Arquitetura em Camadas

Mentor: Carlos Júnior

1 - Arquitetura em Camadas

- O que é?
- Como funciona?

Arquitetura em Camadas

Arquitetura que divide o software em camadas, cada uma com uma responsabilidade específica. Trazendo assim os seguintes benefícios:

- **Organização:** A divisão do software em camadas ajuda a organizar o software e a torná-lo mais fácil de entender.
- **Manutenção:** A arquitetura em camadas torna o software mais fácil de manter, pois cada camada pode ser alterada ou atualizada independentemente das outras camadas.
- **Extensibilidade:** A arquitetura em camadas torna o software mais extensível, pois novas camadas podem ser adicionadas ou camadas existentes podem ser alteradas sem afetar as outras camadas.

Arquitetura em Camadas

Embora possa ter mais camadas ou até mesmo menos, as três camadas descritas abaixo são as mais utilizadas:

- **Camada de apresentação:** A camada de apresentação é responsável por interagir com o usuário. Ela fornece a interface do usuário e processa as entradas do usuário.
- **Camada de negócios:** A camada de negócios é responsável por processar os dados do usuário. Ela implementa os requisitos de negócios do sistema.
- **Camada de dados:** A camada de dados é responsável por armazenar os dados do usuário. Ela fornece acesso aos dados do usuário e garante a integridade dos dados.

Arquitetura em Camadas

Para nosso caso de aplicação, uma das diversas maneiras de separar as camadas seria da seguinte forma:

```
src
├──routes
├──controllers
├──services
├──models
└──databases
```

Arquitetura em Camadas

Camada routes:

Comportaria o conjunto de rotas da aplicação que terão a responsabilidade de chamar suas repectivas **controllers**. Cada arquivo nessa camada representaria uma entidade da aplicação que contém o conjunto de rotas dessa entidade.

```
// src/routes/user.js

const express = require('express')
const { usersController } = require('../controllers')

const router = express.Router()

router.get('/', usersController.listUsers)
router.post('/', /*usersController.validateUsers,*/ usersController.createUsers)

module.exports = router
```

Arquitetura em Camadas

Camada controllers:

Comportaria um conjunto de **middlewares** que terão a responsabilidade de capturar os dados da requisição, chamar a camada services - se preciso - e responder as requisições. Cada arquivo nessa camada representaria uma entidade da aplicação que contém o conjunto de **middlewares** dessa entidade.

```
// src/controllers/users.js

const { usersService } = require('../services')

const listUsers = async (req, res) => {
    // TODO: chamar a próxima camada (usersService.listUsers())

    // TODO: retornar o response
}
// ...
```

```
// src/controllers/users.js

const createUsers = async (req, res) => {
    // TODO: chamar a próxima camada (usersService.createUsers(...))

    // TODO: retornar o response
}

module.exports = {
    listUsers,
    createUsers
}
```

Arquitetura em Camadas

Camada services:

Comportaria um conjunto de **funções** que terão a responsabilidade de receber, se houver, os dados da camada controllers, chamar a camada models - se preciso - e retornar os resultados. Cada arquivo nessa camada representaria uma entidade da aplicação que contém o conjunto de **funções** dessa entidade.

```
// src/services/users.js

const { usersModel } = require('../models')

const listUsers = async () => {
    // TODO: chamar a próxima camada (usersModel.listUsers())

    // TODO: retornar o resultado
}
// ...
```

```
// src/services/users.js

const createUsers = async (user) => {
    // TODO: chamar a próxima camada (usersModel.createUsers(...))

    // TODO: retornar a resposta
}

module.exports = {
    listUsers,
    createUsers
}
```

Arquitetura em Camadas

Camada models:

Comportaria um conjunto de **funções** que terão a responsabilidade de receber - se houver - os dados da camada services, conectar com o banco de dados e retornar os resultados. Cada arquivo nessa camada representaria uma entidade da aplicação que contém o conjunto de **funções** dessa entidade.

```
// src/models/users.js

const pgConnection = require('../databases/pgConnection')

const listUsers = async () => {
    // TODO: abrir conexão com o banco e executar uma operação (pgConnection.query(...))

    // TODO: retornar o resultado
}
// ...
```

```
// src/models/users.js

const createUsers = async (user) => {
    // TODO: abrir conexão com o banco e executar uma operação (pgConnection.query(...))

    // TODO: retornar o resultado
}

module.exports = {
    listUsers,
    createUsers
}
```

Arquitetura em Camadas

Camada databases:

Comportaria um conjunto de **conexões** que terão a responsabilidade de conectar com o banco de dados e possibilitar a execução de consultas. Cada arquivo nessa camada representaria a conexão com um SGBD.

```
// src/databases/pgConnection.js

const { Pool } = require('pg')

const connection = new Pool({
    host: process.env.PG_HOST,
    user: process.env.PG_USER,
    port: process.env.PG_PORT,
    password: process.env.PG_PASSWORD,
    database: process.env.PG_DATABASE
})

module.exports = connection
```

Arquitetura em Camadas

Com tudo isso criado, basta chamar as rotas da camada **routes** em uma instância do **Express.js**:

```
// src/app.js

const { usersRoute } = require('./routes')

const app = express()

app.use(express.json())
app.use('/users', usersRoute)

app.listen(process.env.SERVER_PORT, () => {
    console.log('Server is running...')
})
```

Sua vez!

1. Refatore a API do mini projeto de JavaScript para que agora ela implemente a arquitetura em camadas.