**Aplicação e estrutura de pastas**

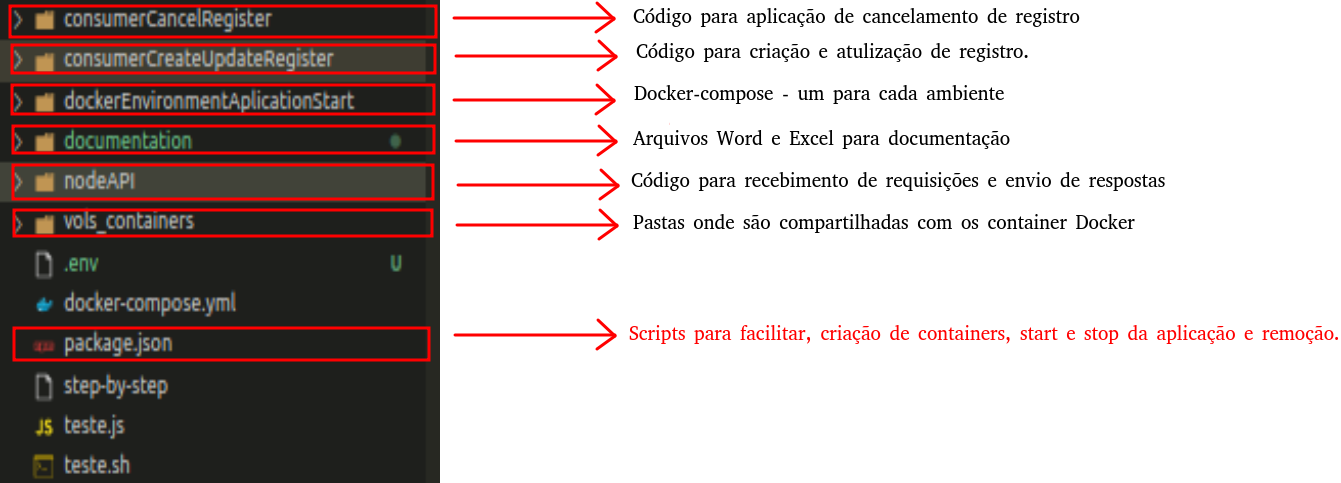
***Versão***

DOCKER:   
 **Client-version**: 20.10 (recomendado)  
 **Server-engine-version**: 20.10 (recomendado)

DOCKER COMPOSE:   
 **Versão**: v2.5.1 (recomendado)

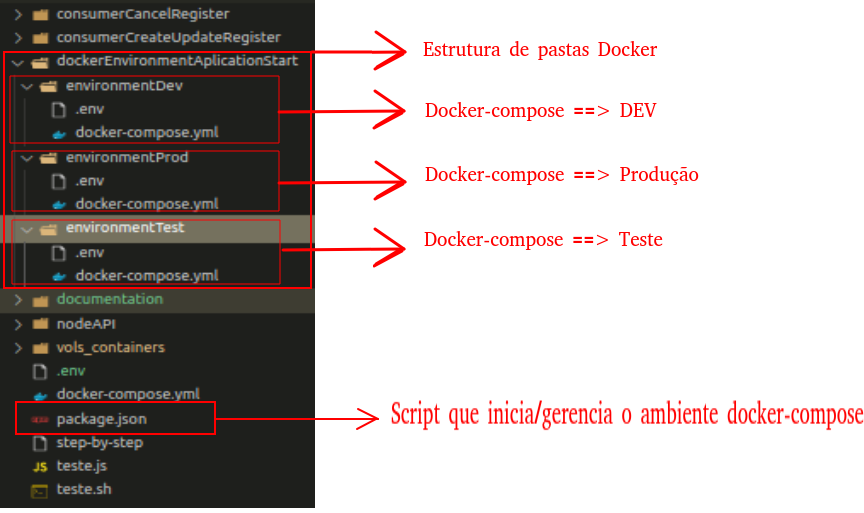
S.O:   
 **Versão**: Qualquer distribuição

**Estrutura de pastas:**

****

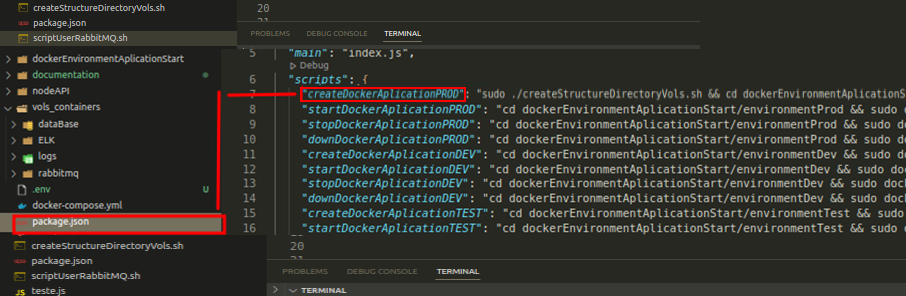
**Docker**

Cada ambiente possui seu arquivo docker-compose junto com seu .env que possui as variáveis de ambiente.  
Para iniciar a aplicação, basta executar o script package-json na raíz do projeto de acordo com o ambiente desejado:

****

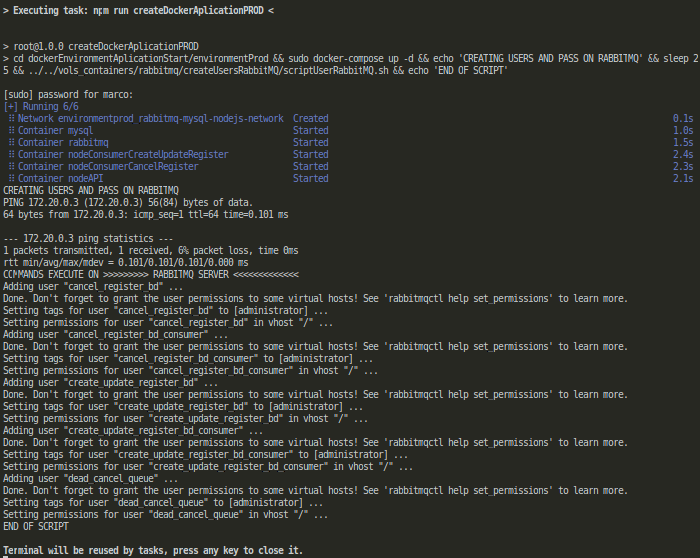
**Iniciar aplicação para Produção**

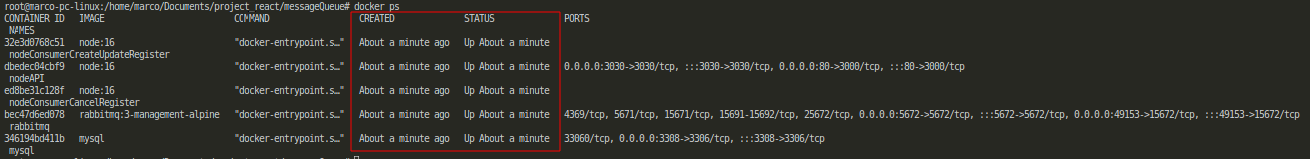
**1: Criar os containers e deixá-los em execução 🡺 *createDockerAplicationPROD***

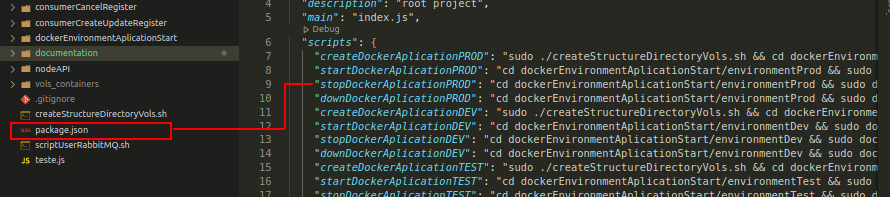
****

* Inicia script *createDockerAplicationPROD.sh* que irá criar (se não existir) a estrutura de volumes usados pelos containers
* Irá até a pasta do docker compose Prod e executará o *docker-compose up –d*
* Após todos os containers iniciarem, executa o script *scriptUserRabbitMQ.sh*
* *Script RabbitMQ tem as seguintes funções:*- Primeiramente o Script verifica qual servidor está ativo: desenvolvimento, teste ou produção.  
  - Acessa o servidor e executa comandos para criar usuários, senhas e permissões no RabbitMQ Server

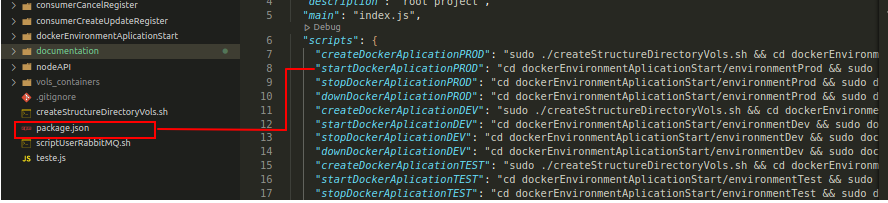
*=>* ***Durante toda execução para criação do ambiente, será mostrado no prompt informações importante das execuções.***

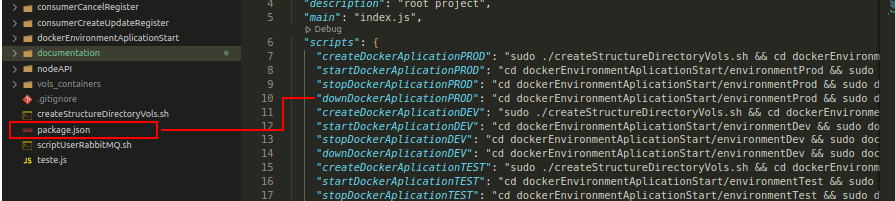


Após a finalização, com o comando “docker ps” podemos ver os servidores em execução:  
  


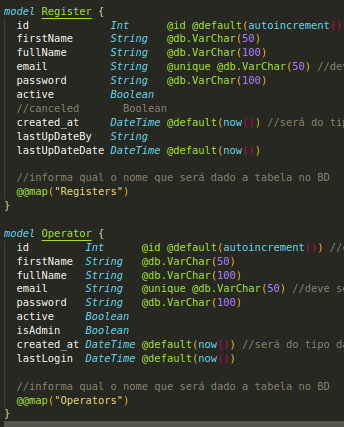
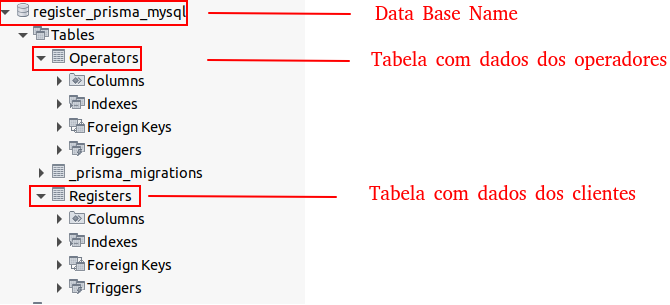
**2: Parar execução dos containers:**  
Para parar os containers, execute stopDockerAplicationPROD  


**3: Iniciar execução dos containers:**

Para iniciar os containers, execute ***startDockerAplicationPROD***  


**4: Remoção dos containers:**  
Para parar e remover os containers, execute ***downDockerAplicationPROD***  
  
  
 **5: Inicialização da aplicação MYSQL - **  
**servidor: Mysql**Após sua inicialização o servidor se comunica através da **porta 3306**, mas recebe na **porta 3308**, onde o Docker faz o redirect.  
Volume: Sincroniza o banco de dados localizado no caminho vols\_containers/dataBase/dataBaseProd para pasta interna /var/lib/mysql

Informações do banco de dados:  
**user: root  
pass: register**

****  


**6: Inicialização da aplicação HABBITMQ -  
servidor: rabbitmq**  
Após sua inicialização o servidor se comunica através da **porta 15672 para gerenciamento WEB e porta 5672** para comunicação via protocolo AMQP.  
Volume:   
Sincroniza as configurações localizadas no caminho ~/docker-conf/rabbitmq/data/ para pasta interna /var/lib/rabbitmq/  
  
Durante a criação do container, são criado credenciais de acesso através do script de inicialização localizado no caminho: *vols\_containers/rabbitmq/createUsersRabbitMQ/scriptUserRabbitMQ.sh.*   
  
As credenciais são utilizadas pela aplicação para abrir conexão, enviar, consumir dados e encerrar:  
  
Acesso Web:  
 \* **user**: register  
 \* **pass**: register

Acesso via protocolo AMQP:  
 \* **user**: cancel\_register\_bd  
 \* **pass**: cancel\_register\_bd  
 \* **user**: cancel\_register\_bd\_consumer  
 \* **pass**: cancel\_register\_bd\_consumer  
 \* **user**: create\_update\_register\_bd  
 \* **pass**: create\_update\_register\_bd  
 \* **user**: create\_update\_register\_bd\_consumer  
 \* **pass**: create\_update\_register\_bd\_consumer  
 \* **user**: dead\_cancel\_queue  
 \* **pass**: dead\_cancel\_queue  
  
Todos user são administradores com permissão full.  
Obs: Assim que o container é criado, o script aguarda 25 segundo para sua execução, pois é necessário aguardar para que o RabbitMQ Server iniciei todas sua funções e então assim aceite alterações de configurações.

Filas:  
create\_update\_register\_bd => recebe dados para criação e atualização de registro  
cancel\_register\_bd => recebe dados para cancelamento de registro  
dead\_cancel\_queue => recebe dados de solicitações de cancelamentos onde o registro já possui o cancelamento solicitado  
  
Sistema Remote procedure call (RPC)  
  
As aplicações Node são configuradas para utilizar o sistema RPC.  
Sempre que o servidor de origem envia algo para uma fila para que alguma aplicação possa receber e tratar, a origem fica aguardando uma resposta.  
E para isso é utilizado o sistema RPC.  
Para cada requisição, antes dos dados serem encaminhados para uma fila no RabbitMQ é criado uma fila temporária de retorno.   
A origem envia os dados para a fila e fica aguardando o retorno na fila temporária.  
Quando o servidor de destino obtém os dados do RabbitMQ ele irá tratar os dados e retornar para a fila temporária. O servidor de origem recebe os dados dessa fila temporária, valida se a chave é a mesma e responde ao operador solicitante com a mensagem da operação solicitada.

**7: Inicialização da aplicação – nodeAPI   
servidor: nodeAPI**Após sua inicialização o servidor se comunica através da **porta 80**, mas recebe na **porta 3000**, onde o Docker faz o redirect.  
Volume:   
Sincroniza os LOGS localizado no caminho vols\_containers/logs para pasta interna /home/node/app/logs  
Sincroniza os CÓDIGOS da aplicação localizado no caminho nodeAPI para pasta interna /home/node/app  
  
Requisições HTTP  
O servidor nodeAPI é responsável por receber requisições HTTP e respondê-las.  
Após o serviço HTTP ser iniciado, disponibiliza as seguintes rotas:  
 \* /login => operador se autentica  
 \* /register => operador cria ou atualiza registro  
 \* /cancel => operador cancela registro

Filas RabbitMQ  
Somente o servidor nodeAPI é responsável durante sua inicialização pela criação das 3 filas RABBIMQ:  
create\_update\_register\_bd  
cancel\_register\_bd  
dead\_cancel\_queue  
Caso ocorra algum erro na tentativa, ele irá fazer uma nova tentativa a cada 5 segundos até o sucesso.

Abertura e fechamento de conexões RabbitMQ  
Para cada requisição o nodeAPi server irá abrir uma nova conexão e enviar os dados para a fila RabbitMQ responsável.  
Para cada conexão é designado um tempo de vida: 20 segundos  
Após 20 segundos a conexão será encerrada

Autenticação com TOKEN  
O servidor nodeAPI possui sistema de autenticação baseada em banco de dados e Json Web Token  
O token é liberado após a autenticação e sempre validado em qualquer outra operação

Banco de dados  
O servidor nodeAPI utiliza o banco de dados somente para validação de credenciais.  
Durante sua inicialização o servidor nodeAPI através da lib Prisma executa ***prisma generate*** (criação do client) e ***prisma db pull*** (Se não existir, cria o schema configurado no banco de dados)  
Quando é iniciado, o servidor nodeAPI cria, **se não existir,** um usuário operador padrão (user: [registerdefaultuser@register.com](mailto:registerdefaultuser@register.com) / senha: Register@jU0OTU4Nzc1fQ). Que é utilizado principalmente pelo ambiente de teste.

**8: Inicialização da aplicação – nodeConsumerCreateUpdateRegister   
servidor: nodeConsumerCreateUpdateRegister**Após sua inicialização o servidor não possui portas abertas para comunicação.  
  
Volume:   
Sincroniza os LOGS localizado no caminho vols\_containers/logs para pasta interna /home/node/app/logs  
Sincroniza os CÓDIGOS da aplicação localizado no caminho consumerCreateUpdateRegister para pasta interna /home/node/app

O servidor **nodeConsumerCreateUpdateRegister** é responsável por consumir dados que estão na fila *create\_update\_register\_bd* do servidor RabbitMQ, executar e responder.  
Ao obter os dados, é verificado primeiramente se o operador é admin, se sim, executar a solicitação e em seguida enviar a resposta para a fila RPC.  
  
RabbitMQ  
Diferente do servidor nodeAPI, o servidor nodeConsumerCreateUpdateRegister não cria uma conexão para cada requisição, mas sim uma única para todas requisições.  
Caso ocorra algum erro na tentativa de criação ou erro durante a execução, ele irá fazer uma nova tentativa a cada 5 segundos até o sucesso.

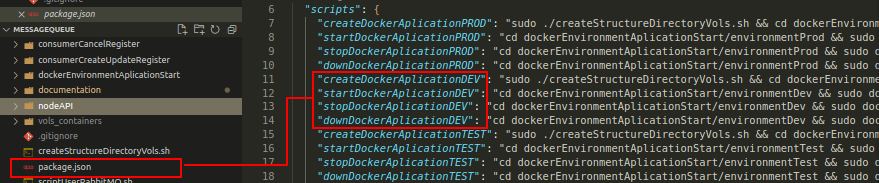
Banco de dados  
O servidor nodeAPI utiliza o banco de dados para verificar se o operador é Admin e criar/alterar registros.  
Durante sua inicialização o servidor nodeAPI através da lib Prisma executa ***prisma generate*** (criação do client).

**9: Inicialização da aplicação – nodeConsumerCancelRegister  
servidor: nodeConsumerCancelRegister**Após sua inicialização o servidor não possui portas abertas para comunicação.  
  
Volume:   
Sincroniza os LOGS localizado no caminho vols\_containers/logs para pasta interna /home/node/app/logs  
Sincroniza os CÓDIGOS da aplicação localizado no caminho consumerCancelRegister para pasta interna /home/node/app

O servidor **consumerCancelRegister**é responsável por consumir dados que estão na fila *cancel\_register\_bd* do servidor RabbitMQ, executar e responder.  
Em alguns casos, poderá enviar dados para a fila *dead\_cancel\_queue*  
Ao obter os dados, é verificado primeiramente se o operador é admin, se sim, executar a solicitação e em seguida enviar a resposta para a fila RPC.  
Se o registro já estiver desativado, os dados serão enviados para a fila *dead\_cancel\_queue* e para a fila de retorno RPC.  
  
RabbitMQ  
Diferente do servidor nodeAPI, o servidor nodeConsumerCreateUpdateRegister não cria uma conexão para cada requisição, mas sim uma única para todas requisições. Exceto para a fila *dead\_cancel\_queue,* para esta fila, uma conexão para cada requisição é criada. Tempo de vida: 20 segundos para cada conexão  
Caso ocorra algum erro na tentativa de criação ou erro durante a execução, ele irá fazer uma nova tentativa a cada 5 segundos até o sucesso.

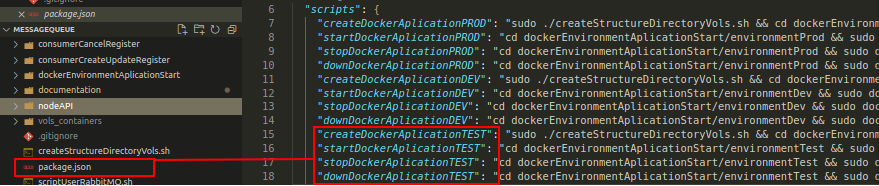
Banco de dados  
O servidor nodeAPI utiliza o banco de dados para verificar:  
Se o operador é admin  
Se o registro está ativo  
Alterar os dados  
Durante sua inicialização o servidor nodeAPI através da lib Prisma executa ***prisma generate*** (criação do client).

**Iniciar aplicação para Desenvolvimento**



O método de criação, inicialização, parada e remoção para o a ambiente de DESENVOLVIMENTO seguem as mesmas regras descritas em produção.

**Iniciar aplicação para Teste**

****

O método de criação, inicialização, parada e remoção para o a ambiente de TESTE seguem as mesmas regras descritas em produção, ***exceto*** para criação dos container 🡺

Durante a execução do script para criação (createDockerAplicationTEST) são criados a estrutura dos volumes, os containers e executado o script RabbbiMQ. Após o término os containers são parados e iniciados novamente pelos seguintes motivos:

* Os Testes que validam rotas e funções não aguardam o script do RabbitMQ executar, fazendo com que apresente erros durante a execução dos testes.
* Para evitar isso, após todas as tarefas para criação e configuração dos containers serem executadas, os containers são reiniciados, pois quando ativam novamente, já estão configurados e prontos para serem testados.
* Após criação e o restart, mesmo desligando e ligando novamente os containers os testes executarão sem erro conforme abaixo:

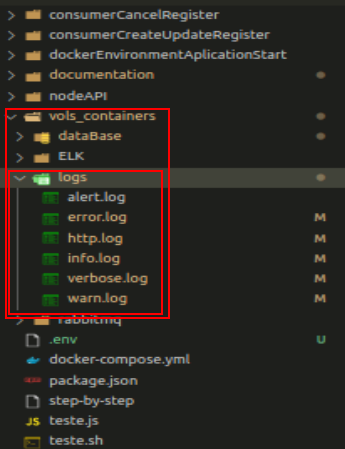


**Autenticação  
**

A autenticação é baseada no sistema de **JSONWebToken**.  
A palavra chave do token varia conforme o ambiente.  
ALG: HS256  
TYP: JWT  
DURAÇÃO: 24 HS

**LOGS  
**

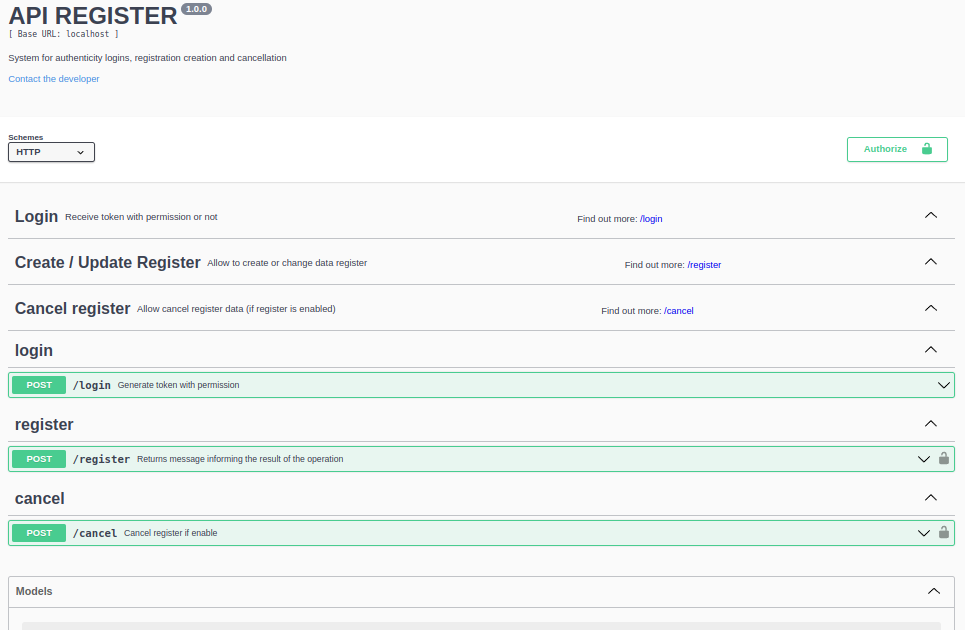
Os 3 containers de aplicação NODEJS utilizam sistema de logs usando a biblioteca **Winston**.  
Os logs são **registrados em JSON**, para uma melhor leitura das aplicações e divididos em níveis para uma melhor organização.



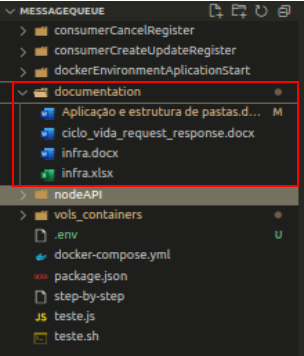
Conforme a imagem acima, a pasta vols\_containers/logs é compartilhada com os containers e armazena os logs da aplicação.  
OBS: Somente o ambiente de **DESENVOLVIMENTO** não registra os logs, mas sim notificam todos os alertas no console.

**Documentação  
**

A aplicação utiliza o **SWAGGER** para documentar a API.



Conforme a imagem acima, podemos testar pela própria aplicação SWAGGER requisições para todas as rotas.  
**OBS**: A documentação SWAGGER só fica disponível em ambiente de **desenvolvimento**.  
Rota: <http://localhost:3030/documentation>

Há também docs e planilha onde há mais detalhes do funcionamento da aplicação:  
  
  


**Leitura escrita Banco de dados  
**

As aplicações Node utilizam a Lib Prisma para leitura e escrita de dados no DataBase e também criações de migrations.  
**Ambiente Produção:** utiliza as configurações .env  
**Ambientes de desenvolvimento e teste:** utiliza as configurações .env.dev