



# Bootcamp: Bootcamp Arquiteto(a) de Software

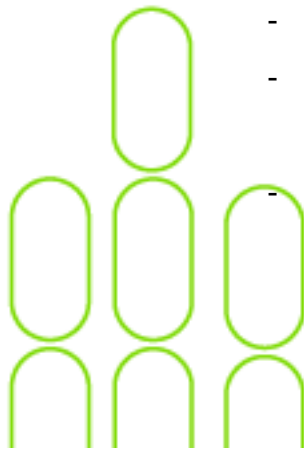
Aluno(a): Raphael de Oliveira Moura

## Relatório de Entrega do Desafio Final

### Desenvolvimento da Solução do Desafio Final

Para a resolução do desafio final, foi criado um repositório no **github**: <https://github.com/RaphaelOliveiraMoura/bootcamp-arquitetura-software-desf5?tab=readme-ov-file> onde no README.md possui mais detalhes técnicos da resolução da construção e documentação da API REST desenvolvida.

Para a construção dos endpoints da API, selecionei o domínio de **Clientes**, foi construído rotas para realizar todas operações de CRUD em cima desse domínio. Segue abaixo os endpoints desenvolvidos na api:

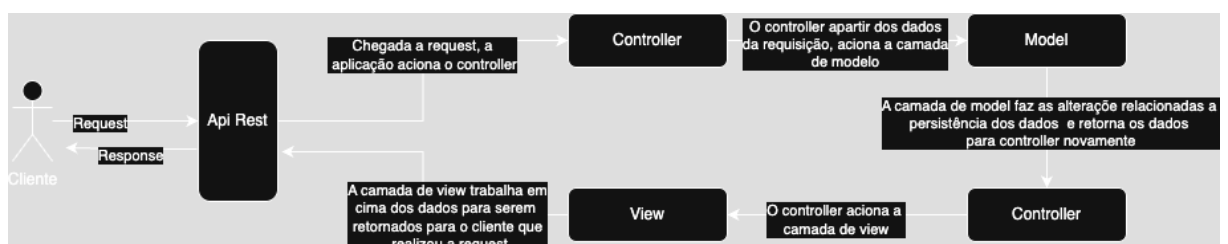
- **GET /clients**: lista todos os clientes do sistema
  - **GET /clients/:id**: retorna detalhes de um cliente específico, baseado no seu id
  - **GET /clients/by-name**: retorna detalhes de um cliente específico, baseado no seu nome
  - **GET /clients/count**: retorna o total de clientes cadastrados na base
  - **POST /clients**: realiza o cadastro de um cliente no sistema
  - **PUT /clients/:id**: realiza a atualização de dados de um cliente, baseado no seu id
  - **DELETE /clients/:id**: realiza a deleção de um cliente, baseado no seu id
- 


Sobre as tecnologias utilizadas, optei por utilizar a linguagem **typescript**, utilizando **nodejs + fastify** para criação do servidor http. Para banco de dados optei por utilizar o banco de dados **postgres**. Para executar o projeto localmente sem a necessidade de instalar todo esse ferramental manualmente, optei por utilizar o **docker**, como meio mais simples de executar o projeto em diferentes ambientes, ou seja, para executar o projeto basta ter o **docker** e **docker-compose** instalados. Lembrando que no [github](https://github.com) possui todas as instruções de como configurar o projeto e executá-lo localmente.

Como ferramenta para documentar os endpoints desenvolvidos optei por usar o **swagger**. Com ele, rodando o projeto localmente, é possível visualizar a documentação de cada rota, além de facilitar nos testes, pois você consegue realizar requisições a partir dessa interface.

POST	/clients	Criar um novo cliente	▼
GET	/clients	Listar todos os clientes	▼
GET	/clients/{id}	Buscar cliente por ID	▼
PUT	/clients/{id}	Atualizar cliente por ID	▼
DELETE	/clients/{id}	Deletar cliente por ID	▼
GET	/clients/by-name/{name}	Buscar clientes por nome	▼
GET	/clients/count	Contar total de clientes	▼

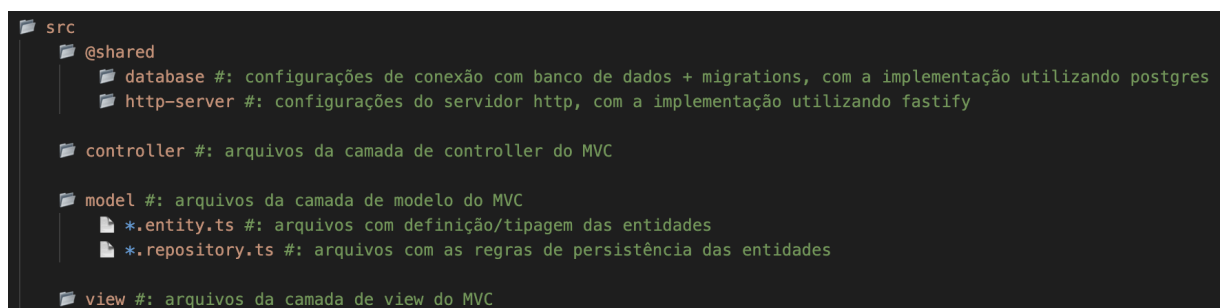
Referente ao padrão utilizado no projeto, foi utilizado o MVC (Model, View, Controller). Segue abaixo o fluxo de dados baseado em cada camada da aplicação:





Esse diagrama foi desenvolvido utilizando a ferramenta **draw.io**. Nele é possível visualizar o fluxo das informações em cada camada da API. Onde a requisição parte do **cliente**, passando assim para a camada de **Controller**, onde a requisição é tratada e aplicada as devidas regras de fluxo das informações, delegando assim para a camada de **Model** onde é realizado a ligação direta com o banco de dados utilizando o padrão de **Repository**, depois de persistidos/ou recuperado os dados a partir da camada **Model**, ela retorna as devidas informações novamente para o **Controller**, que por sua vez aciona a **View**, para tratar os dados a serem retornados para o cliente, e após isso, é de fato finalizado com a resposta enviada para o **cliente**.

Segue abaixo como foi definido a estrutura de pastas do projeto:



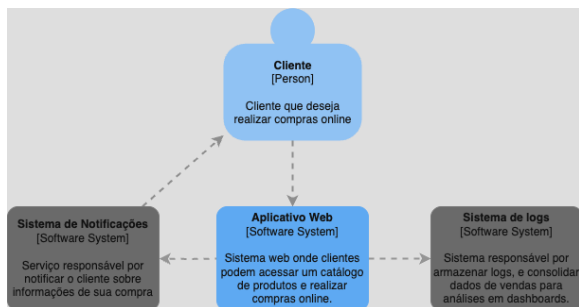
```
src
├── @shared
│   ├── database #: configurações de conexão com banco de dados + migrations, com a implementação utilizando postgres
│   └── http-server #: configurações do servidor http, com a implementação utilizando fastify
├── controller #: arquivos da camada de controller do MVC
├── model #: arquivos da camada de modelo do MVC
│   ├── *.entity.ts #: arquivos com definição/tipagem das entidades
│   └── *.repository.ts #: arquivos com as regras de persistência das entidades
└── view #: arquivos da camada de view do MVC
```

Além disso utilizando o draw.io fiz alguns diagramas c4 da aplicação, para alguns deles acabei adicionando algumas informações fictícias apenas para fins didáticos, como por exemplo no diagrama de contexto onde simulei a necessidade de serviços de notificações e logs, apenas para exemplificar como seria um sistema online de vendas real.

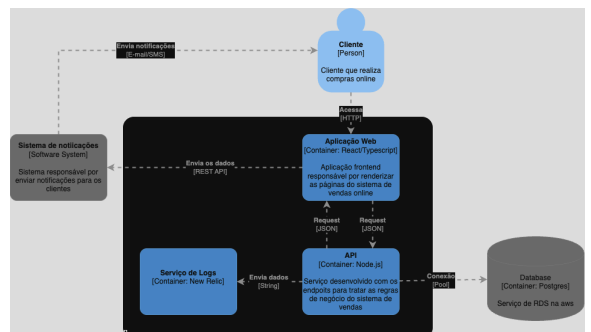
Segue abaixo os diagramas c4:



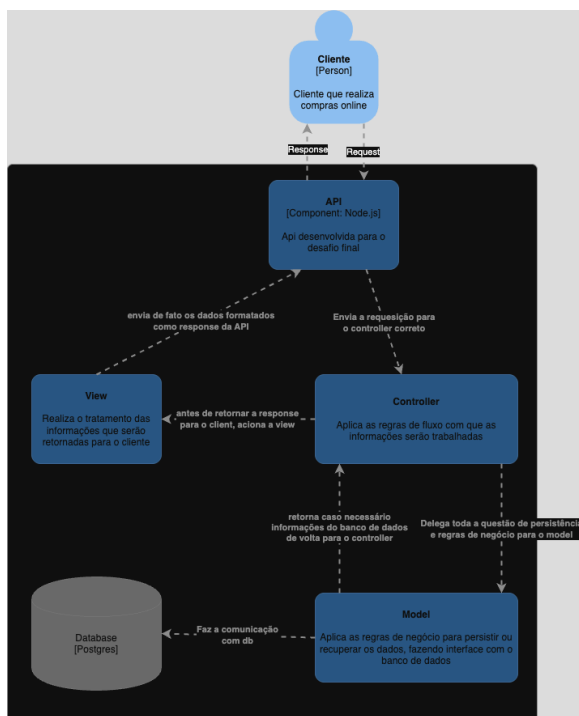
## contexto



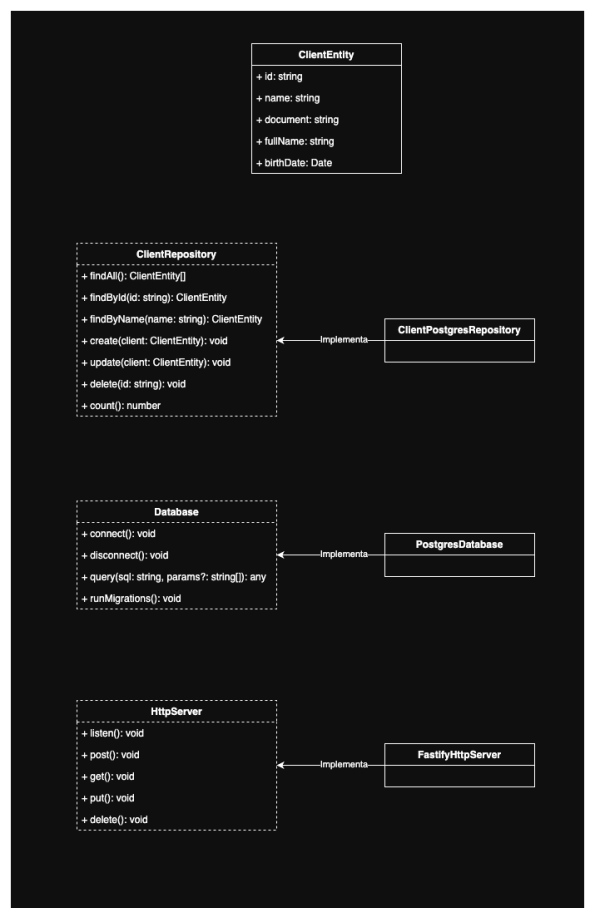
## container



## componente



## classe





## Conclusão

Para a resolução do desafio, optei por utilizar ferramentas que tenho mais familiaridade para construção da API. Tentei a todo momento utilizar aspectos abordados durante o bootcamp, onde pude explorar alguns conceitos de SOLID, boas práticas, uso de design patterns no desenvolvimento da API. Além disso tive que revisar alguns conteúdos referentes à documentação e criação de diagramas, como foi o caso do C4, tema esse que eu não tinha tanta familiaridade. Sobre o resultado final, acredito que consegui atingir um resultado satisfatório, onde consegui aprofundar em bastantes conceitos explorados durante o decorrer do curso. Em relação ao desenvolvimento da API em si, creio que tem bastante abertura para melhorias no código, como por exemplo, explorar melhor a questão de validação de dados, performance, talvez alguns aspectos de segurança, mas como o foco não era esse, optei por não aprofundar tanto nesses aspectos. Tentei ao máximo criar uma solução que seja simples de entender, e ao mesmo tempo, que tenha uma certa profundidade baseado nos temas estudados.

