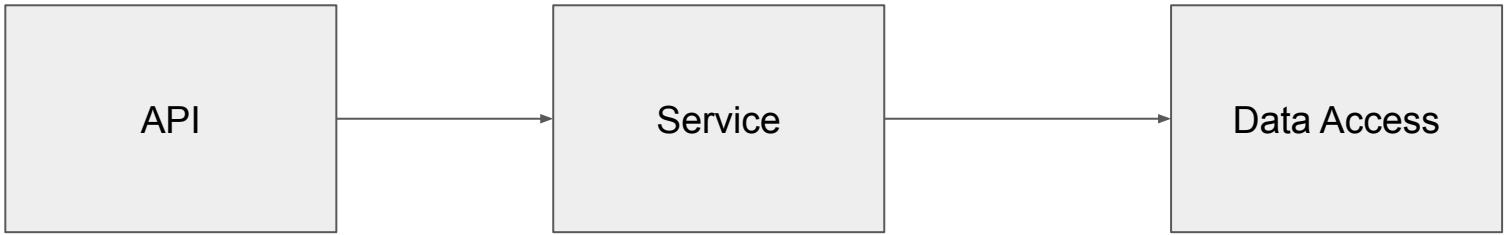
Node.js
Arquitetura e Design Patterns

Camadas de uma aplicação backend

- API / Controller
- Service (Lógica de negócio)
- Data Access (Banco de dados, cache, etc)



Camada API

- Representa os pontos de entrada para a sua aplicação
  - Endpoints
  - Conexões websocket
  - Comandos de terminal
  - Operações batch
  - Eventos
- Responsável por
  - Validação de inputs
  - Autorização de acesso e perfilamento
  - Logs, Rate limiting
- Delega para a camada de service
- Não deve possuir lógica de negócio

Camada Service

- É responsável por toda a lógica de negócio
- Interage com a camada de dados (Data Access)
- Serviços podem ser reutilizados de diversas formas pela camada de API

Camada Data Access

- É responsável por abstrair a camada de dados
  - Banco de dados
  - Cache
  - Outras aplicações
  - Aplicações externas

Chain of Responsibility

- Padrão implementado pela maioria das bibliotecas HTTP do Node.js
- Padrão que passa uma request por diferentes handlers que fazem algum processamento na requisição e podem passar ela adiante para o próximo handler na cadeia.
- Os handlers que não passam adiante a request podemos considerar como os “controllers”, pois são os responsáveis por retornar a resposta ao cliente.
- Os handlers que fazem algum processamento e passam adiante podemos considerar como “middlewares”.
    - Middleware de CORS
    - Middleware para validação de token autenticado

https://refactoring.guru/design-patterns/chain-of-responsibility

12 Factor-App

- Formato declarativo para automatizar a configuração inicial
- Contrato claro com o sistema operacional que o suporta
- Adequado para implantação em plataformas cloud
- Minimiza a divergência entre ambientes de desenvolvimento e produção
- Pode escalar facilmente

12 Factor-App

1. Base de Código
2. Dependências
3. Configurações
4. Serviços de Apoio
5. Construa, publique, execute
6. Processos
7. Vínculo de porta
8. Concorrência
9. Descartabilidade
10. Dev/prod semelhantes
11. Logs
12. Processos Administrativos

https://12factor.net/pt_br/

1 - Base de Código

- Utilização de um sistema de versionamento de código
  - Git
  - TFS
  - SVN…
- Aplicações devem ser deployadas a partir de uma versão do código.
- Atualmente existe o conceito de “mono-repository”, onde em um único repositório se encontram várias aplicações, então não necessariamente 1 repositório = 1 aplicação.

2- Dependências

- Dependências são declaradas de forma explicita
- A aplicação também deve listar dependências de sistema e não somente bibliotecas (ex: Versão do node.js instalada)
- No Node.js utilizamos o package.json para cumprir esse papel

3 - Configurações

- Armazenar as configurações da aplicação no ambiente
- A configuração de uma aplicação deve ser o que deve variar entre um deploy de uma mesma versão
  - Ex: Ambientes de Dev / Homologação / Produção
  - Mesmo código mas configurações diferentes
- Não armazenar configurações no repositório da aplicação
- Depender de variáveis de ambiente para fazer a configuração da aplicação
- No Node.js utilizamos o “process.env” para acessar as variáveis de ambiente
- Podemos utilizar a biblioteca “dotenv” para desenvolvimento, fazendo a configuração do ambiente local a partir de um arquivo .env (que não deve ser rastreado no repositório da aplicação)

4 - Serviços de Apoio

- Trate serviços de apoio como recursos anexados
- Ex: Banco de dados, serviço de cache
- Serviços de apoio devem ser configurados via variáveis de ambiente como indicado no item 3 - Configuração
- Não deve haver distinção entre um serviço local ou de terceira, sendo possível trocar um serviço local por um serviço terceiro
    - Ex: Trocar serviço de email local por um externo, trocar banco de dados por um outro restaurado de backup

5 - Construa, publique, execute (build, release, run)

- Separar os passos de build, deploy e execução
- Passo de build é o processo de transformação do código do repositório em um pacote executável, baixando as dependências e fazendo as compilações necessárias.
- Passo de release pega o pacote gerado pelo passo de build e combina ele com as configurações do ambiente, gerando um artefato pronto para execução
- O passo de run executa um ou mais processos da aplicação no ambiente de execução com o release

6 - Processos

- Executar a aplicação como um ou mais processos que não tem estado
- Qualquer dado que deve ser persistido deve se utilizar um serviço *stateful* para armazenar os dados
  - Banco de dados
  - Serviço de arquivos
- Processos não devem compartilhar nada entre si (ex: memória / sistema de arquivos)
- Memória e sistema de arquivos pode ser usado por um processo como um tipo de cache temporário, e nunca deve assumir que os dados armazenados nesses lugares estarão disponíveis em um novo processo.

7 - Vínculo de porta

- Serviços devem ser expostos em uma porta
- No ambiente de execução, uma camada de roteamento é responsável por direcionar as chamadas para a porta respectiva do serviço
    - Ex: Http utiliza porta 80, https utiliza porta 443, mas o serviço pode rodar em outra porta e a camada de roteamento fará o direcionamento adequado
- No node.js, podemos definir a porta do serviço geralmente utilizando o método “listen” da biblioteca express por exemplo.

8 - Concorrência

- Uma aplicação é representada por um ou mais processos independentes
- Podemos escalonar uma aplicação simplesmente instanciando mais processos de um mesmo deploy
- Camada de roteamento é responsável por distribuir tráfego entre as instâncias do serviço.

9 - Descartabilidade

- Aplicações devem ter rápido tempo de startup e devem lidar com notificações de termino de processo de forma elegante
- Permite o escalonamento rápido da aplicação
- Ao lidarmos com o sinal de sistema *SIGTERM* (sinal de término de processo), podemos encerrar as atividades da aplicação de forma elegante
  - Fechar conexões com banco de dados
  - Devolver uma mensagem que estava em processamento de volta para a fila
  - Interromper um processo batch sem perda de informação
- Também devem ser robustos para suportar o encerramento repentino
  - Serviços de jobs podem retornar um job para a fila de execução caso percam a conexão com a aplicação
- No Node.js podemos implementar isso com “process.on('SIGTERM', handle);”

10 - Dev/prod semelhantes

- Devemos manter os ambientes o mais similar possível
- Aplicações devem ser preparadas para suportarem *deploy contínuo*
- Autores do código devem ser as mesmas pessoas responsáveis pelo deploy
- Tempo entre os intervalos de implantação em desenvolvimento/produção deve ser pequeno
- Evitar usar serviços de apoio diferentes entre desenvolvimento/produção mesmo tendo camadas de abstração, para evitar problemas só encontrados em produção.
    - Ex: Banco de dados SQLite em desenvolvimento, MySQL em produção
-

11 - Logs

- Tratar logs como streams de eventos
- Ordenados por tempo
- Aplicação não é responsável por rotear ou gerenciar os logs
- Não usar arquivos de log
- Localmente, podem ser publicados no *stdout*
- No ambiente de execução, podem ser capturados, agregados e direcionados para um serviço de gerenciamento de logs

12 - Processos Administrativos

- Processos administrativos devem ser considerados processos de execução única
  - Migração de estrutura de banco de dados
  - Criação de dados iniciais
- Devem executar em ambiente identico ao de execução
- Devem ser versionados junto com o código da aplicação
- No Node.js, podemos implementar esses processos administrativos com os *“scripts”* do package.json