmente ao backend por meio das rotas criadas para cada função específica, tornando o backend independente do sistema em si.
A estrutura técnica do sistema é organizada em torno de rotas, controladores e modelos. As rotas são responsáveis por definir os pontos de extremidade da API e os métodos HTTP correspondentes, como GET, POST, PUT e DELETE. Cada rota é mapeada para um controlador específico, que contém a lógica de negócios necessária para processar a solicitação e retornar as respostas apropriadas.
Dentro dos controladores, os modelos são utilizados para interagir com o banco de dados e realizar operações de leitura, gravação, atualização e exclusão de dados através dos serviços, que seriam os métodos de cada objeto.
Ao seguir os princípios REST, utilizamos verbos HTTP adequados para cada ação, garantindo uma API consistente e intuitiva. Por exemplo, para obter informações sobre um cliente, utilizamos uma rota GET específica para clientes. Para criar um cliente, utilizamos uma rota POST na mesma rota de clientes.

Além disso, é importante mencionar que a implementação parcial do sistema de oficina permite uma modularidade eficiente. À medida que novas funcionalidades são desenvolvidas, basta adicionar novas rotas, controladores e serviços conforme necessário, mantendo uma separação clara entre as diferentes partes do sistema.

Por meio dessa arquitetura, nosso sistema de gerenciamento da oficina se torna escalável, flexível e de fácil manutenção. O backend funciona como um serviço independente, pronto para ser conectado e utilizado por diferentes frontends, como aplicativos web e móveis.

b. Ferramenta Utilizadas

A implementação do nosso sistema foi realizada utilizando Node.js em conjunto com TypeScript. Essa escolha foi baseada em diversos fatores estratégicos e técnicos. O Node.js tem ganhado crescente popularidade e se estabelecido como uma das principais linguagens de desenvolvimento no mercado atual. Além disso, a equipe responsável pela implementação já possuía experiência e familiaridade com essa linguagem, o que facilitou o processo de desenvolvimento.

Optamos por utilizar TypeScript em vez de JavaScript devido às suas vantagens em relação à tipagem estática. O TypeScript é um superconjunto do JavaScript que adiciona recursos de tipagem estática, fornecendo uma camada adicional de segurança e robustez ao código. Essa abordagem nos permite detectar erros de tipagem em tempo de compilação, o que facilita a identificação e correção de possíveis problemas antes mesmo da execução do sistema.

Além disso, optamos por utilizar o framework Sequelize para estabelecer a conexão com o banco de dados. Essa escolha foi baseada na familiaridade prévia da equipe com o Sequelize e nos benefícios oferecidos por esse framework. O Sequelize é um ORM (Object-Relational Mapping) para Node.js que simplifica a interação com bancos de dados relacionais, como o SQLite.

Ao utilizar o Sequelize, podemos aproveitar as funcionalidades fornecidas pelo framework para criar, consultar, atualizar e excluir registros no banco de dados de forma mais intuitiva e eficiente. O Sequelize também oferece recursos de validação de dados, mapeamento de objetos para tabelas do banco de dados e gerenciamento de relacionamentos entre as entidades.

Para fins de implementação parcial e testes, decidimos criar um banco de dados local utilizando o SQLite. No entanto, é importante ressaltar que, em um ambiente de produção, seria necessário configurar um banco de dados adequado e seguro para atender às demandas de escalabilidade e segurança.

c. Estrutura

A estrutura do código do sistema de gerenciamento da oficina é organizada da seguinte maneira:

1. Pasta "Eletro Master": Essa pasta abriga todo o código relacionado ao sistema de gerenciamento da oficina.
2. Pasta "server": Dentro da pasta "Eletro Master", a pasta "server" contém todos os arquivos relacionados ao backend do sistema.

3. Arquivos de configuração: Nessa pasta, você encontrará arquivos de configuração das bibliotecas utilizadas, como "jest.config.js", "package-lock.json", "package.json" e "tsconfig.json".
4. Pasta "coverage": Essa pasta contém os arquivos de configuração da biblioteca de testes utilizada, o Jest.
5. Pasta "src": A pasta "src" contém a implementação do sistema de gerenciamento da oficina.
6. Arquivo "server.ts": Esse arquivo é responsável pela inicialização do servidor.
7. Pasta "server-config": Essa pasta contém os arquivos de configuração específicos do servidor.
8. Pasta "database": Dentro dessa pasta, você encontrará os arquivos de configuração relacionados ao banco de dados. Isso inclui a configuração de conexão com o banco de dados, definição de modelos e outras configurações relacionadas ao banco de dados utilizado.
9. Pasta "entidade": Essa pasta contém a implementação dos objetos do sistema, incluindo seus métodos e rotas da API.
    a. Pasta "controllers": Aqui estão localizadas as rotas da API que serão consumidas pelo frontend. Os controladores são responsáveis por receber as solicitações, executar a lógica de negócios necessária e retornar as respostas adequadas.
    b. Pasta "models": Nessa pasta, estão as declarações das classes que representam as entidades do sistema. Os modelos definem os atributos das classes e seus relacionamentos com outras entidades.
    c. Pasta "services": Essa pasta contém a implementação dos serviços, que são os métodos das classes que são chamados pelas rotas da API. Os serviços encapsulam a lógica de negócios e interagem com o banco de dados, se necessário.

Essa estrutura do código permite uma organização clara e modular do sistema, facilitando a manutenção e expansão futura

3. Testes

a. Ferramenta utilizada

Para realizar testes unitários automatizados, escolhemos a biblioteca Jest. O Jest é uma biblioteca de testes amplamente utilizada no ecossistema do Node.js, que possui uma sintaxe simples e intuitiva para escrever testes unitários.

Ao optar pelo Jest, podemos aproveitar uma série de recursos que ele oferece para facilitar a escrita, execução e análise de testes unitários. Alguns desses recursos incluem:

- Sintaxe concisa: O Jest oferece uma sintaxe clara e concisa para descrever casos de teste, expectativas e asserções. Isso torna a escrita dos testes mais legível e fácil de entender.
- Conjunto completo de ferramentas: O Jest inclui uma ampla gama de ferramentas para testes unitários, como asserções, spies (espiões) para

simular comportamentos de funções e módulos, e mocks (simulações) para isolar componentes do sistema durante os testes.

- Cobertura de código: O Jest também possui recursos integrados para medir a cobertura de código dos testes. Isso permite identificar partes do código que não estão sendo testadas e garantir uma cobertura adequada dos testes.
- Velocidade de execução: O Jest é conhecido por sua velocidade de execução, o que é essencial para manter uma suíte de testes eficiente e rápida.

Com a biblioteca Jest, podemos automatizar a execução de testes unitários para garantir que as diferentes partes do sistema, como controladores, modelos e serviços, estejam funcionando corretamente. Os testes unitários nos ajudam a identificar erros, garantir a integridade do código e fornecer confiança na qualidade do sistema.
Jest é um framework criado pelo Facebook para realizar testes unitários em seu sistema e rapidamente se tornou uma das mais utilizadas em sistemas que utilizam Node.Js.

b. Estrutura

Para realizar os testes, implementamos uma estrutura organizada dentro da pasta "services" de cada entidade. Essa abordagem permite testar os métodos de cada classe de forma isolada e precisa.
Para cada classe, foram criados testes para todos os métodos possíveis de serem chamados. A estrutura dos testes segue o seguinte padrão:

- "describe": O teste é descrito de forma clara e concisa, fornecendo informações sobre o que está sendo testado.
- "test": Dentro do bloco de "describe", são definidos testes específicos para cada função a ser testada. É possível ter mais de um teste para uma função, abrangendo diferentes cenários e casos de uso.
- "assert": Os resultados esperados do teste são definidos usando asserções. Isso pode incluir a verificação do valor de um atributo, a contagem de vezes que uma função foi chamada ou os parâmetros que foram passados para uma função específica.

Essa estrutura de teste permite uma organização clara dos casos de teste e facilita a identificação de problemas durante a execução dos testes. Ao verificar os resultados esperados com os resultados reais, é possível determinar se o comportamento do código está correto e se os métodos estão produzindo os resultados desejados.

c. Resultados Obtidos

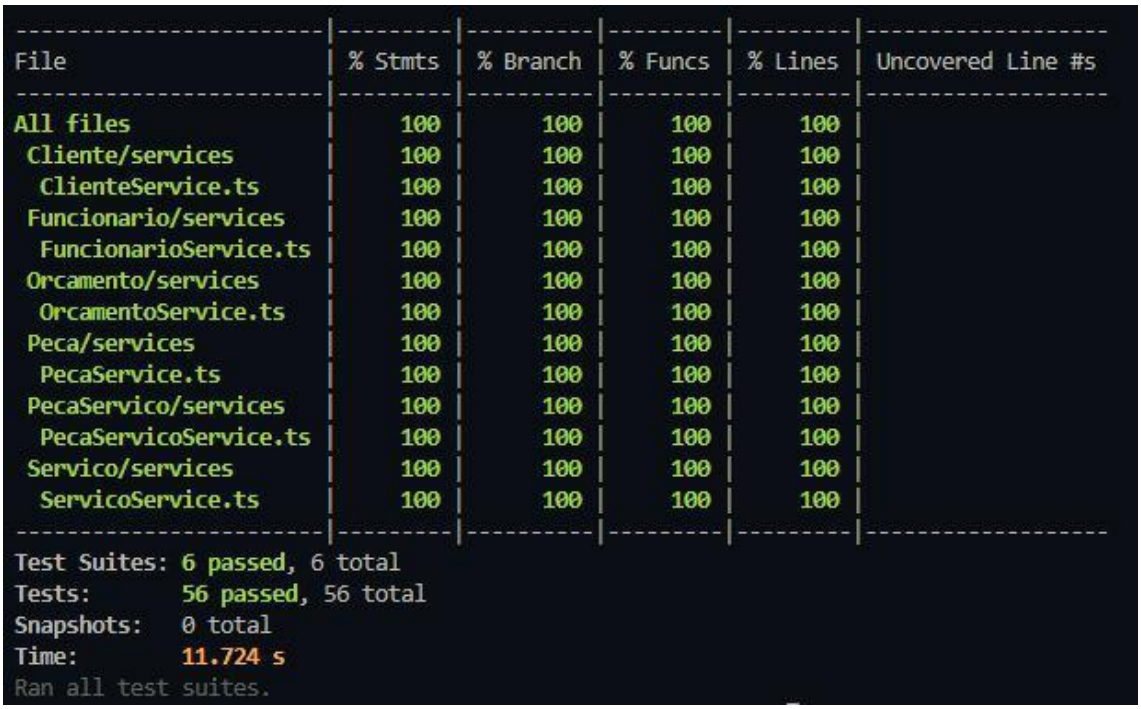
Alcançamos uma cobertura de teste de 100% em todos os aspectos medidos pelo Jest, garantindo uma abrangência completa dos testes. Os aspectos avaliados incluem:

- Statments: Todos os statements (instruções) do código foram testados, assegurando que cada trecho de código tenha sido executado e verificado.

- Branch: Todos os caminhos possíveis do fluxo do sistema foram testados. Isso inclui a verificação de todos os caminhos condicionais e de ramificação para garantir que todas as decisões sejam tratadas corretamente.
- Functions: Todas as funções implementadas no código foram testadas. Isso inclui a execução dos testes para cada função, garantindo que elas produzam os resultados esperados e se comportem corretamente.
- Lines: Todas as linhas de código relevantes foram testadas, garantindo uma cobertura completa em termos de execução de código.

Essa alta cobertura de teste indica que todas as partes essenciais do sistema foram submetidas a testes automatizados e que os resultados obtidos foram validados em conformidade com as expectativas.

É importante ressaltar que alcançar uma cobertura de teste completa não garante a ausência total de erros no código. No entanto, uma cobertura abrangente aumenta significativamente a confiança na qualidade do sistema, permitindo a identificação precoce de problemas e auxiliando na manutenção do código no futuro.



File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
Cliente/services	100	100	100	100	
ClienteService.ts	100	100	100	100	
Funcionario/services	100	100	100	100	
FuncionarioService.ts	100	100	100	100	
Orcamento/services	100	100	100	100	
OrcamentoService.ts	100	100	100	100	
Peca/services	100	100	100	100	
PecaService.ts	100	100	100	100	
PecaServico/services	100	100	100	100	
PecaServicoService.ts	100	100	100	100	
Servico/services	100	100	100	100	
ServicoService.ts	100	100	100	100	

```
Test Suites: 6 passed, 6 total
Tests:       56 passed, 56 total
Snapshots:   0 total
Time:        11.724 s
Ran all test suites.
```